INSTITUTO SUPERIOR DE ENGENHARIA DE LISBOA

Departamento de Engenharia de Eletrónica e Telecomunicações e de Computadores

Malware Detection in Android Applications with Machine Learning Techniques

Catarina Rodrigues Palma

Licenciada em Engenharia Informática, Redes e Telecomunicações

Dissertação para obtenção do Grau de Mestre
em Engenharia Informática e de Computadores

Orientador : Professor Doutor Artur Jorge Ferreira

Júri:

Presidente: Professor Doutor Tiago Miguel Braga da Silva Dias

Vogais: Professor Doutor Rui Manuel Feliciano de Jesus
Professor Doutor Artur Jorge Ferreira

Outubro, 2023

Ao meu irmão, que todos os dias relembro com enorme saudade e carinho. Questiono-me se teria orgulho em mim.

Agradecimentos

Em primeiro lugar, gostaria de agradecer ao professor Artur Ferreira pela oportunidade de realizar esta tese e por todo o apoio, sugestões construtivas, e disponibilidade para ajudar face a qualquer problema ao longo do trabalho desenvolvido.

Agradeço também aos meus colegas e amigos do ISEL pela simpatia, espírito de entreajuda e palavras de motivação.

Ao meu melhor amigo Ricardo, que esteve ao meu lado praticamente durante todo o nosso percurso no ISEL. Sempre presente nos piores e melhores momentos. O seu apoio, conselhos, positivismo e palavras de motivação foram incansáveis. Muito obrigada por tudo.

À minha melhor amiga, de há mais de uma década, Isabela, sempre a motivar-me e a certificar-se que estava tudo bem comigo. Obrigada por todo o apoio e compreensão demonstrada nos momentos mais difíceis.

Quero agradecer a todos os meus amigos e à minha família, em especial à minha mãe e avó materna, por sempre me darem todo o apoio para poder seguir os meus objetivos. Também agradeço à minha pequena Luna, que também faz parte da família, por me alegrar e fazer companhia.

Muito obrigada a todos, sem o vosso apoio nada disto seria possível!

Abstract

The presence of malicious software (malware), for example, in Android applications (apps), has harmful or irreparable consequences to the user and/or the device. Despite the protections app stores provide to restrict apps containing malware, it keeps growing both in sophistication and diffusion.

In this work, we explore the use of Machine Learning (ML) techniques to detect malware in Android apps. The focus is on the study of different data pre-processing, dimensionality reduction, and classification techniques, assessing the generalisation ability of the learned models using standard and public domain datasets. From the literature and our own experimental results, it can be concluded that the classifiers that achieve the best performance in Android malware detection are the Support Vector Machine (SVM) and Random Forest (RF). We also emphasise Feature Selection (FS), which reduces the data's dimensionality and identifies the most relevant features in Android malware classification. Different evaluation metrics are applied to the learned model and compared against the experimental results found in the literature.

The final goal of this study was the development of a prototype that resorts to ML techniques to detect malware in Android apps. Our approach is able to identify the most relevant features to classify an app as malicious. Namely, we conclude that permissions play a prominent role in Android malware detection. The proposed approach reduced the data dimensionality while achieving high accuracy in identifying malware in Android apps.

Keywords: Android Applications, Datasets, Feature Selection, Machine Learning, Malware Detection, Security, Supervised Learning

Resumo

A presença de software malicioso (malware) em, por exemplo, aplicações Android, tem consequências prejudiciais e irreparáveis para o utilizador e/ou o dispositivo. Apesar das *app stores* providenciarem proteções para restringir aplicações contendo malware, este continua a crescer em sofisticação e difusão.

Neste trabalho, exploramos técnicas de Aprendizagem Automática (AA) para deteção de malware em aplicações Android. Com foco no estudo de diferentes técnicas de pré-processamento, redução de dimensionalidade e classificação, avaliando a capacidade de generalização do modelo usando conjuntos de dados *standard* e de domínio público. Com base na literatura e nos nossos resultados experimentais, concluímos que os classificadores que apresentam melhor desempenho na deteção de malware em aplicações Android são *Support Vector Machine* (SVM) e *Random Forest* (RF). É dado ênfase à Seleção de Atributos (SA), que reduz a dimensionalidade dos dados e identifica os atributos mais decisivos para classificação de malware em Android. Aplicam-se diferentes métricas de avaliação ao modelo e comparam-se os resultados experimentais com os reportados na literatura.

O objetivo deste estudo é o desenvolvimento de um protótipo que recorra a técnicas de AA para detetar malware em aplicações Android. A nossa abordagem é capaz de identificar os atributos mais relevantes para classificar uma aplicação como maliciosa. Nomeadamente, concluímos que as permissões se destacam na deteção de malware em Android. A abordagem proposta reduz a dimensionalidade dos dados enquanto apresenta uma alta acurácia na identificação de malware em aplicações Android.

Palavras-chave: Aplicações Android, Aprendizagem Automática, Aprendizagem Supervisionada, Conjuntos de dados, Deteção de Malware, Segurança, Seleção de Atributos.

Contents

List of Figures xv

List of Tables xvii

Acronyms xxi

Glossary xxv

1 Introduction 1

1.1 Context and motivation 1

1.2 Objectives 4

1.3 Contributions 4

1.4 Document organisation 6

2 State of the Art 7

2.1 Android 8

2.1.1 Android architecture 8

2.1.2 Structure of an Android application 9

2.2 Malware on Android applications 11

2.2.1 Existing security measures 12

2.2.2 Malware techniques to avoid detection 13

2.3 Data acquisition 13

2.3.1 Type of analysis . . . 13
2.3.2 Datasets . . . 15
2.4 Pre-processing . . . 17
2.4.1 Data pre-processing . . . 17
2.4.2 Data splitting . . . 20
2.5 Classifiers . . . 22
2.5.1 Random forest . . . 22
2.5.2 Support vector machine . . . 24
2.5.3 K-Nearest Neighbours . . . 25
2.5.4 Naive Bayes . . . 26
2.5.5 Multi-layer Perceptron . . . 27
2.6 Evaluation metrics . . . 27
2.7 Android malware detection using machine learning . . . 31
2.7.1 Deep learning approaches . . . 31
2.7.2 Summary of the existing approaches . . . 32

3 Proposed Approach **37**
3.1 Machine learning module . . . 37
3.2 Complete approach - full block diagram . . . 40
3.3 Software tools . . . 41

4 Experimental Evaluation **45**
4.1 Testing environment . . . 46
4.2 Dataset analysis . . . 46
4.3 Experimental Results: Baseline . . . 50
4.4 Experimental Results: Data Pre-processing . . . 51
4.4.1 Handling missing values . . . 52
4.4.2 Normalisation . . . 53
4.4.3 Numerosity balancing . . . 54
4.5 Experimental Results: Feature Selection . . . 55

4.6 Experimental Results: CV & Hyperparameter tuning 60
4.7 Comparative Analysis of Results 62
4.8 Evaluating the model with real-world applications 63

5 Conclusions 67
5.1 Overview 67
5.2 Future Work 69

References 71

A Experimental Results i
A.1 Experimental Results: Baseline i
A.2 Experimental Results: Data Pre-processing ii
A.2.1 Missing values ii
A.2.2 Normalisation iii
A.2.3 Numerosity balancing iv
A.3 Experimental Results: Feature Selection v
A.3.1 Most relevant features in the Drebin dataset vii
A.3.2 Most relevant features in the CICAndMal2017 dataset ix
A.3.3 Most relevant features in the AM dataset x
A.3.4 Most relevant features in the AMSF dataset x
A.4 Experimental Results: CV & Hyperparameter tuning xiii
A.5 Experimental Results: Real-world Applications xv

List of Figures

1.1 Pie chart representing the mobile OS market share in September 2023 (data published in [45]). 2

1.2 Malware installation packages for smartphone devices [6]. 3

2.1 The Android software stack (inspired by Figure 1 in [53]). 9

2.2 Structure of an Android application (inspired by Figure 1 in [66]). 10

2.3 Statistics regarding the type of analysis used in ML-based Android malware detection papers between 2016 and 2020, extracted from [66]. . . . 14

2.4 Example of 5-fold CV for the training and testing sets and the training and validation sets. 21

2.5 Example of Leave-one-out CV for the training and testing sets and the training and validation sets. 21

2.6 Simplified representation of the ML branches. 22

2.7 Random Forest schema (extracted from [62]). 23

2.8 Example on finding the hyperplane, with the possible hyperplanes on the left and then the optimal hyperplane (that allows for the maximum margin) on the right (extracted from [60]). 24

2.9 Prediction examples for the KNN classifier, with k=3 in (b) and k=7 in (c), to classify the data point in (a). 25

2.10 Multi-layer Perceptron (MLP) schema. 27

2.11 Confusion matrix for binary classification. 29

2.12 The ROC space for a "better" and "worse" classifier, extracted from [55]. . 30

3.1 Partial block diagram of the proposed approach: the data pre-processing stage, which is composed of handling missing values, numerosity balancing, and FS. The vertical arrow points to the continuation of the ML pipeline, and the right-hand side arrow highlights that the approach identifies the most relevant features for the feature extraction module. . 38

3.2 Partial block diagram of the proposed approach: data splitting for training and testing of the model with standard evaluation metrics. With a validation set provided to perform hyperparameter tuning. The right-hand side arrow with input data refers to the use of data from real-world applications. 38

3.3 Full block diagram of the ML module, aggregating all the stages referenced in Figure 3.1 and Figure 3.2 as well as the hyperparameter tunning stage. 39

3.4 Full block diagram of the proposed approach with the ML module and the Android applications and feature extraction modules. 40

4.1 Orange workflow for dataset analysis. 47

4.2 Class distribution ('benign' in green and 'malicious' in red) obtained via the 'Distributions' widget for the Drebin, CICAndMal2017, AM and AMSF datasets, in (a), (b), (c) and (d), respectively. 48

4.3 Accuracy (%) obtained with each classifier (RF, SVM, KNN, NB and MLP) for each dataset, Drebin, CICAndMal2017, AM and AMSF, in (a), (b), (c) and (d), respectively. 51

4.4 Accuracy (%) obtained, with the RF and SVM classifiers, for the AM dataset after applying different methods to deal with missing values. . . 52

4.5 The original numbers of features versus the number of features after applying RRFS with different relevance measures for the Drebin, CICAndMal2017, AM and AMSF datasets presented in (a), (b), (c) and (d), respectively. 57

4.6 Numbers of features in the Drebin, CICAndMal2017, AM and AMSF datasets presented in (a), (b), (c) and (d), respectively, before RRFS and after RRFS with FR for different values of M_s. 58

4.7 Developed Android applications: 'App1', 'App2' and 'App3' in (a) and (b), (c) and (d), respectively. 64

List of Tables

2.1 Summary of some of the results of DL approaches for Android malware detection found in the literature. 32

2.2 Summary of some of the results of ML approaches for Android malware detection found in the literature. 35

4.1 Characteristics of the computational environment where the proposed approach was assessed. 46

4.2 Number of instances (n) and features (d) for each dataset and their online reference. 47

4.3 Number and percentage of benign and malicious instances (n) in each dataset. 49

4.4 Number of categorical and numerical features (d) in each dataset. 49

4.5 Number of missing values in each dataset. 49

4.6 Experimental results, in terms of accuracy (Acc), F1-score and Area Under the Curve - Receiver Operating Characteristic (AUC-ROC) for the CICAndMal2017 and AM datasets and the RF and SVM classifiers with and without the use of Min-Max normalisation. 54

4.7 Accuracy (Acc) and Recall (Rec) values, obtained with the Random Forest (RF) and Support Vector Machine (SVM) classifiers with the different numerosity balancing approaches for the AM dataset. 55

4.8 Accuracy (Acc) obtained with the SVM classifier for each dataset, by not applying FS, applying RRFS with MM or RRFS with FR. 56

4.9 Comparison of the experimental results, in terms of accuracy (%), obtained by Alkahtani and Aldhyani [6] with the SVM classifier, with the ones obtained with the proposed approach using the same classifier. . . 62

A.1 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and classifier. i

A.2 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the CICAndMal2017 dataset and the RF classifier. ii

A.3 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the AM dataset and the RF classifier. ii

A.4 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the CICAndMal2017 dataset and the SVM classifier. ii

A.5 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the AM dataset and the SVM classifier. iii

A.6 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with and without the use of Min-Max normalisation. iii

A.7 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with different numerosity balancing techniques. iv

A.8 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with RRFS with MM and FR (with M_s equal to 0.3). v

A.9 Number of features (d) in each dataset before RRFS and after RRFS with MM and FR (with M_s equal to 0.3). v

