



YAŞAR UNIVERSITY
GRADUATE SCHOOL OF NATURAL AND APPLIED SCIENCES

PHD THESIS

**MALWARE DETECTION FOR THE ANDROID
PLATFORM USING MACHINE LEARNING
TECHNIQUES**

GÖKÇER PEYNİRCİ

THESIS ADVISOR: ASSIST. PROF. DR. KORHAN KARABULUT
CO-ADVISOR: ASSIST. PROF. DR. METE EMİNAĞAOĞLU

COMPUTER ENGINEERING

PRESENTATION DATE: 07.06.2018

BORNOVA / İZMİR
JUNE 2018

We certify that, as the jury, we have read this thesis and that in our opinion it is fully adequate, in scope and in quality, as a thesis for the degree of the Doctor of Philosophy.

Jury Members:

Asst. Prof. Dr. Korhan KARABULUT
Yaşar University

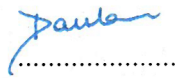
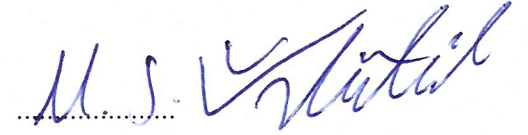
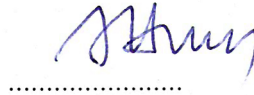
Prof. Dr. Efendi NASİBOV
Dokuz Eylül University

Assoc. Prof. Dr. Mehmet ÜNLÜTÜRK
Yaşar University

Prof. Dr. Aybars UĞUR
Ege University

Asst. Prof. Dr. Damla OĞUZ
Yaşar University

Signature:



Prof. Dr. Cüneyt GÜZELİŞ
Director of the Graduate School

ABSTRACT

Malware Detection for the Android Platform using Machine Learning Techniques
Peynirci, Gökçer

Ph.D., Computer Engineering

Advisor: Assist. Prof. Dr. Korhan KARABULUT

Co-Advisor: Assist. Prof. Dr. Mete EMİNAĞAOĞLU

June 2018

Android is the mobile operating system most frequently targeted by malware in the smartphone market with a significantly higher total market share in comparison to its competitors in addition to a much higher total number of applications. Detection of malware before it is published on the Google Play Store or any unofficial application market is very important owing to the end users' typical security inadequacy. In this Ph.D. thesis, a novel methodology of feature selection is proposed along with an Android malware detection approach that implements the proposed feature selection methodology. The machine learning approach proposed in this thesis makes use of permissions, API calls, and strings as features, which are statically extractable from the Android executables (APK files). In the proposed feature selection approach, a document frequency-based approach was designed and implemented that differs from the existing methods in the literature. The proposed methodology was tested upon two universal benchmark datasets that contain Android malware samples and promising results were obtained by using several binary classification algorithms and some ensemble learning models.

Key Words: information security, Android, malware detection, data mining, machine learning, feature selection

ÖZ

ANDROID PLATFORMU İÇİN MAKİNE ÖĞRENMESİ TEKNİKLERİ KULLANARAK KÖTÜCÜL YAZILIM TESPİTİ

Peynirci, Gökçer

Doktora Tezi, Bilgisayar Mühendisliği

Danışman: Dr. Öğr. Üyesi Korhan KARABULUT

Yardımcı Danışman: Dr. Öğr. Üyesi Mete EMİNAĞAOĞLU

Haziran 2018

Android mobil işletim sisteminin, rakiplerine kıyasla sahip olduğu oldukça yüksek toplam pazar payının yanında toplamda sayısal olarak çok daha fazla uygulamaya sahip olması dolayısıyla kötücül yazılımlar tarafından en sık hedef alınan mobil platform olduğu bilinmektedir. Son kullanıcının, tipik güvenlik yetersizliğine bağlı olarak, kötücül yazılımın Google Play Store veya herhangi bir resmi olmayan uygulama mağazasında yayımlanmadan önce tespit edilmesi hayati bir öneme sahiptir. Bu tezde, makine öğrenmesi teknikleri kullanarak yeni bir Android kötücül yazılım tespit metodolojisi yanında yeni bir öznitelik seçim metodolojisi ortaya konmuştur. Bu çalışmada sunulan makine öğrenmesi yaklaşımı, Android uygulamalarından (APK dosyaları) statik olarak çıkarılabilen, izinler (permissions), Uygulama Programlama Arayüzü çağrılarını (API calls) ve katar (string) özelliklerini kullanmaktadır. Sunulan özellik seçim metodolojisinde literatürdeki mevcut yöntemlerden farklı olarak, belge sıklığı tabanlı (document frequency-based) bir yöntem tasarlanıp uygulanmıştır. Önerilen yöntem, Android kötücül yazılım örnekleri barındıran iki evrensel temel ölçüt veri kümesi ile test edilmiş ve bazı ikili sınıflandırma algoritmaları yanı sıra bazı topluluk (ensemble) yöntemine dayalı algoritmalar da kullanılarak literatürdeki diğer modeller ve yöntemlere göre daha başarılı sayılabilecek yüksek doğrulukta sonuçlar elde edilmiştir.

Anahtar Kelimeler: bilgi güvenliği, Android, kötücül yazılım tespiti, veri madenciliği, makine öğrenmesi, öznitelik seçimi

ACKNOWLEDGEMENTS

I would like to give my special thanks and gratitude to my advisors Assist. Prof. Dr. Korhan KARABULUT and Assist. Prof. Dr. Mete EMINAĞAOĞLU for their precious guidance, efforts and patience during this Ph.D. study.

I would like to give my sincere thanks to my cousin Alican Peynirci for his support in graphic design issues throughout the writing and the printing of this thesis.

I would also like to express gratitude to my parents, who were supportive through the hard times and never lost their trust in me through this long journey.



Gökçer Peynirci
İzmir, 2018

TEXT OF OATH

I declare and honestly confirm that my study, titled “Malware Detection for the Android Platform using Machine Learning Techniques” and presented as a Ph.D. Thesis, has been written without applying to any assistance inconsistent with scientific ethics and traditions. I declare, to the best of my knowledge and belief, that all content and ideas drawn directly or indirectly from external sources are indicated in the text and listed in the list of references.

Gökçer Peynirci

Signature

.....

July 2, 2018

TABLE OF CONTENTS

ABSTRACT	iii
ÖZ	iv
ACKNOWLEDGEMENTS	v
TEXT OF OATH	vi
TABLE OF CONTENTS	vii
LIST OF FIGURES	x
LIST OF TABLES	x
SYMBOLS AND ABBREVIATIONS	xi
CHAPTER 1 INTRODUCTION	1
1.1. Android Malware	2
1.2. Machine Learning and Data Mining	2
1.3. Data Mining for Information Security.....	3
1.4. Data Mining for Cyber Security.....	4
1.5. Machine Learning and Data Mining for Malware Detection	5
1.6. Dealing with Advanced Threats	6
1.7. Thesis Contributions	7
CHAPTER 2 BACKGROUND	8
2.1. Malware: A Brief Definition.....	8
2.2. The Android Software Stack.....	8
2.2.1. Linux Kernel	9
2.2.2. Libraries	9
2.2.3. Android Runtime.....	10
2.2.4. Application Framework	10
2.2.5. Applications	10
2.3. Dalvik Virtual Machine.....	11
2.4. Android Applications	11
2.4.1. Components of an Android Application	12
2.4.2. The Manifest File	14
2.4.3. Application Permission Structure	15

2.4.4. Android Application Programming Interface (API)	16
2.5. Android's Build Process	17
2.6. Malware Analysis Techniques	20
2.6.1. Static Analysis	20
2.6.2. Dynamic Analysis.....	21
2.7. Android Malware Infection Vectors	22
2.7.1. Client-Side Software Vulnerabilities	22
2.7.2. Vulnerabilities in the Smartphone's OS	22
2.7.3. Social Engineering.....	22
2.7.4. Third Party App Stores	22
2.7.5. Brute Forcing the User's Accounts.....	23
2.7.6. Drive-By Downloads on the Android.....	23
2.8. Relevant Data Mining and Machine Learning Background	23
2.8.1. K-Nearest Neighbor (K-NN)	24
2.8.2. Support Vector Machines (SVMs)	24
2.8.3. Decision Trees	25
2.8.4. Naive Bayes.....	27
2.8.5. Neural Networks.....	28
2.8.6. Radial Basis Function (RBF).....	30
2.8.7. Ensemble Learning Methods	31
CHAPTER 3 LITERATURE REVIEW	33
3.1. Approaches for Malware Detection on the Android that use Machine Learning Methods.....	33
3.2. Approaches that Utilize Text Mining for Malware Detection	46
CHAPTER 4 PROPOSED METHODOLOGY	49
4.1. Data Collection and Pre-processing.....	49
4.2. Feature Extraction.....	50
4.2.1 Permissions	50
4.2.2. API Calls.....	51
4.2.3. Strings	52

4.3. A Novel Document Frequency-Based Approach to Feature Selection.....	52
4.3.1. Term Frequency	52
4.3.2. Inverse Document Frequency	53
4.3.3. Description of the Document Frequency-Based Approach	53
4.4. The Architecture of the Android Malware Detection Methodology.....	57
4.4.1. Combined Feature Set Model	58
4.4.2. Classification of Android Malware via Machine Learning	58
4.5. Limitations of the Proposed Methodology	59
CHAPTER 5 EXPERIMENTS AND RESULTS.....	61
5.1. Standard Metrics used for Evaluating the Classifiers	64
5.1.1. Confusion Matrix and the Related Metrics	64
5.2. Cross-validation.....	66
5.2.1. 10-fold Cross-Validation.....	66
5.3. Results from the First MalGenome Dataset.....	67
5.4. Results from the Second MalGenome Dataset.....	68
5.5. Results from the AndroZoo Dataset.....	76
CHAPTER 6 CONCLUSIONS	79
6.1. Summary of Thesis Contributions.....	79
6.2. Directions for Future Work.....	81
6.2.1. New Experiments with Bigger Datasets.....	81
6.2.2. Additional Research for Ensemble Classifiers with the AndroZoo Dataset	81
6.2.3. Alternative Methodologies for Feature Selection	82
6.2.4. Malware Detection Tool for Android Platforms	82
REFERENCES.....	83
APPENDIX 1 – Descriptions of the Selected Attributes for the MalGenome Dataset ..	90
APPENDIX 2 – Descriptions of the Selected Attributes for the AndroZoo Dataset.....	93
APPENDIX 3 – Detailed Results Obtained in WEKA for the First MalGenome Dataset 	95
APPENDIX 4 – Detailed Results Obtained in WEKA for the AndroZoo Dataset	134

LIST OF FIGURES

Figure 2.1. Android Software Stack (ZDNet, 2008)	9
Figure 2.2. Building and Running an Application in Android (stuff.mit.edu, 2016)	17
Figure 2.3. Creation of the Final APK Archive File (stuff.mit.edu, 2016).....	19
Figure 2.4. Support Vector Machine (SVM) (Han, Kamber, & Pei, 2006).....	25
Figure 2.5. Artificial Neural Network that contains one hidden layer.....	29
Figure 4.1. Malware detection process used in this thesis.....	57
Figure 5.1. 10-fold Cross-Validation.....	66

LIST OF TABLES

Table 3.1. Results Comparison of the Approaches in the Literature Review	45
Table 4.1. Selected attributes from the MalGenome dataset	56
Table 4.2. Selected attributes from the AndroZoo dataset.....	56
Table 5.1. Confusion Matrix.....	65
Table 5.2. MalGenome Dataset Accuracy Values	67
Table 5.3. MalGenome Dataset True Positive Rates	67
Table 5.4. MalGenome Dataset True Negative Rates.....	68
Table 5.5. MalGenome 2 nd Dataset Accuracy Values Part 1	69
Table 5.6. MalGenome 2 nd Dataset Accuracy Values Part 2	70
Table 5.7. MalGenome 2 nd Dataset True Positive Rates Part 1	71
Table 5.8. MalGenome 2 nd Dataset True Positive Rates Part 2	72
Table 5.9. MalGenome 2 nd Dataset True Negative Rates Part 1.....	74
Table 5.10. MalGenome 2 nd Dataset True Negative Rates Part 2.....	75
Table 5.11. AndroZoo Dataset Accuracy Values	77
Table 5.12. AndroZoo Dataset True Positive Rates	77
Table 5.13. AndroZoo Dataset True Negative Rates.....	77

SYMBOLS AND ABBREVIATIONS

ABBREVIATIONS:

Malware malicious software

IT Information Technology

APT Advanced Persistent Threat

app applications

malcode malicious code

DDoS Distributed Denial of Service

DOD Department of Defense

IDS Intrusion Detection System

IDF Inverse Document Frequency

API Application Programming Interface

BFGS Broyden–Fletcher–Goldfarb–Shanno

OS Operating System

Dalvik VM Dalvik Virtual Machine

VM Virtual Machine

GPS Global Positioning System

APK Android Package

JIT Just in Time

CFG Control Flow Graph

UID User ID

DEX Dalvik Executable

URIs Uniform Resource Indicators

JRE Java Runtime Environment

ANN Artificial Neural Network

SVM Support Vector Machine

k-NN k-Nearest Neighbor

WEKA Waikato Environment for Knowledge Analysis

MLP Multilayer Perceptron

RBF Radial Basis Function

F-score Fisher score

IDS Intrusion Detection System

PART Partial Decision Trees

RIDOR Ripple-Down Rule

TF Term Frequency

TF-IDF Term Frequency–Inverse Document Frequency

MalGenome Android Malware Genome

C&C Command and Control

NNge Non-Nested Generalized Exemplars

LAC Lazy Associative Classifier

SYMBOLS:

N The number of cases in the training set.

D The training set and the associated class labels.

X A data tuple or a row in database.

H A hypothesis that states the sample X belongs to a predefined class C.

k The number of the training samples.

CHAPTER 1

INTRODUCTION

As the information security industry becomes more aligned with providing protection through utilizing cyber intelligence each year, the need for expertise in data mining and machine learning increases as well. The problem of malicious software (malware) detection on the Android, Microsoft Windows or any other platform has never been solved completely and this can be attributed to security weaknesses in the state of the art computing platforms and digital communication protocols we have in use today.

New types and versions of malware are produced by cyber criminals every day, for which no unique detection signature is readily available. Even if the cybersecurity experts come up with a very good idea and model to detect malware today, malware with unique features may be produced that is unlike any malware we have seen to date, that surpasses the technicality and complexity of the malware we have seen so far.

Even though the field of data mining for malware detection is not very new, it is still in progress for improvement with academic studies focusing on this field and possible industrial applications of the proposed methodologies. Malware developers continuously strive to develop less detectable and more catastrophic (causing more damage, having more functionality, becoming widespread etc.) malware applications. This, in turn, creates a strict requirement of encountering these types of threats that put the security experts in a position of sustained adaptation and response to the dynamically changing Information Technology (IT) landscape shaped every day by improvements in malware and other security threats.

To assess the importance of malware in the current security landscape irrespective of computing platform, we need to look at what Advanced Persistent Threats (APTs), targeted cyber attacks and threats from state-sponsored actors have in common: the common element is the use of malware at some stage of their cyber attacks.

This study was proposed with the motivation that utilization of machine learning and data mining for detection of Android malware applications that could lead to new

opportunities to make the mobile landscape more secure. Being aware of the past successful research in this field, I hope my results, findings and conclusions will be a valuable contribution.

1.1. Android Malware

As our mobile phones integrate into our personal and professional lives more and more each day, they are targeted by cyber criminals more frequently than ever. Since these devices contain valuable private information and access to financial services such as internet banking or e-commerce purchases, making sure that adequate security is provided on the mobile phone is essential for every user.

Android applications (apps) can be obtained either via the official app store (Google Play Store), or via third-party unofficial stores such as GetJar or Slide ME. The Google Play Store was constructed with the intention to make a store that fulfills all app requirements of the typical user and at the same time provides the adequate amount of security for the downloaders of the apps. Towards this endeavor, Android phones come with a security option that prevents app installs from third-party stores, which can be turned off by users. However, it is not recommended to be turned off for typical users.

The reason for such precautions is the possibility that the adversaries may have injected benign looking Android apps with malicious code (malcode). In order to compute and assess the level of threat that the users face from third-party store app download, we need to look at the percentage of malware infections from the Google Play compared to malware infections from the third-party app stores. According to an analysis conducted by Cheetah Mobile (CheetahMobile, 2014), malware coming from third-party markets account for 99.86% of all malware infections compared to only 0.14% from the Google Play.

1.2. Machine Learning and Data Mining

Machine learning is a kind of artificial intelligence whereby an algorithm or method will extract patterns out of data. The aim is to automatically infer and generalize patterns from data. Machine learning can be incorporated into many facets of our digital life including but not limited to: face recognition, handwriting digit recognition, spam filtering in an email, and product recommendations from e-commerce sites. To elaborate on one of the examples, in handwriting digit recognition, the aim is to infer

associations between drawn shapes and particular letters, while taking into account variations of the same letter.

Machine learning lies at the intersection of computer science, engineering, and statistics. Any field that needs to interpret and act on data can benefit from machine learning techniques. Data mining is a similar field to machine learning, in which we use many techniques of machine learning. However, in data mining, the part played by databases is stronger.

Data mining is also known as “knowledge-discovery in databases” and it is an extension of exploratory data analysis and has the same goals: the discovery of unknown and unanticipated structure in the data. The chief distinction lies in the size and dimensionality of the datasets involved. Data mining, in general, deals with much more massive datasets for which highly interactive analysis is not feasible (Wegman, 2002).

Let us note the difference between learning from a fully labeled set of examples and a fully unlabelled set. If learning is being performed from a fully labeled set of examples as it is in the case of this thesis, it is called supervised learning. On the other hand, unsupervised learning is performed on a fully unlabelled set as opposed to a labeled set. The activity of learning from labeled or unlabelled sets is called discovery or mining. There is also a mid-ground between supervised learning and unsupervised learning, which is semi-supervised learning, where a partially set of examples are used for the learning activity.

We can talk about four different styles of learning in data mining applications. The first one is classification learning, where the learning needs to happen from a set of classified examples. The second one is association learning, where any association among features is sought, while the third one is clustering, where groups of examples that belong together are sought and the last one is numeric prediction, where the outcome to be predicted is not a discrete class but a numeric quantity (Witten & Frank, 2005).

1.3. Data Mining for Information Security

Data mining is considered as a promising solution to the ever-growing problem of information security. Application of data mining related solutions to information

security emerges as an alternative method of solving problems. Various information security applications make use of data mining or machine learning techniques such as classification, clustering or association rule mining. Induction algorithms are incorporated in such solutions that explore data in order to discover hidden patterns and build predictive models. Such techniques and algorithms have proved to tackle most of the information security challenges effectively.

In order to prevent and thwart the risks to information security, attack pattern generalization and discovery present a great opportunity for data mining and information security communities. Classification, association rules, and clustering mechanisms can be incorporated into the data at hand, both before and after an information security compromise that maps the attack patterns of each individual attack. Powerful software solutions can be implemented that incorporate aforementioned techniques in order to deal with latest threats and risks such as Distributed Denial of Service (DDoS) attacks, host-based intrusions, access control violations and malicious code detection.

Several data mining use cases applied to information security related issues include (Bhatnagar & Sharma, 2012):

- Identification of various anomalies and malware in the system by classifying the benign and anomalous activities into different groups and classifying incoming data accordingly.
- Extraction of various security requirements, performing fuzzing techniques to identify vulnerabilities, defining and finding audit trails and establishing security policies.
- Detection of various cybercrimes, such as credit card fraud, money laundering frauds and other financial crimes and classifying criminals into classes according to behavior.

1.4. Data Mining for Cyber Security

Intrusion detection and malware detection are two areas heavily researched in data mining for cybersecurity. Even though both areas are fairly new compared to many classical theoretical computer science topics, there has been active research going on

in these topics for 17 years now, considering the first research paper on data mining for malware detection was released in 2001 (Schultz, Eskin, Zadok, & Stolfo, 2001).

Cyberspace is defined as “a global domain within the information environment consisting of the interdependent network of information technology infrastructures, including the Internet, telecommunication networks, computer systems, and embedded processors and controllers” in the Joint Publication 1-02, Department of Defense (DoD) Dictionary of Military and Associated Terms (DoD, 2010).

Massive amounts of data are collected by sensors placed at cyber systems such as firewalls, Intrusion Detection System (IDS), and anti-virus. This data, either network traffic data or log data is ripe for application of data mining for the purpose of unearthing valuable patterns and relationships to be used in security research. Data mining may enable us new capabilities that were not possible before. Apart from all the tactical operations necessary to defend a cyber-system, it has become vital to continuously sift through vast amounts of sensor data that could be made more efficient with advances in data mining techniques to accurately map the attack surface, collect and integrate data, extract knowledge and produce useful visualizations (Blowers et al., 2014). Strategic coordination of all the sources of data is becoming a central piece of effective cyber defense. This accumulation of data from various sources can easily become what we call big data.

Dealing with large and fast-growing sources of data obliged us to build new techniques, models and a new kind of computing infrastructure to process, analyze and store data. Amongst many considerations when dealing with massive amounts of data, one challenge is in having a computing infrastructure that can ingest, validate, and analyze high volumes (size and/or rate) of data. Another challenge is in assessing mixed data (structured and unstructured) from multiple sources. It is often very difficult to deal with unpredictable content with no apparent schema or structure, and often a challenge enabling real-time or near-real-time collection, analysis, and results (Villars, Olofson, & Eastwood, 2011).

1.5. Machine Learning and Data Mining for Malware Detection

The security industry is locked inside the endless loop of generating a specific signature to one kind of malware only for this specific kind of malware to be later modified to evade the present detection mechanisms (David & Netanyahu, 2015). This

never-ending loop puts the security industry in a position of attempting to defend against all attack vectors from the full cyberattack spectrum with the attackers continually improving their tools and attack methods.

Instead of taking the approach of one unique signature to one malware sample, data mining and machine learning propose a fundamentally different approach to tackle the problem of detecting malware.

Once malcode injection occurs, the traffic or the app is not the same as it was before; there are clear indications of that malcode injection somewhere in the traffic or the source code. Such changes that occur through injection of malcode provide us the ability to learn about such changes to measure the degree of maliciousness of an executable in order to determine whether the file in question is a benign one or a malicious one.

Data mining is a kind of prediction in which we look for meaningful patterns amongst data and from these make classifications, form clusters or predict numeric values. Either within the network traffic or source code of an app, it is in a clean state when there are no malcode injections inside the traffic or source code of the app.

1.6. Dealing with Advanced Threats

Cyber adversaries are getting more sophisticated every day and targeting organizations, corporations, and governments. We are facing what we call advanced threats, which is beyond the attack sophistication threshold we are used to and it involves advanced malware, targeted attacks, and APTs. The primary aim of such adversaries is to conduct industrial espionage, undermine business and financial operations, and/or sabotage critical infrastructure. Many organizations today lack the workforce with the adequate skills to combat such threats.

Traditional approaches to security, which can be defined as security systems that leverage a rule, pattern, signature, or algorithm, based approach to detect malware or cyberattacks, are no longer effective against advanced threats. The main pitfall of traditional approaches to security is the requirement of constant updates and influx of rules, signatures or patterns to identify and mitigate each individual malware or threat.

Software and hardware solutions with data analytics at its core are rapidly becoming the cornerstone of protection in cyber and information security domains. Advances in

machine learning is a promising approach to deal with the ever-changing and evolving advanced threats. Machine learning techniques are finding widespread applications and implementations in dealing with a wide range of security issues with many machine learning techniques, algorithms, and tools being used by security experts and researchers to tackle some of the most advanced threats we are facing today.

Machine learning, when utilized in the right way can benefit us with faster identification of previously unidentified vulnerabilities in software or hardware, detection of complex cyber attacks and malware, and mitigation of insider threats through detection of anomalous user behavior.

1.7. Thesis Contributions

The main contributions of this thesis will be twofold:

- A novel methodology of feature selection in the machine learning process for malware detection against malicious Android executables.
- An exploration of ensemble learning methods and techniques among several machine learning algorithms for Android malware detection that utilizes permissions, Application Programming Interface (API) calls, and strings as features.

The feature selection methodology proposed in this thesis differs from the methodologies already present in the literature in accordance with selecting attributes that have the lowest possible Inverse Document Frequency (IDF) values in the malware dataset and at the same time the highest possible IDF values in the benign dataset, which is different from the present approaches in the literature.

Secondly, the methodology of Android malware detection using machine learning techniques is new, based on exploring ensemble learning with different algorithms by using specific sets of selected features.

I hope these two main contributions of this thesis, will be valuable towards the efforts of researchers and the security industry in combating and providing better detection against Android malware detection.

CHAPTER 2

BACKGROUND

In this chapter, a broad academic background of essential topics composing this thesis will be provided in order to make sure the reader is adequately familiar with them to understand the following chapters.

2.1. Malware: A Brief Definition

Referred to by many names including malicious software, malicious code or malcode, malware is basically software produced to harm a target system either by cyber espionage, encrypting of files to ask for ransom (ransomware) or erasing of critical Operating System (OS) files to make it unusable.

This definition platform independently applies to all the programs that could be considered as a malware, the key point for different platforms is that Android malware must be executable on the Android OS, whereas a Microsoft Windows malware must be executable on the Windows platform. Malware that is executable on multiple platforms may also be produced depending on how portable the written code is.

2.2. The Android Software Stack

As illustrated in Figure 2.1., the Android Software Stack is composed of four main layers and the green items are components written in native code (C/C++), while blue items are Java components interpreted and executed by the Dalvik Virtual Machine (Dalvik VM). The bottom red layer is the Linux kernel components that run in the kernel space.

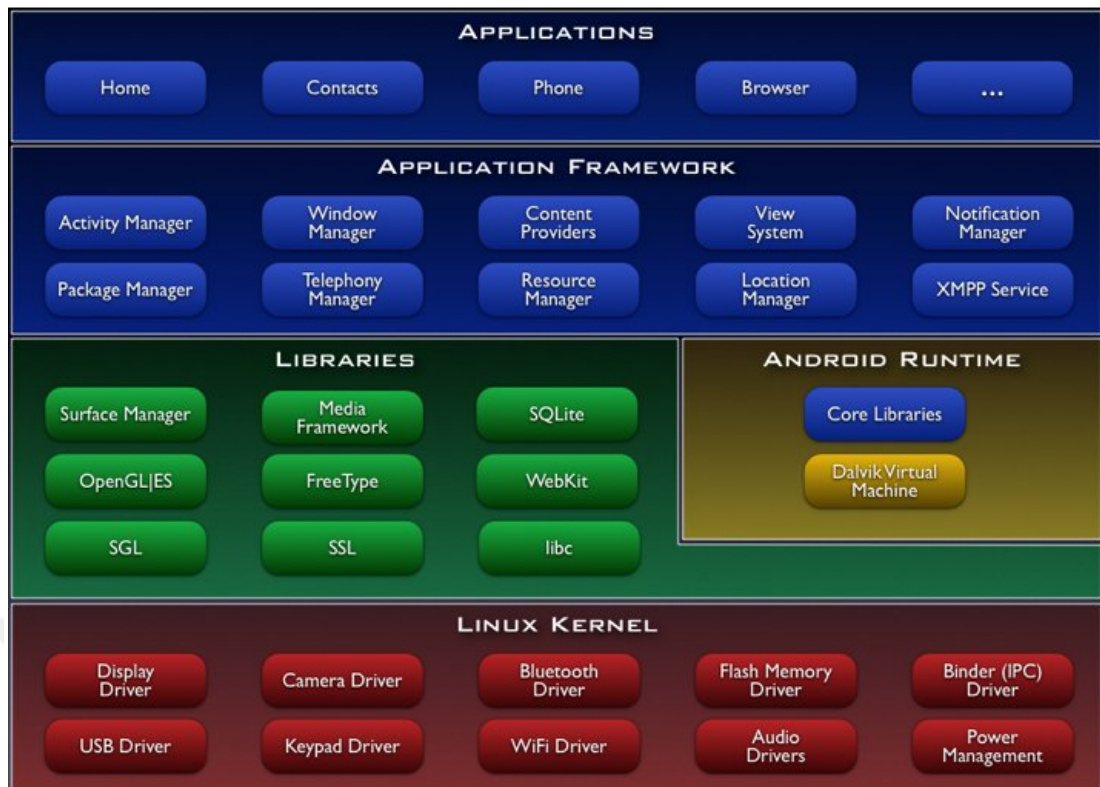


Figure 2.1. Android Software Stack (ZDNet, 2008)

In the following subsections, Android abstraction layers are discussed starting from the Linux Kernel.

2.2.1. Linux Kernel

A specialized version of the Linux Kernel with a few special additions is utilized by the Android. The additions include wakelocks (mechanisms to indicate that apps need to have the device stay on), a memory management system that is more aggressive in preserving memory, the Binder IPC driver, and other features that are important for a mobile embedded platform like the Android.

2.2.2. Libraries

The Libraries component exposes a set of native C/C++ libraries to the Application Framework and Android Runtime. These libraries are mostly external with only minor modifications such as OpenSSL, WebKit, and bzip2. The crucial C libraries, codenamed the Bionic, were ported from BSD's libc and were rewritten to support ARM hardware and Android's own implementation of pthreads based on Linux futexes.

2.2.3. Android Runtime

Android Runtime, which acts as a middleware component, consists of the Dalvik VM and a set of Core Libraries. The Dalvik VM is responsible for the execution of applications on the Android and is discussed in detail in Section 2.3.

The core libraries are an implementation of general purpose APIs and there are two different types of core libraries:

1. Core libraries for the Dalvik VM.
2. Core libraries for Java programming language interoperability.

The first category of core libraries is used in processing or modifying Virtual Machine (VM) specific information and is generally used when bytecode has to be loaded into memory. The second category coming from the Apaches Harmony enables Java interoperability by implementing the popular Java packages such as `java.lang` and `java.util`.

2.2.4. Application Framework

The Application Framework layer provides many higher-level services to apps in the form of Java classes where the developers are allowed to inherit in their own apps. Most components in this layer are implemented as official system apps and run as background processes on the device.