A.10 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with RRFS with FR and different values of M_s. vi

A.11 Number of features (d) in each dataset before RRFS and after RRFS with FR for different values of M_s. vi

A.12 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers without RRFS and with RRFS with FR and M_s equal to 0.3. vii

A.13 Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with and without tuning of the hyperparameters. xiii

A.14 Hyperparameters (deemed more relevant) for the RF classifier optimised for each dataset. xiii

A.15 Hyperparameters (deemed more relevant) for the SVM classifier optimised for each dataset. xiii

A.16 Experimental results in the form of the mean and standard deviation (std) of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers, with LOOCV and 10-fold CV. xiv

A.17 Experimental results obtained using the RF classifier and each dataset that assess if the prototype of the proposed approach correctly predicts or not the existence of malicious content, based on the features extracted from APK files of real-world Android applications. xv

Acronyms

AC Absolute Cosine. 19, 55, 56, 58, 68
AE Autoencoder. 31
AI Artificial Intelligence. 3
AM Android Malware. 16, 46, 47, 48, 49, 50, 52, 53, 54, 55, 56, 57, 58, 60, 65, 67, 68
AMD Android Malware Dataset. 15
AMSF Android Malware static feature. 17, 46, 47, 48, 49, 50, 52, 54, 55, 56, 57, 58, 60, 63, 64, 65, 67, 68
ANN Artificial Neural Networks. 3, 31
API Application Programming Interface. 8, 9, 14, 16, 17
APK Android Package Kit. 8, 9, 10, 14, 17, 33, 63, 65, 69
ART Android Runtime. 8, 10
AUC Area Under the Curve. 30
AUC-ROC Area Under the Curve - Receiver Operating Characteristic. xvii, 30, 53, 54, 55, 69

BiLSTM Bidirectional LSTM. 32

CNN Convolutional Neural Networks. 31
CSV Comma-Separated Values. 17, 46
CV Cross-Validation. 20, 21, 39, 45, 50, 60, 61, 68

DBN Deep Belief Networks. 32
DDoS Distributed Denial-of-Service. 11
DEX Dalvik Executable. 8, 10, 14

DL Deep Learning. 3, 7, 27, 31, 50, 51, 68, 69
DT Decision Tree. 22, 23, 34, 35

FN False Negative. 28, 37
FP False Positive. 28, 37
FPR False Positive Rate. 30
FR Fisher's Ratio. 19, 55, 56, 57, 58, 62, 68
FS Feature Selection. 33, 34, 38, 41, 45, 53, 55, 56, 58, 62, 63, 68

HAL Hardware Abstraction Layer. 8

IDE Integrated Development Environment. 42, 43
iOS iPhone Operating System. 1, 2

KNN K-Nearest Neighbours. 25, 26, 32, 33, 34, 35, 50, 67

LDA Linear Discriminant Analysis. 32, 34, 35
LightGBM Light Gradient Boosting Model. 34
LOO Leave-one-out. 20, 21, 61, 68
LR Logistic Regression. 34
LSTM Long Short-Term Memory. 31

MAE Mean Absolute Error. 28
ML Machine Learning. 3, 4, 6, 7, 13, 17, 18, 20, 22, 23, 24, 26, 27, 28, 31, 33, 34, 35, 39, 41, 45, 46, 50, 51, 60, 61, 62, 63, 65, 67, 68, 69
MLP Multi-layer Perceptron. 27, 34, 50, 51, 68
MM Mean-Median. 18, 55, 56, 68
MSE Mean Squared Error. 28

NB Naive Bayes. 26, 33, 34, 50, 67

OS Operating System. 1, 2, 8, 12, 15, 42

PCA Principal Component Analysis. 18, 34

RAT Remote Access Trojan. 11

RBF Radial Basis Function. 24, 61

RF Random Forest. xvii, 22, 23, 24, 33, 34, 35, 50, 51, 53, 54, 55, 56, 57, 58, 60, 61, 63, 67

RFE Recursive Feature Elimination. 34

ROC Receiver Operating Characteristic. 30

RRFS Relevance-Redundancy Feature Selection. 18, 55, 56, 57, 58, 62, 65, 68, 69

SMOTE Synthetic Minority Over-sampling Technique. 19, 34, 54, 55

SMS Short Message Service. 12, 64

SVM Support Vector Machine. xvii, 18, 24, 25, 32, 33, 34, 35, 50, 51, 53, 54, 55, 56, 58, 60, 61, 63, 67

TN True Negative. 28

TP True Positive. 28, 54

TPR True Positive Rate. 30

VPN Virtual Private Network. 12

XML Extensible Markup Language. 9

Glossary

bias Phenomenon that occurs when an algorithm produces results that are systematically prejudiced due to erroneous assumptions in the Machine Learning process. 18, 20

branch A part of a Decision Tree. 23

class label Discrete attribute whose value is predicted based on the values of other attributes. 20, 22, 30, 47, 48, 56, 68

decision node When a Decision Tree's sub-node splits into further sub-nodes. 22, 23

hyperparameter Parameter whose value is used to control the learning process. 20, 23, 24, 25, 41, 45, 50, 60, 61, 62, 68

leaf node Endpoint of a branch, the final output of a series of decisions. 23

multiclass Each input has only one output class out of more than two classes. 22, 28, 69

multilabel Each input can have multi-output classes out of more than two classes. 22

noise Additional meaningless information present in the data. 25, 26

outlier Data point that differs significantly from the other data points. 18

overfitting Behaviour that occurs when the Machine Learning model gives accurate predictions for training data but not for new data. No generalisation ability of the trained model. 18, 19, 23, 24, 26, 55

oversampling Technique to balance uneven datasets by keeping all of the data in the majority class and increasing the size of the minority class. 19, 54, 55

underfitting Behaviour that occurs when the Machine Learning model can neither model the training data nor generalise to new data. 18, 58

undersampling Technique to balance uneven datasets by keeping all of the data in the minority class and decreasing the size of the majority class. 19, 54, 55

Introduction

This first chapter introduces the context and relevance of the problem addressed by this dissertation, the motivation behind this study and research and, finally, the objectives and contribution of the thesis.

Section 1.1 provides insight into the context and motivation behind this thesis. Section 1.2 presents the primary goals of the thesis. The thesis contribution to this field of study is depicted in Section 1.3. Lastly, Section 1.4 describes this document's organisation and structure.

1.1 Context and motivation

The use of smartphones has grown exponentially over the past years. To offer a more comprehensive view, at the time of writing of this document, the world population is around 8 billion, and the estimated number of smartphone users is a staggering 5.25 billion [30], and it continues to grow.

This growth has been accompanied by the popularisation of Android, an open-source Operating System (OS), based on the Linux kernel and mainly designed for touch-screen mobile devices such as smartphones and tablets. It was first launched in 2008, and since then, it has become one of the most popular mobile OS, and it is estimated that 70% of mobile phone users utilise Android [46]. Currently, the only other OS that presents significant competition to Android is the iPhone Operating System (iOS).

However, Android is still more popular, being the mobile OS with the larger market share, as shown in Figure 1.1.

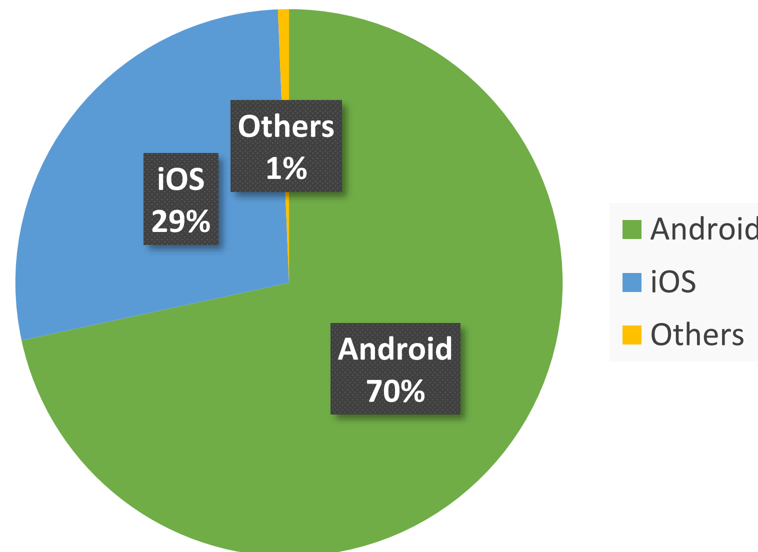

Figure 1.1: Pie chart representing the mobile OS market share in September 2023 (data published in [45]).

Many Android applications exist for various purposes, such as E-commerce, banking, education, social media, and entertainment. In September 2023, the app store Google Play Store had 3.553 million apps available for Android users to download [15].

The rapid wide-scale expansion of the use of smartphone devices, the increased popularity of the Android OS that dominates the market and the wide variety and number of Android applications attract the attention of malware developers. Additionally, due to its open nature and a broader range of devices, Android can be more susceptible to malware attacks when compared to iOS, which has a more closed ecosystem which contributes to a more secure environment [16].

Attackers can target a wide range of applications that deal with a significant amount of user-sensitive data, such as passwords, contacts, and banking information. They can also target the user's data on the smartphone or want to use the device to execute other attacks. Furthermore, from the attacker's perspective, the massive number of users are all targets and potential victims that can download their malware. Millions of users can download one app (that could contain malicious code) in a matter of minutes. Thus, the need to detect malicious applications is a major issue.

Since the Android system has become a popular and profitable target, malicious attacks against Android mobile devices have increased. In 2021, Kaspersky[1] detected 9.5 million malware Android packages, three times more than in 2019 (3.1 million) [6]. Figure 1.2 summarises the increase in malware installation packages for smartphone devices between 2017 and 2021.

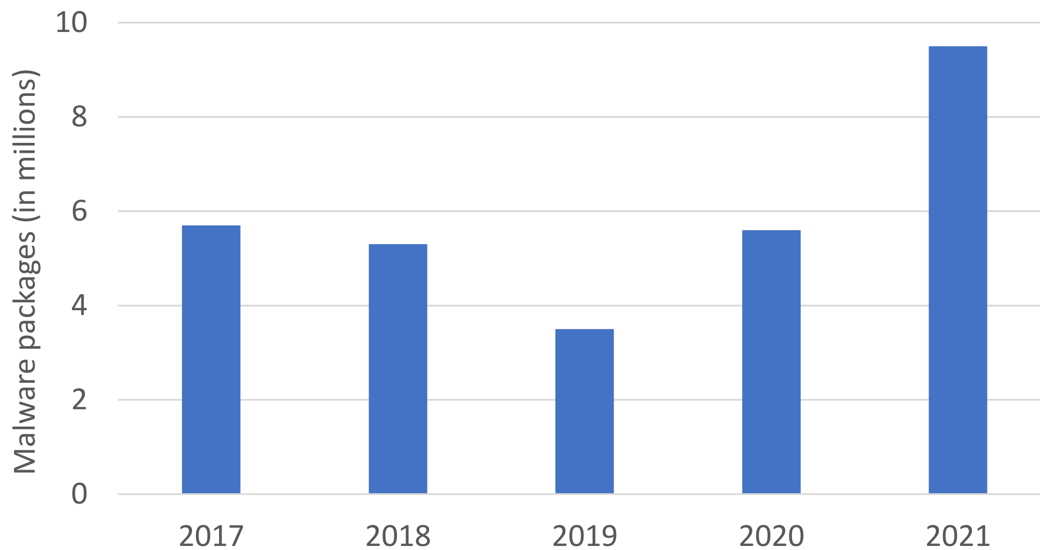

Figure 1.2: Malware installation packages for smartphone devices [6].

There are software and applications focused on security, and major app stores also have security and detection mechanisms to mitigate malicious apps (for example, Google Play Store has Google Play Protect). These are, to some extent, successful, but there's a wide variety of malware, and some can easily bypass them. Malware continues to grow in sophistication and diffusion as its developers constantly work on contouring security mechanisms and exploiting vulnerabilities.

In recent years, Machine Learning (ML) techniques have been proposed to approach the malware detection problem in Android applications. ML focuses on enabling computers to perform tasks without explicit programming, and it is a field of Artificial Intelligence (AI), which is devoted to making machines act like humans. ML includes Deep Learning (DL), which is a subset of ML based on Artificial Neural Networks (ANN) that can also be used to detect malware in Android applications. The focus of this thesis is directed at the ML approaches in specific. As such, DL approaches will be referenced, given their relevance and good results in solving this problem, but not explored or detailed.