Some components are responsible for managing baseline phone functions such as receiving phone calls, receiving text messages or monitoring battery usage, while some have more focused uses cases such as providing access to the system's location services given the phone has Global Positioning System (GPS) capabilities.

2.2.5. Applications

On the top of the Android framework, there are applications. All apps such as home, contact, settings, games, browsers use the Android framework that, in turn, uses the Android runtime and libraries. Android runtime and native libraries use the Linux Kernel. Android apps are discussed in more detail in Section 2.4.

2.3. Dalvik Virtual Machine

A runtime comprises of software instructions that execute when a program is running. These instructions are tasked with translating the applications' code into the machine code that the device is capable of running. Android makes use of a virtual machine as its runtime environment in order to run the Android Package (APK) files that constitute an Android application.

Dalvik has been the default virtual machine that runs applications on top of device hardware since Android was started back in 2007. Dalvik runtime uses Just-in-Time (JIT) compilation method first introduced in Android 2.2 Froyo, in order to interpret the bytecode. JIT means applications are partially compiled and built, in addition, meaning each time an application is launched, and it must be compiled first. Introduced as an improvement over the previous conventional interpreter approach that compiled and ran code line by line, the downside is the huge overhead when launching applications.

Two major advantages of this approach are as follows: Firstly, as the code is isolated from the core, in case of an intentional or unintentional security threat, the risk is contained within the virtual machine, thus, not affecting the primary OS. Secondly, the code can be compiled on another platform and can still be executed on the mobile platform using the virtual machine.

2.4. Android Applications

Android applications are in an executable format as APK files. APK files are signed compressed files that contain the app's bytecode along with all its data, resources, third-party libraries and a manifest file that describes the app itself.

Android apps are run in a sandboxed environment to improve security. Apps receive a unique Linux User ID (UID) from the Android OS during installation on the device. Access permissions to the files of an app are set in a way to enable the app itself only to access them. Each app is granted its own VM during runtime resulting in app code that is completely isolated from other applications.

2.4.1. Components of an Android Application

The essential building blocks of an Android application are components. Each Android application is composed of four standard components that manage different parts of the application's functions. These four types of application components are explained below:

2.4.1.1. Activities

Activities provide a single screen and an interface. Each application may have activities for carrying out different tasks such as reading e-mail for an e-mail application or depicting available routes in a navigation application. Each activity works independently from each other to form a cohesive user experience for a specific app. A different app can start activities belonging to another app (as long as permission is given), for example, a camera app may start the activity in the e-mail app in order to e-mail a photograph.

2.4.1.2. Services

Services provide background functionality for long-term operations that do not require an interface. Services are similar to activities; on the other hand, the only main difference is that there is no requirement of an interface for each activity. An example of a service may be music playing in the background while the user is using another app or downloading data over the network without blocking user interaction by use of an activity. Services can be started by other components of the app such as an activity or a broadcast receiver.

2.4.1.3. Content providers

Shared app data are managed by content providers. The data may be stored in the file system, an SQLite database or any other persistent storage location the app can access. As long as the content provider allows it, other apps can query or even modify the data. Android system provides a content provider that manages the user's contacts information and an app with the required permissions can send queries to the content provider in order to read and write information about a particular contact.

2.4.1.4. Broadcast receivers

A broadcast receiver listens for specific system-wide broadcast announcements to pick them up if they were the intended recipient. While many broadcasts originate from the

system such as low battery, apps can also initiate broadcasts, for instance, to let other apps know that some data has been downloaded to the device and is available for them to use.

Broadcast receivers are generally used to act as a gateway to other components and do not have a user interface. They are tasked with initiating a background service to perform a task based on a specific event.

There are two types of broadcasts: non-ordered and ordered. Non-ordered broadcasts are sent to all interested receivers at the same time, on the other hand, ordered broadcasts are first sent to the receiver with the highest priority, before being forwarded to the receiver with the second highest priority. An example of a non-ordered broadcast is the battery low announcement, while an example to the ordered broadcast is an incoming SMS text message announcement.

Upon receiving an ordered broadcast, the receiver may decide to abort the broadcast so that it is not forwarded to other receivers. This allows vendors to develop alternatives to the official Android apps such as an alternative text message manager that can disable the official Android messaging application by using a higher priority receiver and aborting the broadcast after handling the incoming message.

2.4.1.5. Intents

Android's solution to establishing communication amongst application components is handled using a message routing system based on Uniform Resource Indicators (URIs). Asynchronous messages called "intents" are used to activate components such as activities, services and broadcast receivers. Intents are nearly equivalent to parameters passed to API calls and the fundamental differences between API calls and intents' way of invoking components are listed below:

- API calls are synchronous while intent based invocations are asynchronous.
- API calls are compile-time bindings while intent based calls are run-time bindings.

In order to listen for an intent, intent filters need to be implemented that specifies the types of intent that an activity, service, or broadcast receiver can respond to. An intent filter declares the capabilities of a component. It specifies the tasks an activity or

service can achieve and which types of broadcasts a receiver can handle. It allows the corresponding component to receive Intents of the declared type.

Intent filters are typically defined in the `AndroidManifest.xml` file. An Intent Filters is defined by its category, action and data filters. It can also contain additional metadata.

When an intent is broadcast and received by the relevant listener, the intent filter is invoked by the Android platform to accomplish the job. This means, both of the components are not aware of each other's existence and can still work together to give the desired result for the end user.

Some intents may require specific permissions to be sent, while system intents can be sent by processes that have the system's UID. The latter, cannot be sent by an application regardless of the permissions they hold and they can only be sent by the system processes.

2.4.2. The Manifest File

Every Android application must have an "`AndroidManifest.xml`" file. It is some sort of a configuration file in which references to the implemented components exist. This file describes each component of the application and the components' interaction among themselves. All the components of an application must be declared in this file, which resides in the root of the app project directory. Activities and services that are not declared in the manifest cannot be run.

Broadcast receivers, on the other hand, can either be declared in the manifest or may be registered later dynamically via the `registerReceiver()` method. The manifest additionally specifies application requirements such as special hardware requirements (e.g., camera, temperature sensor), or the minimal API version required to run the app.

In order to access the protected components (e.g., external storage, accessing the contact's list), an application must be granted the corresponding permission. The list of permissions required by the app must be defined in the app's `AndroidManifest.xml`. This way, during runtime, the Android OS can prompt the user to grant the specific required permission(s) to enable the app to access these components through specific APIs.

Inside the OS, the protected components have a unique Linux group ID, granting of the corresponding permission makes the app's VM a member of the corresponding unique group, thus enabling access to the restricted components.

2.4.3. Application Permission Structure

Permissions in the Android OS are enforced using permission validation mechanisms that must be invoked by some key components. Particularly, the system process is tasked with implementing the permission validation mechanism with several invocations spread throughout the API. The key components forming the Android permission enforcement model will be explained below.

Third-party applications on the Android platform are provided with an extensive API that provides access to phone hardware, settings, and user data. Access to security or privacy sensitive parts of the API is governed by the permissions security feature. Prior to installation of each application, the user is presented with a list of permission she/he must accept in order for the application to function properly. Each application developer must determine the required list of permissions beforehand and if the user spots something dubious, they can cancel the installation altogether.

This gives a user the chance to evaluate an application as potentially dangerous or benign to some level. However, the fact that most people are negligent about reading such information is the weakest point of this security technique employed by the Android OS.

There are 134 (Felt et al., 2012) officially defined application permissions in total divided into four protection levels, each level enforcing a different security policy. From low risk to high risk there are:

2.4.3.1. Normal Permissions

Includes permissions that present the lowest risk to the user thanks to the use of API calls that cannot be used to do harm. They provide access only to isolated application-level features, posing minimal risk to other applications, the system, or the user. Such types of permissions are given automatically without requiring the explicit consent of the user.

2.4.3.2. Dangerous Permissions

Permissions falling in this category have access to API calls that could be potentially malicious and could enable access to private user data. They provide a wider access range to device resources and give requesting applications control over the device that can affect the user negatively. Applications requiring these kinds of permissions will ask for the user's explicit approval prior to installation.

2.4.3.3. Signature Permission

A type of permission that the system grants only if the requesting application is signed with the same certificate as the application that declared the permission. Given the certificates match, the system automatically grants the permission without asking for the user's explicit approval.

2.4.3.4. SignatureOrSystem Permission

This permission type is granted only to applications that reside in the system image or that are signed with the same certificate as the application that declared the permission. This permission type should normally be avoided, as the protection level provided by the signature permission should be enough regardless of where an application is stored. This permission is reserved for certain special situations such as when multiple vendors want to embed applications inside the system image and need to share specific features of the application explicitly.

This model of Android security is static as an application needs to obtain a permission once and the list of permissions the application has cannot be modified during the lifetime of the application on the device.

2.4.4. Android Application Programming Interface (API)

The Android public API is composed of 8,648 distinct methods (Chin, Felt, Greenwood, & Wagner, 2011), some of which are protected by permissions. However, no centralized policies are in place to execute permission checks when an API is called.

The Android API framework has two distinct parts, one of them is a library residing in each application's virtual machine and another is an implementation of the API that runs as a system process. The library provides the necessary means for interacting with the API implementation.

The API implementation in the system process is not bound by the restrictions brought about by the permissions systems, while the API library runs bound by the set of permissions accepted during the installation of the application.

There are three steps to handle API calls in the Android OS:

Firstly, the application invokes the public API in the library, secondly, the library invokes a private interface (an RPC stub) and lastly, the RPC stub starts an RPC request that makes the system process instruct a system service to perform the desired operation.

The API implementation in the system process holds the corresponding permission checks for each application. The permission validation mechanism is called to check whether the invoking application has the necessary permissions.

2.5. Android's Build Process

The Android build process consists of compiling the Android modules and packaging them into .apk files according to the given build settings. The .apk file for each application contains all the information necessary to run an application on a device including compiled .dex files (.class files converted to Dalvik bytecode), a binary version of the AndroidManifest.xml file, compiled resources (resources.arsc) and decompiled resource files for an application.

Given the developer is using Android development tools, the build system can sign the application when building it for debugging, whereas a certificate to sign the app must be obtained and used when building the app for release.

An illustration of building and running an application can be seen in Figure 2.2.

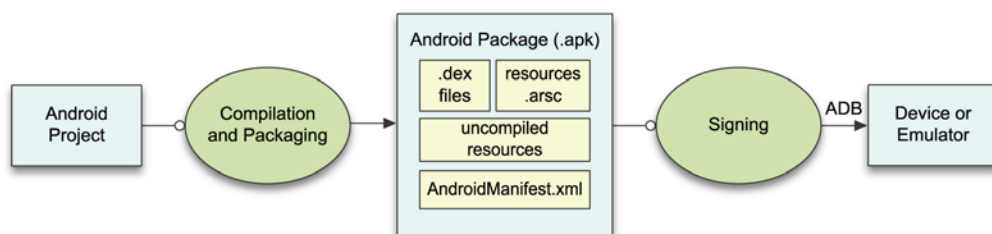


Figure 1.2. Building and Running an Application in Android (stuff.mit.edu, 2016)

The steps required to be carried in order to create the final APK file from an application's resource files are explained in the following paragraphs.

The Android Asset Packaging Tool takes and compiles an application's resource files such as the AndroidManifest.xml file and the XML files for the activities. Any existing .aidl interfaces are converted into Java interfaces. All the Java code, including the R.java and .aidl files, is compiled and .class files are output. The dex tool converts the .class files and third-party libraries into Dalvik bytecode in order to package them in the final .apk file.

Apkbuilder tool takes all non-compiled resources (such as images), compiled resources, and the .dex files for packaging into an .apk file. After building the APK file, it must be signed with either a debug or release key before installing it to a device. As the last step, given the application is being signed in release mode, it must be aligned using the zipalign tool in order to decrease memory usage during execution. A detailed illustration of the above process can be seen in Figure 2.3.

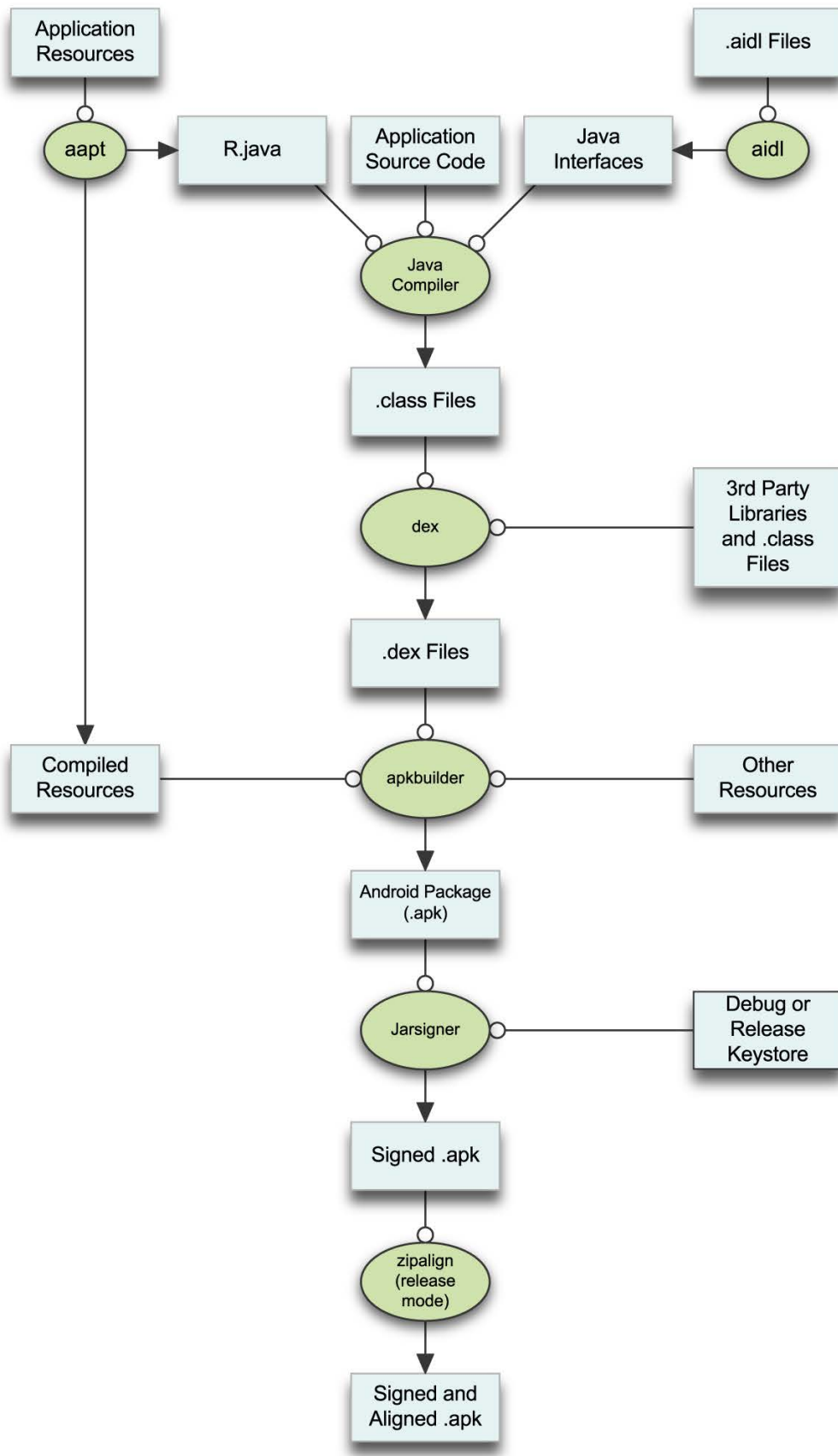


Figure 2.3. Creation of the Final APK Archive File (stuff.mit.edu, 2016)

2.6. Malware Analysis Techniques

If we consider a mobile anti-virus company that uses a signature-based Android malware detection method, the following scenario would be possible. After receiving a sample worth analyzing, they need to find a pattern that correctly identifies the sample. The identified signature should be generic enough to match the other variants of the same malware and at the same time should not produce false positives (Egele, Scholte, Kirda, & Kruegel, 2008). This manual inspection method is both labor intensive and likely to produce errors. Assuming that the source code is not available for analysis, as is the general case, the analysis needs to be performed on the binary level.

In this section, two main techniques of analyzing malware, which are static and dynamic analysis, will be explained.

2.6.1. Static Analysis

In static analysis or reverse engineering of a malicious binary, the file in question is examined without actually executing its code. In static analysis, the aim is to arrive at an intermediate representation of the program that has the same control and data flows as the original program source code. This includes the use of techniques such as disassembling, decompiling, unpacking etc., in order to arrive at a human understandable piece of code.

This type of analysis can provide useful results in analyzing compiler-generated binaries. However, it is mostly inadequate in identifying obfuscated, polymorphic or metamorphic code. In order to conduct static analysis of obfuscated malware, it must be de-obfuscated first by using corresponding de-obfuscation techniques.

When compiling the source code of a program into a binary executable, some information such as the size of data structures or variables are lost which can make the task of analysis and detection harder through static analysis. However, static analysis can still bear useful results, for example, producing a call graph can give an analyst an overview of what functions are invoked from what parts of the code (Egele et al., 2008).

2.6.2. Dynamic Analysis

In dynamic analysis, the behavior of the executable is monitored during execution, thus giving us the ability to oversee the tasks performed by it. The program's instructions are executed either by single step execution, dynamic instrumentation or whole system emulation to obtain a trace of the application's executed instructions (Roundy & Miller, 2010). The obtained trace is used to construct an analysis artifact, which is used to shed light on the program's unpacking behavior.

The main drawback of this approach is that some malware may contain functionality triggered only under certain conditions, meaning the malware may hide its true function unless certain conditions are met and may be resistant against running in virtual environments or use of sandboxing techniques.

Different techniques can be used to conduct a dynamic analysis of an executable, such as function call monitoring, function parameter analysis or information flow tracking. A brief description of each technique is given below.

Firstly, in function call monitoring, function calls are intercepted by a process called hooking by the invocation of a special hook function. This function implements the required analysis functionality by recording its invocation to a log file or analyzing input parameters. Secondly, in function parameter analysis, the focus is on the actual values that are passed when a function is invoked. This tracking of parameters and function return values enables the correlation of individual function calls that operate on the same object. Lastly, in information flow tracking, the focus is on how the program processes data and the aim is to shed light on the propagation of "interesting" data throughout program execution (Egele et al., 2008).

The analysis component that supervises the operations of the program under inspection needs to be executed in a higher OS privilege level compared to that of the program in order to prevent access to the analysis component. Otherwise, if both are on the same privilege level, the analysis component will have to employ stealth techniques to hide its presence from the program (Egele et al., 2008).

A hybrid approach can be employed in which the code is dynamically analyzed for pieces of code where static analysis fails.

2.7. Android Malware Infection Vectors

Malware may use various infection vectors to infect a smartphone. In this section, the most likely vectors for Android malware infection are detailed.

2.7.1. Client-Side Software Vulnerabilities

Vulnerabilities in applications may be exploited to execute code on a target system remotely. This software may take the form of a web browser, PDF reader, Java Runtime Environment (JRE) or any program with exploitable bugs in its source code. Old versions of these applications or even up to date versions may share common vulnerabilities that can be exploited by malware for infection and spreading.

2.7.2. Vulnerabilities in the Smartphone's OS

Given the smartphone's OS is not entirely free of security vulnerabilities, hackers may exploit such vulnerabilities to infect the phone with malware. In other words, keeping the Android OS up to date can be a great barrier against malware infection.

2.7.3. Social Engineering

Mostly overlooked, but a common vector of infection for malware is through use of social engineering techniques. This technique exploits not software vulnerabilities but human psychology through elements of shock and surprise. An e-mail claiming to come from your bank with a malicious attachment or requiring the installation of malware disguised, as fake codecs to view a shocking video are examples of how social engineering attacks may be carried out. The goal of such techniques is to trick a user to install the malware on their system by their own actions.

2.7.4. Third Party App Stores

The third party app stores for Android are known for their weak security precautions and they are responsible for the biggest rate of malware infection on the Android platform. This is considered as the most popular way for malware distribution for most malware authors.

2.7.5. Brute Forcing the User's Accounts

Any weak or default password of a user account for a remotely accessible service, be it FTP or SSH, is susceptible to remote brute-force attacks. Likewise, some malware are programmed to launch brute-force attacks autonomously on specific user accounts in order to propagate inside the network to be able to infect other devices.

2.7.6. Drive-By Downloads on the Android

Another common infection vector for malware is drive-by downloads. It usually works by exploiting a vulnerability in the target's web browser to download and execute malicious code on the target computer. Drive-by downloads follow a pull-based scheme (Provos, Mavrommatis, Rajab, & Monroe, 2008), namely, the user unknowingly initiates the connection to download malicious code in contrast to push based schemes in which attackers actively discover and exploit vulnerabilities present on services of a network or a computer.

2.8. Relevant Data Mining and Machine Learning Background

Data mining is the processing large amounts of data to uncover unseen patterns and the types of learning that can be used are classification, clustering, association or anomaly detection. The work in this thesis falls under the classification (malware or benign) type of data mining using supervised learning. In supervised learning, a set of labeled input is fed into the data mining model for learning, which in turn classifies it into a class, in our case, malicious or benign.

Data mining can be further categorized as predictive data mining and descriptive data mining. In predictive data mining, the aim is to deduct values from given datasets and in descriptive data mining, the aim is to deduct patterns that describe the dataset.

Many data mining models and algorithms can be used for the task of malware detection such as Artificial Neural Networks (ANNs), Support Vector Machines (SVMs), decision trees, association rule mining, k-Nearest Neighbour (k-NN) and others. Each of these models can perform better under certain circumstances, as well as performing worse under certain circumstances, namely, each one has its strengths and weaknesses. No one algorithm or model can meet all the expectations at once.

The group of machine learning algorithms and techniques that are essential for an adequate understanding of this Ph.D. work will be explained in detail in this section.

2.8.1. K-Nearest Neighbor (K-NN)

The method k-NN aka Instance-based Learner (Aha, Kibler, & Albert, 1991) has a high computation cost when given large datasets and was first described in the early 1950s. Based on learning by analogy, which is a comparison of a chosen instance with instances that are close to it in distance wise. When given an unknown tuple, a k-NN classifier searches the pattern space for the k training tuples that are closest to the unknown tuple. These k training tuples are the k “nearest neighbors” of the unknown tuple. “Closeness” is defined in terms of a distance metric, such as the Euclidean distance metric or the Manhattan distance metric.

The Euclidean distance between two points or tuples, for $x_1 = (x_{11}, x_{12}, \dots, x_{1n})$ and $x_2 = (x_{21}, x_{22}, \dots, x_{2n})$ is computed by:

$$\text{dist}(x_1, x_2) = \sqrt{\sum_{i=1}^n (x_{1i} - x_{2i})^2}$$

2.8.2. Support Vector Machines (SVMs)

SVMs (Cortes & Vapnik, 1995) are a very powerful method that has been utilized in a wide variety of applications. It can be used to classify both linear and nonlinear data. For nonlinear data, a nonlinear mapping technique is used to transform the original training data into a higher dimension. The linear optimal separating hyperplane is searched within this new dimension. The hyperplane classifier or linear separability is the basic concept in SVMs. To achieve linear separability, SVM applies two basic ideas: margin maximization and kernels, namely mapping input space to a higher dimension space. The expectation is that the hyperplane with the larger margin will be more accurate at classifying future data tuples than the hyperplane with the smaller margin.

Owing to their ability to model complex nonlinear decision boundaries they are highly accurate and much less prone to overfitting compared to other data mining models, SVMs can be used for prediction or classification in areas including but not limited to handwritten digit recognition, object recognition, and speaker identification.

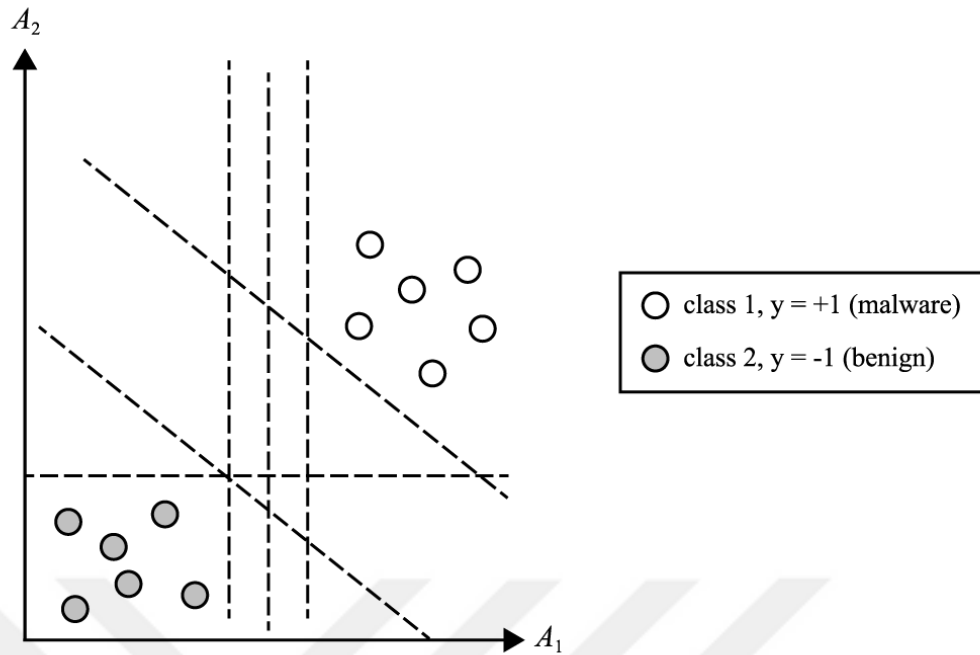


Figure 3.4. Support Vector Machine (SVM) (Han, Kamber, & Pei, 2006)

2.8.3. Decision Trees

Classification by decision tree involves constructing a decision tree, it is essentially a collection of decision nodes in which the internal nodes of a decision tree correspond to attributes and leaf nodes correspond to class labels (Kolter & Maloof, 2004). Attributes and values of an instance are used by a performance element to traverse the tree from the root to a leaf. A predictive model is used that maps observations about an item to conclusions about the item's target value.

In classification trees, the target variable can take a finite set of values. Leaves represent class labels and branches represent conjunctions of features that lead to those class labels. In regression trees, the target variable can take continuous values.

A tree is traversed from the root to a leaf to collect values of attributes for prediction of the leaf node. The attribute that best splits the training set into their corresponding classes is selected to build the decision tree.

A node, branches, and children are assigned for each attribute and its values, the examples are distributed to corresponding child nodes. This process is repeated recursively until a node contains examples of the same class and lastly, it stores the class label. The gain ratio is mostly used for attribute selection which is based on

information gain (Patel & Upadhyay, 2012). In order to prevent or lessen overfitting in decision trees, pruning of the tree is mostly employed, which removes sub-trees that are likely to perform poorly on test data and are likely to be redundant information.

2.8.3.1. C4.5

C4.5 (Quinlan, 1986, 1993) is a decision tree learner algorithm that precedes C5.0, which is a proprietary learner and closed source. As the source code for C4.5 is publicly available, it will be described instead of the C5.0. J48 is the decision tree algorithm that was used during the experiments in this Ph.D. work, and it is an open source Java based implementation of the C4.5 Decision Tree algorithm that is available in the Waikato Environment for Knowledge Analysis (WEKA) (Weka, 2018) Platform.

C4.5 constructs decision trees using a top-down recursive divide-and-conquer manner. A top-down approach is followed by most decision trees that starts with a training set and their corresponding class labels.

C4.5 recursively partitions the training set into smaller subsets during the tree construction process. The attributes are selected depending on the gain ratio criterion and a decision tree is built up in which a specific classification rule is represented by each path from the root to any selected leaf.

2.8.3.2. Random Forest

The Random Forest (Breiman, 2001; Tin Kam, 1995) decision tree classifier, classifies a sample based on the majority vote of classification generated by multiple classification trees. K trees are independently generated by this classifier, which makes it easily parallelizable.

A full binary tree of a given depth is constructed for each tree and the features used in each tree are selected in a random way, in other words, the same feature can be present more than one time. This derives an algorithm that votes the classification results of K random trees, in other words, it is an ensemble learner composed of K decision trees.

Each tree is grown using the following procedure (Breiman & Cutler, 2011):

- Let us denote the number of cases in the training set as “ N ”. From the original data, sample N cases at random with replacement are selected. This sample will compose the training set for growing the tree.

- Given there are “M” input variables, a number “ $m \ll M$ ” is specified in a way that at each node, m variables are selected randomly out of the M and the best split on these m is used for splitting the node. During the process of forest growing, the value of m is held constant.
- Each tree is grown as much as possible without pruning.

Given most of the selected features are relevant this algorithm produces the best results, as the feature subset selected for any given tree are small.

2.8.4. Naive Bayes

The Naive Bayes classifier is categorized as a statistical classifier, specializing in the prediction of class membership probabilities, such as predicting the probability of a given sample belonging to a particular class. Bayes’ theorem is named after Thomas Bayes, a scientist who did early work in probability and the decision theory.

Let X be a data tuple or a row in the database. In terms of the Bayes’ theorem, X is considered as an "evidence". Hence, it must be described by measurements made on a set of n attributes. Let H be a hypothesis that states the sample X belongs to a predefined class C . In classification problems, one needs to determine $P = (H|X)$, where P is the probability that the hypothesis H holds given the "evidence" or the observed sample X . Namely, the probability that the sample X belongs to the class C has to be found, given that we know the attribute description of X .

The working principles of the Naïve Bayes classifier will be given in formulas below (Han et al., 2006):

Let D stand for the training set and the associated class labels. An n -dimensional attribute vector represents each sample that are denoted by $X = (x_1, x_2, \dots, x_n)$ and depicts n measurements made on the sample from n attributes, which are given by A_1, A_2, \dots, A_n , respectively.