ML approaches have proved very efficient and versatile in various fields, such as

[1] https://www.kaspersky.com/

health, education, finances, business, and retail. It has already proven to be a milestone in the tech industry. However, there are no perfect solutions, and malware is constantly evolving. Thus, whatever the chosen ML approach, it will not be able to identify the existence of malware in all cases. However, it will help, in many cases, to mitigate its threat in Android applications.

1.2 Objectives

This thesis focuses on studying the existing ML techniques and their application scenarios, namely, which techniques provide a better performance in detecting malware in Android applications. Thus, this study aims to:

- Identify the most decisive features for Android malware classification.
- Recognise the ML classifiers that provide the most satisfying results in detecting malware in Android applications.
- Assess the impact of different data pre-processing techniques applied to the datasets of this problem.
- Develop a prototype that resorts to ML techniques to detect malware in Android applications.

To accomplish these goals, public domain datasets are used, namely, datasets referenced in the literature, to enable results comparison.

The development of this thesis involved several phases, with the problem being vast and complex. Additionally, there was no prior knowledge on our part regarding the Android system. Thus, **the study of Android also integrated a significant portion of the work** put onto this thesis. Some basic Android applications were developed to consolidate this intended knowledge and test the developed prototype.

1.3 Contributions

This thesis provides an analysis and experimental evaluation of the use of ML techniques for malware detection in Android applications, contributing to this field with:

- The development of a prototype that resorts to ML techniques to detect malicious apps and draw conclusions about its performance in mitigating this problem.

- Conclusions and comparisons between the obtained experimental results and the ones reported in the literature.

- Enrichment of the literature by assessing the impact of different data pre-processing techniques using four different datasets. Data pre-processing is an essential step, and to the best of our knowledge, this aspect is lacking attention in the literature on this problem.

- Enriching the literature by identifying the most decisive features for malware detection among the public-domain datasets used and identifying the ML classifiers that provide the best results.

- Expanding the literature by using real-world Android applications (developed and existing) to extend test scenarios over the ones made available by the datasets. To the best of our knowledge, no previous work uses real-world applications for malware model testing.

- From this thesis, the following papers have been published.

  – Catarina Palma, Artur Ferreira, and Mário Figueiredo, "*On the use of machine learning techniques to detect malware in mobile applications*" [17], Simpósio em Informática[2] (INForum), September 2023, Porto, Portugal

  – Catarina Palma, Artur Ferreira, and Mário Figueiredo, "*A study on the role of feature selection for malware detection on Android applications*" [18], Portuguese Conference on Pattern Recognition[3] (RECPAD), October 2023, Coimbra, Portugal

- Additionally, the following paper has been submitted and, at the time of writing this document, is pending the decision of the editor.

  – Catarina Palma, Artur Ferreira, and Mário Figueiredo, "*An Approach for Explainability on Malware Detection on Android Applications with Machine Learning*", *Information* journal, Multidisciplinary Digital Publishing Institute (MDPI)

Additionally, all the developed code is available and documented on GitHub in [20].

[2] https://www.inforum2023.org/
[3] https://recpad2023.isec.pt/

1.4 Document organisation

The remaining document is organised as follows:

- Chapter 2 (State of the Art) approaches the existing knowledge related to this problem. Namely, the Android system, existent malware techniques, how data regarding this problem is acquired, some existing datasets, the data pre-processing and splitting stages, ML classifiers, and evaluation metrics. Finally, some of the approaches found in the literature are summarised and analysed.
- Chapter 3 (Proposed Approach) describes the proposed approach to the problem and the software tools used in its development and implementation.
- Chapter 4 (Experimental Evaluation) presents the testing environment, an analysis of the used datasets, the results obtained from the various experiments and a comparison of these with the ones found in the literature.
- Chapter 5 (Conclusions) contemplates this study's findings and presents a self-assessment resulting in possible improvements.
- Appendix A (Experimental Results) exhibits the complete experimental results.

2

State of the Art

This chapter aims to provide a better understanding of the thesis goals and context by giving an overview of the current state of knowledge about the presented challenge. Its relevance, variants, the existing approaches, and what techniques, algorithms and datasets are considered when applying a ML approach in Android malware detection are addressed.

Section 2.1 provides an overview of the Android system, namely, its architecture and the general composition of its applications.

Section 2.2 presents some examples of malware that targets Android applications, its motivation and goals, existing security measures, and some techniques malware developers use to contour them.

Information regarding data acquisition, namely, the different types of analysis and the existing datasets, is provided in Section 2.3.

Section 2.4 provides an overview of some ML techniques concerning data pre-processing and data splitting.

Section 2.5 introduces some ML algorithms considered in the approach to this problem.

Section 2.6 presents some evaluation metrics used to validate the ML approaches.

Lastly, Section 2.7 contextualises the problem in the ML scope, namely, it summarises the results from the literature of the application of ML approaches, as well as some DL approaches to this problem.

2.1 Android

Android is an open-source OS, based on the Linux Kernel and mainly designed for touchscreen mobile devices such as smartphones and tablets, and first launched in 2008 [65]. Since then, it has had many versions, with a new major Android release occurring every few months. New features are added from version to version, aiming for better performance, security, user experience and an increased number of functionalities.

To better understand how malware can exploit the Android system, basic knowledge regarding the structure of an Android application and the Android OS is needed. Knowing what files malicious software uses to exploit the Android system makes it easier to speculate what files should be scrutinised. Thus, this section provides an overview of the Android OS architecture and the composition of an Android application.

2.1.1 Android architecture

The Android architecture [46], as shown in Figure 2.1, comprises many components. The primary layers of the architecture are, very superficially, described next.

Android is based on a modified version of the **Linux Kernel**, the base of the OS. The Kernel provides critical security features to the OS, which include application sandboxing and process isolation. As such, applications are run in dedicated processes, and each application has a specific data directory, which can only be read or written by the application, with no other application having permission to access it. It is also responsible for device drivers, power, memory and device management.

Next, the **Hardware Abstraction Layer (HAL)** consists of several interfaces that allow the higher-level Java Application Programming Interface (API) framework access to different hardware components, such as the camera or Bluetooth module with a library module from HAL being loaded by the OS.

Android Runtime (ART), as the name suggests, is a virtual machine where the Android apps run and convert Java/Kotlin code to Dalvik Executable (DEX) bytecode format, which is more compact and efficient than class files, considering an Android mobile device's limited memory and battery power. Together with other resources like libraries, DEX files are then packed into Android Package Kit (APK) files [28].

Native C/C++ Libraries are essential since some Android system components are built from native code, including HAL interfaces and ART.

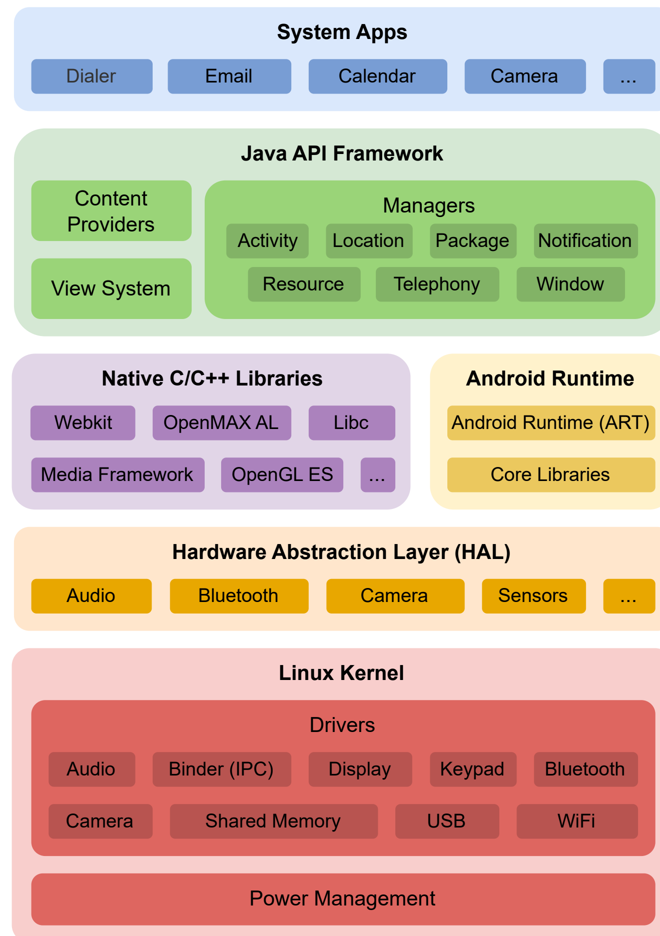

Figure 2.1: The Android software stack (inspired by Figure 1 in [53]).

The **Java API framework** layer provides the API with the building blocks for any Android application. One of the most essential parts of the Application Framework layer is the Activity Manager, responsible for controlling the life cycle of applications [28].

The **System Apps** layer holds core applications pre-installed in Android, such as calendars, emails and browsers.

2.1.2 Structure of an Android application

The Android build system is organised around a specific directory tree structure. Each Android project consists of key elements included in the root directory. Figure 2.2 exhibits the structure of an Android application.

Synthesising some of the depicted components [13]:

- **AndroidManifest.xml -** Extensible Markup Language (XML) file that describes essential information (such as name, version, and contents of the APK file) about

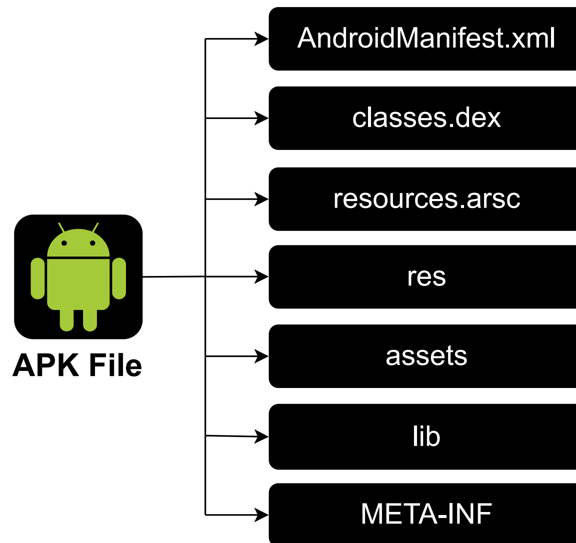

Figure 2.2: Structure of an Android application (inspired by Figure 1 in [66]).

the app. Among others, this file declares the components of the app (activities, services, receivers, and providers), with each component defining basic properties such as the name of its Kotlin/Java class. It also contains the **permissions** the app requires to access restricted parts of the system or other apps and the hardware and software features it requires [14].

- **classes.dex -** contains classes compiled in the DEX file format executed by the ART. It essentially contains the application logic, the app's source code compiled into DEX format.
- **resources.arsc -** binary file that contains precompiled application resources.
- **res -** directory including all non-precompiled resources (not compiled into the 'resources.arsc' file) that the app needs in runtime, such as images and layout.
- **assets -** directory containing the raw resource files that developers bundle with the app, this is, the application's assets, which the AssetManager can retrieve.
- **lib -** directory with the compiled native libraries the app uses, organised in many directories, one for each supported processor architecture.
- **META-INF -** directory containing APK metadata (such as the signature), the AndroidManifest.xml file and the list of resources.

This brief overview of the Android architecture and the structure of an Android application allows for a better comprehension of some of the critical security aspects of the

Android system. For instance, **the 'AndroidManifest.xml' file is of significant relevance in determining if an app is malicious** as it will be explained in Section 2.3.1.

2.2 Malware on Android applications

With the rapid growth of Android devices and applications, the Android environment faces more security threats. Besides the increasing number of attacks, malware also takes many forms, resulting in various types. Malware often reaches the victim's device by luring users (for example, through social engineering) to install applications containing malware [8]. Besides this, malware can explore system vulnerabilities, design weaknesses and security flaws in many Android applications, thus threatening end-users.

To provide a clearer understanding of the wide variety of malware, some popular examples of types [3] of malware are:

- **Remote Access Trojan (RAT)** - Mainly used to gain access to the device and leak data from it. It is a type of phishing that also allows hackers to turn the user's device into a bot. Then, it is possible to form botnets with several infected devices, which are used to execute different malicious attacks, such as Distributed Denial-of-Service (DDoS) attacks [6].

- **Banking Trojans** - Targets mobile banking apps and aims to collect login information and details to transfer funds to anonymous accounts owned by the malware authors.

- **Ransomware** - Encrypts all the mobile device data of the user, and then, the attackers demand payment.

- **Adware** - Aims to infect the device and generate revenue for the attackers by leading the user to click on unwanted advertisements.