If there are m classes denoted by C_1, C_2, \dots, C_m and there is a sample X , then the classifier will predict that X belongs to the class with the highest posterior probability, conditioned on X . In other words, the naïve Bayesian classifier predicts that the sample X belongs to the class C_i if and only if,

$$P(C_i|X) > P(C_j|X), \text{ for } 1 \leq j \leq m, j \neq i.$$

Hence, $P(C_i|X)$ is maximized for the class C_i and it is called the maximum posteriori hypothesis. By Bayes' theorem,

$$P(C_i|X) = \frac{P(X|C_i)P(C_i)}{P(X)}$$

In this equation, only $P(X|C_i)P(C_i)$ need be maximized as $P(X)$ is constant for all classes. Given the class prior probabilities are unknown, it is assumed that the classes are equally likely, implying, $P(C_1) = P(C_2) = \dots = P(C_m)$, and thus $P(X|C_i)$ would need to be maximized. Otherwise, $P(X|C_i)P(C_i)$ would have to be maximized.

The fact that datasets commonly have many attributes, makes computing $P(X|C_i)$ exceedingly computationally expensive. The naive assumption of class conditional independence is made in order to reduce the computational cost of computing $P(X|C_i)$. This assumption implies that the given the class label of the tuple, the values of the attributes are conditionally independent from one another, in other words, there are no dependence relationships among the attributes. Thus,

$$\begin{aligned} P(X|C_i) &= \prod_{k=1}^n P(x_k|C_i) \\ &= P(X_1|C_i) \times P(X_2|C_i) \times \dots \times P(X_n|C_i) \end{aligned}$$

2.8.5. Neural Networks

A Neural Network is a fine-grained, parallel, distributed model of computation characterized by the following list of distinctive features:

- It contains a large number of simple, neuron-like processing elements, which are called nodes or, units.
- Typically, a large number of weighted (positive or negative real values), directed connections are present between pairs of nodes.
- Local processing carried out by each node computes a function, which is also dependent on the outputs of a number of other nodes inside the network.

Each node computes a simple function from their input values, which are the weighted outputs from the other nodes. Given there are n inputs to a node, the node's output, or activation, is definable by:

$$a_i^{(k)} = g \left(\sum_{j=0}^n \theta_{i,j}^{(k)} x_j \right)$$

In this equation $a_i^{(k)}$ is the activation of the node “i” in the layer “k”, and θ^k is a matrix of weights controlling function mapping from the layer “k” to the layer “k + 1”. Hence, each node computes a function g of a linear combination of its inputs.

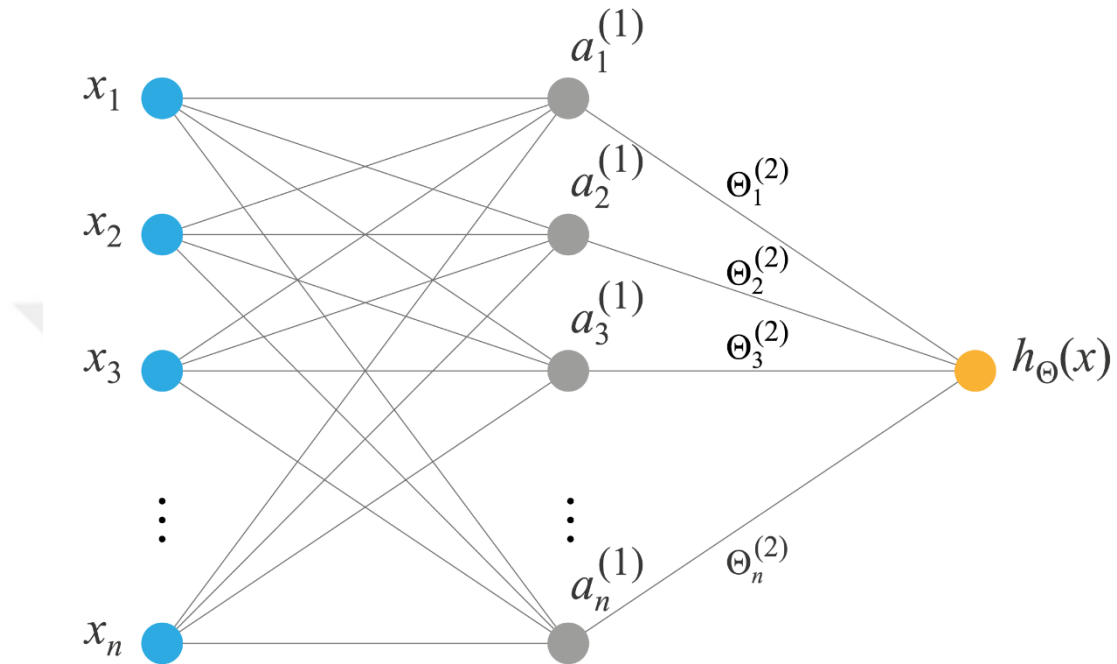


Figure 4.5. Artificial Neural Network that contains one hidden layer

2.8.5.1. Multilayer Perceptron (MLP)

A Multilayer Perceptron (MLP) is a feedforward artificial neural network that generates a set of outputs from a set of inputs. It is characterized by several layers of input nodes which are connected as a directed graph between the input and output layers. An MLP has at least three layers of nodes and each node is a neuron that utilizes a nonlinear activation function, except for the input nodes. MLP uses a supervised learning technique called backpropagation for training the network (Rumelhart, Hinton, & Williams, 1986; Van Der Malsburg, 1986). The fact that it has multiple layers and uses non-linear activation make it different from a linear perceptron. Thus, it can distinguish data that is not linearly separable (Cybenko, 1989). Since there are multiple layers of neurons, MLP is a deep learning technique.

MLP is widely used for solving problems that require supervised learning as well as in research areas such as computational neuroscience and parallel distributed

processing. Current application fields include but not limited to speech recognition, image recognition and machine translation.

2.8.5.2. Backpropagation Algorithm

In artificial neural networks, backpropagation, short for "backward propagation of errors," is a method that is used to calculate a gradient required in the calculation of the weights that will be used in the network (Goodfellow, Bengio, & Courville, 2016). The method, when input an artificial neural network and an error function is used to calculate the gradient of the error function with respect to the neural network's weights, The calculation of the gradient proceeds backwards through the network adds to the algorithm's name "backwards", in the procedure of calculating gradient proceeds backwards through the network the gradient of the final layer of weights are calculated first and the gradient of the first layer of weights is calculated last. During the computation of the gradient for the previous layer, partial computations of the gradient from one layer are reused. Efficient computation of the gradient at each layer is achieved through this backwards flow of the error information.

2.8.6. Radial Basis Function (RBF)

Radial Basis Functions (RBFs) focus on the rebuilding of unknown functions from known data (supervised learning). The functions are generally multivariate, and they may be solutions of partial differential equations that satisfy specific additional conditions. However, the rebuilding of multivariate functions from data can only be accomplished if the space furnishing the "trial" functions is not fixed in advance, but is data dependent (Mairhuber, 1956).

A radial basis function, RBF, $\phi(x)$ is a function defined in relation to the origin or a certain point c , that is $\phi(x) = f(\|x-c\|)$ where the norm is generally the Euclidean norm but can also be another type of measure.

The RBF learning model assumes that the hypothesis set $h(x)$ is influenced by the dataset $D = (x_n, y_n), n = 1 \dots N$ for a new observation x , in the following way:

$$h(x) = \sum_{n=1}^N w_n x \exp(-\gamma \|x - x_n\|^2)$$

This equation implies that each x_i of the dataset influences the observation in a Gaussian shape. Given a data point that is far away from the observation, its influence becomes residual due to the exponential decay of the tails of the Gaussian. This is an example of a localized function ($x \rightarrow \infty \Rightarrow \phi(x) \rightarrow 0$); however, other types of radial functions can also be used, such as:

Multi-quadratic: $\phi(x) = \sqrt{x^2 + \gamma^2}$

Thin plate spline: $\phi(x) = x^2 \ln(x)$

2.8.7. Ensemble Learning Methods

Ensemble classifiers are utilized due to the fact that the classification accuracy of a group of classifiers is typically higher compared to the accuracy of a single classifier (Dietterich, 2000; Martínez-Muñoz & Suárez, 2007; Polikar, 2006). A binary tree is constructed by the base classifier in which each node operates on one of the features from the dataset. Weighted voting is used to combine the predictions of the individual base classifier to decide the class of the test samples. When selecting the individual classifiers there are two important criteria that must be satisfied: accuracy must be above a certain threshold, in other words, it must be at least better than results from a random guess and a degree of diversification between the classifiers must be present that often leads to different errors on the new dataset (Saha, Pal, & Konar, 2016).

2.8.7.1. Boosting

Boosting is a method of enhancing the performance of individual classifiers by combining multiple machine learning algorithms. During the boosting process, a set of weighted models are produced via iterative learning of the model from a weighted dataset. Afterwards, the model is evaluated, and the dataset is reweighted on the basis of the model's performance.

Boosting emerged from the idea of combining simple rules to form an ensemble in order to enhance the performance of a single ensemble member. Let h_1, h_2, \dots, h_T denote a set of hypotheses, and consider the following composite ensemble hypothesis:

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

In this equation, each $h(x)$ is a classifier that produces values ± 1 , and α_t denotes the coefficient with which the member h_t is combined. During the boosting procedure, both α_t and the hypothesis h_t are to be learned.

The boosting procedure updates the weight of each sample at each iteration, to ensure that the misclassified ones get more weighting during the next iteration. In other words, boosting focuses on the samples which are harder to classify.

The AdaBoost algorithm which refers to adaptive boosting (Bishop, 2006; Han et al., 2006; Hastie, Tibshirani, & Friedman, 2001) is the most widely known and used boosting algorithm, it was also utilized in the experiments of this Ph.D. thesis.

The AdaBoost algorithm works as follows: the weights for all the classified samples are updated each time AdaBoost creates a new weak classifier, given the process is iterated T times. The weights of the samples that were misclassified are increased and the weights of the samples that were correctly classified are decreased. This process continues in a loop with each new set of weighted samples (Saha et al., 2016).

It is called adaptive due to the fact that it focuses on samples that were misclassified in the previous iterations.

2.8.7.2. Bagging

In Bagging or Bootstrap Aggregating, the classifiers are trained by different datasets which are obtained from bootstrapping the original dataset, in other words, a subset of the dataset is constructed via randomly selecting n samples with replacement from the original dataset. This resampling procedure, which is repeated T times, explores the diversity among the weak classifiers. At the final step, majority voting on the outputs of the weak learners determines the class of an unknown sample (Saha et al., 2016).

CHAPTER 3

LITERATURE REVIEW

In this chapter, a broad literature review on the topic of malware detection approaches on the Android using machine learning techniques is given.

3.1. Approaches for Malware Detection on the Android that use Machine Learning Methods

Approaches to detecting malware on Android can be broadly categorized into two categories. The first category aims to detect Android malware prior to its installation on the device (static analysis) and the second category aims to detect after installation via monitoring of run-time behavior (dynamic analysis). Some of the low-level features extracted in the static analysis include calls to external libraries, strings, and byte sequences, while more detailed features such as the list of API calls or the graphical representations of the control flow may be utilized as well (Cesare & Xiang, 2010). The main problem with the dynamic analysis is the significant overhead incurred compared to the overhead incurred when using a static analysis method.

One of the first studies on the Android permission model, its internal components and interactions were published by (Enck, Ongtang, & McDaniel, 2009). Following this study, a solution based on monitoring events occurring at the Linux kernel was proposed by Schmidt et al. (A. D. Schmidt et al., 2008). Features which were extracted from the Linux kernel, such as system calls and latest modified files, were used to create a baseline model of smartphone usage. Because Android was still in infancy at the time, no tests on real Android devices could be carried out.

Shortly after, the same authors (A.-D. Schmidt et al., 2009) implemented a method for static analysis of the APK files to extract function calls in the Android environment. In their study, they compared function call lists to malware APK files in order to classify the function calls.

In order to identify specific privacy violations, a methodology for static analysis of Android applications was implemented by SCanDroid (Fuchs, Chaudhuri, & Foster,

2009) that works by extracting the security specifications from the manifest file and checking whether all data flows are consistent with the predefined specifications.

Bläsing et al. (Bläsing, Batyuk, Schmidt, Camtepe, & Albayrak, 2010) proposed a dynamic analysis methodology that makes use of an Android application sandbox. After performing static analysis by disassembling Android APK files in order to detect malicious patterns, dynamic analysis is carried out that execute applications in a sandbox. During the execution of the file in question in the sandbox, relevant events such as opened files, accessed files, and battery consumption are monitored. An application simulating user interaction called “ADB Monkey” was made use of, which can be seen as the main drawback of this proposed method due to the fact that the simulated user interaction may not clearly represent the real world user interaction patterns.

Wu et al. proposed DroidMat (D. Wu, Mao, Wei, Lee, & Wu, 2012) that makes use of several features including permissions, deployment of components, intent messages and API calls. They collected their malware samples from the “Contagio mobile” (ContagioMobile, 2016) website and the benign samples were downloaded from the Google Play Store. They had 238 malware and 1500 benign application samples in their dataset. A number of different types of clustering algorithms were used to classify applications as benign or malicious and a detection rate of 97.87% was achieved.

Developers of DroidAPIMiner (Aafer, Du, & Yin, 2013) proposed an approach that relies on API level information within the bytecode representation of APK files. More specifically, they utilize requested permissions, critical API calls, package-level information and app parameters in their machine learning process for Android malware detection. They developed a tool called the DroidAPIMiner built on top of the Androguard libraries and used the RapidMiner (RapidMiner, 2017) application to build their machine learning classification models. Their dataset was collected from three different sources, for the malware samples they used McAfee repositories and the MalGenome project, for the benign samples they used the Google Play store. They analyzed around 20,000 apps, out of which 3987 of them are malware apps and about 16000 of them are apps collected from the Google Play Store. In their approach, they used the ID5 decision tree (Quinlan, 1986), C4.5 decision tree (Quinlan, 1993), k-NN (Aha et al., 1991) and SVM (V Vapnik, 1995) machine learning algorithms. They

achieved 99% accuracy and 2.2% false positive rates on their dataset using the K-NN classifier.

TaintDroid (Enck et al.), an information flow tracking tool for Android, enables dynamic taint tracking capability. In order to provide enhanced visibility and control over how an application uses private data, it simultaneously monitors multiple sources of sensitive data. Real-time analysis of applications is achieved by leveraging Android's virtualized execution environment, the Dalvik VM. Multiple sources of private data are labeled according to four levels of monitoring, namely variable level, method level, file level and message level. Similar to the working principle of an IDS, an alarm is raised in case the labeled data leaves the system through an untrusted third-party application.

A study on Android permissions focusing on how to conduct effective risk communication was published by (Peng et al., 2012). Specifically, they proposed applying probabilistic learning methods to calculate risk scores according to the requested permissions of an Android app. They conducted their experiments on real-world datasets and their results showed that probabilistic general models significantly outperform the existing approaches, and that the Naive Bayes models provide for a promising risk scoring approach.

Droidbox (Lantz, Desnos, & Yang) is built on top of TaintDroid with the added functionality of monitoring the Android API and reporting on the file system and network activity as well as the use of cryptographic operations and cell phone usage by using a patched Dalvik VM. A timeline view of the monitored activity is presented to the user which makes it useful for manually identifying malware by viewing its observed behavior.

ComDroid (Chin et al., 2011) performs static analysis of decompiled bytecode of Android applications to find Android Intents sent with weak permissions.

Crowdroid proposed by Burguera et al. (Burguera, Zurutuza, & Nadjm-Tehrani, 2011), is a dynamic analysis and machine learning based framework. It recognizes Trojan-like malware on Android smartphones, by analyzing the number of times a system call has been issued that requires user interaction by an application during the processing of a request. The monitored features are analyzed in the cloud and the collected

observations are classified using K-Means, reaching a claimed detection rate of 100% for the malware samples they had implemented themselves.

Yerima et al. (Yerima, Sezer, McWilliams, & Muttik, 2013) 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. These models were evaluated with real malware samples reaching the highest detection rate of 90.60%.

A classification approach with high accuracy was proposed by (Elish, Shu, Yao, Ryder, & Jiang, 2015) in which a data-flow feature on how user inputs trigger sensitive API invocations was extracted statically. Their approach was evaluated on 1433 malware and 2684 benign samples and achieved 2.1% false negative and 2.0% false positive rates. They based their classification method on enforcing carefully chosen benign properties in apps that are observable in benign samples, while not observable in malware samples. It was claimed by the authors that, this enforcement of benign properties through app classification will give the defenders an edge against combating malware.

A broad analysis of Android applications to detect on-device malware was performed by (Arp et al., 2014). The researchers built an app called “Drebin” and favored conducting initial extensive static analysis in order to avoid depletion of limited resources of a smartphone quickly. Their space of extracted features was decided to be as large as possible and is organized in sets of strings, that include permissions, API calls, and network addresses. All the extracted features are later embedded in a joint vector space with the aim of automatically identifying typical patterns indicative of maliciousness. Their experiments were conducted with 5,560 malware samples and 123,453 benign apps from a number of app markets. Malware detection rate of 94% and false-positive rate of 1% were achieved by utilizing the SVM machine learning classifier.

Malicious Android applications Detection through String analysis (MADS) (Sanz et al., 2013), extracts strings from Android APK files in order to construct machine learning classifiers to detect Android malware. 333 unique malware samples and 333 unique benign samples were used in their experiments. They employed text mining to represent each different Android application. 6 different algorithms which are, Naive

Bayes, Bayesian Network, SVM, k-NN, J48 and Random Forest were used in their experiments within the WEKA machine learning platform, with each algorithm configured to use the default configuration. They used the k-fold cross-validation technique in order to get their test results. They achieved the highest accuracy of 94.70% using the SVM Poly algorithm.

Authors of (S. Wu, Wang, Li, & Zhang, 2016) proposed an Android malware detection method that uses dataflow APIs as features within a machine learning approach. They conducted a thorough analysis to extract the dataflow-related API-level features and to improve the k-NN classification model. By further optimizing the dataflow-related API list, the efficiency of the sensitive data transmission analysis was increased considerably. Their dataset contained 1160 benign and 1050 malicious samples, and they obtained accuracy rates as high as 97.66%. The dataflow API was the only feature used during the whole data mining process.

Constituting a real-time malware detection system, TStructDroid (Shahzad, Akbar, Khan, & Farooq, 2013) uses various methods such as time-series feature logging, segmentation and frequency information extraction from the data. A novel information processing framework was proposed to extract the hidden patterns in the execution traces. Their detection system depends on majority voting on a segment of feature instances to make a decision about an executing application. J48 decision tree based classifier was preferred to be used by the framework. In standard cross-validation tests, they received accuracy in the range of 90 - 93.6% and false positive (FP) rate of 5.4 - 7.3%. However, in real-world scenarios, in which they used zero-day malware, they claim to reach promising results with 98% accuracy and less than 1% false alarm rate with a dataset containing only 110 benign and 110 malware Android APK files.

DroidScope (Yan & Yin, 2012) is a multilevel semantic analysis tool that performs dynamic profiling and information tracking to detect malicious behavior and privacy leaks on the Android platform. It performs instruction trace and API call monitoring and uses taint analysis to discover leakage of sensitive information. Has been tested on a dataset of just two Android malicious files.

A multilevel anomaly detection technique for detecting Android malware (MADAM) was proposed by Dini et al. (Dini, Martinelli, Saracino, & Sgandurra, 2012) that operates in the kernel and the user-space at the same time. The multilevel view of the

system makes it possible to deduce a rich feature space that enables detection of previously unknown malware. The framework operates in training, learning and operation phases. In the classification phase, the k-NN algorithm is used. By incorporating new feature vectors in training and learning sets at runtime, the machine learning model aims to improve detection rates. They reached an average accuracy of 93% and a false positive rate of 5% on 10 malicious and 50 benign apps.

Smartdroid, proposed by (Zheng et al., 2012), is a hybrid automatic malware detection framework that monitors user's interaction with the interface. Activity control graphs and function call graphs are built by a static path selector during static analysis. Function call graphs are updated for indirect and event-driven API calls. For dynamic analysis, Android framework code is modified and a restrictive component is added to limit the new activities that are created after interacting with the user interface.

An approach utilizing ensemble learning was proposed by Yerima et al. (Yerima, Sezer, & Muttik, 2015) for Android malware detection. Their aim was improving Android malware detection accuracy by combining static analysis with the efficiency and performance advantages of utilizing the ensemble of machine learning algorithms. 2925 malware and 3938 benign applications collected from the McAfee's internal repository were used in their experiments. The method proposed by them, utilized a large feature space to leverage the power of ensemble learning and achieved detection accuracy rates ranging between 97.3% and 99%.

Authors of Mlifdetect (Wang, Zhang, Su, & Li, 2017) claim that the traditional machine learning based malware detection methods have limited detection accuracy due to the utilization of single classification algorithms. They propose a novel approach that leverages parallel machine learning and information fusion techniques for better Android malware detection. They extract eight types of features via static analysis that include permissions, API calls, and deployment of components. They developed a parallel machine learning detection model in order to speed up the process of classification. Their dataset is composed of 8,385 apps, out of which 3,982 are malware and the rest are benign. Their malware samples were collected from the Drebin (Arp et al., 2014) and MalGenome (Zhou & Jiang, 2012) projects, and their benign samples were downloaded from the Google Play Store. The proposed approach can classify Android benign and malware apps with 99.7% accuracy.

Authors in (Alzaylaee, Yerima, & Sezer, 2017) utilize real phones in their research for automated feature extraction in an attempt to alleviate the problem of anti-emulator techniques' usage by Android malware in order to evade detection. Dynamic features were automatically extracted from Android phones and through several experiments a comparative analysis of emulator based vs. device based detection were performed. They used 222 malware samples obtained from the MalGenome project and 1222 benign samples obtained from Intel Security (McAfee Labs). A real phone and an emulator environment created by a Santoku Linux Virtual Box (SantokuLinux, 2018) based machine were used during the experiments. WEKA data mining platform was utilized in their experiments and from the 178 extracted features, 100 of them were selected after ranking them by using the InfoGain (information gain) feature ranking algorithm in WEKA. They used the linear SVM, Naive Bayes, Simple Logistic, MLP, PART (Partial Decision Trees), Random Forest, and J48 Decision Tree algorithms in their experiments. Several different experiments were conducted, in which emulator to real device machine learning detection was compared, and comparison of the results to several previous works was made (Amos, Turner, & White, 2013; W.-C. Wu & Hung, 2014). Their results from the phone-based analysis achieved up to 0.926 F-measure along with 93.1% true positive rate (TPR), but also a poor performance of 14.85% false positive rate (FPR) using the Random Forest classifier and phone-based experiments were found to achieve better rates overall compared to emulator based experiments.

Authors in (Yerima, Sezer, & Muttik, 2014) propose a parallel machine learning based classification approach for early detection of Android malware. From the parallel combination of heterogeneous classifiers, a composite classification model was developed. Static features are extracted to be used in the learning phase of the model development. Three features are extracted which are: API related features, app permissions and the standard OS as well as Android framework commands. The machine learning classifiers used during the experiments are Decision Tree, SL, Naive Bayes, PART, and RIDOR (Ripple-Down Rule). A composite model of the aforementioned heterogeneous classifiers was utilized in various parallel combination machine learning schemes. 125 permissions along with 54 API calls and commands related features were extracted. 6,863 applications from the McAfee's internal repository were collected with 2,925 malicious apps and 3,938 benign apps. Among

all the classifier results, the PART classifier achieved the highest detection ratio at 95.8%.

Another approach that uses permissions and API calls as features within a machine learning context was proposed by Peiravian and Zhu (Peiravian & Zhu, 2013). Several classifiers such as SVMs, Decision tree and Bagging were utilized in the experiments. They constructed three benchmark datasets, each with the same number of samples, but with different numbers of features. They gathered 2510 samples, out of which 1260 of them are malware and the remaining 1250 are benign samples. They eliminated malware with identical feature values from the dataset and ended up with 610 malware samples from 49 different malware families of the MalGenome dataset. The benign applications were downloaded from the Google Play Store's 25 different app categories. Their approach achieved the highest detection ratios using the SVM classifier with an accuracy percentage of 96.88% and a recall percentage of 94.8%.

In a more recent study published by Alatwi et al. (Alatwi, Oh, Fokoue, & Stackpole, 2016), the authors argue that category based machine learning classifiers enable considerable higher average detection rates in comparison to non-category based classifiers. For each category, a malware detection classifier is trained separately. They associate between the apps' features and the features that are needed to deliver its category's functionality, in other words, an association is formed between the features that the app requests and a common set of features for its category. APK files were reverse engineered into their native JAVA source codes and the permissions, broadcast receivers and API calls features were extracted. They constructed three datasets, 70% of each dataset was used for training and the 30% for testing. The datasets were also randomly shuffled in each round of the 50 iterations that were used to average the performance of the classifiers. Category based and non-category based SVM classifiers were used during the experiments and their results were compared. It was shown that, category based classifier achieves an average accuracy ratio of 98.72%, compared to the non-category based classifier's average accuracy ratio of 94.58%.

In the study published by (Mariconti et al., 2017), the authors present MAMADROID, which is an Android malware detection system that relies on app behavior. MAMADROID builds a behavioral model from the sequence of abstracted API calls as a Markov chain, and uses it to extract features and perform classification. Its performance was tested on a dataset of 8.5K benign and 35.5K malicious apps

collected over a period of six years. The fact that MAMADROID relies on the sequence of abstracted API calls performed by an app rather than their use or frequency is its novelty. They claim that using the sequence of abstracted calls as features enables modelling of behavior in a more complex way. By building a statistical model to represent the transitions between the API calls performed by an app, they model these transitions as Markov chains, and use them to extract features and perform classification. In the classification phase, four different classifiers which are: Random Forest, 1-NN, 3-NN and SVM were used. Achieving the highest F-measure of 99%, it was shown to maintain a good detection performance ratio over three years: by achieving an F-measure value of 87% one year after the model has been trained, and an F-measure value of 73% after two years.

A recent study published by (Onwuzurike et al., 2018) builds on MAMADROID to analyze the performance of static, dynamic and hybrid analysis methods, using the same modeling approach as (Mariconti et al., 2017). In order to port MAMADROID to dynamic analysis, CHIMP (Almeida et al., 2018) was modified, which is a platform recently proposed to crowdsource human inputs for app testing. API calls' sequences were extracted from the traces produced while executing the app on a CHIMP virtual device. The developed system was named AUNTIEDROID and it is instantiated by using both pseudorandom input generators such as Monkey (Monkey, 2018) and user-generated inputs. The main distinction of AUNTIEDROID from MAMADROID is that, the former is based on behavioral models extracted through dynamic analysis, not static analysis as in the case of the latter. For their dataset, the authors re-used the set of 2,568 benign apps labeled as "new benign" in (Mariconti et al., 2017), and 2,692 Android malware from VirusShare (VirusShare, 2018). The authors note that combining static and dynamic analysis yields the best results by achieving an F-measure of 0.92 and static analysis is claimed to be at least as effective as dynamic analysis.

An improved Naive Bayes classifier, called the Normalized Bernoulli Naïve Bayes was proposed by (Sayfullina et al., 2015) to detect Android malware. They extract most of their 13 groups of features from three main Android files, which are AndroidManifest.xml, classes.dex and resources.arsc. Over 120,000 files from the year of 2014 with trustworthy malware or benign labels were obtained from F-Secure. The Bernoulli Naive Bayes model was chosen due to the relevance of using binary

features in two class classification problems. The Bernoulli Naive Bayes model was modified by tuning the Laplace smoothing parameter and normalizing the sum of log-factors by the length of the file. The Normalized Bernoulli Naive Bayes that they presented outperformed Bernoulli Naive Bayes in the accuracy rates and enabled better overall class separation. On 10,000 training and 10,000 test samples they achieved TPR (True Positive Rate) of 82.10% and FPR (False Positive Rate) of just 0.1%.

In a study published by (Dhaya & Poongodi, 2014), the authors apply the n-gram concept to find vulnerabilities in the apps by considering the source code as signatures. They decompile the APK files and extract the API call, Call flow and Device memory features. The main tasks carried out by the system are using static analysis to find bugs in the apps before publication at an app store and comparison of the n-gram signatures of samples to signatures which were already stored. The authors claim their method is effective in detecting malware in Android.

A hybrid approach combining static and dynamic analysis was proposed by (Lindorfer, Neuschwandtner, & Platzer, 2015) called MARVIN. It leverages machine learning techniques to assess the risk associated with Android apps by producing a malice score for each. Both static and dynamic analysis are performed in an off-device manner and a comprehensive feature set is utilized to represent the properties and the behavioral aspects of each app. Their feature selection is based on using the F-score (Fisher score) (Chen & Lin, 2006) metric. Their dataset was comprised of 135,000 Android apps that included 15,000 malicious samples. MARVIN achieved a 98.24% detection rate and an FPR of less than 0.04%.

In another study that employs online machine learning, the authors (Abdurrahman Pektaş, Çavdar, & Acarman, 2016) utilized the Cuckoo Sandbox (CuckooSandbox, 2018) file analysis reports to extract behavioral features from Android malware. They obtained their malware samples from the VirusShare (VirusShare, 2018) malware sharing platform and evaluated their online machine learning algorithm on 2000 samples belonging to 18 families. They employed the 10-fold cross-validation approach and achieved the highest classification accuracy of 89%.

In a study published by (Milosevic, Dehghantanha, & Choo, 2017), the authors present two machine learning aided approaches for static analysis of Android malware: one based on utilizing the requested permissions of an app as features, the other based on

source code analysis using a bag of words representation model. The MODroid (Damshenas, Dehghantanha, Choo, & Mahmud, 2015) dataset that contains only 200 malicious and 200 benign samples, was utilized for the training and testing of the machine learning models. Their permission-based analysis is computationally lightweight and using the modified Weka 3.6.6 library for Android, was integrated into the permission scanner in OWASP Seraphimandroid application (Milosevic, 2018). In their source code based approach, they first decompile the APK files, and afterwards the decompiled code is utilized within a text mining classification approach that uses the bag of words model. They tokenize the source codes they obtain from the APK files into unigrams that are used as a bag of words. Several machine learning algorithms were utilized for the experiments including J48, Naive Bayes, SVM with SMO, Random Forests, JRIP, Logistic Regression and AdaBoostM1 with SVM base as well as ensemble learning with combinations of three and five algorithms that use majority voting. Their source code based classification achieved an F-score of 95.1%, while the permissions based approach obtained an F-measure of 89%.