- **Spyware** - Collects data without the user's consent and knowledge. It can perform different types of operations that violate the privacy of personal information, such as recording the keys pressed by the user (useful to obtain credentials) or keeping a record of the web pages viewed and monitoring the searches made on the internet to find more information about the victim. This can result in illegal actions such as stealing people's email and bank credentials or in "personalised advertising" pop-ups, spam and scams.

- **Scareware** - Manipulates victims (using social engineering) into downloading or buying malicious software. It achieves this by claiming the existence of a virus or other issue on the victim's device and that the software (containing malware) will resolve the problem. If it successfully tricks the user, the scammer may gain access to personal information, putting the user at risk of identity theft or other forms of fraud [64].

- **Premium Text (SMS)** - Common attack where people register to a premium Short Message Service (SMS) without their knowledge. This may cause monetary loss since these SMS charge more than normal text messages [46].

There are already well-documented cases of Android malware families which are even named, such as *ExpensiveWall*, *HummingBad*, *FalseGuide*, *Judy*, *AdultSwine*, *Chamois*, among others. These are embedded or hidden in dozens of apps on app stores and then downloaded by millions of users. For instance, in early 2017, a new modified version of the *HummingBad* malware was discovered, hidden in over 20 apps on the Google Play Store, which over 12 million users downloaded before Google removed them. Another example is *AdultSwine*, which was embedded in around 60 apps on Google Play Store and could steal banking credentials [8].

2.2.1 Existing security measures

As mentioned in Section 2.1.1, since the Android OS is Linux-based, the Kernel already provides some key security features to the Android system, **application sandboxing** and **process isolation**. With these, applications are run in dedicated processes. Each application is given a specific data directory which can only be read or written by the application, and no other has permission to access it [46].

App stores have security mechanisms in place to detect and remove malicious apps. With pre-publication checks and ongoing monitoring being performed. One of these security mechanisms is, for example, Google Play Protect, a security feature built into the Google Play Store that scans apps for malware and other security threats before they are downloaded.

Other security measures to mitigate malware attacks include device encryption, the use of a secure Virtual Private Network (VPN), the installation of anti-virus apps, maintaining the apps and OS up-to-date and user awareness, for example, when it comes to responding to permission requests made by the Android apps and downloading apps from any app store.

2.2.2 Malware techniques to avoid detection

The measures previously presented to prevent malware attacks are widely used and, to some extent, successful. However, many malware apps can easily bypass them, with malware developers constantly searching for ways to contour them. As such, malware only grows in both sophistication and diffusion.

For example, Chamois malware [21] had its tactics improved to contour security measures. When Google Play Store scanning tools became more efficient and effective at recognising it, its later versions switched tactics, tricking app developers and device manufacturers into incorporating the code directly into their apps.

To avoid detection, installed malware can attempt to download additional stages of its attack chain [29]. With this technique, the malware splits its functionality to remain as stealthy as possible, further complicating the detection of the malware.

Some sophisticated malicious apps can recognise if they are being executed in emulated environments by checking for the presence of specific system properties or characteristics that are unique to the emulator and avoid detection [39]. Some can also recognise if antivirus software is running by checking for specific processes, services or files that are known to be associated with antivirus software. Others can check for active firewalls or other security-related system components.

Due to these challenges, researchers and developers have, in the last years, been studying ML (and Deep Learning) approaches focused on detecting malware on Android applications. Moreover, some approaches focus on vulnerability analysis to prevent malware attacks [6].

2.3 Data acquisition

This section provides some understanding regarding the different types of analysis used to extract the features from Android applications and insight into some of the standard datasets for Android malware detection found in the literature.

2.3.1 Type of analysis

The type analysis (for feature extraction) to find malware in Android applications can be organised into three types: static, dynamic, and hybrid. From the literature, as depicted in Figure 2.3, it can be concluded that static analysis is much more popular

than the other two. The second most popular is dynamic analysis, followed by the hybrid analysis.

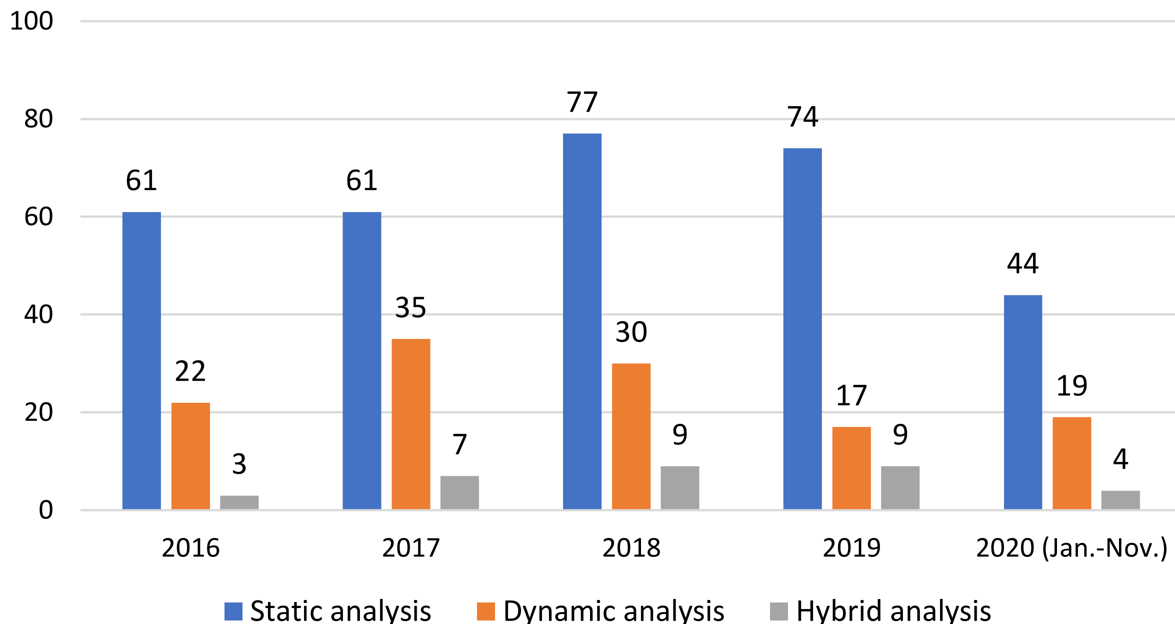

Figure 2.3: Statistics regarding the type of analysis used in ML-based Android malware detection papers between 2016 and 2020, extracted from [66].

Static analysis

In static analysis, the application is examined and analysed in a non-runtime environment. This can be performed by unpacking and reverse engineering an APK file and then scanning the produced code file(s) for critical information.

There are available tools for static analysis, such as APKtool[1] and Androguard[2]. Both of them can parse the 'AndroidManifest.xml' file and parse DEX files, enabling extraction of features from both [46]. From the 'AndroidManifest.xml' file analysis, features such as permissions, intents, activities, services, and providers can be extracted. Meanwhile, with code analyses, API calls, information flow, opcodes, native code, and others can be extracted [39].

Since in static analysis, the application is not run on an emulator or an actual device, it is generally a quicker and more straightforward type of analysis to perform than the other two. Another significant advantage to dynamic analysis is full code coverage, which can cover all code and resource files. While dynamic analysis can hardly cover all code execution paths, resulting in an incomplete feature extraction [66]. Static analysis has proven efficient, although it tends to be ineffective against code obfuscation

[1] https://ibotpeaches.github.io/Apktool/, accessed on 25/10/2023
[2] https://github.com/androguard/androguard, accessed on 25/10/2023

and encryption techniques [39].

Dynamic analysis

Dynamic analysis adopts the opposite approach to static analysis, taking place during the app's regular operation when it is running. Features are extracted by, for example, analysing network traffic, system calls, system resources and other behaviours while the application runs in a monitored environment [39].

Often, dynamic analysis requires more computational power than static analysis [46]. However, as an advantage, malware detection models fed with additional features extracted through dynamic analysis can typically cope significantly better with more recent and challenging malware.

Overall, there is a lack of up-to-date tools for dynamic analysis, representing a significant issue when considering this approach. Hence, not being a prevalent approach among the existent studies. For example, DroidBox[3] uses Android 4.1.2 (a 2012 version), and it has not been updated. As such, it is incompatible with the more recent versions. Due to this and the rapid changes in the Android OS with a new version every couple of months, the results are unlikely to be accurate even if a study uses this tool.

Hybrid analysis

Hybrid analysis incorporates static and dynamic analysis. As it enables the extraction of static and dynamic features, in theory, it should be the best approach, and it is reported [46] to present high accuracy rates. However, as with dynamic analysis, researchers are discouraged by the time and computational resources it requires and its complexity, leading it to be the less popular approach.

2.3.2 Datasets

Several standard datasets for malware detection in Android applications are mentioned in the literature. Some of the most commonly used [66] are, for example, Drebin, MalGenome, VirusShare, Android Malware Dataset (AMD), AndroZoo, Contagio, and CICAndMal2017. Among these, it is worth highlighting the Drebin dataset, which is quite frequently used in the literature. Alkahtani and Aldhyani [6], Algahtani *et al.* [8], Kouliaridis and Kambourakis [39], Muzaffar *et al.* [46] and Wu *et al.* [66] have considered it.

[3] `https://github.com/pjlantz/droidbox` accessed on 25/10/2023

Unfortunately, in many cases, it is not easy to obtain the standard datasets referenced in the literature. Often, the access is restricted, involving payment or authorisation. In other cases, the sources might not be trustworthy, resulting in security warnings when attempting to obtain the datasets.

The authors Alkahtani and Aldhyani, previously mentioned, used two datasets (Drebin and CICAndMal2017) in their study [6]. These are available in [25] and [22], respectively, on the Kaggle[4] website, which allows access to a variety of datasets for different problems. To compare results with their study, these two datasets seemed appropriate to perform experiments.

The Drebin dataset [25], or 'Android Malware Dataset for Machine Learning' as described in Kaggle, was first published in 2014. It contains 215 features extracted from 15036 applications, with 9476 benign apps and 5560 malware apps from 179 different malware families.

The CICAndMal2017 dataset [22], or 'Android Permission Dataset' as described in Kaggle, was developed by the Canadian Institute[5] and published in 2018 [6]. It contains 183 features and 29999 instances extracted from several sources. Most benign samples were applications published between 2015 and 2017 from the Google Play Store. The malware samples can be organised into four categories: Adware, Ransomware, Scareware and SMS Malware. In total, it contains data from 42 unique malware families.

Additionally, further exploring the Kaggle website, two more datasets, the Android Malware dataset and the Android Malware static feature dataset, available in [11] and [12], respectively, were selected.

The Android Malware dataset [11] was developed by Martín *et al.* in 2016, in the context of their study [42]. The dataset contains meta information of benign and malware Android samples. The authors gathered apps from the Aptoide[7] app store and then analysed each one using an API provided by the VirusTotal[8] online portal, to analyse the apps with 56 different antivirus engines. The results obtained from the VirusTotal website were used to label all the samples, with an app considered malicious if a single antivirus gave a positive detection. The dataset possesses 183 features and 11476 instances. This dataset will be mentioned in this thesis with an acronym, Android Malware (AM), for simplification.

The Android Malware static feature dataset [12] is divided into six datasets, each in

[4]https://www.kaggle.com/
[5]https://www.canadianinstitute.com/
[6]https://www.unb.ca/cic/datasets/andmal2017.html, accessed on 25/10/2023
[7]https://www.aptoide.com/
[8]https://www.virustotal.com/

a different Comma-Separated Values (CSV) file. Each dataset has different features: permissions, intents, system commands, API calls, API packages and opcodes. There are 1062 features, with 38 in the permission dataset, 221 in the opcodes dataset, 93 in the intents dataset, 523 in the API calls dataset, 91 in the API packages dataset and 96 in the system commands. It is a balanced dataset with 2508 benign and 2511 malicious samples, totalling 5019 instances. The benign apps were collected from the Google Play Store and APKPure, while malicious apps were collected from VirusShare. All six datasets were extracted from the same APK files, which enables column-wise merging to obtain a single dataset with all features. This dataset will be mentioned in this thesis with an acronym, Android Malware static feature (AMSF), for simplification.

2.4 Pre-processing

This section provides an overview of some data pre-processing techniques since data generally contains missing values and may be in an unusable format that can not be directly used by the ML models. Preparing the data and making it suitable for the ML model significantly impacts its accuracy and efficiency [23]. A brief overview of data splitting is also provided in this section.

2.4.1 Data pre-processing

Data pre-processing techniques [24] can be generalised and aggregated into four steps: data cleaning, data integration, data reduction, and data transformation.

- **Data Cleaning** includes:
  - **Missing values processing** can be done in various ways, such as discarding the tuples with missing data, imputing the values with a constant or a prediction of the missing values using a ML method (such as Naive Bayes [47]) or with values resulting from numerical methods like the mean, median or mode.
  - **Reformatting the data** involves making data format changes to a standard format to ensure that, for example, making dates have a similar format throughout.

  - **Attribute conversions** can also take place since, for example, some ML techniques require only numerical inputs (such as SVM [60]). Different conversions can be applied, with techniques such as one-hot encoding or label encoding. The first turns each value of the categorical feature into a binary feature, with its main drawback being the resulting increase in dimensionality. The latter enables the preservation of an existing ordinal relationship between the categorical values of a feature. For example, 'weak', 'medium', and 'strong' would be converted to 0, 1, and 2, respectively. However, if the labels have no specific order of preference, it can add unintended bias.
  - Lastly, data cleaning also includes the **identification of outliers and smoothing of noisy data**.

- **Data Integration** is the process of merging data from multiple sources into a single dataset.

- **Data Reduction** techniques aim to derive a reduced representation of the data in terms of volume while maintaining the integrity of the original data. With high dimensional data, the training process is more difficult due to underfitting/overfitting problems. Some of the main strategies for data reduction are dimensionality reduction and numerosity reduction, which include instance sampling.

  - **Dimensionality reduction** diminishes the number of features to be considered. One of the most predominant dimensionality reduction techniques is Principal Component Analysis (PCA) [54] that works by searching for a set of orthogonal vectors, which is smaller than the original feature vectors, that can best represent the data, thus resulting in dimensionality reduction. Thus compounding the original features into an alternative, smaller set. Another example is feature selection, which involves discarding the features considered weakly relevant or redundant and maintaining the relevant features that add more value to the model. This can be achieved, for example, by applying the Relevance-Redundancy Feature Selection (RRFS) filter approach [5]. with the unsupervised Mean-Median (MM) relevance metric given by

$$MM_i = |\overline{X_i} - median(X_i)|, \tag{2.1}$$

with $\overline{X_i}$ denoting the sample mean of feature X_i. We also consider the supervised Fisher's Ratio (FR) relevance metric

$$FR_i = \frac{\left|\overline{X}_i^{(-1)} - \overline{X}_i^{(1)}\right|}{\sqrt{\mathrm{var}(X_i)^{(-1)} + \mathrm{var}(X_i)^{(1)}}}, \tag{2.2}$$

where $\overline{X}_i^{(-1)}$, $\overline{X}_i^{(1)}$, $\mathrm{var}(X_i)^{(-1)}$, and $\mathrm{var}(X_i)^{(1)}$, are the sample means and variances of feature X_i, for the patterns of each class. The redundancy analysis between two features, X_i and X_j, is done with the Absolute Cosine (AC) given by

$$AC_{X_i,X_j} = |cos(\theta_{X_iX_j})| = \left|\frac{\langle X_i, X_j\rangle}{||X_i||||X_j||}\right| = \frac{\sum_{k=1}^{n} X_{ik}X_{jk}}{\sqrt{\sum_{k=1}^{n} X_{ik}^2 \sum_{k=1}^{n} X_{jk}^2}}, \tag{2.3}$$

where $\langle , \rangle$ and $||.||$ denote the inner product and L_2 norm, respectively.

– **Numerosity reduction** involves replacing the original data with a smaller form of data representation. Numerosity reduction includes **instance sampling**, a method that balances imbalanced data.

  To balance imbalanced data, undersampling or oversampling can be performed. Undersampling consists of removing samples of the majority class, yielding information loss. The latter doesn't reduce numerosity since it involves duplicating instances of the minority class (yielding a higher chance of overfitting) but balancing the data. Some techniques [52], such as Synthetic Minority Over-sampling Technique (SMOTE), perform oversampling by creating synthetic data instead of performing a copy of the existing instances.

- **Data Transformation** aims to change the data's value, structure, or format to shape it into an appropriate form. The most widely used techniques are normalisation and discretisation. The first involves scaling attributes to ensure they fit within a specified range. One of the most popular techniques is the Min-max normalisation, given by

$$v' = \frac{v - min(A)}{max(A) - min(A)}(new_max(A) - new_min(A)) + new_min(A). \tag{2.4}$$

Meanwhile, discretisation involves reducing the number of values for a continuous attribute by partitioning the attribute range into intervals to replace actual data values.

2.4.2 Data splitting

In ML approaches, data is typically split into two or three sets: the training, testing, and validation sets. The training set is the portion of data used to train the model. The testing set is the data portion used to test the final model. The validation set is the portion of data used to evaluate the model and help tune its hyperparameters.

Regarding the proportion of data split for each set, the training set should always have a more considerable amount of data since it's the set that will be used to train the model. A typical ratio used in the literature, for example, in the experiments conducted by Alkahtani and Aldhyani [6], is 70-30 for training and testing data, respectively.

Data can be split based on different data sampling methods, such as random or stratified sampling. The first one helps prevent bias but can also present issues such as an uneven distribution of class labels. The latter randomly selects data samples within specific parameters, ensuring a more even distribution between class labels.

Cross-Validation (CV)

CV is a resampling method that splits the data into subsets and rotates their usage among them. For example, considering only training and testing, the subsets in each iteration would all be used for training the model except one, which would be used for testing. The testing fold should be rotated until all subsets have become a testing fold once and only once.

The nested CV strategy uses the training, testing and validation sets and consists of an outer loop and an inner loop. The outer loop deals with the training and testing sets and performs a generalisation error estimated by averaging test set scores over several dataset splits. The inner loop deals with the training and validation sets, with both sets being obtained from the training set of the outer loop. In the inner loop, the score is approximately maximised by fitting a model to each training set and then directly maximised by tuning hyperparameters over the validation set.

There are many types of CV, such as Hold-out CV and Leave-p-out CV. Two of the most widely used are Stratified K-fold CV and Leave-one-out (LOO) CV. In stratified K-fold CV, the data is divided into k-folds (typically ten folds) of approximately the same size with stratified sampling. Figure 2.4 shows an example considering 5-fold CV applied to training and testing sets and the training and validation sets.

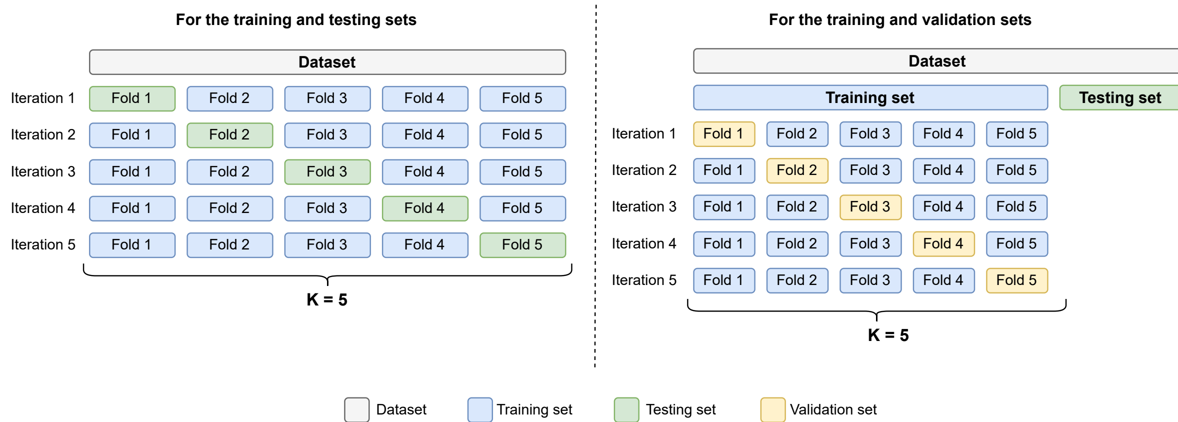

Figure 2.4: Example of 5-fold CV for the training and testing sets and the training and validation sets.

LOOCV is the exhaustive holdout splitting approach that k-fold enhances, being a particular case of k-Fold CV where k is equal to the number of instances (n). In the example depicted in Figure 2.5, each sample is considered the validation set, and the rest (N-1) is regarded as the training set. Since it requires each holdout instance to be tested using a model, it becomes computationally costly.

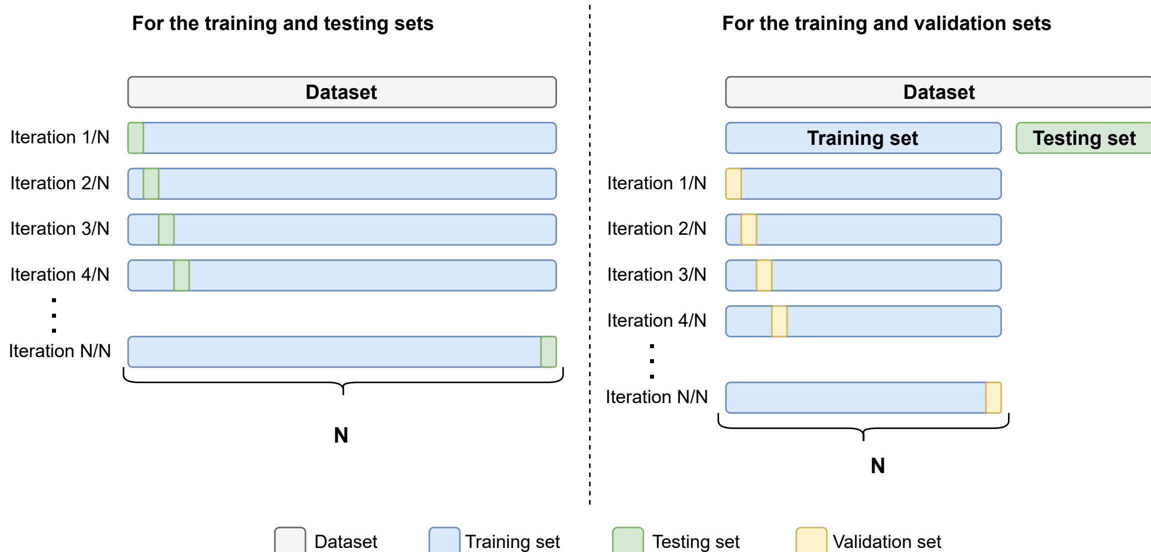

Figure 2.5: Example of Leave-one-out CV for the training and testing sets and the training and validation sets.

2.5 Classifiers

This section briefly introduces some of the literature's most used ML algorithms. To understand their placement in the ML branches, a simplified representation of the ML branches is presented in Figure 2.6.

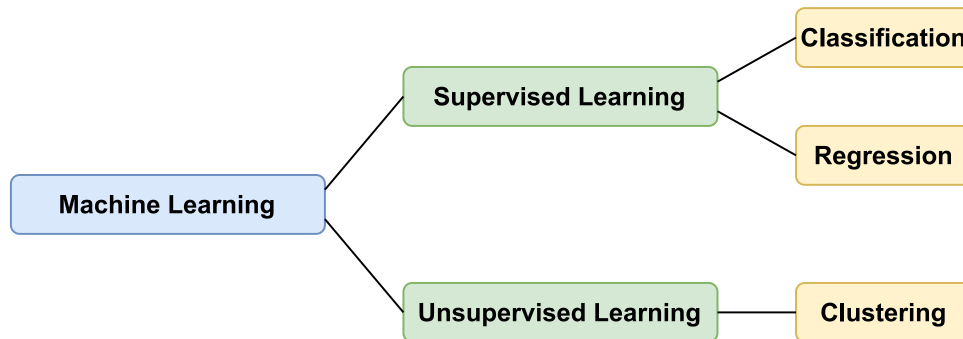

Figure 2.6: Simplified representation of the ML branches.

First, ML techniques can be organised into two major categories: Supervised Learning and Unsupervised Learning. In the first one, the class labels are provided to the model, meaning the model is trained with a labelled dataset. In the latter, the class labels are not provided.

Supervised techniques can be generally segregated into classification and regression techniques. Classification algorithms categorise data into a class or category by predicting a discrete class label. Classification can be binary, multiclass, and multilabel. Regression algorithms predict a continuous variable. In Android malware detection, supervised learning is typically used to train classifier models that can determine whether an unknown application is benign or malware. In some cases, classification is also used to classify malware applications according to their malware families [46].

Regarding unsupervised techniques, these usually lead to the clustering approach (although it can also lead to association and dimensionality reduction), where the unlabelled data is grouped based on their similarities or differences.

2.5.1 Random forest

RF, is a supervised ML algorithm that can be used for classification and regression problems. It is considered an ensemble method since it combines the output of multiple Decision Tree (DT) to reach a single result (as shown in Figure 2.7).