In Dendroid (Suarez-Tangil, Tapiador, Peris-Lopez, & Blasco, 2014), the authors use text mining approaches to automatically analyze smartphone malware samples and families based on the code structures present in their software components. The code structures represent the Control Flow Graph (CFG) of each method available in the app classes. They constructed their dataset by selecting a subsample from the samples available in the MalGenome Project, containing 1231 malware samples grouped into 33 families. They extract all different code structures from their dataset and a vector space model is used to associate a unique feature vector with each malware sample and family. By reformulating the modelling process followed in text mining applications, they achieved measurement of similarity between malware samples, which in turn was used to automatically classify them into families. They also investigate the application of hierarchical clustering over the feature vectors obtained for each malware family. The resulting dendograms resemble the so-called phylogenetic trees for biological species, enabling them to reason about evolutionary relationships among the malware families. Their experimental results came out with promising high detection accuracies.

In the approach proposed by Coronado-De-Alba et al. (Coronado-De-Alba, Rodríguez-Mota, & Ambrosio, 2016), the authors present a meta-ensemble classifier

for Android malware detection and employ static analysis of the samples. They gathered 1531 malware and 1531 benign samples, while the malware samples were collected from the Drebin (Arp et al., 2014) project, the benign samples were collected from the Google Play Store. Their original dataset contained 3062 samples with 660 features which were selected from permissions, intents, hardware and software. Chi-squared and Relief feature selection algorithms were used on their balanced and unbalanced datasets. WEKA was used for running the classifiers by using the default parameter settings for each classifier. 10-fold cross-validation was used for getting the results. On their datasets, they applied all the possible classifiers in accordance with the characteristics of the datasets. They obtained the highest detection rate of 97.56% using the meta-ensemble RandomForest with 200 trees based in RandomCommittee.

The largest measurement of Android malware behavior reported in the literature was carried out by (Suarez-Tangil & Stringhini, 2018), in which they analyzed over 1.28 million malicious samples belonging to 1.2K families collected from the years 2010 to 2017. As human analysis efforts fell short due to the scale of the study, specific tools were developed for the automated analysis of the dataset. Focusing specifically on repackaging malware, they aim to understand how the behavior of Android malware evolved from 2010 to 2017. In this type of threats, benign apps are piggybacked with a malicious payload, called a rider. To address the problem of separating the malicious part from the benign part of the repackaging malware, they employed differential analysis. All their samples, both benign and malware were collected from AndroZoo. In order to establish a systematic way of extracting rider methods, they mine methods that are common to members of the same family. For identifying which methods are common among samples of the same family, they build on top of Dendroid, the work that was mentioned previously. They extended Dendroid to recursively extract fragments from all available resources within the app of type DEX (Dalvik Executable) or APK. From the 1.2 million apps in their dataset they observed 155.7 million number of methods, out of which about 1.3 million were rider methods. Their research notes Android malware evolved rapidly from 2010 to 2017 and evidences the importance of developing anti-malware systems that are resilient to such changes.

Another recent study utilized deep learning to construct an automatic framework for Android malware detection. In their paper, the authors propose MalDozer (Karbab, Debbabi, Derhab, & Mouheb, 2018), which uses neural networks on API method calls

for Android app classification. By automatically extracting the raw sequence of an app's API method calls, MalDozer learns the malicious and the benign patterns from the actual samples in order to detect Android malware. MalDozer was evaluated on the following datasets: The MalGenome Dataset, Drebin Dataset, their MalDozer dataset that contain 20,000 samples and a merged dataset of 33,000 malware samples along with 38,000 benign apps that were downloaded from the Google Play Store. In the experiments conducted on the aforementioned datasets, MalDozer achieved an F1-score between 96% and 99%. Additionally, on the same datasets, MalDozer correctly classified each malware to their corresponding families with an F1-score between 96% and 98%.

Table 3.1. Results Comparison of the Approaches in the Literature Review

Name	Year	Features	# of Malware	# of Benign	Accuracy
DroidMat	2012	Permissions, deployment of components, intent messages and API calls	238	1500	97.87%
DroidAPIMiner	2013	Requested permissions, critical API calls, package-level information and app parameters	3987	16000	99%
Elish et al.	2015	A data-flow feature	1433	2684	2.1% (FN) 2.0% (FP)
MADS	2013	Strings	333	333	94.70%
Wu et al.	2016	Dataflow APIs	1160	1050	97.66%
TStructDroid	2013	-	110	110	90 - 93.6%
MADAM	2012	-	10	50	93%
Yerima et al.	2015	Large feature space	2925	3938	97.3% - 99%
Mlifdetect	2017	Eight types of features extracted via static analysis	3982	4403	99.7%

Yerima et al.	2014	API related features, app permissions and Android framework commands	2925	3938	95.8%
Alzaylaee et al.	2017	Dynamic features	222	1222	93.1% (TPR)
Peiravian et al.	2013	Permissions and API calls	1260	1250	96.88%
Alatwi et al.	2016	Permissions, broadcast receivers and API calls	-	-	98.72%
MARVIN	2015	Comprehensive feature set	15,000	120,000	98.24%
Coronado-De-Alba et al.	2016	Permissions, intents, hardware and software	1531	1531	97.56%
MalDozer	2018	Raw sequence of an app's API method calls	33,000	38,000	96% - 99% (F1-score)
Pektas et al.	2018	Behavioral features	2000	-	89%.
Milosevic et al.	2017	Source code analysis	200	200	95.1% (F-measure)
Sayfullina et al.	2015	13 groups of features	10,000	10,000	82.10% (TPR)
MAMADROID	2017	Sequence of abstracted API calls	35,500	8,500	99% (F-measure)
AUNTIEDROID	2018	Behavioral models extracted through dynamic analysis	2568	2692	92% (F-measure)

3.2. Approaches that Utilize Text Mining for Malware Detection

In this section existing approaches that utilize text mining in their malware detection research will be mentioned by including the Windows OS in addition to the Android. In (Firdausi, Lim, Erwin, & Nugroho, 2010) the authors used Term Frequency (TF) weight in data pre-processing and built their dataset upon it.

In (Moskovitch et al., 2008) authors heavily used the Term Frequency-Inverse Document Frequency (TF-IDF) approach, parsed Windows binaries and used the n-gram method by taking each n-gram term as equal to a word in the textual domain. Representing a text file as a bag of words, they used the vector space model (Salton, Wong, & Yang, 1975). They have looked for terms with the highest Document Frequency (DF) values.

Authors of (Sanz et al., 2013) used a text mining approach to represent each Android APK file on the vector space model. They constructed a textual representation of an executable that is formed by strings. They employed the TF-IDF weighing schema to assign weights to each APK file.

In another study published by Lin et al. (Lin, Wang, Xiao, & Eckert, 2015) a generic and efficient algorithm to classify malware was proposed that combines the selection and the extraction of features. They extract n-gram feature space data from behavior logs; build an SVM classifier for malware classification; select a subset of features; transform high-dimensional feature vectors into low-dimensional feature vectors; and finally select the models. Their experiments were conducted on a real-world dataset with 4,288 samples from nine families. In feature selection analysis using the TF-IDF algorithm they calculate the effective feature set, specifically they based the feature weightings on the TF-IDF value to determine which feature set yields the optimal accuracy and learning times. Different from the approach proposed in this thesis, they applied the n-gram model in their data. Another considerable difference is the fact that they consider samples with the largest IDF values, such as the ones that appear only in one sample, as having substantial effect, which is in direct contrast to the feature selection approach proposed by this thesis.

In the study published by (A. Pektaş & Acarman, 2018), the TF and IDF metrics are used to assign weights to features extracted as a result of n-gram search over API call sequences. They compute the weight of each feature as the product of these two metrics. Similar to the above studies, they carry out feature selection starting from the features with the lowest IDF values as they represent the features that rarely occur in the document corpus.

Different from the above approaches, the approach proposed in this thesis does not take a classical approach to text mining. Relevant permission, API call, and strings

feature selections were made according to their Delta-IDF values. The Delta-IDF introduced in this Ph.D thesis enables selection of attributes with the lowest possible IDF values in the malware while at the same time having the highest possible IDF values in the benign samples. This indicates the novelty of the feature selection approach proposed in this Ph.D. research.



CHAPTER 4

PROPOSED METHODOLOGY

To the best of my knowledge, no other past study in the literature makes use of the combination of permissions, API calls and strings as features in the machine learning process for Android malware detection. Each of the three features collected from APK files using reverse engineering present opportunities for classifying an Android executable as malicious or benign. This chapter gives the details of the proposed methodology for malware detection on the Android platform.

4.1. Data Collection and Pre-processing

Two distinct academically recognized and available datasets were utilized during this Ph.D. study. The first one is prepared from the Android Malware Genome (MalGenome) (Zhou & Jiang, 2012) project, and provides 1,260 Android malware samples belonging to 49 different families. The second one is prepared with the permission from the authors in (Allix, Bissyandé, Klein, & Traon, 2016), by downloading malware samples from the vast collection of APK repository in the AndroZoo dataset.

Malware samples from the MalGenome project were downloaded as a ZIP file, from which randomly selected 250 malware files were used for constructing the first dataset.

A subset of the CSV file provided by AndroZoo was transferred into a MySQL database in order to be able to run SQL queries on it. By utilizing the SQL queries, I was able to obtain 1300 Malware samples with a VirusTotal detection number of at least 20. VirusTotal is an online malware scanning service that works by aggregating the scanning capabilities of many commercial antivirus products. The benign samples totaling 300 were manually downloaded from the Google Play Store. In other words, the AndroZoo malware samples that were detected as malicious by at least 20 different antivirus products were included in the second dataset.

Two sets of malware samples collected from the MalGenome and AndroZoo datasets along with the 300 benign samples downloaded from the Google Play Store were added to another PostgreSQL database by using a Python script I developed. All the tuples inside the database contained the following columns: app_name, app_path, app_perms, app_apis, and app_strings. Once APK samples are added to the database, each one is reverse engineered to extract the list of requested permissions, API calls, and strings.

4.2. Feature Extraction

One of the core parts of this research has been based upon extensive feature extraction work. In trying to get the best results from a predictive model, a key requirement is making the best of the extracted data or data at hand by selecting the best set of features to be used in the data mining process.

Feature extraction can be explained as the process of transforming raw data into a set of features that adequately input the underlying problem to the predictive models, with the aim of improving model accuracy for never before processed data.

The file features from the APK samples were extracted through the use of reverse engineering. Libraries from the open source Androguard tool (Androguard, 2017) written in Python for disassembling and decompiling Android applications were utilized for extraction of the permissions, API calls and the strings features.

The extracted features were converted into bit vectors with 1s representing the presence and 0s representing the absence of a feature for a given APK sample respectively. These were stored in files in CSV format, which were later used in the machine learning process.

4.2.1 Permissions

Android for a long time depended on presenting users with a list of all of the requested permissions by the app for acceptance before app installation. With Android's version 6.0 (API level 23) update, this method of asking a user for accepting the list of all the permissions the app requires was revamped into, asking for permission acceptance at the time it was required by the app during runtime.

All the permissions an application requires are declared in the `AndroidManifest.xml` file. The format for declaring a permission is:

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

Each Android application contains several strings for permission usage such as “`android.permission.CAMERA`” or “`android.permission.READ_LOGS`”. As can be seen from the examples, a permission name typically includes the package name as a prefix. The default format of the Manifest file is binary XML format, in other words, it needs to be parsed programmatically in order to extract the permission strings.

Permission names are deemed suitable for use in machine learning for malware detection due to a number of factors. First, the requirement that an app must be granted specific permissions to be able to use certain phone features through API calls. For instance, an app without the `CAMERA` permission will not be able to use the phone’s camera to take photos or record videos. Second, certain combinations of permissions may be associated with malicious behaviour as such a malware that utilizes SMS to spread to the list of contacts in your phone’s address book, while at the same time carrying out cyber espionage operations must possess a specific combination of permissions including `READ_CONTACTS`, `SEND_SMS`, `INTERNET`, `CAMERA`, and `RECORD_AUDIO`.

4.2.2. API Calls

Permissions enable implementation of API calls by an app, while the list of officially defined permissions is limited to 134 (Felt et al., 2012), the total number of API calls is much higher, precisely, there are 8,648 (Chin et al., 2011) distinct API Calls.

Similar to permissions, API calls can be effectively utilized as a feature in the data mining process as an indicator of maliciousness for detection of malware at a lower level compared to permissions. Androguard was utilized in the extraction of API Calls. As it is not possible to access the original source code of an Android app from its APK file (executable) an intermediate version of the code from the file must be produced which in the case of this thesis is the Dalvik Bytecode. By using the code that utilizes the Androguard libraries, Dalvik Bytecode representations of each APK sample were produced by reverse engineering each.

This bytecode representation of each APK sample that I extracted, was utilized for counting the number of each API calls' appearance in each sample's source code and for calculating the TF and the IDF values corresponding to each distinct API call within each APK sample.

4.2.3. Strings

Strings are a general feature of an executable that can be utilized both in the Windows and Android platforms for conducting static malware analysis. Extracting strings from an executable can be easy as running a one-line command in the Windows command line, just like it is the case in the Android. For mass string extraction from the APK samples, the Androguard libraries were utilized. In contrast to its relative simplicity of obtaining from an executable, it carries valuable information that can be linked to maliciousness, such as keywords indicating potential malware or IP addresses that can indicate the call back IP address for a malicious executable that aims to connect to the Command and Control (C&C) servers for data exfiltration.

4.3. A Novel Document Frequency-Based Approach to Feature Selection

After extraction of the three distinct features from the set of collected APK samples, the requirement to make attribute selection amongst all the extracted set of attributes from each feature emerged. In this section, a novel TF-IDF approach to feature selection phase in machine learning for malware detection on the Android is presented.

The extracted data belonging to the three categories of features were input to the TF and IDF calculating Python code for the calculation of each attributes' corresponding values. The output values were used as the main criteria of selection in the feature selection phase of the data mining process.

4.3.1. Term Frequency

TF (Term Frequency) is generally used within a text mining, document classification, or information retrieval context. TF is calculated by counting the number of times a word appears in a document, the document in the case of this thesis is the APK file. The term frequency is the count of a feature's (such as a permission, API call or string) appearance in an APK file, calculated by:

$$TF = \frac{f_i}{\sum_{j=1}^n f_j}$$

In the above equation, f_i is the number of times the string i appears in the APK file and $\sum_{j=1}^n f_j$ is the sum of frequencies of each of the string in the APK file.

4.3.2. Inverse Document Frequency

The IDF is the measurement of a specific term's occurrence within a document. In this Ph.D. work, the presence of specific strings within APK files were considered as the occurrence of specific features. It can be calculated as follows:

$$IDF = \log \frac{T}{C}$$

In the above equation, T is the total number of APK files and C is the number of APK files containing the given string.

TF and IDF values are used together as a weighted specific metric named as TF-IDF in text mining and document classification contexts and it is calculated as follows:

$$TF-IDF = TF \times IDF$$

The IDF and TF-IDF values start becoming closer to zero, given an attribute or feature appears in greater numbers of files. The weighted TF-IDF is normalized in some alternative approaches by using the below equation:

$$TF - IDF = \left(0.5 + 0.5 \frac{f_{t,q}}{\max_t f_{t,q}} \right) \cdot \log_{10} \frac{N}{n_t}$$

As a result of the above normalization, solutions differ typically by a few percent and such error ranges are prevalent in the literature. The error may be due to simulation rather than the analytical models when a study compares its analytical results to the simulation and reports errors of only a few percent.

4.3.3. Description of the Document Frequency-Based Approach

Contrary to the classical TF-IDF approaches of feature selection that was utilized by the following authors on the Windows platform in (Moskovitch et al., 2008) the approach in this thesis proposes selecting attributes with the lowest possible IDF values in the malware samples and at the same time the highest possible IDF values in the benign samples. This thesis argues that attributes with the lowest IDF values

represent the most evenly distributed attributes among the complete sample sets' extracted collection of attributes, while attributes with the highest IDF values are the ones that appear the most sparsely. This even distribution in turn indicates a higher frequency of occurrence for the corresponding attribute amongst the total set of attributes. Hence, selecting attributes with this methodology enables better learning of malware features, as the selected attributes are the ones that appear most frequently and evenly distributed in the malware dataset while appearing the least possible frequently and the most sparsely in the benign dataset.

In case of duplicating IDF values amongst the attributes, TF values of the corresponding attributes can be compared to make the final decision about which attribute to include in the data mining process. The one with a higher TF value would be selected, since a higher TF indicates a higher frequency of occurrence for an attribute among the executables (APK files).

In order to achieve selection of attributes that appear most frequently in the malware dataset while appearing the least possible frequently in the benign dataset, the results of the calculation of Delta-IDF is used and the ones with the highest values are taken to conduct learning for categorizing according to maliciousness. Delta-IDF value can be simply calculated as:

$$\text{Delta-IDF} = (\text{IDF}_{\text{Benign}}) - (\text{IDF}_{\text{Malware}})$$

In deciding the number of features to be selected from the whole extracted feature set, some basic statistical measures such as the percentile have been used. In this approach, the features that are above the 90th percentile, or in other words within the top 10% segment have been chosen after sorting them according to their *Delta-IDF* values.

If the dataset size was much larger, after z-score standardization the normalized *Delta-IDF* values would be observed and the attributes that have z-scores above +2 sigma or +1 sigma would have been chosen. The Delta-IDF value can also be simplified as below:

Let N be the total number of samples.

n_b : total number of records that an attribute or string occur in the benign samples.

n_m : total number of records that an attribute or string occur in the malicious samples.

$$\text{IDF}_{\text{benign}} = \log_{10} \frac{N}{n_b}$$

$$\text{IDF}_{\text{malware}} = \log_{10} \frac{N}{n_m}$$

$$\text{Delta-IDF} = \log_{10} \frac{N}{n_b} - \log_{10} \frac{N}{n_m} = \log_{10} \frac{n_m}{n_b}$$

As a result, Delta-IDF values can be effectively found out by calculating $\log_{10} \frac{n_m}{n_b}$.

In order to derive smaller sample datasets, the methodology of statistical random sampling without replacement was employed.

For the parameters given below:

N = population size

μ = distribution ratio

E = margin of error

σ = standard deviation of the population

$Z_{\alpha/2}$ = upper critical value for the standard normal distribution within the chosen confidence level, the sample size n can be calculated as follows:

$$n = \frac{\mu(1 - \mu)}{\left(\frac{E}{Z_{\alpha/2}}\right)^2}$$

Since the population is limited, by using the correction factor the formula given above can be evaluated as follows:

$$n = \frac{N\mu(1 - \mu)}{\mu(1 - \mu) + (N - 1) \left(\frac{E}{Z_{\alpha/2}}\right)^2}$$

For the first subsample dataset, the new sample size was chosen depending on the following criteria:

Confidence interval (accepted margin of error) = 2%

Confidence level = 98%

The sample size for the second subsample was chosen similarly depending on the following criteria:

Confidence interval (accepted margin of error) = 1%

Confidence level = 95%

Using these statistical sampling parameters and the equations given above, the sample sizes for the first and second subsample datasets were determined to be 3315 and 9100, respectively, which satisfies the minimum sample size limits regarding the original dataset size and the chosen sampling parameters. To sum up, these two subsample datasets were derived with these given sizes by random selection and sampling from the original dataset.

By implementing the methodology explained above for feature selection, 18 permissions, 8 API calls and 11 strings were selected for the MalGenome dataset, and 13 permissions, 9 API calls and 4 strings were selected for the AndroZoo dataset. The selected attributes are given in tables Table 4.1. and Table 4.2.

Table 4.1. Selected attributes from the MalGenome dataset

Permissions	Permissions (cont.)	API calls	Strings
BROADCAST_STICKY	SEND_SMS	getSubscriberId	Parse
CALL_PHONE	SYSTEM_ALERT_WINDOW	sendTextMessage	Add
CAMERA	WRITE_APN_SETTINGS	createFromPdu	iterator
DISABLE_KEYGUARD	WRITE_CONTACTS	getSimSerialNumber	schedule
GET_ACCOUNTS	WRITE_EXTERNAL_STORAGE	getExtraInfo	setId
READ_CALENDAR	WRITE_HISTORY_BOOKMARKS	getPaint	Digest
READ_LOGS	WRITE_SMS	seekTo	Callback
READ_PHONE_STATE		TruncateAt	entrySet
READ_SMS			findPointerIndex
RECEIVE_SMS			getEdgeFlags
RESTART_PACKAGES			getFinalX

Table 4.2. Selected attributes from the AndroZoo dataset

Permissions	Permissions (cont.)	API calls	Strings
REORDER_TASKS	SEND_SMS	getAttributeUnsignedIntValue	remoteexception
CLEAR_APP_CACHE	WRITE_CONTACTS	setKeywords	Credentials
MANAGE_DOCUMENTS	RESTART_PACKAGES	setAppName	remoteinput
WRITE_HISTORY_BOOKMARKS	BROADCAST_STICKY	getCurrencyCode	doubleclick
READ_HISTORY_BOOKMARKS		isSurrogatePair	
KILL_BACKGROUND_PROCESSES		allowCoreThreadTimeOut	
ACCESS MOCK_LOCATION		requestAd	
ACCESS_GPS		displayAd	
ACCESS_LOCATION_EXTRA_COMMANDS		FilenameFilter	

According to the differing datasets at hand for other application scenarios, the actual selected attributes will be different, however feature selection with this novel TF-IDF approach will enable achieving the best possible learning of the malicious features

within a dataset of Android samples, resulting in high detection accuracy and low false positive rates.

4.4. The Architecture of the Android Malware Detection

Methodology

The high-level view of my Android malware detection via machine learning and data mining methodology is depicted in Figure 4.1.

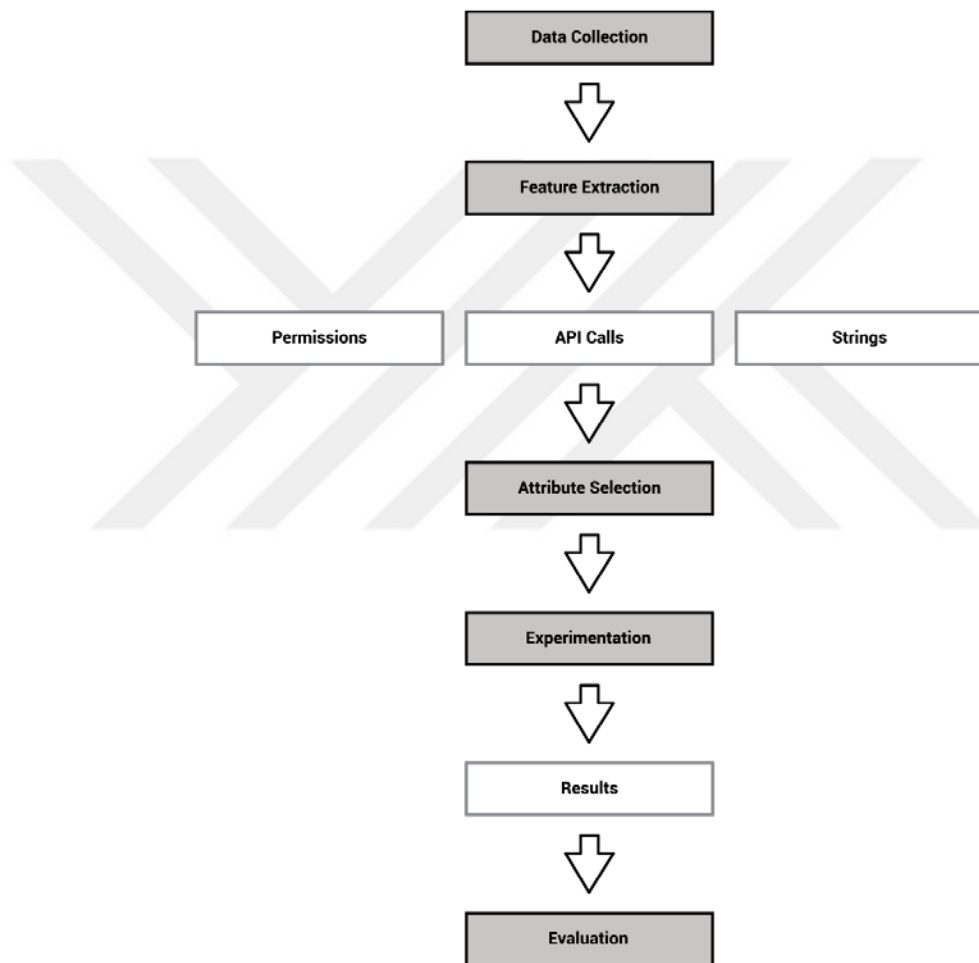


Figure 4.1. Malware detection process used in this thesis

After extraction of the three different feature sets from each APK sample, a thorough analysis was carried out to include only the best discriminative features inside the data mining process for malware detection on the Android. Feature vectors are generated from each training instance using the selected feature set. The feature vectors are used

to train the selected classifiers. When a new APK sample must be tested, first the features selected during training are extracted from the executable in order to be able to generate a feature vector. This feature vector is classified using the selected classifier to predict whether the executable is benign or malicious.

After extraction of the three different feature sets which are permissions, API calls and strings, the Android malware detection methodology proposed in this thesis merge them into one feature set, called the combined feature set. These features are used for the training of the machine learning algorithms (classifiers) in WEKA that were later run for detection of malicious Android apps.

Three different sets of features are selected to maximize the differentiating factors in the combined feature set between malicious and benign instances. This implies that permissions or API calls alone may not lead to high degrees of maliciousness when viewed in singularity, however a combination of a specific permissions, a group of API calls and specific strings contained in one APK sample, may indicate a very high probability of maliciousness, leading to a detection by the malware detection tool.

The aim of selecting this specific set of features is to enable learning of malicious associations between combinations of certain sets of permissions, API calls, and strings.

4.4.1. Combined Feature Set Model

In the combined feature set model, the three types of features, which are permissions, API calls, and strings, were extracted individually from each sample, however, were later merged inside the combined feature set. This combination and reduction processes of the feature set were possible as a result of the Delta-IDF value scoring based feature selection which was detailed in the previous section.

Python scripts were used for putting together the combined feature set among the whole possible sets of attributes that were extracted in the feature extraction phase of the data mining process. The script selected attributes from the three feature sets based on the calculated Delta-IDF values.

4.4.2. Classification of Android Malware via Machine Learning

Research and development of automatic classification methods have become essential due to the exponential increase and proliferation of Android malware. Static extraction

of APIs has been a widely used method in successful detection efforts against malware historically.

The framework developed in this thesis makes adequate use of supportive features that are permissions and strings in addition to the API calls. I was able to construct a combined feature set that enabled high detection rates against Android malware via an extensive study on the theoretical aspects, as well as the practical aspects of the research topic.

The implementation of the methodology presented in this thesis is not an automatic way of classifying Android malware. However, the proposed methodology could be integrated into an equivalent automatic Android malware detection implementation without any constraints or limitations.

The way the proposed method was implemented in the experiments conducted for this thesis was by making use of the data mining platform WEKA. The results produced by the machine learning algorithms on WEKA were used as detection ratios to simulate how the selected algorithms would perform in real-world use cases given the specific datasets. The results from the WEKA data mining experiments provided the classification of the APK samples inside the datasets as malware or benign, the True Positive (TP) designated the ratio of successful malware classifications, the True Negative (TN) designated the ratio of successful benign classifications and the weighted average designated the accuracy of the approach. The weighted average in that sense corresponds to the detection ratio with successful malware and benign detections combined, while TP corresponds to successful malware detections alone and TN corresponds to successful benign classifications alone.

4.5. Limitations of the Proposed Methodology

The model of Android detection on the Android presented in this thesis has a few limitations and in this section, they will be discussed.

Firstly, it does not directly handle obfuscated API calls or encrypted/packed APK files. There are tools available to unpack the packed executables automatically and they may be applied for de-obfuscation/decryption to use their output within this methodology. Such a feature was not implemented, but may be integrated into the data mining procedure in the future.

Secondly, the current implementation is an off-device detection mechanism, which means it was not directly deployed on an Android device to detect malware. The off-device detection mechanism can be particularly useful in the case of security scans before publishing on an app store.

Finally, this malware detection methodology exhibits the typical limitation of static analysis that is the inability to detect the type of malware attacks, in which the payload is stored in a remote host and retrieved after the app is executed on the target platform.

However, the objectives of this thesis were to obtain an acceptable degree of detection accuracy and to provide a new methodology for feature selection in machine learning based Android malware detection, which were both achieved successfully.



CHAPTER 5

EXPERIMENTS AND RESULTS

In this chapter, the experiments run on the WEKA will be evaluated and analyzed to determine the employed machine algorithms' suitability for malware detection on the Android platform.

Attributes from the three features were selected by using the approach proposed in this thesis and the WEKA's ". arff" extension files with the selected attributes were prepared for use in the data mining experiments using WEKA.

The majority of the well-known data mining and machine learning algorithms, including some combinations of ensemble algorithms, were experimented and tested with the Weka version 3.9.1. Test results of adequately performing algorithms will be given in tables of accuracy, TPR and true negative rate (TNR).

Several machine learning algorithms for binary classification present in Weka were applied to the MalGenome and AndroZoo datasets. While a few of these algorithms were decision tree models such as ID3 and J48, RandomForest, which is an ensemble learner of different decision trees, was also used in the experiments. A Naive Bayes algorithm and an instance based learner algorithm the k-NN were also used.