To better understand the RF algorithm, the DT algorithm will be briefly described. DT apply a succession of conditions to the data. These are the decision nodes of the tree,

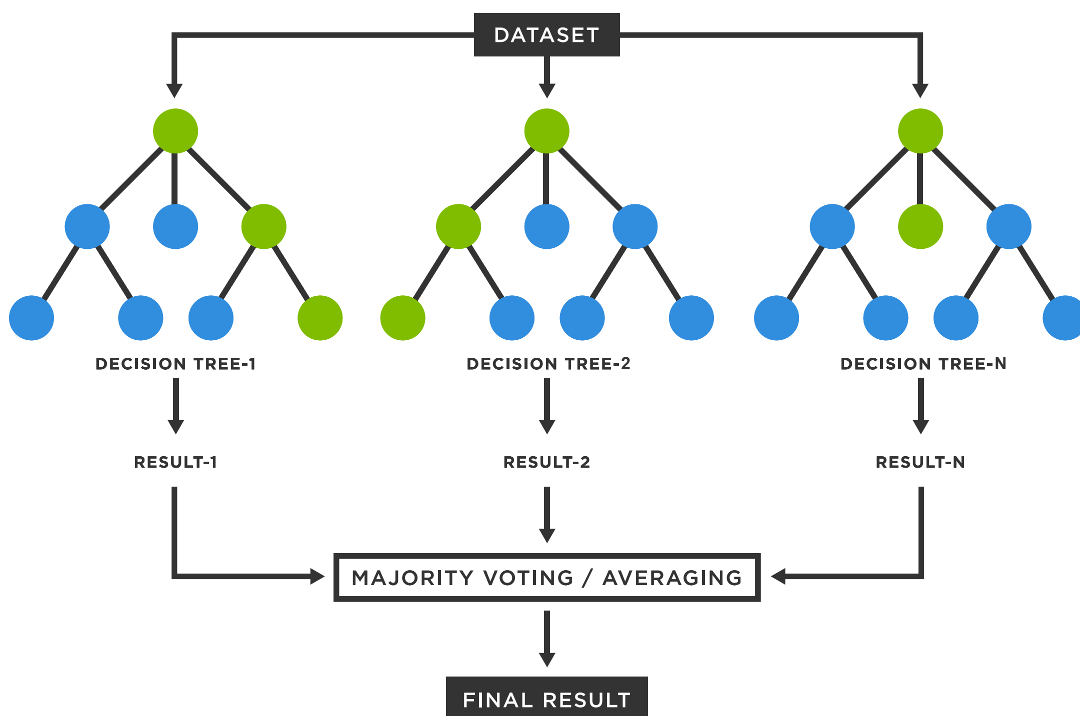

Figure 2.7: Random Forest schema (extracted from [62]).

and their purpose is to split the data. Each node is a stepping stone to arrive at a final prediction, denoted by the leaf node. Data that fits the decision node criteria will follow the respective branch, and those that don't will follow the alternate path(s) until a leaf node is reached [61].

In the RF algorithm, all predictions from all the DT are aggregated to identify the result, with the average (in a regression task) or majority (in a classification task) of the model's predictions providing the final prediction.

Some of the most relevant hyperparameters [31], of the RF algorithm are the maximum depth of the trees, the number of trees and the function to measure the quality of a decision node split.

Besides being flexible by being able to solve both regression and classification problems, RF also reduces the risk of overfitting. DT tend to overfit, as they tightly fit all the samples within training data. However, since there is a considerable number of DT in a RF, the classifier won't overfit the model since the averaging of uncorrelated trees lowers the overall variance and prediction error.

The RF algorithm also presents some challenges. Namely, it is time-consuming (data is being processed by each DT) and requires more computational resources.

RF is one of the more used ML algorithms mainly because it can be used for classification and regression tasks, combined with its nonlinear nature, making it highly

adaptable to a range of data and situations. In the literature, the use of RF is prevalent among the studies related to Android malware detection, being one of the models that present the best results, as reported in Section 2.7.2 of this chapter.

2.5.2 Support vector machine

SVM is a supervised ML algorithm for classification and regression problems. It aims to maximise the predictive accuracy of a model without overfitting the training data. It works by mapping data to a high-dimensional feature space so that data points can be categorised, even when the classes are not otherwise linearly separable. A separator between the categories is found, and then the data is transformed so that the separator can be drawn as a hyperplane. SVM tries to find a hyperplane that best splits a dataset into two classes. The data will continue to be mapped into higher and higher dimensions until a hyperplane can be formed to segregate it and transform the linearly inseparable data into linearly separable data. The SVM algorithm uses Kernel functions (chosen as an hyperparameter), with the linear and Radial Basis Function (RBF) being the more widely used.

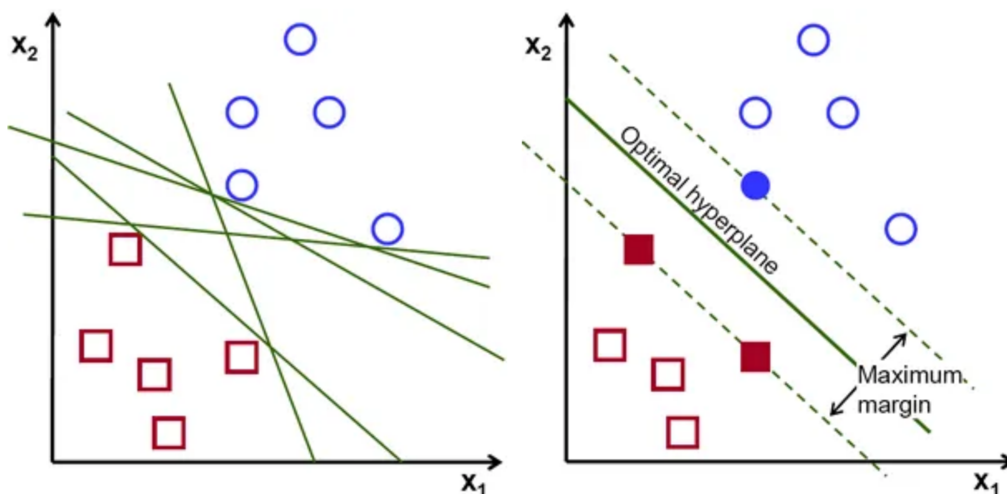

Figure 2.8: Example on finding the hyperplane, with the possible hyperplanes on the left and then the optimal hyperplane (that allows for the maximum margin) on the right (extracted from [60]).

The distance between the hyperplane and the nearest data point (called support vector) from either set is known as the margin (depicted in Figure 2.8). The goal is to choose a hyperplane with the broadest possible margin between the hyperplane and any point within the training set. This increases the chance of new data being classified correctly [59]. Intuitively, the farthest from the hyperplane the data points lie, the more

confident we are that they have been correctly classified. Therefore, the ideal is for the data points to be as far away from the hyperplane as possible while still being on the correct side of it. When new testing data is added, whatever side of the hyperplane it lands will decide the class assigned.

SVM works well with high-dimensional data, can handle non-linearly separable data and is relatively memory efficient. However, it does not perform very well when there is noise in the data.

2.5.3 K-Nearest Neighbours

K-Nearest Neighbours (KNN) is a supervised algorithm that can be used for classification and regression problems [36]. For classification, KNN classifies a data point by a plurality vote of its neighbours, with the data point being assigned to the class most common among its k nearest neighbours (k is a positive integer, typically small).

Figure 2.9 briefly exemplifies how the classifier works. As shown in (a), there are data points from class A (in green) and data points from class B (in blue), as well as a data point we aim to classify. In (b) with k=3, and according to the distance metric chosen, the three closest data points are considered. Two are from class B, and one is from class A. Thus, the data point is classified as class B. Meanwhile, in (c) with k=7, the closest data points are three from class B and four from class A. Thus, the data point is classified as class A.

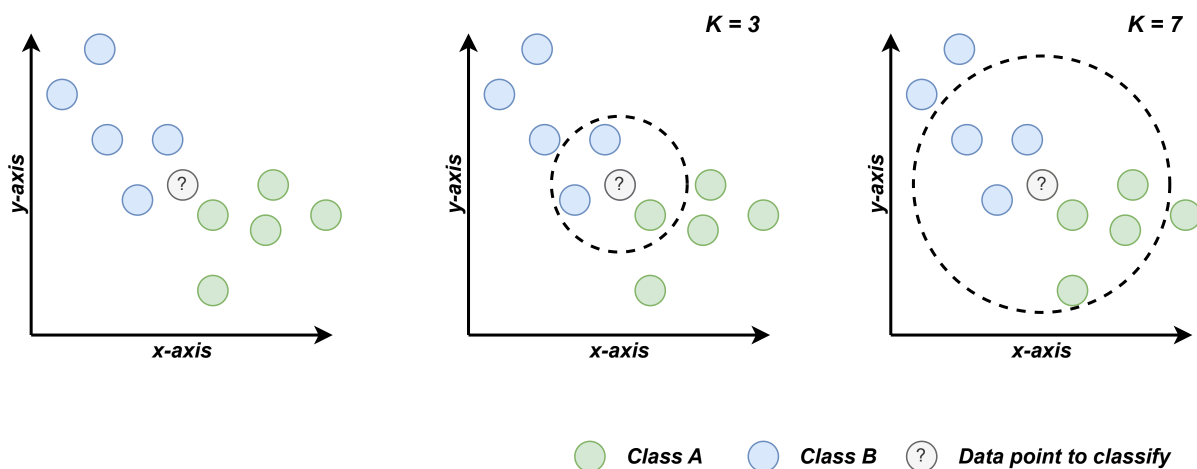

Figure 2.9: Prediction examples for the KNN classifier, with k=3 in (b) and k=7 in (c), to classify the data point in (a).

The KNN classifier adapts quickly to new data, since all training data is stored in memory, and has few hyperparameters, with the most relevant ones being the number of

neighbours (k), the distance metric (such as Euclidean, Manhattan or Hamming distance), and the weight function used in prediction.

Regarding the number of neighbours, generally, a larger value reduces the effect of noise on the classification but makes boundaries between classes less distinct [36]. Additionally, it should not be a multiple of the number of classes to avoid ties.

KNN also presents some disadvantages, such as not scaling well since it takes up more memory and data storage than other classifiers. Additionally, it tends [37] to fall victim to the curse of dimensionality and, due to this, is also prone to overfitting.

2.5.4 Naive Bayes

Naive Bayes (NB) is a supervised ML algorithm used for classification tasks [26]. It is a probabilistic classifier based on Bayes' Theorem [19], and it relies on incorporating prior probability distributions to generate posterior probabilities. It is given by

$$P(A|B) = \frac{P(B \cap A)}{P(B)} = \frac{P(B|A) \times P(A)}{P(B)}. \tag{2.5}$$

Where

$P(A)$ = Probability of event A

$P(B)$ = Probability of event B

$P(B \cap A)$ = Probability of both A and B occurring

$P(A \mid B)$ = Conditional probability of A given B

$P(B \mid A)$ = Conditional probability of B given A

There is more than one type of NB classifier. The most popular types differ based on the distributions of the feature values. Some of these are the Gaussian NB, which is more appropriate for continuous data that follows a normal distribution; the Multinomial NB, which is more suitable for discrete data representing counts or frequencies; the Bernoulli NB, which is more appropriate if the data is binary; the Categorical NB, which is more appropriate if the data is discrete and represents unordered categories.

Compared to other classifiers, NB is a simpler classifier since the parameters are easier to estimate. It is considered a fast and efficient classifier that is reasonably accurate when its conditional independence assumption holds. It also has low storage requirements and can handle high-dimensional data.

However, it also presents disadvantages, namely, the classifier is considered 'naive' since it makes unrealistic assumptions about the data. Namely, it assumes that each

input variable is independent and equally relevant. Thus, when these assumptions are not held, it leads to incorrect classifications.

2.5.5 Multi-layer Perceptron

The Multi-layer Perceptron (MLP) classifier is a DL algorithm, often used to solve complex problems stochastically, allowing approximate solutions to challenging issues.

MLP consists of three or more layers (an input and an output layer with one or more hidden layers) of perceptrons, as shown in Figure 2.10. Each perceptron in one layer connects with a certain weight to every perceptron in the following layer, thus being fully connected. Each also has a function that maps the weighted inputs to the output of each perceptron. Learning occurs in the perceptron by changing these connection weights after each piece of data is processed based on the amount of error in the output compared to the expected result [44]. Thus, it is supervised learning carried out through backpropagation [63].