Additionally, several basic function-based classifiers such as radial basis function and some rule-based classifiers such as FURIA, MODLEM, OneR, LAC, and NNGE were also included in the tests, combinations of ensemble learning using either majority voting or average of probabilities were also included and the ones with the best accuracy values are given in the result tables.

AdaBoost M1, which is a meta-learner, was used with NNGE, which is a rule-based classifier. AdaBoost M1's results are also included taking into consideration their promising accuracy values. Several different types of artificial neural networks such as MLP and SVMs were also used in the experiments. A brief description of the machine learning algorithms used during the tests on WEKA will be given below:

ID3 (Iterative Dichotomiser 3) (Quinlan, 1986): An algorithm invented by Ross Quinlan, which is used to generate a decision tree from a dataset in decision tree learning. ID3 is generally used in the machine learning and natural language processing domains and is the precursor to the C4.5 algorithm (Quinlan, 1993).

J48: An implementation of the algorithm ID3 developed by the WEKA project team, uses an enhanced implementation of C4.5.

RandomForest (Breiman, 2001): An meta-ensemble learner and classifier algorithm that constructs and uses a forest of trees. It is composed of K decision trees such as CART, C4.5 or ID3, however in WEKA, it is implemented as K “random” decision trees.

FURIA (Fuzzy Unordered Rule Induction Algorithm) (Hühn & Huellermeier, 2009): A rule-based classifier that uses fuzzy logic and fuzzy rule induction, FURIA is an extension of the well-known RIPPER algorithm, which is a state-of-the-art rule learner, the advantages of RIPPER such as simple and comprehensible rule sets are preserved. FURIA includes a number of modifications and extensions such as learning fuzzy rules instead of conventional rules and contains unordered rule sets instead of rule lists. Additionally, FURIA uses an efficient rule stretching method when dealing with uncovered examples. It was shown that, in terms of classification accuracy, FURIA significantly outperforms the original RIPPER, along with other machine learning algorithms such as the C4.5.

MODLEM (Stefanowski, 1998): A machine learning algorithm that induces a minimum set of rules. These rules can be adopted as a classifier. It is a sequential covering algorithm, which was invented to cope with numeric data without using discretization. This algorithm considers nominal and numeric attributes as equal, and to find the best rule condition during rule induction, attribute's space is searched. However, numeric attribute's conditions are more precise and closely describe the class. Some aspects of this algorithm were taken from the Rough Set Theory that states the class definition can be described according to its lower or upper approximation.

NNge (Non-nested Generalized Exemplars) (Salzberg, 1991): A rule-based and nearest-neighbor-like classifier algorithm that uses non-nested generalized exemplars where these hyper-rectangles are implemented as “if-then” rules. Generalization is performed by merging exemplars and constructing hyper-rectangles in attribute space

which represent conjunctive rules with internal disjunction. Each time a new sample is added to the database, generalization is carried out by combining the sample with the nearest neighbor of the same class.

LAC (Lazy Associative Classifier) (Veloso, Jr., & Zaki, 2006): Uses associative rules to execute classifications. Unlike other a-priori-based classifiers, the LAC algorithm computes association rules on a demand-driven basis. LAC projects the training data only on features in the test dataset, in other words from all the training instances, only the samples that share at least one feature with the test samples are used. For each instance to be classified, it filters the training set, thus producing only useful rules for that instance. LAC outperforms traditional associative classifiers in both speed and accuracy.

OneR (Holte, 1993): A simple rule-based classifier that builds and uses only one single rule for classification of the instances in a way that uses the minimum-error attribute for prediction, discretizes the numeric attributes. In order to create a rule for a predictor, a frequency table for each predictor constructed against the target. OneR has been shown to produce rules that are only slightly less accurate than state-of-the-art machine learning algorithms while at the same time producing rules that are simple for humans to read and interpret.

AdaBoost M1 (Freund & Schapire, 1996): A meta-learner type of algorithm that boosts the classification performance of a classifier by using the Adaboost M1 method.

Radial Basis Function (Frank, 2014): A classifier algorithm that implements radial basis function networks, which is trained in total supervised type of machine learning. Radial Basis Function uses WEKA's optimization class by minimizing squared error with the Broyden–Fletcher–Goldfarb–Shanno (BFGS) method.

MLP (Haykin, 2009): An artificial neural network that uses the feed-forward learning model that has one or more hidden layers. Achieves the differential error and the corresponding weight updates by backpropagation with the gradient descent methodology.

SVM with Sequential Minimal Optimization (SMO): A specialized support vector machine classifier which implements John Platt's SMO (Platt, 1998) algorithm for the training phase. Given labeled training data (supervised learning), SVM (V. Vapnik & Lerner, 1963) is a discriminative classifier that constructs a separating hyperplane with

maximum margin, which categorizes the test samples into their predicted classes. In two dimensional space, this hyperplane becomes a line that divides a plane into two parts in which each class lays in either side.

Naive Bayes (John & Langley, 1995): A probabilistic classifier based on Bayes' theorem that assumes strong independence between the features.

k-NN: An instance-based learner type of algorithm which is based on the comparison of an undefined sample with the k training samples that happen to be the nearest neighbors of the undefined sample.

Train, 10-fold cross-validation and test set methods of running machine learning algorithms in Weka were used in the experiments. The highest level of attention was given to the obtained rates of TPR, representing the rate of malware samples correctly detected as malware and the true negative rate, representing the percentage of benign samples correctly identified as benign. The weighted average in Weka, which is the mathematical average of TP and TN rates, is the accuracy and malware detection rate of the approach proposed by this Ph.D. thesis.

For evaluation of the conducted experiments, four standard metrics were used:

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

5.1. Standard Metrics used for Evaluating the Classifiers

In order to measure the performance of each classifier, important values such as the numbers of correct or incorrect classifications were taken into consideration. Detailed descriptions of the used classification metrics will be given next.

5.1.1. Confusion Matrix and the Related Metrics

In table 4.2., a confusion matrix for computing the evaluation indicators is described.

Table 5.1. Confusion Matrix

		Predicted Class	
		Positive	Negative
Actual Class	Positive	True Positive (TP)	False Negative (FN)
	Negative	False Positive (FP)	True Negative (TN)

A confusion matrix contains the following entries:

- True Positives: malware samples predicted to be malware.
- True Negatives: benign samples predicted to be benign.
- False Positives: benign samples predicted to be malware.
- False Negatives: malware samples predicted to be benign.

The following metrics are calculable by the aforementioned confusion matrix components.

TPR signifies the ratio of correctly identified malware applications, given by:

$$TPR = \frac{TP}{TP + TN}$$

Likewise, TNR signifies the ratio of correctly identified benign applications and it is calculated by:

$$TNR = \frac{TN}{TN + FP}$$

FPR represents the ratio of malware-infested applications incorrectly identified as benign, given by:

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

The accuracy of a classifier is defined by the probability of correctly predicting the class of an unclassified sample (Baldi, Brunak, Chauvin, Andersen, & Nielsen, 2000). In the case of this thesis accuracy is an indicator representing the system's ratio of successful detection, expressed as the proportion of accurately identified benign and malware samples. We can calculate accuracy by:

$$Accuracy = \frac{(TP + TN)}{(TP + TN + FP + FN)}$$

As the number of positive (malicious) and benign (negative) samples in each dataset used in the experiments are equal, there is no need to use metrics such as Precision, F-Score or Kappa Statistics and it is enough to take into account the accuracy and TPR metrics in evaluating the success of the malware detection approach.

5.2. Cross-validation

A standard statistical method used for estimating the generalization error of a predictive model, cross-validation is a method of model validation that works by dividing data into two segments. One segment is used to train the machine learning classifier while the other segment is used to test it.

5.2.1. 10-fold Cross-Validation

10-fold cross-validation was used as a method of evaluating the success of the data mining algorithms on WEKA on the AndroZoo and MalGenome datasets. The procedure for carrying out 10-fold cross-validation is as follows: using the other 9 subsets as the training set a model is built and its performance is evaluated on the current subset. Each subset is used for testing exactly once and the result of the cross-validation is the average of the performances obtained from the 10 rounds.

The main disadvantage of this method is requiring the training phase be rerun from scratch 10 times, meaning it will take 10 times as many computations to perform the evaluation.

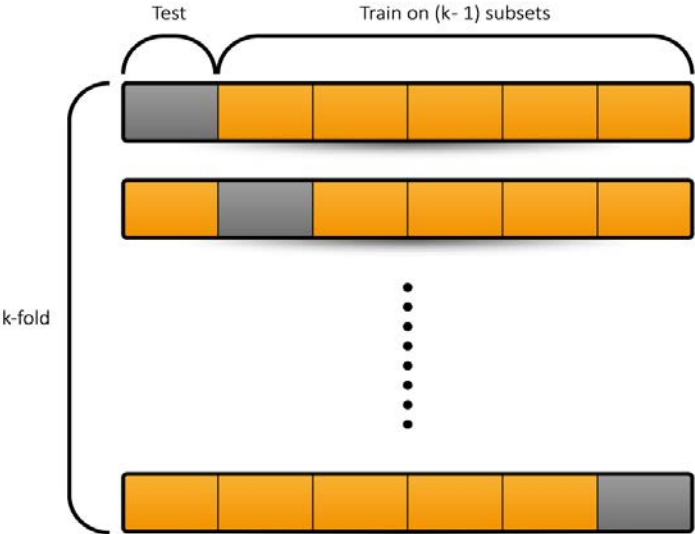


Figure 5.1. 10-fold Cross-Validation

5.3. Results from the First MalGenome Dataset

The dataset constructed using the MalGenome malware samples was subsampled into five different datasets with one training set containing 250 benign samples and 250 malware samples and 4 test sets composed of 125 malware and 125 benign samples. From the 500 benign records collected from the Google Play Store, benign samples were chosen randomly, similarly, from the 1000 malware records collected from the MalGenome Project, malware samples were chosen randomly. Each of the malware samples among the training dataset and each of the test datasets were chosen in a way ensuring that no duplicate records were present in the dataset. The machine learning algorithms that achieved the highest detection ratios during the experiments for the first MalGenome dataset, including several well-known types of classification algorithms such as decision trees, functional models, artificial neural networks, Naive Bayes and rule-based classifiers are given in tables Table 5.2., Table 5.3., and Table 5.4. The FURIA algorithm achieved the highest 10-fold cross-validation accuracy ratio of 0.984 among all the machine learning algorithms.

Table 2.2. MalGenome Dataset Accuracy Values

Algorithm Name and Parameters	Train (500 instances)	10-fold x-val. (500 instances)	Test set 1 (250 instances)	Test set 2 (250 instances)	Test set 3 (250 instances)	Test set 4 (250 instances)
Radial Basis Function	0.984	0.980	0.984	0.956	0.956	0.976
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.992	0.978	0.972	0.960	0.940	0.964
Id3 (decision tree)	0.998	0.970	0.948	0.960	0.968	0.964
k-NN (k nearest neighbors, k = 1)	0.998	0.976	0.952	0.940	0.932	0.956
NaiveBayes	0.962	0.962	0.984	0.956	0.944	0.960
J48 (improved version of C4.5 decision tree)	0.986	0.972	0.940	0.960	0.964	0.960
SVM (Support Vector Machine with Sequential Minimal Optimization)	0.986	0.978	0.972	0.956	0.952	0.972
FURIA (fuzzy unordered rule induction classifier)	0.984	0.984	0.940	0.960	0.960	0.964
RandomForest (ensemble of decision trees)	0.998	0.980	0.960	0.948	0.940	0.964
NNGE / AdaBoostM1 (Non-nested Generalized Exemplars rule based classification with AdaBoost M1 meta-learner)	0.998	0.974	0.972	0.972	0.960	0.972
Ensemble by Voting (majority voting) 3 classifiers (LAC, NNGE, MODLEM)	0.998	0.972	0.996	0.976	0.980	0.992
Ensemble by Voting (majority voting) 3 classifiers (OneR, NNGE, MODLEM)	0.998	0.970	1.000	0.988	0.992	0.992
Ensemble by Voting (majority voting) 4 classifiers (OneR, LAC, NNGE, MODLEM)	0.980	0.968	0.996	0.992	0.980	0.976

Table 5.3. MalGenome Dataset True Positive Rates

Algorithm Name and Parameters	Train (500 instances)	10-fold x-val. (500 instances)	Test set 1 (250 instances)	Test set 2 (250 instances)	Test set 3 (250 instances)	Test set 4 (250 instances)
Radial Basis Function	0.972	0.972	0.968	0.912	0.912	0.952
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.990	0.972	0.928	0.912	0.900	0.914
Id3 (decision tree)	0.996	0.968	0.896	0.920	0.936	0.928
k-NN (k nearest neighbors, k = 1)	0.996	0.964	0.904	0.880	0.864	0.912
NaiveBayes	0.972	0.972	0.984	0.936	0.952	0.984
J48 (improved version of C4.5 decision tree)	0.972	0.964	0.880	0.920	0.928	0.920
SVM (Support Vector Machine with Sequential Minimal Optimization)	0.976	0.968	0.944	0.912	0.904	0.944

FURIA (fuzzy unordered rule induction classifier)	0.972	0.972	0.880	0.920	0.920	0.928
RandomForest (ensemble of decision trees)	0.996	0.972	0.920	0.896	0.880	0.928
NNGE / AdaBoostM1 (Non-nested Generalized Exemplars rule based classification with AdaBoost M1 meta-learner)	0.996	0.968	0.944	0.944	0.920	0.944
Ensemble by Voting (majority voting) 3 classifiers (LAC, NNGE, MODLEM)	0.996	0.968	0.992	0.952	0.960	0.984
Ensemble by Voting (majority voting) 3 classifiers (OneR, NNGE, MODLEM)	0.996	0.972	1.000	0.976	0.984	0.984
Ensemble by Voting (majority voting) 4 classifiers (OneR, LAC, NNGE, MODLEM)	0.976	0.972	0.992	0.984	0.984	0.984

Table 5.4. MalGenome Dataset True Negative Rates

Algorithm Name and Parameters	Train (500 instances)	10-fold x-val. (500 instances)	Test set 1 (250 instances)	Test set 2 (250 instances)	Test set 3 (250 instances)	Test set 4 (250 instances)
Radial Basis Function	0.996	0.988	1.000	1.000	1.000	1.000
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.994	0.984	1.016	1.008	0.980	1.014
Id3 (decision tree)	1.000	0.972	1.000	1.000	1.000	1.000
k-NN (k nearest neighbors, k = 1)	1.000	0.988	1.000	1.000	1.000	1.000
NaiveBayes	0.952	0.952	0.984	0.976	0.936	0.936
J48 (improved version of C4.5 decision tree)	1.000	0.980	1.000	1.000	1.000	1.000
SVM (Support Vector Machine with Sequential Minimal Optimization)	0.996	0.988	1.000	1.000	1.000	1.000
FURIA (fuzzy unordered rule induction classifier)	0.996	0.996	1.000	1.000	1.000	1.000
RandomForest (ensemble of decision trees)	1.000	0.988	1.000	1.000	1.000	1.000
NNGE / AdaBoostM1 (Non-nested Generalized Exemplars rule based classification with AdaBoost M1 meta-learner)	1.000	0.980	1.000	1.000	1.000	1.000
Ensemble by Voting (majority voting) 3 classifiers (LAC, NNGE, MODLEM)	1.000	0.976	1.000	1.000	1.000	1.000
Ensemble by Voting (majority voting) 3 classifiers (OneR, NNGE, MODLEM)	1.000	0.968	1.000	1.000	1.000	1.000
Ensemble by Voting (majority voting) 4 classifiers (OneR, LAC, NNGE, MODLEM)	0.984	0.964	1.000	1.000	0.976	0.968

5.4. Results from the Second MalGenome Dataset

In an attempt to get results from a larger dataset that include all of the MalGenome dataset's malware samples, a new dataset was formed that include all the 1260 malware samples in the MalGenome Project. The number of available benign samples remained 300, however by using SMOTE (Synthetic Minority Over-sampling Technique), the number of benign samples was up-sampled (over-sampled) to construct a balanced dataset in WEKA. Results for the balanced and the imbalanced dataset constructed only with the physically available benign samples will be presented in the same tables. The machine learning algorithms that performed the best results in this second MalGenome dataset are given in tables Table 5.5, Table 5.6, Table 5.7, Table 5.8, Table 5.9 and Table 5.10.

The experiment types and the dataset details in tables Table 5.5, Table 5.7 and Table 5.9 will be detailed in the following sentence. In the first column of these tables, the 10-fold cross-validation results of 1260 malware and 250 benign samples are given, in the second column, 10-fold cross-validation of 1260 malware and 1260 benign

samples (produced by SMOTE) are given, the third column correspond to the train results of the 840 malware and 840 benign samples and the fourth column correspond to the results of using a test set that contains 420 malware and 420 benign samples.

The experiment types and the dataset details in tables Table 5.6, Table 5.8 and Table 5.10 will be detailed in the following sentence. In the first column of these tables, 1/3 test split (hold-out) results of 1510 instances with a test size of 422 malware and 81 benign samples are given, in the second column, 1/3 test split (hold-out) results of 2520 instances with a test size of 420 malware and 420 benign samples are given, the third column correspond to the train results of 840 malware and 840 benign samples, and the fourth column correspond to using a test set of 420 malware and 420 benign samples.

Table 5.5. MalGenome 2nd Dataset Accuracy Values Part 1

Algorithm Name and Parameters	10-fold-xval (1510 instances) 1260 mal + 250 ben	SMOTE- balanced-10- fold-xval (2520 instances) 1260 mal + 1260 ben	SMOTE- balanced- TRAIN (1680 instances) 840 mal + 840 ben	SMOTE- balanced- TEST (840 instances) 420 mal + 420 ben
Radial Basis Function	0.984	0.994	0.992	0.994
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.983	0.992	0.997	0.99
Multilayer Perceptron (default param. in Weka lr=0.3, mom=0.2, hidden: 19. 500 iter.)	0.981	0.992	0.997	0.99
1-NN (1/d weighted)	0.983	0.992	0.999	0.99
NaiveBayes	0.983	0.958	0.958	0.961
BayesianLogisticRegression	0.984	0.991	0.993	0.992
VotedPerceptron	0.985	0.991	0.99	0.989
SVM (Support Vector Machine)	0.986	0.992	0.995	0.988
Lazy Bayesian Rules Classifier (LBR)	0.988	0.992	0.995	0.992
FURIA (fuzzy unordered rule induction)	0.983	0.993	0.996	0.987
MODLEM (rough set-based rule classifier with default param.)	0.97	0.986	0.999	0.988
LAC (lazy associative rule-based with default param.)	0.953	0.959	0.958	0.961
OneR (rule-based)*	0.982	0.957	0.957	0.957
NNGE (rule-based with default param.)	0.977	0.989	0.999	0.993
JRIP (rule-based Repeated Incremental Pruning to Produce Error Reduction (RIPPER))	0.985	0.993	0.994	0.993
RIDOR (rule-nased RIPple-DOWn Rule learner)	0.982	0.993	0.996	0.993
Id3 (decision tree)	0.976	0.99	0.999	0.992
J48 (improved version of C4.5 decision tree)	0.984	0.993	0.994	0.993

CART (decision tree)	0.983	0.992	0.994	0.993
Functional Tree (decision tree)	0.988	0.99	0.997	0.989
DecisionStump (decision tree)*	0.982	0.957	0.957	0.957
NBTree (NaiveBayes decision tree)	0.985	0.992	0.993	0.989
LMT (logistic model trees - decision tree)	0.983	0.992	0.999	0.992
RandomForest (ensemble of decision trees)	0.983	0.992	0.999	0.992
NNGE / AdaBoostM1	0.981	0.992	0.999	0.99
Ensemble by Voting (majority voting) 2 classifiers (LAC, NNGE)	0.977	0.989	0.999	0.993
Ensemble by Voting (majority voting) 3 classifiers (OneR, J48, RBF classifier)	0.983	0.994	0.993	0.994
Ensemble by Voting (majority voting) 3 classifiers (OneR, FunctionalTree, LazyBayesianRules)	0.989	0.994	0.996	0.99
Ensemble by Voting (majority voting) 3 classifiers (J48, FunctionalTree, LazyBayesianRules)	0.989	0.994	0.996	0.993
JRIP / RealAdaBoost	0.983	0.993	0.998	0.989
Ensemble by Voting (majority voting) 4 classifiers (JRIP, OneR, FunctionalTree, LazyBayesianRules)	0.988	0.993	0.995	0.992

Table 5.6. MalGenome 2nd Dataset Accuracy Values Part 2

Algorithm Name and Parameters	1/3 Test split (hold-out) (1510 instances) test size: 422 mal + 81 ben	SMOTE-balanced-1/3 Test split (hold-out) (2520 instances) test size: 420 mal + 420 ben	Imbalanced-TRAIN (1005 instances) 840 mal + 165 ben	Imbalanced-TEST (505 instances) 420 mal + 85 ben
Radial Basis Function	0.988	0.995	0.983	0.984
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.988	0.996	0.995	0.988
Multilayer Perceptron (default param. in Weka lr=0.3, mom=0.2, hidden: 19. 500 iter.)	0.982	0.996	0.995	0.982
1-NN (1/d weighted)	0.986	0.996	0.999	0.984
NaiveBayes	0.988	0.956	0.982	0.984
BayesianLogisticRegression	0.988	0.996	0.987	0.988
VotedPerceptron	0.99	0.994	0.981	0.982
SVM (Support Vector Machine)	0.986	0.995	0.99	0.98
Lazy Bayesian Rules Classifier (LBR)	0.99	0.992	0.982	0.984
FURIA (fuzzy unordered rule induction)	0.988	0.996	0.993	0.978
MODLEM (rough set-based rule classifier with default param.)	0.964	0.995	0.999	0.972
LAC (lazy associative rule-based with default param.)	0.952	0.957	0.956	0.952
OneR (rule-based)*	0.986	0.956	0.982	0.982
NNGE (rule-based with default param.)	0.976	0.993	0.999	0.982

JRIP (rule-based Repeated Incremental Pruning to Produce Error Reduction (RIPPER))	0.988	0.996	0.99	0.982
RIDOR (rule-nased RIpPLE-DOWn Rule learner)	0.988	0.988	0.987	0.98
Id3 (decision tree)	0.98	0.993	0.999	0.98
J48 (improved version of C4.5 decision tree)	0.99	0.996	0.989	0.982
CART (decision tree)	0.99	0.996	0.982	0.982
Functional Tree (decision tree)	0.988	0.995	0.994	0.984
DecisionStump (decision tree)*	0.986	0.956	0.982	0.982
NBTree (NaiveBayes decision tree)	0.86	0.99	0.99	0.99
LMT (logistic model trees - decision tree)	0.988	0.994	0.988	0.984
RandomForest (ensemble of decision trees)	0.984	0.996	0.999	0.988
NNGE / AdaBoostM1	0.984	0.995	0.999	0.984
Ensemble by Voting (majority voting) 2 classifiers (LAC, NNGE)	0.964	0.967	0.999	0.982
Ensemble by Voting (majority voting) 3 classifiers (OneR, J48, RBF classifier)	0.986	0.996	0.983	0.984
Ensemble by Voting (majority voting) 3 classifiers (OneR, FunctionalTree, LazyBayesianRules)	0.99	0.995	0.984	0.984
Ensemble by Voting (majority voting) 3 classifiers (J48, FunctionalTree, LazyBayesianRules)	0.99	0.996	0.99	0.986
JRIP / RealAdaBoost	0.992	0.995	0.999	0.978
Ensemble by Voting (majority voting) 4 classifiers (JRIP, OneR, FunctionalTree, LazyBayesianRules)	0.988	0.995	0.989	0.988

Table 5.7. MalGenome 2nd Dataset True Positive Rates Part 1

Algorithm Name and Parameters	10-fold-xval (1510 instances) 1260 mal + 250 ben	SMOTE- balanced-10-fold- xval (2520 instances) 1260 mal + 1260 ben	SMOTE-balanced- TRAIN (1680 instances) 840 mal + 840 ben	SMOTE-balanced- TEST (840 instances) 420 mal + 420 ben
Radial Basis Function	0.993	0.99	0.99	0.99
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.989	0.988	0.994	0.988
Multilayer Perceptron (default param. in Weka lr=0.3, mom=0.2, hidden: 19. 500 iter.)	0.989	0.988	0.994	0.988
1-NN (1/d weighted)	0.991	0.988	0.998	0.986
NaiveBayes	0.991	0.992	0.993	0.99
BayesianLogisticRegression	0.994	0.988	0.99	0.988
VotedPerceptron	0.991	0.987	0.988	0.983
SVM (Support Vector Machine)	0.993	0.989	0.993	0.988
Lazy Bayesian Rules Classifier (LBR)	0.991	0.986	0.992	0.986

FURIA (fuzzy unordered rule induction)	0.991	0.99	0.994	0.986
MODLEM (rough set-based rule classifier with default param.)	0.993	0.99	0.998	0.988
LAC (lazy associative rule-based with default param.)	0.994	0.992	0.993	0.99
OneR (rule-based)*	0.994	0.994	0.994	0.993
NNGE (rule-based with default param.)	0.989	0.989	0.998	0.99
JRIP (rule-based Repeated Incremental Pruning to Produce Error Reduction (RIPPER))	0.989	0.99	0.993	0.988
RIDOR (rule-nased Ripple-DOWn Rule learner)	0.94	0.99	0.993	0.988
Id3 (decision tree)	0.989	0.989	0.998	0.988
J48 (improved version of C4.5 decision tree)	0.99	0.991	0.993	0.988
CART (decision tree)	0.99	0.989	0.993	0.988
Functional Tree (decision tree)	0.972	0.99	0.994	0.988
DecisionStump (decision tree)*	0.994	0.994	0.994	0.993
NBTree (NaiveBayes decision tree)	0.992	0.99	0.993	0.988
LMT (logistic model trees - decision tree)	0.992	0.99	0.998	0.988
RandomForest (ensemble of decision trees)	0.991	0.99	0.998	0.988
NNGE / AdaBoostM1	0.991	0.99	0.998	0.988
Ensemble by Voting (majority voting) 2 classifiers (LAC, NNGE)	0.989	0.989	0.998	0.99
Ensemble by Voting (majority voting) 3 classifiers (OneR, J48, RBF classifier)	0.994	0.995	0.993	0.99
Ensemble by Voting (majority voting) 3 classifiers (OneR, FunctionalTree, LazyBayesianRules)	0.994	0.993	0.994	0.988
Ensemble by Voting (majority voting) 3 classifiers (J48, FunctionalTree, LazyBayesianRules)	0.992	0.991	0.994	0.988
JRIP / RealAdaBoost	0.99	0.99	0.998	0.986
Ensemble by Voting (majority voting) 4 classifiers (JRIP, OneR, FunctionalTree, LazyBayesianRules)	0.994	0.992	0.994	0.988

Table 5.8. MalGenome 2nd Dataset True Positive Rates Part 2

Algorithm Name and Parameters	1/3 Test split (hold-out) (1510 instances) test size: 422 mal + 81 ben	SMOTE-balanced-1/3 Test split (hold-out) (2520 instances) test size: 420 mal + 420 ben	Imbalanced-TRAIN (1005 instances) 840 mal + 165 ben	Imbalanced-TEST (505 instances) 420 mal + 85 ben
Radial Basis Function	0.998	0.991	0.993	0.993
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.993	0.993	0.994	0.986
Multilayer Perceptron (default param. in Weka lr=0.3, mom=0.2, hidden: 19, 500 iter.)	0.995	0.993	0.994	0.988

1-NN (1/d weighted)	0.995	0.993	1	0.988
NaiveBayes	0.995	0.993	0.992	0.99
BayesianLogisticRegression	0.998	0.993	0.994	0.993
VotedPerceptron	0.998	0.993	0.993	0.993
SVM (Support Vector Machine)	0.998	0.991	0.993	0.981
Lazy Bayesian Rules Classifier (LBR)	0.995	0.984	0.992	0.99
FURIA (fuzzy unordered rule induction)	0.998	0.993	0.993	0.981
MODLEM (rough set-based rule classifier with default param.)	0.998	0.998	1	0.99
LAC (lazy associative rule-based with default param.)	0.998	0.993	0.995	0.993
OneR (rule-based)*	0.998	0.995	0.994	0.993
NNGE (rule-based with default param.)	0.993	0.991	0.999	0.99
JRIP (rule-based Repeated Incremental Pruning to Produce Error Reduction (RIPPER))	0.995	0.993	0.99	0.979
RIDOR (rule-nased Ripple-Down Rule learner)	0.991	0.977	0.994	0.988
Id3 (decision tree)	0.998	0.993	1	0.986
J48 (improved version of C4.5 decision tree)	0.993	0.993	0.994	0.988
CART (decision tree)	0.993	0.993	0.994	0.993
Functional Tree (decision tree)	0.995	0.993	0.994	0.99
DecisionStump (decision tree)*	0.998	0.995	0.994	0.993
NBTree (NaiveBayes decision tree)	0.991	0.991	0.992	0.99
LMT (logistic model trees - decision tree)	0.998	0.993	0.994	0.993
RandomForest (ensemble of decision trees)	0.995	0.993	0.999	0.99
NNGE / AdaBoostM1	0.998	0.991	0.999	0.99
Ensemble by Voting (majority voting) 2 classifiers (LAC, NNGE)	0.995	0.991	0.999	0.99
Ensemble by Voting (majority voting) 3 classifiers (OneR, J48, RBF classifier)	0.998	0.993	0.994	0.993
Ensemble by Voting (majority voting) 3 classifiers (OneR, FunctionalTree, LazyBayesianRules)	0.998	0.993	0.994	0.993
Ensemble by Voting (majority voting) 3 classifiers (J48, FunctionalTree, LazyBayesianRules)	0.995	0.993	0.994	0.99
JRIP / RealAdaBoost	0.998	0.993	1	0.99
Ensemble by Voting (majority voting) 4 classifiers (JRIP, OneR, FunctionalTree, LazyBayesianRules)	0.995	0.993	0.994	0.99

Table 5.9. MalGenome 2nd Dataset True Negative Rates Part 1

Algorithm Name and Parameters	10-fold-xval (1510 instances) 1260 mal + 250 ben	SMOTE- balanced-10- fold-xval (2520 instances) 1260 mal + 1260 ben	SMOTE- balanced- TRAIN (1680 instances) 840 mal + 840 ben	SMOTE- balanced-TEST (840 instances) 420 mal + 420 ben
Radial Basis Function	0.94	0.998	0.994	0.998
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.952	0.997	1	0.993
Multilayer Perceptron (default param. in Weka lr=0.3, mom=0.2, hidden: 19. 500 iter.)	0.944	0.996	1	0.993
1-NN (1/d weighted)	0.94	0.996	1	0.995
NaiveBayes	0.94	0.923	0.924	0.931
BayesianLogisticRegression	0.936	0.994	0.995	0.995
VotedPerceptron	0.952	0.994	0.992	0.995
SVM (Support Vector Machine)	0.952	0.994	0.998	0.988
Lazy Bayesian Rules Classifier (LBR)	0.972	0.998	0.998	0.998
FURIA (fuzzy unordered rule induction)	0.94	0.995	0.998	0.988
MODLEM (rough set-based rule classifier with default param.)	0.852	0.982	1	0.988
LAC (lazy associative rule-based with default param.)	0.748	0.926	0.924	0.931
OneR (rule-based)*	0.924	0.92	0.919	0.921
NNGE (rule-based with default param.)	0.916	0.989	1	0.995
JRIP (rule-based Repeated Incremental Pruning to Produce Error Reduction (RIPPER))	0.964	0.995	0.995	0.998
RIDOR (rule-nased RIpple-DOWn Rule learner)	0.99	0.996	0.999	0.998
Id3 (decision tree)	0.912	0.991	1	0.995
J48 (improved version of C4.5 decision tree)	0.952	0.995	0.995	0.998
CART (decision tree)	0.944	0.994	0.995	0.998
Functional Tree (decision tree)	0.991	0.991	1	0.99
DecisionStump (decision tree)*	0.924	0.92	0.919	0.921
NBTree (NaiveBayes decision tree)	0.948	0.994	0.993	0.99
LMT (logistic model trees - decision tree)	0.94	0.994	1	0.995
RandomForest (ensemble of decision trees)	0.94	0.994	1	0.995
NNGE / AdaBoostM1	0.932	0.994	1	0.993
Ensemble by Voting (majority voting) 2 classifiers (LAC, NNGE)	0.916	0.989	1	0.995
Ensemble by Voting (majority voting) 3 classifiers (OneR, J48, RBF classifier)	0.928	0.992	0.994	0.998

Ensemble by Voting (majority voting) 3 classifiers (OneR, FunctionalTree, LazyBayesianRules)	0.964	0.994	0.998	0.993
Ensemble by Voting (majority voting) 3 classifiers (J48, FunctionalTree, LazyBayesianRules)	0.972	0.998	0.999	0.998
JRIP / RealAdaBoost	0.948	0.995	0.999	0.993
Ensemble by Voting (majority voting) 4 classifiers (JRIP, OneR, FunctionalTree, LazyBayesianRules)	0.96	0.994	0.996	0.995

Table 5.10. MalGenome 2nd Dataset True Negative Rates Part 2

Algorithm Name and Parameters	1/3 Test split (hold-out) (1510 instances) test size: 422 mal + 81 ben	SMOTE-balanced-1/3 Test split (hold-out) (2520 instances) test size: 420 mal + 420 ben	Imbalanced-TRAIN (1005 instances) 840 mal + 165 ben	Imbalanced-TEST (505 instances) 420 mal + 85 ben
Radial Basis Function	0.938	1	0.933	0.941
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.963	1	1	1
Multilayer Perceptron (default param. in Weka lr=0.3, mom=0.2, hidden: 19. 500 iter.)	0.914	1	1	0.953
1-NN (1/d weighted)	0.938	1	0.994	0.965
NaiveBayes	0.951	0.917	0.933	0.953
BayesianLogisticRegression	0.938	1	0.952	0.965
VotedPerceptron	0.951	0.995	0.921	0.929
SVM (Support Vector Machine)	0.926	1	0.976	0.976
Lazy Bayesian Rules Classifier (LBR)	0.963	1	0.933	0.953
FURIA (fuzzy unordered rule induction)	0.938	1	0.994	0.965
MODLEM (rough set-based rule classifier with default param.)	0.79	0.993	0.994	0.882
LAC (lazy associative rule-based with default param.)	0.716	0.92	0.758	0.753
OneR (rule-based)*	0.926	0.915	0.921	0.929
NNGE (rule-based with default param.)	0.889	0.995	1	0.941
JRIP (rule-based Repeated Incremental Pruning to Produce Error Reduction (RIPPER))	0.951	1	0.988	1
RIDOR (rule-nased RIpPLE-Down Rule learner)	0.975	1	0.952	0.941
Id3 (decision tree)	0.889	0.993	0.994	0.953
J48 (improved version of C4.5 decision tree)	0.975	1	0.964	0.953
CART (decision tree)	0.975	1	0.921	0.929

Functional Tree (decision tree)	0.951	0.998	0.994	0.953
DecisionStump (decision tree)*	0.926	0.915	0.921	0.929
NBTree (NaiveBayes decision tree)	0.963	0.99	0.982	0.988
LMT (logistic model trees - decision tree)	0.938	0.995	0.958	0.941
RandomForest (ensemble of decision trees)	0.926	1	1	0.976
NNGE / AdaBoostM1	0.914	1	1	0.953
Ensemble by Voting (majority voting) 2 classifiers (LAC, NNGE)	0.802	0.942	1	0.941
Ensemble by Voting (majority voting) 3 classifiers (OneR, J48, RBF classifier)	0.926	1	0.927	0.941
Ensemble by Voting (majority voting) 3 classifiers (OneR, FunctionalTree, LazyBayesianRules)	0.951	0.998	0.933	0.941
Ensemble by Voting (majority voting) 3 classifiers (J48, FunctionalTree, LazyBayesianRules)	0.963	1	0.97	0.965
JRIP / RealAdaBoost	0.963	0.998	0.994	0.918
Ensemble by Voting (majority voting) 4 classifiers (JRIP, OneR, FunctionalTree, LazyBayesianRules)	0.951	0.998	0.964	0.976

5.5. Results from the AndroZoo Dataset

The dataset constructed from the AndroZoo malware samples was subsampled into four different datasets including one training set with 300 malware and 300 benign samples, and three different test sets out of which two have 150 malware and 150 benign, and one has 200 malware and 200 benign samples. Similar to the construction of the train and test datasets of the MalGenome dataset, the benign samples were chosen randomly from the 600 benign records collected from the Google Play Store, and the malware samples were chosen randomly from 1300 malware records collected from the AndroZoo Project. The data mining process execution methods of train sets, test sets, and 10-fold cross-validation were executed on Weka using many of the available machine learning algorithms. The machine learning algorithms achieving the highest detection ratios among the experiments for the AndroZoo dataset including several well-known types of classification algorithms are given in tables Table 5.11, Table 5.12, and Table 5.13. For this dataset, FURIA, ID3, and J48 machine learning algorithms achieved the highest 10-fold cross-validation accuracy ratio of 0.988.

Table 5.11. AndroZoo Dataset Accuracy Values

Algorithm Name and Parameters	Train (600 instances)	10-fold x-val. (600 instances)	Test set 1 (300 instances)	Test set 2 (300 instances)	Test set 3 (400 instances)
Radial Basis Function	0.987	0.982	0.980	0.980	0.980
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.992	0.983	0.983	0.983	0.985
Id3 (decision tree)	0.992	0.988	0.983	0.987	0.988
k-NN (k nearest neighbors, k = 1)	0.992	0.985	0.983	0.987	0.988
NaiveBayes	0.960	0.960	0.957	0.950	0.960
J48 (improved version of C4.5 decision tree)	0.988	0.988	0.980	0.983	0.980
SVM (Support Vector Machine with Sequential Minimal Optimization)	0.990	0.985	0.983	0.983	0.983
FURIA (fuzzy unordered rule induction classifier)	0.990	0.988	0.983	0.983	0.983
RandomForest (ensemble of decision trees)	0.992	0.985	0.987	0.987	0.988
NNGE / AdaBoostM1 (Non-nested Generalized Exemplars rule based classification with AdaBoost M1 meta-learner)	0.992	0.987	0.987	0.987	0.988
Ensemble by Voting (average of probabilities) 3 classifiers (LAC, NNGE, MODLEM)	0.992	0.978	0.987	0.987	0.988
Ensemble by Voting (average of probabilities) 3 classifiers (OneR, NNGE, MODLEM)	0.992	0.983	0.987	0.987	0.988
Ensemble by Voting (average of probabilities) 4 classifiers (OneR, LAC, NNGE, MODLEM)	0.992	0.983	0.987	0.987	0.988

Table 5.12. AndroZoo Dataset True Positive Rates

Algorithm Name and Parameters	Train (600 instances)	10-fold x-val. (600 instances)	Test set 1 (300 instances)	Test set 2 (300 instances)	Test set 3 (400 instances)
Radial Basis Function	0.990	0.990	0.987	0.980	0.975
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.997	0.987	0.993	0.987	0.985
Id3 (decision tree)	0.997	0.990	0.987	0.987	0.985
k-NN (k nearest neighbors, k = 1)	0.997	0.997	0.993	0.993	0.990
NaiveBayes	1.000	1.000	0.993	0.993	0.990
J48 (improved version of C4.5 decision tree)	0.990	0.990	0.980	0.980	0.970
SVM (Support Vector Machine with Sequential Minimal Optimization)	0.993	0.987	0.987	0.980	0.975
FURIA (fuzzy unordered rule induction classifier)	0.993	0.990	0.987	0.980	0.975
RandomForest (ensemble of decision trees)	0.997	0.987	0.993	0.987	0.985
NNGE / AdaBoostM1 (Non-nested Generalized Exemplars rule based classification with AdaBoost M1 meta-learner)	0.997	0.993	0.993	0.987	0.985
Ensemble by Voting (average of probabilities) 3 classifiers (LAC, NNGE, MODLEM)	0.997	0.990	0.993	0.987	0.985
Ensemble by Voting (average of probabilities) 3 classifiers (OneR, NNGE, MODLEM)	0.997	0.987	0.993	0.987	0.988
Ensemble by Voting (average of probabilities) 4 classifiers (OneR, LAC, NNGE, MODLEM)	0.997	0.987	0.993	0.987	0.985

Table 5.13. AndroZoo Dataset True Negative Rates

Algorithm Name and Parameters	Train (600 instances)	10-fold x-val. (600 instances)	Test set 1 (300 instances)	Test set 2 (300 instances)	Test set 3 (400 instances)
Radial Basis Function	0.984	0.974	0.973	0.980	0.985
Multilayer Perceptron (learning rate=0.3, momentum=0.2, one hidden layer with 8 nodes, 500 iterations)	0.987	0.977	0.973	0.979	0.985
Id3 (decision tree)	0.987	0.986	0.979	0.987	0.991
k-NN (k nearest neighbors, k = 1)	0.987	0.973	0.973	0.981	0.986
NaiveBayes	0.920	0.920	0.921	0.907	0.930

J48 (improved version of C4.5 decision tree)	0.986	0.986	0.980	0.986	0.990
SVM (Support Vector Machine with Sequential Minimal Optimization)	0.987	0.983	0.979	0.986	0.991
FURIA (fuzzy unordered rule induction classifier)	0.987	0.986	0.979	0.986	0.991
RandomForest (ensemble of decision trees)	0.987	0.983	0.981	0.987	0.991
NNGE / AdaBoostM1 (Non-nested Generalized Exemplars rule based classification with AdaBoost M1 meta-learner)	0.987	0.981	0.981	0.987	0.991
Ensemble by Voting (average of probabilities) 3 classifiers (LAC, NNGE, MODLEM)	0.987	0.966	0.981	0.987	0.991
Ensemble by Voting (average of probabilities) 3 classifiers (OneR, NNGE, MODLEM)	0.987	0.979	0.981	0.987	0.982
Ensemble by Voting (average of probabilities) 4 classifiers (OneR, LAC, NNGE, MODLEM)	0.987	0.979	0.981	0.987	0.991



CHAPTER 6

CONCLUSIONS

In this concluding chapter, the contributions of this thesis and their potential impacts are discussed.

6.1. Summary of Thesis Contributions

A novel feature selection methodology is proposed and implemented in this Ph.D. study along with a new model for malware detection on the Android. The proposed approach was tested using nearly all the available machine learning algorithms on WEKA along with some ensembles of certain machine learning algorithms. High accuracy values and malware detection ratios were obtained, which prove the usefulness of the approach in real world cases.

The malware detection using machine learning methodology proposed in this thesis achieved the highest TPRs ranging between 99.6% and 100% for the first MalGenome dataset and ranging between 99.3% and 100% for the AndroZoo dataset.

In terms of accuracy rates in the first MalGenome dataset, ensemble learner by majority voting of three classifiers (OneR, NNGE, MODLEM) achieved the best rates with 99.8%, 97.2%, 99.6%, 97.6% and 99.2% for the train, 10-fold cross-validation, test set 1, test set 2, test set 3 and test set 4 experiments, respectively.

Considering the accuracy results in the second MalGenome dataset, the JRIP / RealAdaBoost classifier achieved the highest accuracy rate of 99.2% in the 1/3 test split (hold-out) experiment that contains 1510 instances.

Majority voting of three classifiers (OneR, J48 and the RBF classifier) achieved the highest TPR of 99.5% in the 10-fold cross validation experiment that include 1260 malware and 1260 benign samples.

Modlem achieved the highest TPR of 99.8% in the 1/3 hold-out experiment that contain 2520 instances with test sizes of 420 malware and 420 benign samples.

The following ensembles that contain three classifiers: OneR, J48 and the RBF classifier; OneR, FunctionalTree and the LazyBayesianRules; J48, FunctionalTree and the LazyBayesianRules along with the ensemble that contain four classifiers: JRIP, OneR, FunctionalTree and the LazyBayesianRules, achieved some of the best results by obtaining the highest accuracy of 99.6% and the highest TPR of 99.8%.

For the AndroZoo dataset, the highest malware detection accuracy rates were observed to be ranging from 98.7% to 98.8%, and these results slightly outperform the results of the study conducted by Wu et al. (S. Wu et al., 2016), where they had achieved the highest detection ratio of 97.66% using the k-NN classifier.

In terms of accuracy rates with the AndroZoo dataset, NNGE / AdaBoostM1 achieved the best rates with 99.2%, 98.7%, 98.7%, 98.7% and 98.8% for the train, 10-fold cross-validation, test set 1, test set 2 and test set 3 experiments, respectively.

Additionally, the TPR results of the AndroZoo dataset given in Table 5.12, range from 97% to 100%. In terms of the TPR values by each classifier, Naive Bayes achieved the best results with 100% for train and 10-fold cross-validation, and 99.3% for test sets 1 and 2, and 99% for the test set 3. The k-NN classifier produced the second best results in terms of TPR with 99.7% for train and 10-fold cross-validation, and 99.3% for test sets 1 and 2, and 99% for the test set 3.

Several ensemble learners that use majority voting achieved the highest accuracy ratios upon four different test sets derived from the MalGenome project. Three different ensemble models which include, one model with three classifiers (LAC, NNGE, MODLEM), another one with three classifiers (OneR, NNGE, MODLEM), and another ensemble model with four classifier algorithms (OneR, LAC, NNGE, MODLEM) were the ensemble learners that achieved the highest accuracy ratios. The test accuracies of these ensemble learners were observed to be ranging between 0.992 and 1.

Concerning the AndroZoo dataset, it was further observed that besides these three ensemble models, another ensemble learner which is the RandomForest, AdaBoost M1 meta-learner that uses the NNGE classifier, k-NN, and ID3 achieved the highest test accuracy values of 0.987 and 0.988.

It can be asserted that the feature selection methodology proposed in this thesis is successful owing to the outstanding and promising accuracy values and malware

detection performances. A novel approach for the feature selection process that makes use of IDF values in a different way has been designed and developed. Additionally, a new malware detection approach that implements the proposed methodology for feature selection and explores ensemble learning with different algorithms, has been implemented and evaluated.

It can be further concluded that, the machine learning algorithms utilized in this work were shown to obtain considerably high accuracy rates for malware detection on the Android platform by using the feature selection and malware detection methodologies proposed in this Ph.D. thesis.

6.2. Directions for Future Work

Further studies and directions for future work are discussed in this section.

6.2.1. New Experiments with Bigger Datasets

Due to time constraints and download limitations (mass download methods were not available) in the Google Play Store, a high amount of benign application samples could not be utilized in this study. Therefore, one of the future works is to increase the number of benign samples included in the experiments. This has already been carried out for the MalGenome dataset by utilizing the up-sampling statistical method of increasing sample size and it will also be carried out for the AndroZoo dataset by downloading large numbers of Android benign apps from the AndroZoo Project. The similar experiments and tests will be carried out with the same and potentially some extra classifiers on the new dataset to observe how the methodology scales to greater numbers of malware and benign samples.

6.2.2. Additional Research for Ensemble Classifiers with the AndroZoo Dataset

In the MalGenome dataset, ensemble classifiers achieved the best TPR scores, in contrast to this, in the AndroZoo dataset, basic single classifiers such as Naive Bayes and the k-NN achieved the best TPR. The ensemble learners would have been expected to provide the best TPR scores in the AndroZoo dataset as well. This may be the consequence of a number of factors such as the fact that various malware types with different features and symptoms were included within each dataset, or the fact that

train and test sets were constructed randomly. The second future work will be carrying out some additional research into this situation.

6.2.3. Alternative Methodologies for Feature Selection

The novel feature selection methodology proposed in this thesis could be combined with different known feature selection methodologies to provide an alternative hybrid model of feature selection. In such a scenario, the attributes would be extracted and selected according to the original proposed methodology and afterwards, by using various ranker (such as Information Gain, Mutual Information, Chi-Square) methods and/or subset selection and/or Wrapper methods, from the currently selected attributes, some additional attributes could be discarded. The same set of classifiers would be used to get the results from the train and test sets constructed with this hybrid methodology. The results obtained by this alternative methodology would be compared against the results obtained using the original methodology to measure the success of using such a hybrid methodology.

6.2.4. Malware Detection Tool for Android Platforms

The fourth and final potential future study is to implement this new malware detection methodology as an Android application that could be deployed on mobile devices. As it was previously noted, the methodology proposed in this thesis is an off-line detection methodology. Within the time constraints of this thesis, it was not possible to build the actual Android application for malware detection that implements the proposed methodology. However, within a larger timeframe, a mobile application that implements the proposed malware detection methodology could be implemented and published on the Google Play Store.

REFERENCES

- Aafer, Y., Du, W., & Yin, H. (2013). DroidAPIMiner: Mining API-level features for robust malware detection in Android. *SecureComm, volume 127 of Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering, Springer*, 86–103.
- Aha, D. W., Kibler, D., & Albert, M. K. (1991). Instance-Based learning algorithms. *Machine Learning*, 6, 37-66.
- Alatwi, H. A., Oh, T., Fokoue, E., & Stackpole, B. (2016). *Android malware detection using category-based machine learning classifiers* Paper presented at the Proceedings of the 17th Annual Conference on Information Technology Education, Boston, Massachusetts, USA.
- Allix, K., Bissyandé, T. F., Klein, J., & Traon, Y. L. (2016). *AndroZoo: Collecting millions of Android apps for the research community*. Paper presented at the 2016 IEEE/ACM 13th Working Conference on Mining Software Repositories (MSR), Austin, TX.
- Almeida, M., Bilal, M., Finamore, A., Leontiadis, I., Grunenberger, Y., Varvello, M., & Blackburn, J. (2018). *CHIMP: Crowdsourcing Human Inputs for Mobile Phones*. Paper presented at the Proceedings of the 2018 World Wide Web Conference, Lyon, France.
- Alzaylaee, M. K., Yerima, S. Y., & Sezer, S. (2017). *EMULATOR vs REAL PHONE: Android malware detection using machine learning*. Paper presented at the 3rd ACM International Workshop on Security and Privacy Analytics, Scottsdale, Arizona, USA.
- Amos, B., Turner, H., & White, J. (2013, 1-5 July 2013). *Applying machine learning classifiers to dynamic Android malware detection at scale*. Paper presented at the 9th International Wireless Communications and Mobile Computing Conference (IWCMC).
- Androguard. (2017). from <https://github.com/androguard> on 30 October 2017
- Arp, D., Spreitzenbarth, M., Hübner, M., Gascon, H., Rieck, K., & Siemens, C. (2014). *Drebin: Effective and explainable detection of Android malware in your pocket*. Paper presented at the Annual Symposium on Network and Distributed System Security (NDSS).
- Baldi, P., Brunak, S., Chauvin, Y., Andersen, C. A., & Nielsen, H. (2000). Assessing the accuracy of prediction algorithms for classification: an overview. *Bioinformatics*, 16(5), 412–424.
- Bhatnagar, V., & Sharma, S. (2012). Data mining: a necessity for information security. *Journal of Knowledge Management Practice*, 13(1).
- Bishop, C. (2006). *Pattern recognition and machine learning*: Springer-Verlag New York.
- Bläsing, T., Batyuk, L., Schmidt, A.-D., Camtepe, S. A., & Albayrak, S. (2010). An Android application sandbox system for suspicious software detection. *MALWARE'10*, 55–62.
- Blowers, M., Fernandez, S., Froberg, B., Williams, J., Corbin, G., & Nelson, K. (2014). Data mining in cyber operations. In R. E. Pino, A. Kott & M. Shevenell (Eds.), *Cybersecurity Systems for Human Cognition Augmentation* (Vol. 61, pp. 61-73): Springer International Publishing.

- Breiman, L. (2001). Random forests. *Machine Learning*, 45(1), 5-32.
- Breiman, L., & Cutler, A. (2011). Random Forests. Retrieved April 2018, from https://www.stat.berkeley.edu/~breiman/RandomForests/cc_home.htm
- Burguera, I., Zurutuza, U., & Nadjm-Tehrani, S. (2011). *Crowdroid: Behavior based malware detection system for Android*. Paper presented at the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices, SPSM '11, New York, NY, USA.
- Cesare, S., & Xiang, Y. (2010). *Classification of malware using structured control flow*. Paper presented at the 8th Australasian Symposium on Parallel and Distributed Computing.
- CheetahMobile. (2014). 2014 Half Year Security Report. Retrieved June 2016, from <https://www.cmcm.com/blog/2014-07-18/186.html>.
- Chen, Y.-W., & Lin, C.-J. (2006). Combining SVMs with various feature selection strategies. In I. Guyon, M. Nikraves, S. Gunn & L. A. Zadeh (Eds.), *Feature Extraction: Foundations and Applications* (pp. 315-324). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Chin, E., Felt, A. P., Greenwood, K., & Wagner, D. (2011). *Analyzing inter-application communication in Android*. Paper presented at the 9th International Conference on Mobile Systems, Applications, and Services, MobiSys '11, New York, NY, USA.
- ContagioMobile. (2016). Retrieved June 2016, from <http://contagiominedump.blogspot.com/>.
- Coronado-De-Alba, L. D., Rodríguez-Mota, A., & Ambrosio, P. J. E. (2016, 15-17 Nov. 2016). *Feature selection and ensemble of classifiers for Android malware detection*. Paper presented at the 2016 8th IEEE Latin-American Conference on Communications (LATINCOM).
- Cortes, C., & Vapnik, V. (1995). Support-Vector networks. *Machine Learning, AT&T Bell Labs, Holmdel, NJ*, 273-297.
- CuckooSandbox. (2018). Retrieved June 2018, from <https://cuckoosandbox.org/>
- Cybenko, G. (1989). Approximation by superpositions of a sigmoidal function. *Mathematics of Control, Signals and Systems*, 2(4), 303-314. doi: 10.1007/bf02551274
- Damshenas, M., Dehghantanha, A., Choo, K.-K. R., & Mahmud, R. (2015). M0Droid: An Android behavioral-based malware detection model. *Journal of Information Privacy and Security*, 11(3), 141-157. doi: 10.1080/15536548.2015.1073510
- David, O. E., & Netanyahu, N. S. (2015, 12-17 July 2015). *DeepSign: Deep learning for automatic malware signature generation and classification*. Paper presented at the 2015 International Joint Conference on Neural Networks (IJCNN).
- Dhaya, R., & Poongodi, M. (2014, 8-10 May 2014). *Detecting software vulnerabilities in Android using static analysis*. Paper presented at the 2014 IEEE International Conference on Advanced Communications, Control and Computing Technologies.
- Dietterich, T. G. (2000). An experimental comparison of three methods for constructing ensembles of decision trees: bagging, boosting, and randomization. *Machine Learning*, 40(2), 139-157. doi: 10.1023/a:1007607513941
- Dini, G., Martinelli, F., Saracino, A., & Sgandurra, D. (2012). *Madam: A multi-level anomaly detector for Android malware*. Paper presented at the 6th International

- Conference on Mathematical Methods, Models and Architectures for Computer Network Security: Computer Network Security, MMM-ACNS'12.
- DoD. (2010). Joint Publication 1-02 Dictionary of Military and Associated Terms. Retrieved November 2010, from http://www.dtic.mil/doctrine/new_pubs/jp1_02.pdf.
- Egele, M., Scholte, T., Kirda, E., & Kruegel, C. (2008). A survey on automated dynamic malware-analysis techniques and tools. *ACM Computing Surveys*, 44(2), 1-42. doi: 10.1145/2089125.2089126
- Elish, K. O., Shu, X., Yao, D. D., Ryder, B., & Jiang, X. (2015). Profiling user-trigger dependence for Android malware detection. *Computers & Security*, 49(0), 255–273.
- Enck, W., Gilbert, P., Chun, B., Cox, L. P., Jung, J., McDaniel, P., & Sheth, A. N. *TaintDroid: an information-flow tracking system for realtime privacy monitoring on smartphones*. Paper presented at the 9th USENIX Conference on Operating Systems Design and Implementation, OSDI'10, Berkeley, CA, USA.
- Enck, W., Ongtang, M., & McDaniel, P. D. (2009). Understanding Android security. *IEEE Security & Privacy*, 7(1), 50–57.
- Felt, A. P., Ha, E., Egelman, S., Haney, A., Chin, E., & Wagner, D. (2012). *Android permissions: user attention, comprehension, and behavior*. Paper presented at the Proceedings of the Eighth Symposium on Usable Privacy and Security, Washington, D.C.
- Firdausi, I., Lim, C., Erwin, A., & Nugroho, A. S. (2010, 2-3 Dec. 2010). *Analysis of machine learning techniques used in behavior-based malware detection* Paper presented at the Second International Conference on Advances in Computing, Control and Telecommunication Technologies (ACT).
- Frank, E. (2014). Fully supervised training of Gaussian radial basis function networks in WEKA. *Department of Computer Science, University of Waikato*, 1-4.
- Freund, Y., & Schapire, R. E. (1996). *Experiments with a new boosting algorithm*. Paper presented at the Thirteenth International Conference on Machine Learning, San Francisco.
- Fuchs, A. P., Chaudhuri, A., & Foster, J. S. (2009). SCanDroid: automated security certification of Android applications *Technical Report CS-TR-4991, Department of Computer Science, University of Maryland, College Park*.
- Goodfellow, I., Bengio, Y., & Courville, A. (2016). *Deep Learning*: MIT Press.
- Han, J., Kamber, M., & Pei, J. (2006). *Data Mining: Concepts and Techniques* (2nd ed.): Morgan Kaufmann Publishers Inc.
- Hastie, T., Tibshirani, R., & Friedman, J. (2001). *The Elements of Statistical Learning*. New York, NY, USA: Springer New York Inc.
- Haykin, S. (2009). *Neural Networks and Learning Machines* (3rd ed.). New Jersey: Pearson Education Inc.
- Holte, R. C. (1993). Very simple classification rules perform well on most commonly used datasets. *Machine Learning*, 11, 63-91.
- Hühn, J., & Huellermeier, E. (2009). FURIA: an algorithm for unordered fuzzy rule induction *Data Mining and Knowledge Discovery*, 19(3), 293-319.
- John, G. H., & Langley, P. (1995). *Estimating continuous distributions in Bayesian classifiers*. Paper presented at the The Eleventh Conference on Uncertainty in Artificial Intelligence.
- Karbab, E. B., Debbabi, M., Derhab, A., & Mouheb, D. (2018). MalDozer: Automatic framework for Android malware detection using deep learning. *Digital*

- Investigation*, 24, S48-S59. doi: <https://doi.org/10.1016/j.diin.2018.01.007>
- Kolter, J. Z., & Maloof, M. A. (2004). *Learning to detect malicious executables in the wild*. Paper presented at the Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining Seattle, WA, USA: ACM, Seattle, WA, USA.
- Lantz, P., Desnos, A., & Yang, K. Droidbox: Android application sandbox. *available at: <https://code.google.com/p/droidbox/>*.
- Lin, C.-T., Wang, N.-J., Xiao, H., & Eckert, C. (2015). Feature selection and extraction for malware classification. *Journal of Information Science and Engineering*, 31(3), 965-992.
- Lindorfer, M., Neugschwandtner, M., & Platzer, C. (2015). *MARVIN: Efficient and comprehensive mobile app classification through static and dynamic analysis*. Paper presented at the IEEE 39th Annual Computer Software and Applications Conference.
- Mairhuber, J. C. (1956). On Haar's theorem concerning Chebychev approximation problems having unique solutions. *Proceedings of the American Mathematical Society*, 7(4), 609-615.
- Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E. D., Ross, G., & Stringhini, G. (2017). MaMaDroid: Detecting Android malware by building markov chains of behavioral models. *arXiv:1612.04433*.
- Martínez-Muñoz, G., & Suárez, A. (2007). Using boosting to prune bagging ensembles. *Pattern Recognition Letters*, 28(1), 156-165.
- Milosevic, N. (2018). OWASP Seraphimdroid GitHub page. Retrieved June 2018, from <https://github.com/nikolamilosevic86/owasp-seraphimdroid>
- Milosevic, N., Dehghantaha, A., & Choo, K.-K. R. (2017). Machine learning aided Android malware classification. *Computers & Electrical Engineering*, 61, 266-274. doi: <https://doi.org/10.1016/j.compeleceng.2017.02.013>
- Monkey. (2018). Retrieved June 2018, from <https://developer.android.com/studio/test/monkey>
- Moskovitch, R., Feher, C., Tzachar, N., Berger, E., Gitelman, M., Dolev, S., & Elovici, Y. (2008). Unknown malcode detection using opcode representation. In D. Ortiz-Arroyo, H. L. Larsen, D. D. Zeng, D. Hicks & G. Wagner (Eds.), *Intelligence and Security Informatics: First European Conference, EuroISI 2008, Esbjerg, Denmark, December 3-5, 2008. Proceedings* (pp. 204-215). Berlin, Heidelberg: Springer Berlin Heidelberg.
- Onwuzurike, L., Almeida, M., Mariconti, E., Blackburn, J., Stringhini, G., & Cristofaro, E. D. (2018). A family of droids: Analyzing behavioral model based Android malware detection via static and dynamic analysis. *arXiv:1803.03448*.
- Patel, N., & Upadhyay, S. (2012). Study of various decision tree pruning methods with their empirical comparison in WEKA. *International Journal of Computer Applications*, 60 (12), 20-25.
- Peiravian, N., & Zhu, X. (2013, 4-6 Nov. 2013). *Machine learning for Android malware detection using permission and API calls*. Paper presented at the 2013 IEEE 25th International Conference on Tools with Artificial Intelligence.
- Pektaş, A., & Acarman, T. (2018). Malware classification based on API calls and behaviour analysis. *IET Information Security*, 12(2), 107-117. doi: 10.1049/iet-ifs.2017.0430
- Pektaş, A., Çavdar, M., & Acarman, T. (2016). *Android malware classification by applying online machine learning*. Paper presented at the International Symposium on Computer and Information Sciences (ISCIS).