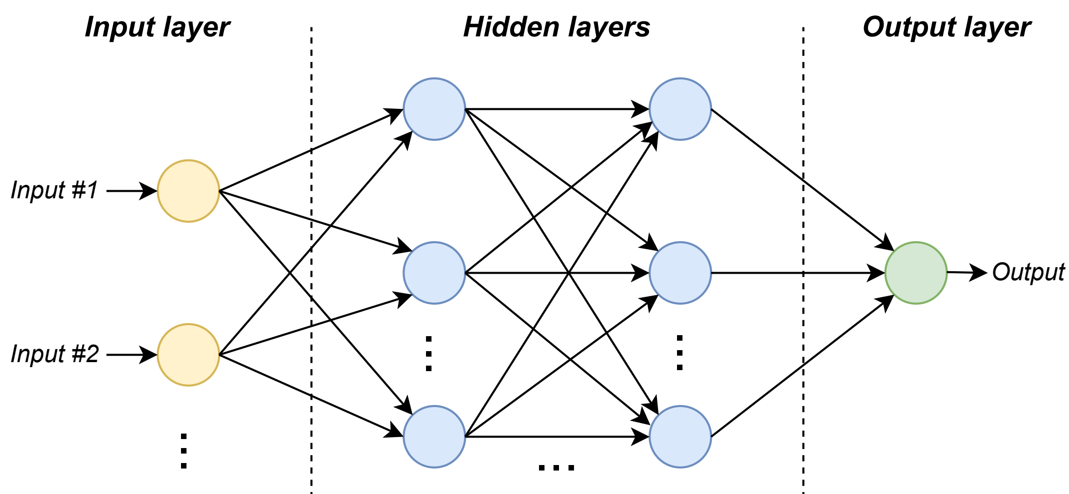

Figure 2.10: Multi-layer Perceptron (MLP) schema.

MLP has the advantage of learning non-linear models, the ability to train models in real-time (online learning), handling large amounts of input data and making quick predictions after training. However, it is more computationally costly than other classifiers and is sensitive to feature scaling [2].

2.6 Evaluation metrics

To build an effective ML model, it is essential to evaluate its performance with different evaluation metrics. These allow an assessment of how well the model performed for

the given data. Not all metrics apply to all types of problems (for example, regression or classification). The focus is on metrics appropriate to classification problems. As such, evaluation metrics such as Mean Squared Error (MSE) and Mean Absolute Error (MAE) [56], adequate for regression and not for classification problems, are not taken into consideration. The evaluation metrics considered are briefly described next.

Confusion matrix

The confusion matrix metric is widely used for classification problems with two (binary) or more classes (multiclass). It consists of a simple N x N matrix/table, where N is the number of target classes, being the same for the two existing dimensions ('actual' and 'predicted' values). It compares the actual target values with those predicted by the ML model under evaluation.

In this thesis, it is important to consider the following terminology:

- **False Positive (FP) -** The actual value was negative (-) but the model predicted a positive (+) value. It is also known as Type 1 Error.

- **False Negative (FN) -** The actual value was positive (+) but the model predicted a negative (-) value. It is also known as Type 2 Error.

- **True Positive (TP) -** The actual value was positive (+), and the model predicted a positive (+) value.

- **True Negative (TN) -** The actual value was negative (-), and the model predicted a negative (-) value.

These also form the base for other metrics. Assuming a simple binary classification problem, there are two classes: positive (1) and negative (-1), leading to a 2 x 2 matrix, with the four combinations resulting from the predicted (rows) and actual values (columns), similar to the one represented in Figure 2.11.

Predicted Values	Actual Values: Positive (1)	Actual Values: Negative (-1)
Positive (1)	TP	FP
Negative (-1)	FN	TN

Figure 2.11: Confusion matrix for binary classification.

Accuracy

The accuracy evaluation metric conveys the fraction of correct predictions made by the model. It is given by

$$Accuracy = \frac{Number\ of\ Correct\ Predictions}{Total\ Number\ of\ Predictions}. \tag{2.6}$$

Assuming a binary classification problem, the accuracy can be computed as

$$Accuracy = \frac{TP + TN}{TP + TN + FP + FN}. \tag{2.7}$$

Although it is the most used evaluation metric throughout the literature, it is important to notice that, in some cases, it can be misleading, for example, in cases of severely imbalanced data.

Precision

Precision, also known as positive predictive value, is the proportion of positive predictions that, among all positive predictions, are truly positive. It is defined as follows

$$Precision = \frac{TP}{TP + FP}. \tag{2.8}$$

Recall

Recall, also known as true positive rate or sensitivity, corresponds to the proportion of positive predictions that are correctly considered positive concerning all positives. It is given by

$$Recall = \frac{TP}{TP + FN}. \tag{2.9}$$

F1-score

The F1-score is an evaluation metric achieved by the harmonic mean of the precision and recall metrics. Thus, the higher the precision and recall values, the higher the F1-score value. F1-score is defined as follows

$$F_1\text{-score} = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}. \tag{2.10}$$

AUC-ROC

The Receiver Operating Characteristic (ROC) is a probability curve for different classification thresholds, while the Area Under the Curve (AUC) represents the area under it corresponding to the degree or measure of separability. As shown in Figure 2.12, the higher the AUC, the better the model distinguishes between class labels. Thus, when AUC is one, we have a perfect classifier since True Positive Rate (TPR) is one and False Positive Rate (FPR) is zero. Meanwhile, the model cannot distinguish between classes if AUC is approximately 0.5. Thus behaving as a random classifier. If the AUC value is close to zero, the model provides the exact opposite predictions (a negative class as a positive class and vice versa).

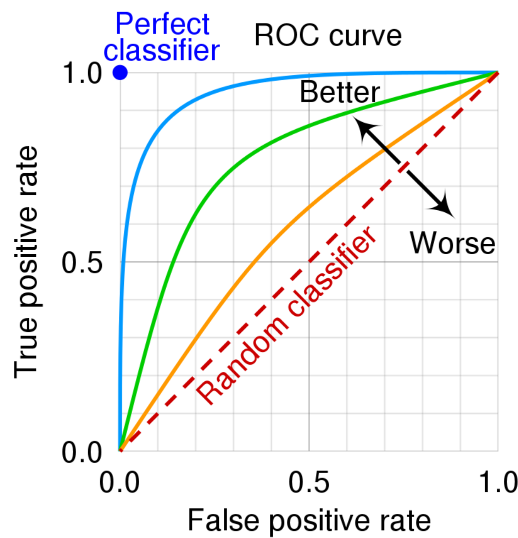

Figure 2.12: The ROC space for a "better" and "worse" classifier, extracted from [55].

The ROC curve is plotted with TPR (or recall) on the y-axis and FPR on the x-axis. FPR is defined as follows

$$FPR = \frac{FP}{TN + FP}. \tag{2.11}$$

2.7 Android malware detection using machine learning

This section provides an overview of some of the ML approaches found in the literature for malware detection in Android applications.

2.7.1 Deep learning approaches

DL is a branch of ML based on ANN. Although DL techniques are not the focus of this research, it is important to mention their satisfactory results in detecting malware on Android applications.

Alkahtani and Aldhyani [6] applied the Long Short-Term Memory (LSTM), Convolutional Neural Networks (CNN) with LSTM (CNN-LSTM) and Autoencoder (AE) algorithms, which were successful in identifying malware in mobile environments. The authors report that the CNN-LSTM and LSTM algorithms provide the most satisfying results. The LSTM model achieved an accuracy of 99.40% using the Drebin dataset, and the CNN-LSTM model achieved an accuracy of 95.05% with the CICAndMal2017 dataset.

Wu *et al.* [66] as well as AlOmari *et al.* [7] provided insight to many DL approaches found in the literature. Table 2.1 summarises some of the results from these approaches. The most popular DL algorithm in the literature is CNN, and, in general, we can conclude that DL approaches provide good results at the expense of computational time and resources.

Table 2.1: Summary of some of the results of DL approaches for Android malware detection found in the literature.

Reference	Year	DL classifier	Dataset	Accuracy (%)
[6]	2022	LSTM	CICAndMal2017	94.58
[6]	2022	CNN-LSTM	CICAndMal2017	95.07
[6]	2022	AE	CICAndMal2017	75.79
[6]	2022	LSTM	Drebin	99.40
[6]	2022	CNN-LSTM	Drebin	56.65
[6]	2022	AE	Drebin	75.79
[9]	2020	Bidirectional LSTM (BiLSTM) LSTM CNN Deep Belief Networks (DBN)	AMD Debrin VirusShare	99.90
[27]	2017	CNN	MalGenome Debrin Apk mirror Apk4fun	93.00
[40]	2018	DBN	Debrin Google Play	90.00
[41]	2020	BiLSTM	AMD Google Play	97.22
[48]	2017	CNN	Contagio Third-party app stores	99.40
[68]	2018	CNN	Debrin Chinese app markets	97.40

2.7.2 Summary of the existing approaches

Alkahtani and Aldhyani [6] applied the SVM, KNN and Linear Discriminant Analysis (LDA) algorithms to identify malware in mobile environments, using two standard Android malware applications datasets: CICAndMal2017 and Drebin. SVM achieved a 80.71% accuracy with the Drebin dataset. The results with the CICAndMal2017 dataset were especially positive, with the authors claiming an accuracy of 100%. Overall, it was shown that the SVM algorithm successfully detects malware. Regarding KNN, it achieved 81.57% with the Drebin dataset and 90% with the CICAndMal2017 dataset. It successfully detected malware, but SVM was still more effective. Lastly, the authors applied LDA, and overall, the results could have been more adequate since nonlinear algorithms are not the most appropriate choice for this problem. The accuracy of LDA

was 45.32% in the CICAndMal2017 dataset, a percentage that reached 81% with the Drebin dataset.

Muzaffar *et al.* [46] concluded that many existent studies cite high accuracy rates in identifying malware. However, there are issues with existing approaches that may limit their real-world performance. These include the widespread use of outdated datasets and inappropriate and/or incomplete metrics that may give a misleading view of performance.

Kouliaridis and Kambourakis [39] attempted to schematise the so far ML-powered malware detection approaches. Based on the surveyed works, the authors concluded the following:

- Static analysis is the most common approach.
- Most publicly available and standard datasets are not recent or up-to-date.
- The ML techniques commonly applied are base models, followed by ensemble learning.
- The most popular evaluation metric is accuracy.

The authors also analysed a number of studies from 2014 to 2021 regarding the used base classification model. They concluded that the RF algorithm is the most used in the literature, followed by SVM.

Wu *et al.* [66] explored the statistics of static feature usage in ML-based Android detection research papers from January 2019 to November 2020. The authors concluded that the most used features are native opcodes, raw APK files, intents and permissions. In addition, they provided some insight into the most popular datasets (such as Drebin and MalGenome) and the types of analysis used in the literature. They also presented statistics showing the most used ML algorithms for Android malware detection from January 2019 to November 2020. They concluded that the most popular was SVM followed by RF and KNN.

Keyvanpour *et al.* [35] conducted experiments with the Drebin dataset and proposed embedding Feature Selection (FS) in a learning model. The authors opted for effective FS with RF. They also conducted tests with other classifiers, such as KNN and NB. The authors concluded that RF outperformed other models based on several metrics. Based on the FS method, the RF algorithm employs the selected features to detect malicious applications. FS was shown to improve the performance of the RF classifier.

Islam *et al.* [32] utilised the CCCS-CIC-AndMal2020 dataset, with 12 major malware categories, 53439 instances, and 141 features. Concerning pre-processing, the authors performed missing data imputation using the 'mean' strategy. SMOTE was applied to deal with class imbalance. Additionally, they used the Min-max normalisation and transformed the categorical data into numerical data via one-hot encoding. To perform FS, the authors applied Recursive Feature Elimination (RFE), discarding 60.2% of features. The authors concluded that the reduced set of features lessened the complexity and improved the accuracy. Their approach was based on multi-classification and dynamic analysis, with an ensemble ML approach with weighted voting that incorporates RF, KNN, MLP, DT, SVM, and Logistic Regression (LR). Their proposed weighted voting method showed an accuracy of 95%.

AlOmari *et al.* [7] proposed a multi-classification approach with five classes: Adware, Banking Malware, SMS Malware, Mobile Riskware, and Benign, using the CICMalDroid2020 dataset. The dataset contains 11598 instances and 470 features. Regarding pre-processing, the authors applied z-score normalisation. They used SMOTE and PCA, concluding that SMOTE and z-score normalisation improved the results, while PCA was not beneficial. Their approach was based on the Light Gradient Boosting Model (LightGBM), but they also analysed the performance of several other algorithms, such as KNN, RF, DT, LDA, and NB. The LightGBM showed the best accuracy and F1-score values, achieving 95.49% and 95.47%, respectively.

Finally, Table 2.2 summarises some of the results of ML approaches for Android malware detection found in the literature. These results were obtained from the literature surveyed, with some studies referencing others.