- Peng, H., Gates, C., Sarma, B., Li, N., Qi, Y., Potharaju, R., Rotaru, C. N., Molloy, I. (2012). Using probabilistic generative models for ranking risks of Android apps. Paper presented at the 2012 ACM Conference on Computer and Communications Security (CCS), New York, USA.
- Platt, J. (1998). *Advances in Kernel Methods - Support Vector Learning*. USA: MIT Press.
- Polikar, R. (2006). Ensemble based systems in decision making. *IEEE Circuits and Systems Magazine*, 6(3), 21–45.
- Provos, N., Mavrommatis, P., Rajab, M. A., & Monrose, F. (2008). *All your iFRAMES point to us*. Paper presented at the 17th conference on Security symposium, San Jose, CA.
- Quinlan, J. R. (1986). Induction of Decision Trees. *Machine Learning*, 1(1), 81-106.
- Quinlan, J. R. (1993). *C4.5: Programs for machine learning*: Morgan Kaufmann.
- RapidMiner. (2017). from <https://rapidminer.com/products/studio/>
- Roundy, K., & Miller, B. (2010). Hybrid analysis and control of malware. In S. Jha, R. Sommer & C. Kreibich (Eds.), *Recent Advances in Intrusion Detection* (Vol. 6307, pp. 317-338): Springer Berlin Heidelberg.
- Rumelhart, D. E., Hinton, G. E., & Williams, R. J. (1986). Learning internal representations by error propagation. In E. R. David, L. M. James & C. P. R. Group (Eds.), *Parallel distributed processing: Explorations in the microstructure of cognition, vol. 1* (pp. 318-362): MIT Press.
- Saha, S., Pal, M., & Konar, A. (2016). *Ensemble classifier approach to gesture recognition in health care using a Kinect sensor*. FL, Florida, USA: Taylor & Francis Group.
- Salton, G., Wong, A., & Yang, C. S. (1975). A vector space model for automatic indexing. *Communications of the ACM*, 18, 613-620.
- Salzberg, S. (1991). A nearest hyperrectangle learning method. *Machine Learning*, 6(3), 251–276.
- SantokuLinux. (2018). Retrieved April 2018, from <https://santoku-linux.com/>
- Sanz, B., Santos, I., Nieves, J., Laorden, C., Alonso-Gonzalez, I., & Bringas, P. G. (2013). *MADS: Malicious Android applications detection through string analysis*. Berlin Heidelberg: Springer
- Sayfullina, L., Eirola, E., Komashinsky, D., Palumbo, P., Miche, Y., Lendasse, A., & Karhunen, J. (2015, 20-22 Aug. 2015). *Efficient detection of zero-day Android malware using normalized Bernoulli Naive Bayes*. Paper presented at the 2015 IEEE Trustcom/BigDataSE/ISPA.
- Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Camtepe, S. A., & Albayrak, S. (2009). *Static analysis of executables for collaborative malware detection on Android*. Paper presented at the Proceedings of the 2009 IEEE international conference on Communications, Dresden, Germany.
- Schmidt, A. D., Schmidt, H. G., Clausen, J., Yuksel, K. A., Kiraz, O., Camtepe, A., & Albayrak, S. (2008). Enhancing security of Linux-based Android devices. *Proceedings of the 15th International Linux Kongress, Lehmann*.
- Schultz, M. G., Eskin, E., Zadok, E., & Stolfo, S. J. (2001). *Data mining methods for detection of new malicious executables*. Paper presented at the IEEE Symposium on Security and Privacy, Oakland, CA, USA.
- Shahzad, F., Akbar, M., Khan, S., & Farooq, M. (2013). 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*.

- Stefanowski, J. (1998). *The rough set based rule induction technique for classification problems*. Paper presented at the The 6th European Congress on Intelligent Techniques and Soft Computing, Aachen, Germany.
- stuff.mit.edu. (2016). *Building and Running | Android Developers*. Retrieved March 2016, from <https://stuff.mit.edu/afs/sipb/project/android/docs/tools/building/index.html>.
- Suarez-Tangil, G., & Stringhini, G. (2018). Eight years of rider measurement in the Android malware ecosystem: Evolution and lessons learned.
- Suarez-Tangil, G., Tapiador, J. E., Peris-Lopez, P., & Blasco, J. (2014). Dendroid: A text mining approach to analyzing and classifying code structures in Android malware families. *Expert Systems with Applications*, 41(4, Part 1), 1104-1117. doi: <https://doi.org/10.1016/j.eswa.2013.07.106>
- Tin Kam, H. (1995, 14-16 Aug 1995). *Random decision forests*. Paper presented at the 3rd International Conference on Document Analysis and Recognition.
- Van Der Malsburg, C. (1986). *Frank Rosenblatt: Principles of Neurodynamics: Perceptrons and the Theory of Brain Mechanisms*, Berlin, Heidelberg.
- Vapnik, V. (1995). *The Nature of Statistical Learning Theory*. New York - USA: Springer-Verlag.
- Vapnik, V., & Lerner, A. (1963). Pattern recognition using generalized portrait method. *Automation and Remote Control*, 24(6), 774-780.
- Veloso, A., Jr., W. M., & Zaki, M. (2006). *Lazy Associative Classification*. Paper presented at the The Sixth International Conference on Data Mining (ICDM), Washington, DC, USA.
- Villars, R. L., Olofson, C. W., & Eastwood, M. (2011). Big data: What it is and why you should care. *White Paper, IDC*
- VirusShare. (2018). Retrieved June 2018, from <https://virusshare.com/>
- Wang, X., Zhang, D., Su, X., & Li, W. (2017). Mlifdetect: Android malware detection based on parallel machine learning and information fusion *Security and Communication Networks*, 2017. doi: 10.1155/2017/6451260
- Wegman, E. J. (2002). Visual Data Mining. *Statistics in Medicine*, 22, 1383-1397.
- Weka. (2018). Weka 3: Data Mining Software in Java. Machine Learning Group at the University of Waikato. Retrieved April 2018, from <http://www.cs.waikato.ac.nz/ml/weka/>
- Witten, I. H., & Frank, E. (2005). *Data Mining: Practical Machine Learning Tools and Techniques*, Second Edition: Morgan Kaufmann Publishers Inc.
- Wu, D., Mao, C., Wei, T., Lee, H., & Wu, K. (2012). DroidMat: Android malware detection through Manifest and API calls tracing. *Seventh Asia Joint Conference on Information Security (Asia JCIS)*, 62-69.
- Wu, S., Wang, P., Li, X., & Zhang, Y. (2016). Effective detection of Android malware based on the usage of data flow APIs and machine learning. *Information and Software Technology*, 75, 17-25.
- Wu, W.-C., & Hung, S.-H. (2014). *DroidDolphin: A dynamic Android malware detection framework using big data and machine learning*. Paper presented at the 2014 Conference on Research in Adaptive and Convergent Systems, Towson, Maryland.
- Yan, L. K., & Yin, H. (2012). *DroidScope: Seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis*. Paper presented at the 21st USENIX Security Symposium, Bellevue, WA. <https://www.usenix.org/conference/usenixsecurity12/technical-sessions/presentation/yan>

- Yerima, S. Y., Sezer, S., McWilliams, G., & Muttik, I. (2013). *A new Android malware detection approach using Bayesian classification*. Paper presented at the IEEE 27th International Conference Advanced Information Networking and Applications (AINA).
- Yerima, S. Y., Sezer, S., & Muttik, I. (2014). *Android malware detection using parallel machine learning classifiers*. Paper presented at the 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies.
- Yerima, S. Y., Sezer, S., & Muttik, I. (2015). High accuracy Android malware detection using ensemble learning. *IET Information Security*, 9(6), 313-320. doi: 10.1049/iet-ifs.2014.0099
- ZDNet. (2008). How Android works: The big picture. Retrieved September 2016, from <http://www.zdnet.com/article/how-android-works-the-big-picture/>
- Zheng, C., Zhu, S., Dai, S., Gu, G., Gong, X., Han, X., & Zou, W. (2012). *Smartdroid: An automatic system for revealing UI-based trigger conditions in Android applications*. Paper presented at the Second ACM Workshop on Security and Privacy in Smartphones and Mobile Devices.
- Zhou, Y., & Jiang, X. (2012). *Dissecting Android malware: characterization and evolution*. Paper presented at the 33rd IEEE Symposium on Security and Privacy, Oakland, San Francisco, CA.

APPENDIX 1 – Descriptions of the Selected Attributes for the MalGenome Dataset

Permissions

BROADCAST_STICKY: This is a permission to be able to send a sticky broadcast. A sticky broadcast is a broadcast that stays around following the moment it is announced to the system. Most broadcasts are sent, processed within the system and become quickly inaccessible. However, sticky broadcasts announce information that remains accessible beyond the point at which they are processed. A typical example is the battery level broadcast. Unlike most broadcasts, the battery level can be retrieved within applications beyond the point at which it was sent through the system. This means that apps can find out whatever the last battery level broadcast was.

CALL_PHONE: This is the permission that enables making phone calls.

CAMERA: This is the permission to be able to utilize the device's camera given it has one.

DISABLE_KEYGUARD: Allows an app to disable the keylock in addition to any associated password security. For example, upon receiving an incoming phone call, the phone disables the keylock, and re-enables the keylock when the call is finished.

GET_ACCOUNTS: Enables an app to get the list of accounts known by the phone. This may include any accounts created by the apps installed by the phone owner.

READ_CALENDAR: Grants read access to the phone's calendar.

READ_LOGS: Grants read access to the phone's logs.

READ_PHONE_STATE: Required to obtain the UUID (Universally unique identifier) of the device.

READ_SMS: Grants read access to the received, drafted and sent SMS messages.

RECEIVE_SMS: Enables an app to intercept an incoming SMS. In other words, an SMS incoming to the device may also be received by the app possessing this permission.

RESTART_PACKAGES: Required to restart a running package.

SEND_SMS: Enables an app to send SMS messages.

SYSTEM_ALERT_WINDOW: Allows an app to display content over another app. In other words, this permission allows a developer to display content on the screen of your Android device after a predefined trigger event.

WRITE_APN_SETTINGS: Allows applications to write the Access Point Name (APN) settings. APN settings are required to set up a connection to the gateway between your carrier's cellular network and the public Internet.

WRITE_CONTACTS: Allows an app to write to the device's contacts data.

WRITE_EXTERNAL_STORAGE: Allows an app to write to the external storages (such as a MicroSD Card or USB Flash) connected to the device.

WRITE_HISTORY_BOOKMARKS: Allows an app to modify the browser's history or bookmarks stored on a phone. This may allow the app to erase or modify browser data, additionally providing access to the history of websites visited using the browser apps on the device.

WRITE_SMS: Allows an app to write to SMS messages stored on a phone or SIM card. Malicious apps may delete sensitive SMS messages using this permission.

API calls

getSubscriberId: Returns the unique subscriber ID, such as the IMSI (International Mobile Subscriber Identity) for a GSM (Global System for Mobile communication) phone.

sendTextMessage: Must be called by applications that send SMS text messages.

createFromPdu: Used for constructing an SMS text message using a raw PDU (Protocol Description Unit) with the specified message format.

getSimSerialNumber: Returns the serial number of the SIM, if applicable. Returns null if it is unavailable.

getExtraInfo: Report extra information concerning the network state, given it was provided by the lower networking layers.

getPaint: Returns the paint required to draw the shape.

seekTo: Implemented to adjust the playback position of an audio/video in the media player.

TruncateAt: Required to truncate an input text after the specified position.

Strings

Parse: This string is indicative of some sort of parsing code in the Android executable.

Add: This string indicates an add operation by the corresponding sample.

Iterator: Presence of this string indicates use of an iterator method.

Schedule: This string indicates some sort of scheduling of program execution was carried out.

setId: Indicative of use of the setId method.

Digest: This string is indicative of the use of the “digest” package in Java required to generate an MD5 digest for a string.

Callback: Indicative of the use of the “callback” method in Java that gets called when an event occurs.

entrySet: Indicative of the use of the “entrySet” method in Java used for iterating over an object that implements the map interface.

findPointerIndex: Indicates the use of the findPointerIndex(int) method that is required for obtaining the pointer index for a given pointer id in the corresponding motion event (mouse, pen, finger and trackball movements).

getEdgeFlags: Indicates use of the getEdgeFlags() method that returns a bit field indicating which edges, if any, were touched by the corresponding motion event.

getFinalX: Indicates use of the getFinalX() method that returns where the scroll will end in the Android Scroller class that extends Object.

APPENDIX 2 – Descriptions of the Selected Attributes for the AndroZoo Dataset

Permissions

REORDER_TASKS: Enables the app to move tasks to the foreground and background. The app may also do this without any user input.

CLEAR_APP_CACHE: Required for deleting files in the cache directories of other applications, as a side effect other applications may start up more slowly due to having to re-retrieve their data.

MANAGE_DOCUMENTS: Allows an app to manage the storage for documents.

READ_HISTORY_BOOKMARKS: Enables an app to read the history of all URLs that the Browser has visited, and all of the Browser's bookmarks.

KILL_BACKGROUND_PROCESSES: Gives an app the permission to end background processes of other apps, effectively causing other apps to stop running.

ACCESS_MOCK_LOCATION: This permission allows an app to override the location or status returned by other location sources such as GPS or other positional information providers.

ACCESS_GPS: Enables an app to utilize the GPS functions of the device.

ACCESS_LOCATION_EXTRA_COMMANDS: Allows an app to access extra location provider commands. This may allow the app to interfere with the operation of the GPS or other positional information providing sources.

API calls

getAttributeUnsignedIntValue: Returns the Boolean value of 'attribute' that is formatted as an unsigned value.

setKeywords: A method in the Android API used for data passing.

setAppName: A method in the Android API to change the Android app's name.

getCurrencyCode: A method in the Android API to get the code of a currency specific to a country.

isSurrogatePair: Determines whether the specified pair of char values is a valid Unicode surrogate pair.

allowCoreThreadTimeOut: A method of the ThreadPoolExecutor public class, sets the policy governing whether core threads may time out and terminate if no tasks arrive within the keep-alive time, become replaced if required when new tasks arrive.

requestAd: Is a method to request advertisements from a server.

displayAd: Is a method required for displaying advertisements.

FilenameFilter: Presence of this string is indicative of implementation of the “FilenameFilter” public interface that is used to filter file names. The classes that implement this interface can filter directory listings in the “list” method of the File class.

Strings

remoteexception: This string indicates a remote method invoked by the app on another service did not complete and an exception was thrown accordingly.

Credentials: Indicates the use of the credentials class which is a class for representing UNIX credentials passed via ancillary data on UNIX domain sockets.

remoteinput: Indicates utilization of the RemoteInput object that specifies input to be collected from a user to be passed along with an intent.

doubleclick: Presence of this string indicates a possible <http://ad.doubleclick.net> redirecting piece of code, which is a common malicious behavior employed by several Android malware families.

APPENDIX 3 – Detailed Results Obtained in WEKA for the First MalGenome Dataset

Radial Basis Function

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,004	0,996	0,972	0,984	0,968	0,991	0,994	(m)
	0,996	0,028	0,973	0,996	0,984	0,968	0,991	0,987	(b)
Weighted Avg.	0,984	0,016	0,984	0,984	0,984	0,968	0,991	0,991	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

1 249 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,012	0,988	0,972	0,980	0,960	0,989	0,992	(m)
	0,988	0,028	0,972	0,988	0,980	0,960	0,989	0,983	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,989	0,988	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

3 247 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,968	0,000	1,000	0,968	0,984	0,968	1,000	1,000	(m)
	1,000	0,032	0,969	1,000	0,984	0,968	1,000	1,000	(b)
Weighted Avg.	0,984	0,016	0,984	0,984	0,984	0,968	1,000	1,000	

=== Confusion Matrix ===

a b <-- classified as

121 4 | a = (m)

0 125 | b = (b)

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,912	0,000	1,000	0,912	0,954	0,916	1,000	1,000	(m)
	1,000	0,088	0,919	1,000	0,958	0,916	1,000	1,000	(b)
Weighted Avg.	0,956	0,044	0,960	0,956	0,956	0,916	1,000	1,000	

=== Confusion Matrix ===

a b <-- classified as

114 11 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,912	0,000	1,000	0,912	0,954	0,916	0,999	0,999	(m)
	1,000	0,088	0,919	1,000	0,958	0,916	0,999	0,999	(b)
Weighted Avg.	0,956	0,044	0,960	0,956	0,956	0,916	0,999	0,999	

==== Confusion Matrix ====

a b <-- classified as
114 11 | a = (m)
0 125 | b = (b)

Test Set 4

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,952	0,000	1,000	0,952	0,975	0,953	0,999	0,999	(m)
	1,000	0,048	0,954	1,000	0,977	0,953	0,999	0,999	(b)
Weighted Avg.	0,976	0,024	0,977	0,976	0,976	0,953	0,999	0,999	

==== Confusion Matrix ====

a b <-- classified as
119 6 | a = (m)
0 125 | b = (b)

Multilayer Perceptron

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,000	1,000	0,984	0,992	0,984	0,997	0,997	(m)
	1,000	0,016	0,984	1,000	0,992	0,984	0,997	0,996	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,984	0,997	0,997	

=== Confusion Matrix ===

a b <-- classified as

246 4 | a = (m)

0 250 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,976	0,020	0,980	0,976	0,978	0,956	0,982	0,988	(m)
	0,980	0,024	0,976	0,980	0,978	0,956	0,982	0,969	(b)
Weighted Avg.	0,978	0,022	0,978	0,978	0,978	0,956	0,982	0,979	

=== Confusion Matrix ===

a b <-- classified as

244 6 | a = (m)

5 245 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,944	0,000	1,000	0,944	0,971	0,945	0,995	0,996	(m)
	1,000	0,056	0,947	1,000	0,973	0,945	0,995	0,994	(b)
Weighted Avg.	0,972	0,028	0,973	0,972	0,972	0,945	0,995	0,995	

=== Confusion Matrix ===

```
a b <-- classified as  
118 7 | a = (m)  
0 125 | b = (b)
```

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,974	0,982	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,974	0,966	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,974	0,974	

=== Confusion Matrix ===

```
a b <-- classified as  
115 10 | a = (m)  
0 125 | b = (b)
```

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,880	0,000	1,000	0,880	0,936	0,886	0,981	0,986	(m)
	1,000	0,120	0,893	1,000	0,943	0,886	0,981	0,978	(b)
Weighted Avg.	0,940	0,060	0,946	0,940	0,940	0,886	0,981	0,982	

=== Confusion Matrix ===

a b <-- classified as

110 15 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,928	0,000	1,000	0,928	0,963	0,930	0,987	0,991	(m)
	1,000	0,072	0,933	1,000	0,965	0,930	0,987	0,981	(b)
Weighted Avg.	0,964	0,036	0,966	0,964	0,964	0,930	0,987	0,986	

=== Confusion Matrix ===

a b <-- classified as

116 9 | a = (m)

0 125 | b = (b)

Id3

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,996	0,000	1,000	0,996	0,998	0,996	1,000	1,000	(m)
	1,000	0,004	0,996	1,000	0,998	0,996	1,000	1,000	(b)
Weighted Avg.	0,998	0,002	0,998	0,998	0,998	0,996	1,000	1,000	

=== Confusion Matrix ===

```
a b <-- classified as
249 1 | a = (m)
0 250 | b = (b)
```

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,968	0,028	0,972	0,968	0,970	0,940	0,969	0,957	(m)
	0,972	0,032	0,968	0,972	0,970	0,940	0,969	0,954	(b)
Weighted Avg.	0,970	0,030	0,970	0,970	0,970	0,940	0,969	0,955	

=== Confusion Matrix ===

```
a b <-- classified as
242 8 | a = (m)
7 243 | b = (b)
```

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,896	0,000	1,000	0,896	0,945	0,901	0,946	0,948	(m)
	1,000	0,104	0,906	1,000	0,951	0,901	0,946	0,902	(b)
Weighted Avg.	0,948	0,052	0,953	0,948	0,948	0,901	0,946	0,925	

==== Confusion Matrix ====

a b <-- classified as

112 13 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,958	0,960	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,958	0,922	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,958	0,941	

==== Confusion Matrix ====

a b <-- classified as

115 10 | a = (m)

0 125 | b = (b)

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,936	0,000	1,000	0,936	0,967	0,938	0,967	0,968	(m)
	1,000	0,064	0,940	1,000	0,969	0,938	0,967	0,938	(b)
Weighted Avg.	0,968	0,032	0,970	0,968	0,968	0,938	0,967	0,953	

=== Confusion Matrix ===

a b <-- classified as

117 8 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,928	0,000	1,000	0,928	0,963	0,930	0,963	0,964	(m)
	1,000	0,072	0,933	1,000	0,965	0,930	0,963	0,931	(b)
Weighted Avg.	0,964	0,036	0,966	0,964	0,964	0,930	0,963	0,947	

=== Confusion Matrix ===

a b <-- classified as

116 9 | a = (m)

0 125 | b = (b)

k-NN

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,996	0,000	1,000	0,996	0,998	0,996	1,000	1,000	(m)
	1,000	0,004	0,996	1,000	0,998	0,996	1,000	1,000	(b)
Weighted Avg.	0,998	0,002	0,998	0,998	0,998	0,996	1,000	1,000	

=== Confusion Matrix ===

a b <-- classified as

249 1 | a = (m)

0 250 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,964	0,012	0,988	0,964	0,976	0,952	0,980	0,985	(m)
	0,988	0,036	0,965	0,988	0,976	0,952	0,980	0,966	(b)
Weighted Avg.	0,976	0,024	0,976	0,976	0,976	0,952	0,980	0,975	

=== Confusion Matrix ===

a b <-- classified as

241 9 | a = (m)

3 247 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,904	0,000	1,000	0,904	0,950	0,908	0,976	0,977	(m)
	1,000	0,096	0,912	1,000	0,954	0,908	0,976	0,961	(b)
Weighted Avg.	0,952	0,048	0,956	0,952	0,952	0,908	0,976	0,969	

==== Confusion Matrix ====

a b <-- classified as

113 12 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,880	0,000	1,000	0,880	0,936	0,886	0,961	0,967	(m)
	1,000	0,120	0,893	1,000	0,943	0,886	0,961	0,942	(b)
Weighted Avg.	0,940	0,060	0,946	0,940	0,940	0,886	0,961	0,955	

==== Confusion Matrix ====

a b <-- classified as

110 15 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,864	0,000	1,000	0,864	0,927	0,872	0,954	0,963	(m)
	1,000	0,136	0,880	1,000	0,936	0,872	0,954	0,934	(b)
Weighted Avg.	0,932	0,068	0,940	0,932	0,932	0,872	0,954	0,948	

=== Confusion Matrix ===

a b <-- classified as

108 17 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,912	0,000	1,000	0,912	0,954	0,916	0,964	0,968	(m)
	1,000	0,088	0,919	1,000	0,958	0,916	0,964	0,946	(b)
Weighted Avg.	0,956	0,044	0,960	0,956	0,956	0,916	0,964	0,957	

=== Confusion Matrix ===

a b <-- classified as

114 11 | a = (m)

0 125 | b = (b)

NaiveBayes

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,048	0,953	0,972	0,962	0,924	0,988	0,992	(m)
	0,952	0,028	0,971	0,952	0,962	0,924	0,988	0,982	(b)
Weighted Avg.	0,962	0,038	0,962	0,962	0,962	0,924	0,988	0,987	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

12 238 | b = (b)

10-fold x-val

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,048	0,953	0,972	0,962	0,924	0,985	0,990	(m)
	0,952	0,028	0,971	0,952	0,962	0,924	0,985	0,971	(b)
Weighted Avg.	0,962	0,038	0,962	0,962	0,962	0,924	0,985	0,981	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

12 238 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,016	0,984	0,984	0,984	0,968	0,999	0,999	(m)
	0,984	0,016	0,984	0,984	0,984	0,968	0,999	0,999	(b)
Weighted Avg.	0,984	0,016	0,984	0,984	0,984	0,968	0,999	0,999	

=== Confusion Matrix ===

a b <-- classified as

123 2 | a = (m)

2 123 | b = (b)

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,936	0,024	0,975	0,936	0,955	0,913	0,997	0,997	(m)
	0,976	0,064	0,938	0,976	0,957	0,913	0,997	0,997	(b)
Weighted Avg.	0,956	0,044	0,957	0,956	0,956	0,913	0,997	0,997	

=== Confusion Matrix ===

a b <-- classified as

117 8 | a = (m)

3 122 | b = (b)

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,952	0,064	0,937	0,952	0,944	0,888	0,980	0,977	(m)
	0,936	0,048	0,951	0,936	0,944	0,888	0,980	0,985	(b)
Weighted Avg.	0,944	0,056	0,944	0,944	0,944	0,888	0,980	0,981	

==== Confusion Matrix ====

```
a b <-- classified as
```

```
119 6 | a = (m)
```

```
8 117 | b = (b)
```

Test Set 4

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,064	0,939	0,984	0,961	0,921	0,986	0,983	(m)
	0,936	0,016	0,983	0,936	0,959	0,921	0,986	0,989	(b)
Weighted Avg.	0,960	0,040	0,961	0,960	0,960	0,921	0,986	0,986	

==== Confusion Matrix ====

```
a b <-- classified as
```

```
123 2 | a = (m)
```

```
8 117 | b = (b)
```

J48

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,000	1,000	0,972	0,986	0,972	0,987	0,988	(m)
	1,000	0,028	0,973	1,000	0,986	0,972	0,987	0,975	(b)
Weighted Avg.	0,986	0,014	0,986	0,986	0,986	0,972	0,987	0,982	

=== Confusion Matrix ===

```
a b <-- classified as
243 7 | a = (m)
0 250 | b = (b)
```

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,964	0,020	0,980	0,964	0,972	0,944	0,965	0,971	(m)
	0,980	0,036	0,965	0,980	0,972	0,944	0,965	0,927	(b)
Weighted Avg.	0,972	0,028	0,972	0,972	0,972	0,944	0,965	0,949	

=== Confusion Matrix ===

```
a b <-- classified as
241 9 | a = (m)
5 245 | b = (b)
```

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,880	0,000	1,000	0,880	0,936	0,886	0,919	0,959	(m)
	1,000	0,120	0,893	1,000	0,943	0,886	0,919	0,923	(b)
Weighted Avg.	0,940	0,060	0,946	0,940	0,940	0,886	0,919	0,941	

==== Confusion Matrix ====

a b <-- classified as

110 15 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,967	0,982	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,967	0,967	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,967	0,975	

==== Confusion Matrix ====

a b <-- classified as

115 10 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,928	0,000	1,000	0,928	0,963	0,930	0,944	0,971	(m)
	1,000	0,072	0,933	1,000	0,965	0,930	0,944	0,927	(b)
Weighted Avg.	0,964	0,036	0,966	0,964	0,964	0,930	0,944	0,949	

=== Confusion Matrix ===

a b <-- classified as

116 9 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,959	0,978	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,959	0,943	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,959	0,960	

=== Confusion Matrix ===

a b <-- classified as

115 10 | a = (m)

0 125 | b = (b)

SVM

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,976	0,004	0,996	0,976	0,986	0,972	0,986	0,984	(m)
	0,996	0,024	0,976	0,996	0,986	0,972	0,986	0,975	(b)
Weighted Avg.	0,986	0,014	0,986	0,986	0,986	0,972	0,986	0,979	

=== Confusion Matrix ===

a b <-- classified as

244 6 | a = (m)

1 249 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,968	0,012	0,988	0,968	0,978	0,956	0,978	0,972	(m)
	0,988	0,032	0,969	0,988	0,978	0,956	0,978	0,963	(b)
Weighted Avg.	0,978	0,022	0,978	0,978	0,978	0,956	0,978	0,968	

=== Confusion Matrix ===

a b <-- classified as

242 8 | a = (m)

3 247 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,944	0,000	1,000	0,944	0,971	0,945	0,972	0,972	(m)
	1,000	0,056	0,947	1,000	0,973	0,945	0,972	0,947	(b)
Weighted Avg.	0,972	0,028	0,973	0,972	0,972	0,945	0,972	0,959	

=== Confusion Matrix ===

a b <-- classified as

118 7 | a = (m)

0 125 | b = (b)

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,912	0,000	1,000	0,912	0,954	0,916	0,956	0,956	(m)
	1,000	0,088	0,919	1,000	0,958	0,916	0,956	0,919	(b)
Weighted Avg.	0,956	0,044	0,960	0,956	0,956	0,916	0,956	0,938	

=== Confusion Matrix ===

a b <-- classified as

114 11 | a = (m)

0 125 | b = (b)

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,904	0,000	1,000	0,904	0,950	0,908	0,952	0,952	(m)
	1,000	0,096	0,912	1,000	0,954	0,908	0,952	0,912	(b)
Weighted Avg.	0,952	0,048	0,956	0,952	0,952	0,908	0,952	0,932	

==== Confusion Matrix ====

a b <-- classified as

113 12 | a = (m)

0 125 | b = (b)

Test Set 4

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,944	0,000	1,000	0,944	0,971	0,945	0,972	0,972	(m)
	1,000	0,056	0,947	1,000	0,973	0,945	0,972	0,947	(b)
Weighted Avg.	0,972	0,028	0,973	0,972	0,972	0,945	0,972	0,959	

==== Confusion Matrix ====

a b <-- classified as

118 7 | a = (m)

0 125 | b = (b)

FURIA

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,004	0,996	0,972	0,984	0,968	0,986	0,984	(m)
	0,996	0,028	0,973	0,996	0,984	0,968	0,986	0,975	(b)
Weighted Avg.	0,984	0,016	0,984	0,984	0,984	0,968	0,986	0,979	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

1 249 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,004	0,996	0,972	0,984	0,968	0,986	0,984	(m)
	0,996	0,028	0,973	0,996	0,984	0,968	0,986	0,974	(b)
Weighted Avg.	0,984	0,016	0,984	0,984	0,984	0,968	0,986	0,979	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

1 249 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,880	0,000	1,000	0,880	0,936	0,886	0,940	0,940	(m)
	1,000	0,120	0,893	1,000	0,943	0,886	0,940	0,893	(b)
Weighted Avg.	0,940	0,060	0,946	0,940	0,940	0,886	0,940	0,916	

==== Confusion Matrix ====

```
a b <-- classified as
110 15 | a = (m)
0 125 | b = (b)
```

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,960	0,960	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,960	0,926	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,960	0,943	

==== Confusion Matrix ====

```
a b <-- classified as
115 10 | a = (m)
0 125 | b = (b)
```

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,960	0,960	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,960	0,926	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,960	0,943	

=== Confusion Matrix ===

a b <-- classified as

115 10 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,928	0,000	1,000	0,928	0,963	0,930	0,964	0,964	(m)
	1,000	0,072	0,933	1,000	0,965	0,930	0,964	0,933	(b)
Weighted Avg.	0,964	0,036	0,966	0,964	0,964	0,930	0,964	0,948	

=== Confusion Matrix ===

a b <-- classified as

116 9 | a = (m)

0 125 | b = (b)

RandomForest

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,996	0,000	1,000	0,996	0,998	0,996	1,000	1,000	(m)
	1,000	0,004	0,996	1,000	0,998	0,996	1,000	1,000	(b)
Weighted Avg.	0,998	0,002	0,998	0,998	0,998	0,996	1,000	1,000	

=== Confusion Matrix ===

```
a b <-- classified as
249 1 | a = (m)
0 250 | b = (b)
```

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,012	0,988	0,972	0,980	0,960	0,987	0,991	(m)
	0,988	0,028	0,972	0,988	0,980	0,960	0,987	0,976	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,987	0,983	

=== Confusion Matrix ===

```
a b <-- classified as
243 7 | a = (m)
3 247 | b = (b)
```

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	1,000	1,000	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	1,000	1,000	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	1,000	1,000	

=== Confusion Matrix ===

a b <-- classified as

115 10 | a = (m)

0 125 | b = (b)

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,896	0,000	1,000	0,896	0,945	0,901	1,000	0,999	(m)
	1,000	0,104	0,906	1,000	0,951	0,901	1,000	0,999	(b)
Weighted Avg.	0,948	0,052	0,953	0,948	0,948	0,901	1,000	0,999	

=== Confusion Matrix ===

a b <-- classified as

112 13 | a = (m)

0 125 | b = (b)

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,880	0,000	1,000	0,880	0,936	0,886	1,000	1,000	(m)
	1,000	0,120	0,893	1,000	0,943	0,886	1,000	1,000	(b)
Weighted Avg.	0,940	0,060	0,946	0,940	0,940	0,886	1,000	1,000	

=== Confusion Matrix ===

a b <-- classified as

110 15 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,928	0,000	1,000	0,928	0,963	0,930	0,999	0,999	(m)
	1,000	0,072	0,933	1,000	0,965	0,930	0,999	0,998	(b)
Weighted Avg.	0,964	0,036	0,966	0,964	0,964	0,930	0,999	0,998	

=== Confusion Matrix ===

a b <-- classified as

116 9 | a = (m)

0 125 | b = (b)

NNGE / AdaBoostM1

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,996	0,000	1,000	0,996	0,998	0,996	1,000	1,000	(m)
	1,000	0,004	0,996	1,000	0,998	0,996	1,000	1,000	(b)
Weighted Avg.	0,998	0,002	0,998	0,998	0,998	0,996	1,000	1,000	

=== Confusion Matrix ===

a b <-- classified as

249 1 | a = (m)

0 250 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,968	0,020	0,980	0,968	0,974	0,948	0,981	0,988	(m)
	0,980	0,032	0,968	0,980	0,974	0,948	0,981	0,955	(b)
Weighted Avg.	0,974	0,026	0,974	0,974	0,974	0,948	0,981	0,971	

=== Confusion Matrix ===

a b <-- classified as

242 8 | a = (m)

5 245 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,944	0,000	1,000	0,944	0,971	0,945	0,996	0,996	(m)
	1,000	0,056	0,947	1,000	0,973	0,945	0,996	0,996	(b)
Weighted Avg.	0,972	0,028	0,973	0,972	0,972	0,945	0,996	0,996	

==== Confusion Matrix ====

a b <-- classified as

118 7 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,944	0,000	1,000	0,944	0,971	0,945	0,991	0,993	(m)
	1,000	0,056	0,947	1,000	0,973	0,945	0,991	0,990	(b)
Weighted Avg.	0,972	0,028	0,973	0,972	0,972	0,945	0,991	0,991	

==== Confusion Matrix ====

a b <-- classified as

118 7 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,920	0,000	1,000	0,920	0,958	0,923	0,995	0,995	(m)
	1,000	0,080	0,926	1,000	0,962	0,923	0,995	0,995	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,995	0,995	

=== Confusion Matrix ===

a b <-- classified as

115 10 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,944	0,000	1,000	0,944	0,971	0,945	0,990	0,992	(m)
	1,000	0,056	0,947	1,000	0,973	0,945	0,990	0,987	(b)
Weighted Avg.	0,972	0,028	0,973	0,972	0,972	0,945	0,990	0,990	

=== Confusion Matrix ===

a b <-- classified as

118 7 | a = (m)

0 125 | b = (b)

Ensemble by Voting (majority voting) 3 classifiers (LAC, NNGE, MODLEM)

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,996	0,000	1,000	0,996	0,998	0,996	0,998	0,998	(m)
	1,000	0,004	0,996	1,000	0,998	0,996	0,998	0,996	(b)
Weighted Avg.	0,998	0,002	0,998	0,998	0,998	0,996	0,998	0,997	

==== Confusion Matrix ====

a b <-- classified as

249 1 | a = (m)

0 250 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,968	0,024	0,976	0,968	0,972	0,944	0,987	0,991	(m)
	0,976	0,032	0,968	0,976	0,972	0,944	0,987	0,978	(b)
Weighted Avg.	0,972	0,028	0,972	0,972	0,972	0,944	0,987	0,985	

==== Confusion Matrix ====

a b <-- classified as

242 8 | a = (m)

6 244 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,992	0,000	1,000	0,992	0,996	0,992	1,000	1,000	(m)
	1,000	0,008	0,992	1,000	0,996	0,992	1,000	1,000	(b)
Weighted Avg.	0,996	0,004	0,996	0,996	0,996	0,992	1,000	1,000	

==== Confusion Matrix ====

a b <-- classified as

124 1 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,952	0,000	1,000	0,952	0,975	0,953	1,000	1,000	(m)
	1,000	0,048	0,954	1,000	0,977	0,953	1,000	1,000	(b)
Weighted Avg.	0,976	0,024	0,977	0,976	0,976	0,953	1,000	1,000	

==== Confusion Matrix ====

a b <-- classified as

119 6 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,960	0,000	1,000	0,960	0,980	0,961	1,000	1,000	(m)
	1,000	0,040	0,962	1,000	0,980	0,961	1,000	1,000	(b)
Weighted Avg.	0,980	0,020	0,981	0,980	0,980	0,961	1,000	1,000	

==== Confusion Matrix ====

a b <-- classified as

120 5 | a = (m)

0 125 | b = (b)

Test Set 4

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,000	1,000	0,984	0,992	0,984	0,999	0,999	(m)
	1,000	0,016	0,984	1,000	0,992	0,984	0,999	0,999	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,984	0,999	0,999	

==== Confusion Matrix ====

a b <-- classified as

123 2 | a = (m)

0 125 | b = (b)

Ensemble by Voting (majority voting) 3 classifiers (OneR, NNGE, MODLEM)

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,996	0,000	1,000	0,996	0,998	0,996	0,998	0,998	(m)
	1,000	0,004	0,996	1,000	0,998	0,996	0,998	0,996	(b)
Weighted Avg.	0,998	0,002	0,998	0,998	0,998	0,996	0,998	0,997	

=== Confusion Matrix ===

a b <-- classified as

249 1 | a = (m)

0 250 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,032	0,968	0,972	0,970	0,940	0,970	0,955	(m)
	0,968	0,028	0,972	0,968	0,970	0,940	0,970	0,957	(b)
Weighted Avg.	0,970	0,030	0,970	0,970	0,970	0,940	0,970	0,956	

=== Confusion Matrix ===

a b <-- classified as

243 7 | a = (m)

8 242 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	1,000	0,000	1,000	1,000	1,000	1,000	1,000	1,000	(m)
	1,000	0,000	1,000	1,000	1,000	1,000	1,000	1,000	(b)
Weighted Avg.	1,000	0,000	1,000	1,000	1,000	1,000	1,000	1,000	

==== Confusion Matrix ====

a b <-- classified as

125 0 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,976	0,000	1,000	0,976	0,988	0,976	0,988	0,988	(m)
	1,000	0,024	0,977	1,000	0,988	0,976	0,988	0,977	(b)
Weighted Avg.	0,988	0,012	0,988	0,988	0,988	0,976	0,988	0,982	

==== Confusion Matrix ====

a b <-- classified as

122 3 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,000	1,000	0,984	0,992	0,984	0,992	0,992	(m)
	1,000	0,016	0,984	1,000	0,992	0,984	0,992	0,984	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,984	0,992	0,988	

=== Confusion Matrix ===

a b <-- classified as

123 2 | a = (m)

0 125 | b = (b)

Test Set 4

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,000	1,000	0,984	0,992	0,984	0,992	0,992	(m)
	1,000	0,016	0,984	1,000	0,992	0,984	0,992	0,984	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,984	0,992	0,988	

=== Confusion Matrix ===

a b <-- classified as

123 2 | a = (m)

0 125 | b = (b)

Ensemble by Voting (majority voting) 4 classifiers (OneR, LAC, NNGE, MODLEM)

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,976	0,016	0,984	0,976	0,980	0,960	0,980	0,972	(m)
	0,984	0,024	0,976	0,984	0,980	0,960	0,980	0,969	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,980	0,970	

==== Confusion Matrix ====

a b <-- classified as

244 6 | a = (m)

4 246 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,972	0,036	0,964	0,972	0,968	0,936	0,968	0,951	(m)
	0,964	0,028	0,972	0,964	0,968	0,936	0,968	0,955	(b)
Weighted Avg.	0,968	0,032	0,968	0,968	0,968	0,936	0,968	0,953	

==== Confusion Matrix ====

a b <-- classified as

243 7 | a = (m)

9 241 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,992	0,000	1,000	0,992	0,996	0,992	0,996	0,996	(m)
	1,000	0,008	0,992	1,000	0,996	0,992	0,996	0,992	(b)
Weighted Avg.	0,996	0,004	0,996	0,996	0,996	0,992	0,996	0,994	

==== Confusion Matrix ====

a b <-- classified as

124 1 | a = (m)

0 125 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,000	1,000	0,984	0,992	0,984	0,992	0,992	(m)
	1,000	0,016	0,984	1,000	0,992	0,984	0,992	0,984	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,984	0,992	0,988	

==== Confusion Matrix ====

a b <-- classified as

123 2 | a = (m)

0 125 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,024	0,976	0,984	0,980	0,960	0,980	0,969	(m)
	0,976	0,016	0,984	0,976	0,980	0,960	0,980	0,972	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,980	0,970	

==== Confusion Matrix ====

a b <-- classified as

123 2 | a = (m)

3 122 | b = (b)

Test Set 4

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,984	0,032	0,969	0,984	0,976	0,952	0,976	0,961	(m)
	0,968	0,016	0,984	0,968	0,976	0,952	0,976	0,968	(b)
Weighted Avg.	0,976	0,024	0,976	0,976	0,976	0,952	0,976	0,965	

==== Confusion Matrix ====

a b <-- classified as

123 2 | a = (m)

4 121 | b = (b)

APPENDIX 4 – Detailed Results Obtained in WEKA for the AndroZoo Dataset

Radial Basis Function

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,017	0,983	0,990	0,987	0,973	0,995	0,992	(m)
	0,983	0,010	0,990	0,983	0,987	0,973	0,995	0,995	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,995	0,994	

=== Confusion Matrix ===

a b <-- classified as

297 3 | a = (m)

5 295 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,027	0,974	0,990	0,982	0,963	0,993	0,991	(m)
	0,973	0,010	0,990	0,973	0,982	0,963	0,993	0,995	(b)
Weighted Avg.	0,982	0,018	0,982	0,982	0,982	0,963	0,993	0,993	

=== Confusion Matrix ===

a b <-- classified as

297 3 | a = (m)

8 292 | b = (b)

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,027	0,974	0,987	0,980	0,960	0,990	0,988	(m)
	0,973	0,013	0,986	0,973	0,980	0,960	0,990	0,989	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,990	0,989	

=== Confusion Matrix ===

a b <-- classified as

148 2 | a = (m)

4 146 | b = (b)

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,980	0,020	0,980	0,980	0,980	0,960	0,992	0,991	(m)
	0,980	0,020	0,980	0,980	0,980	0,960	0,992	0,991	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,992	0,991	

=== Confusion Matrix ===

a b <-- classified as

147 3 | a = (m)

3 147 | b = (b)

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,975	0,015	0,985	0,975	0,980	0,960	0,990	0,991	(m)
	0,985	0,025	0,975	0,985	0,980	0,960	0,990	0,979	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,990	0,985	

=== Confusion Matrix ===

a b <-- classified as

195 5 | a = (m)

3 197 | b = (b)

Multilayer Perceptron

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,998	0,996	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,998	0,997	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,998	0,996	

=== Confusion Matrix ===

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,023	0,977	0,987	0,982	0,963	0,996	0,996	(m)
	0,977	0,013	0,987	0,977	0,982	0,963	0,996	0,996	(b)
Weighted Avg.	0,982	0,018	0,982	0,982	0,982	0,963	0,996	0,996	

=== Confusion Matrix ===

```
a b <-- classified as
296 4 | a = (m)
7 293 | b = (b)
```

Test Set 1

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,027	0,974	0,993	0,983	0,967	0,992	0,992	(m)
	0,973	0,007	0,993	0,973	0,983	0,967	0,992	0,988	(b)
Weighted Avg.	0,983	0,017	0,984	0,983	0,983	0,967	0,992	0,990	

=== Confusion Matrix ===

```
a b <-- classified as
149 1 | a = (m)
4 146 | b = (b)
```

Test Set 2

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,991	0,993	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,991	0,979	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,991	0,986	

=== Confusion Matrix ===

a b <-- classified as

148 2 | a = (m)

3 147 | b = (b)

Test Set 3

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,015	0,985	0,985	0,985	0,970	0,990	0,993	(m)
	0,985	0,015	0,985	0,985	0,985	0,970	0,990	0,976	(b)
Weighted Avg.	0,985	0,015	0,985	0,985	0,985	0,970	0,990	0,984	

=== Confusion Matrix ===

a b <-- classified as

197 3 | a = (m)

3 197 | b = (b)

Id3

Train

=== Detailed Accuracy by Class ===

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,998	0,997	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,998	0,997	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,998	0,997	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,013	0,987	0,990	0,988	0,977	0,992	0,992	(m)
	0,987	0,010	0,990	0,987	0,988	0,977	0,992	0,987	(b)
Weighted Avg.	0,988	0,012	0,988	0,988	0,988	0,977	0,992	0,989	

==== Confusion Matrix ====

a b <-- classified as

297 3 | a = (m)

4 296 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,991	0,988	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,991	0,982	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,991	0,985	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,013	0,987	0,987	0,987	0,973	0,992	0,990	(m)
	0,987	0,013	0,987	0,987	0,987	0,973	0,992	0,984	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,992	0,987	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,010	0,990	0,985	0,987	0,975	0,991	0,990	(m)
	0,990	0,015	0,985	0,990	0,988	0,975	0,991	0,983	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,991	0,987	

==== Confusion Matrix ====

a b <-- classified as

197 3 | a = (m)

2 198 | b = (b)

k-NN

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,998	0,997	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,998	0,997	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,998	0,997	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,027	0,974	0,997	0,985	0,970	0,995	0,995	(m)
	0,973	0,003	0,997	0,973	0,985	0,970	0,995	0,994	(b)
Weighted Avg.	0,985	0,015	0,985	0,985	0,985	0,970	0,995	0,995	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

8 292 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,027	0,974	0,993	0,983	0,967	0,995	0,992	(m)
	0,973	0,007	0,993	0,973	0,983	0,967	0,995	0,990	(b)
Weighted Avg.	0,983	0,017	0,984	0,983	0,983	0,967	0,995	0,991	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

4 146 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,995	0,994	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,995	0,992	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,995	0,993	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

3 147 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,015	0,985	0,990	0,988	0,975	0,994	0,993	(m)
	0,985	0,010	0,990	0,985	0,987	0,975	0,994	0,990	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,994	0,991	

==== Confusion Matrix ====

a b <-- classified as

198 2 | a = (m)

3 197 | b = (b)

NaiveBayes

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	1,000	0,080	0,926	1,000	0,962	0,923	0,990	0,984	(m)
	0,920	0,000	1,000	0,920	0,958	0,923	0,990	0,992	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,990	0,988	

==== Confusion Matrix ====

a b <-- classified as

300 0 | a = (m)

24 276 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	1,000	0,080	0,926	1,000	0,962	0,923	0,990	0,986	(m)
	0,920	0,000	1,000	0,920	0,958	0,923	0,990	0,993	(b)
Weighted Avg.	0,960	0,040	0,963	0,960	0,960	0,923	0,990	0,989	

==== Confusion Matrix ====

a b <-- classified as

300 0 | a = (m)

24 276 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,080	0,925	0,993	0,958	0,916	0,986	0,980	(m)
	0,920	0,007	0,993	0,920	0,955	0,916	0,986	0,988	(b)
Weighted Avg.	0,957	0,043	0,959	0,957	0,957	0,916	0,986	0,984	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

12 138 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,093	0,914	0,993	0,952	0,903	0,985	0,982	(m)
	0,907	0,007	0,993	0,907	0,948	0,903	0,984	0,977	(b)
Weighted Avg.	0,950	0,050	0,953	0,950	0,950	0,903	0,985	0,980	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

14 136 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,070	0,934	0,990	0,961	0,922	0,984	0,984	(m)
	0,930	0,010	0,989	0,930	0,959	0,922	0,984	0,975	(b)
Weighted Avg.	0,960	0,040	0,962	0,960	0,960	0,922	0,984	0,979	

==== Confusion Matrix ====

a b <-- classified as

198 2 | a = (m)

14 186 | b = (b)

J48

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,013	0,987	0,990	0,988	0,977	0,993	0,987	(m)
	0,987	0,010	0,990	0,987	0,988	0,977	0,993	0,993	(b)
Weighted Avg.	0,988	0,012	0,988	0,988	0,988	0,977	0,993	0,990	

==== Confusion Matrix ====

a b <-- classified as

297 3 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,013	0,987	0,990	0,988	0,977	0,984	0,973	(m)
	0,987	0,010	0,990	0,987	0,988	0,977	0,984	0,982	(b)
Weighted Avg.	0,988	0,012	0,988	0,988	0,988	0,977	0,984	0,978	

==== Confusion Matrix ====

a b <-- classified as

297 3 | a = (m)

4 296 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,980	0,020	0,980	0,980	0,980	0,960	0,989	0,979	(m)
	0,980	0,020	0,980	0,980	0,980	0,960	0,989	0,989	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,989	0,984	

==== Confusion Matrix ====

a b <-- classified as

147 3 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,980	0,013	0,987	0,980	0,983	0,967	0,989	0,983	(m)
	0,987	0,020	0,980	0,987	0,983	0,967	0,989	0,985	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,989	0,984	

==== Confusion Matrix ====

a b <-- classified as

147 3 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,970	0,010	0,990	0,970	0,980	0,960	0,989	0,984	(m)
	0,990	0,030	0,971	0,990	0,980	0,960	0,989	0,983	(b)
Weighted Avg.	0,980	0,020	0,980	0,980	0,980	0,960	0,989	0,984	

==== Confusion Matrix ====

a b <-- classified as

194 6 | a = (m)

2 198 | b = (b)

SVM

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,013	0,987	0,993	0,990	0,980	0,990	0,984	(m)
	0,987	0,007	0,993	0,987	0,990	0,980	0,990	0,987	(b)
Weighted Avg.	0,990	0,010	0,990	0,990	0,990	0,980	0,990	0,985	

==== Confusion Matrix ====

a b <-- classified as

298 2 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,017	0,983	0,987	0,985	0,970	0,985	0,977	(m)
	0,983	0,013	0,987	0,983	0,985	0,970	0,985	0,979	(b)
Weighted Avg.	0,985	0,015	0,985	0,985	0,985	0,970	0,985	0,978	

==== Confusion Matrix ====

a b <-- classified as

296 4 | a = (m)

5 295 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,983	0,974	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,983	0,977	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,983	0,975	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,980	0,013	0,987	0,980	0,983	0,967	0,983	0,977	(m)
	0,987	0,020	0,980	0,987	0,983	0,967	0,983	0,974	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,983	0,975	

==== Confusion Matrix ====

a b <-- classified as

147 3 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,975	0,010	0,990	0,975	0,982	0,965	0,983	0,978	(m)
	0,990	0,025	0,975	0,990	0,983	0,965	0,983	0,971	(b)
Weighted Avg.	0,983	0,018	0,983	0,983	0,982	0,965	0,983	0,974	

==== Confusion Matrix ====

a b <-- classified as

195 5 | a = (m)

2 198 | b = (b)

FURIA

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,013	0,987	0,993	0,990	0,980	0,992	0,985	(m)
	0,987	0,007	0,993	0,987	0,990	0,980	0,992	0,990	(b)
Weighted Avg.	0,990	0,010	0,990	0,990	0,990	0,980	0,992	0,988	

==== Confusion Matrix ====

a b <-- classified as

298 2 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,013	0,987	0,990	0,988	0,977	0,988	0,982	(m)
	0,987	0,010	0,990	0,987	0,988	0,977	0,988	0,983	(b)
Weighted Avg.	0,988	0,012	0,988	0,988	0,988	0,977	0,988	0,983	

==== Confusion Matrix ====

a b <-- classified as

297 3 | a = (m)

4 296 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,986	0,977	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,986	0,983	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,986	0,980	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,980	0,013	0,987	0,980	0,983	0,967	0,986	0,980	(m)
	0,987	0,020	0,980	0,987	0,983	0,967	0,986	0,980	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,986	0,980	

==== Confusion Matrix ====

a b <-- classified as

147 3 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,975	0,010	0,990	0,975	0,982	0,965	0,987	0,982	(m)
	0,990	0,025	0,975	0,990	0,983	0,965	0,987	0,980	(b)
Weighted Avg.	0,983	0,018	0,983	0,983	0,982	0,965	0,987	0,981	

==== Confusion Matrix ====

a b <-- classified as

195 5 | a = (m)

2 198 | b = (b)

RandomForest

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,998	0,996	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,998	0,997	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,998	0,997	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,017	0,983	0,987	0,985	0,970	0,996	0,996	(m)
	0,983	0,013	0,987	0,983	0,985	0,970	0,996	0,997	(b)
Weighted Avg.	0,985	0,015	0,985	0,985	0,985	0,970	0,996	0,996	

==== Confusion Matrix ====

a b <-- classified as

296 4 | a = (m)

5 295 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,996	0,994	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,996	0,995	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,996	0,994	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,013	0,987	0,987	0,987	0,973	0,994	0,993	(m)
	0,987	0,013	0,987	0,987	0,987	0,973	0,994	0,989	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,994	0,991	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,010	0,990	0,985	0,987	0,975	0,993	0,992	(m)
	0,990	0,015	0,985	0,990	0,988	0,975	0,993	0,986	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,993	0,989	

==== Confusion Matrix ====

a b <-- classified as

197 3 | a = (m)

2 198 | b = (b)

NNGE / AdaBoostM1

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,998	0,996	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,998	0,997	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,998	0,997	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,995	0,995	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,995	0,995	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,995	0,995	

==== Confusion Matrix ====

a b <-- classified as

298 2 | a = (m)

6 294 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,997	0,994	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,997	0,995	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,997	0,995	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,013	0,987	0,987	0,987	0,973	0,994	0,993	(m)
	0,987	0,013	0,987	0,987	0,987	0,973	0,994	0,989	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,994	0,991	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,010	0,990	0,985	0,987	0,975	0,993	0,992	(m)
	0,990	0,015	0,985	0,990	0,988	0,975	0,993	0,987	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,993	0,989	

==== Confusion Matrix ====

a b <-- classified as

197 3 | a = (m)

2 198 | b = (b)

Ensemble by Voting (majority voting) 3 classifiers (LAC, NNGE, MODLEM)

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,992	0,985	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,992	0,990	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,992	0,988	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,990	0,033	0,967	0,990	0,979	0,957	0,978	0,963	(m)
	0,967	0,010	0,990	0,967	0,978	0,957	0,978	0,973	(b)
Weighted Avg.	0,978	0,022	0,979	0,978	0,978	0,957	0,978	0,968	

==== Confusion Matrix ====

a b <-- classified as

297 3 | a = (m)

10 290 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,987	0,977	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,987	0,983	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	(m)
	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,010	0,990	0,985	0,987	0,975	0,988	0,983	(m)
	0,990	0,015	0,985	0,990	0,988	0,975	0,988	0,980	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,988	0,981	

==== Confusion Matrix ====

a b <-- classified as

197 3 | a = (m)

2 198 | b = (b)

Ensemble by Voting (majority voting) 3 classifiers (OneR, NNGE, MODLEM)

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,997	0,013	0,987	0,997	0,992	0,983	0,992	0,985	(m)
	0,987	0,003	0,997	0,987	0,992	0,983	0,992	0,990	(b)
Weighted Avg.	0,992	0,008	0,992	0,992	0,992	0,983	0,992	0,988	

==== Confusion Matrix ====

a b <-- classified as

299 1 | a = (m)

4 296 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,983	0,974	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,983	0,977	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,983	0,975	

==== Confusion Matrix ====

a b <-- classified as

296 4 | a = (m)

6 294 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,987	0,977	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,987	0,983	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	(m)
	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,010	0,990	0,985	0,987	0,975	0,988	0,983	(m)
	0,990	0,015	0,985	0,990	0,988	0,975	0,988	0,980	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,988	0,981	

==== Confusion Matrix ====

a b <-- classified as

197 3 | a = (m)

2 198 | b = (b)

Ensemble by Voting (majority voting) 4 classifiers (OneR, LAC, NNGE, MODLEM)

Train

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,983	0,974	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,983	0,977	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,983	0,975	

==== Confusion Matrix ====

a b <-- classified as

296 4 | a = (m)

6 294 | b = (b)

10-fold x-val.

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,020	0,980	0,987	0,983	0,967	0,983	0,974	(m)
	0,980	0,013	0,987	0,980	0,983	0,967	0,983	0,977	(b)
Weighted Avg.	0,983	0,017	0,983	0,983	0,983	0,967	0,983	0,975	

==== Confusion Matrix ====

a b <-- classified as

296 4 | a = (m)

6 294 | b = (b)

Test Set 1

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,993	0,020	0,980	0,993	0,987	0,973	0,987	0,977	(m)
	0,980	0,007	0,993	0,980	0,987	0,973	0,987	0,983	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	

==== Confusion Matrix ====

a b <-- classified as

149 1 | a = (m)

3 147 | b = (b)

Test Set 2

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	(m)
	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	(b)
Weighted Avg.	0,987	0,013	0,987	0,987	0,987	0,973	0,987	0,980	

==== Confusion Matrix ====

a b <-- classified as

148 2 | a = (m)

2 148 | b = (b)

Test Set 3

==== Detailed Accuracy by Class ====

	TP Rate	FP Rate	Precision	Recall	F-Measure	MCC	ROC Area	PRC Area	Class
	0,985	0,010	0,990	0,985	0,987	0,975	0,988	0,983	(m)
	0,990	0,015	0,985	0,990	0,988	0,975	0,988	0,980	(b)
Weighted Avg.	0,988	0,013	0,988	0,988	0,987	0,975	0,988	0,981	

=== Confusion Matrix ===

a b <-- classified as

197 3 | a = (m)

2 198 | b = (b)