To conclude, regarding the used datasets, as mentioned in section 2.3.2, the Drebin dataset is the most popular. Regarding ML algorithms for the detection of malware in Android, RF and SVM are present in the majority of the studies, achieving the best results, at least in accuracy.

Aside from these approaches, there are many others. For example, online learning [46] has been proposed in various studies. Shaojie Yang *et al.* [67] proposed an Android malware detection approach based on contrastive learning. Pektaş *et al.* [1] proposed Android malware classification by applying online ML. Adebayo and Aziz [4] proposed an improved malware detection model using the A-priori association rule algorithm.

Table 2.2: Summary of some of the results of ML approaches for Android malware detection found in the literature.

Reference	Year	ML classifier	Dataset	Accuracy (%)
[6]	2022	SVM	CICAndMal2017	100.0
[7]	2023	LightGBM	CICMalDroid2020	95.49
[6]	2022	SVM	Drebin	80.71
[6]	2022	KNN	CICAndMal2017	90.00
[6]	2022	KNN	Drebin	81.57
[6]	2022	LDA	CICAndMal2017	45.32
[6]	2022	LDA	Drebin	81.35
[27]	2019	SVM	Drebin	93.70
[33]	2019	RF	MalGenome, Kaggle, Androguard	93.00
[34]	2019	RF (with 1000 DT)	Drebin	98.70
[38]	2018	DT	Drebin	97.70
[43]	2016	RF	Drebin	97.00
[51]	2019	RF	Drebin	94.00
[57]	2019	RF	Drebin	96.70
[69]	2019	RF	Drebin	96.00
[32]	2023	Ensemble method	CCCS-CIC-AndMal2020	95.00

3

Proposed Approach

This chapter presents the proposed approach, providing insight into its development and implementation and the tools used. In Section 3.1, the ML approach is formulated, presenting it step by step. In Section 3.2, the component of our approach using real-world applications is described. Section 3.3 presents the software tools used to develop and implement the proposed approach.

3.1 Machine learning module

The aim is to classify a given Android application as malicious or benign. Thus, the problem is formulated as a **binary classification problem**. A benign app is considered a negative, and a malicious app a positive. Thus, a FP is classifying an application as malicious when it is benign, and a FN is classifying an application as benign when it is malicious. Although both, FP and FN should be mitigated, we consider FN more critical than a FP, given that it is preferable to wrongly label a benign app as malicious and continue safe, rather than to indicate that the app is safe when in fact it has malicious code compromising the security.

Figure 3.1 depicts the first segment of the proposed approach, showing that we use binary classification datasets for which we apply different data pre-processing techniques.

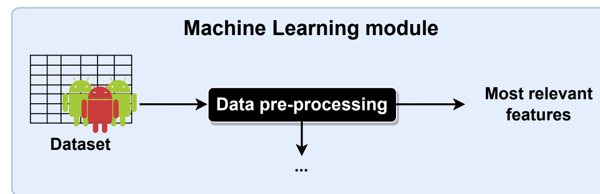

Figure 3.1: Partial block diagram of the proposed approach: the data pre-processing stage, which is composed of handling missing values, numerosity balancing, and FS. The vertical arrow points to the continuation of the ML pipeline, and the right-hand side arrow highlights that the approach identifies the most relevant features for the feature extraction module.

Data on Android apps is obtained from a dataset, such as the Drebin or CICAndMal2017 datasets. Next, data pre-processing, namely, techniques to deal with missing values, for numerosity balancing and FS, are applied (shown in Section 2.4.1) to properly prepare the data and to assess their impact on the performance of the model. Additionally, a set of the most relevant features will be obtained with a FS technique. Figure 3.2 describes the following steps of the proposed approach, after properly preparing the data.

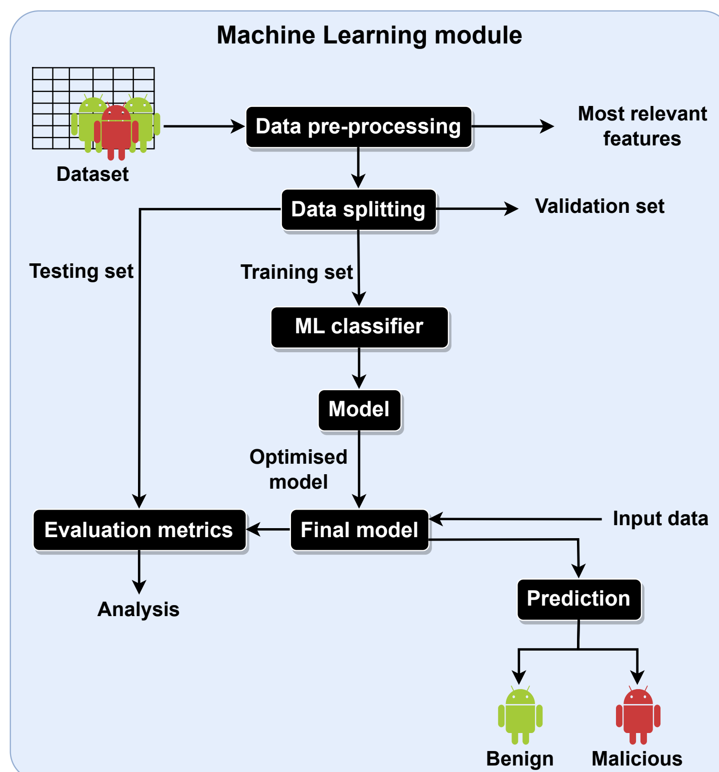

Figure 3.2: Partial block diagram of the proposed approach: data splitting for training and testing of the model with standard evaluation metrics. With a validation set provided to perform hyperparameter tuning. The right-hand side arrow with input data refers to the use of data from real-world applications.

After the data pre-processing stage, three data subsets are obtained from the data splitting action: training, testing, and validation sets. The training set is used to train/learn the model , which employs the ML classifier, that, given input data, can make a prediction, in this case, to classify an app as benign or malicious. The testing set enables the analysis of the model through standard evaluation metrics. Based on the values reported by the evaluation metrics, the techniques used in the data pre-processing and data splitting phases can be changed or improved, thus leveraging the model's performance. The standard metrics also allow comparisons with the existing studies, as reported in Section 2.6.

Figure 3.3 depicts how the use of the validation set improves the model by evaluating it via the CV procedure and allowing for the tuning of the hyperparameters of the ML algorithms. This diagram also depicts the complete ML module, developed in the Python programming language, that is responsible for building, improving, and evaluating the model that will classify Android apps as benign or malicious.

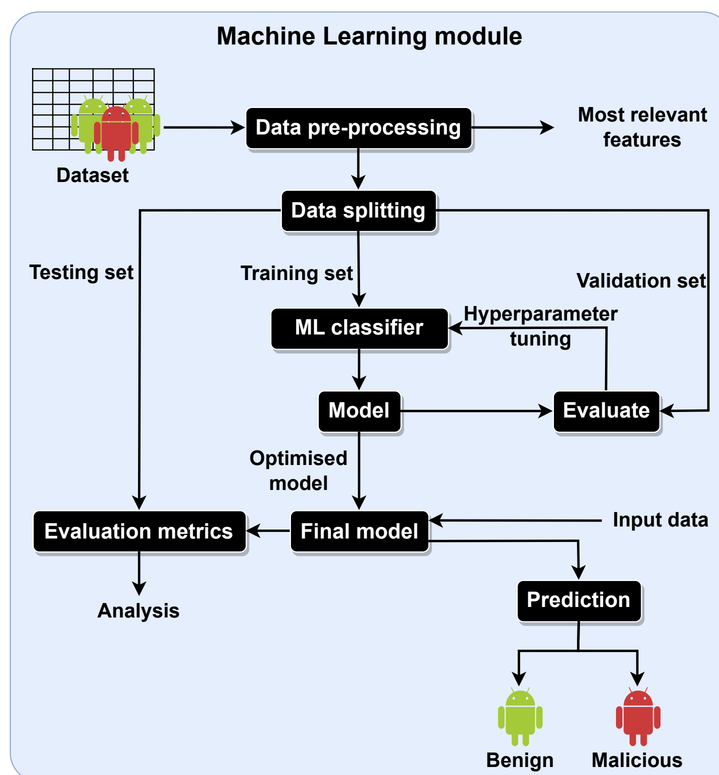

Figure 3.3: Full block diagram of the ML module, aggregating all the stages referenced in Figure 3.1 and Figure 3.2 as well as the hyperparameter tunning stage.

3.2 Complete approach - full block diagram

A diagram representing the complete proposed approach is depicted in Figure 3.4. It incorporates the ML module from Figure 3.3, as well as the feature extraction module and Android applications described next.

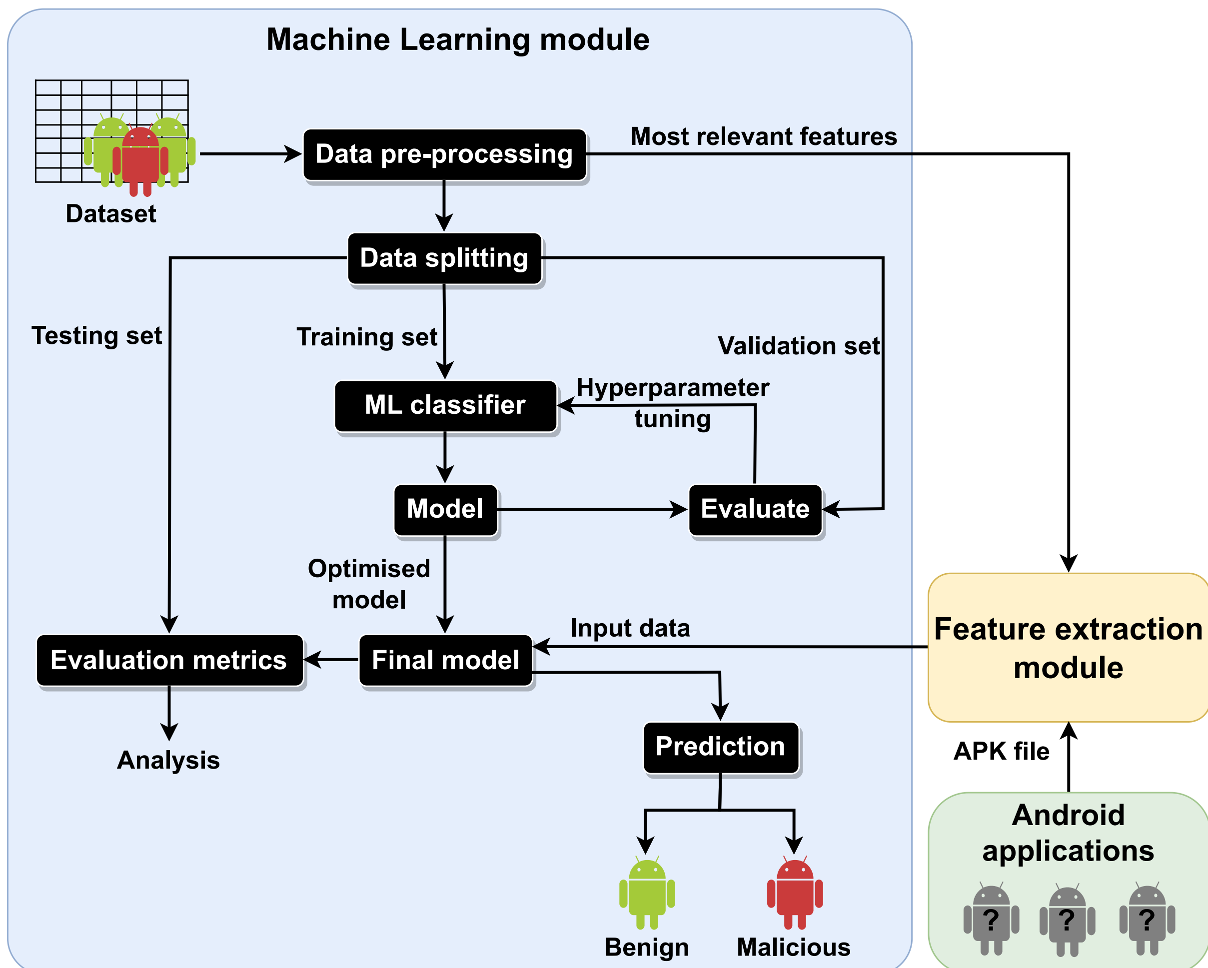

Figure 3.4: Full block diagram of the proposed approach with the ML module and the Android applications and feature extraction modules.

Feature extraction module

The feature extraction module follows a static analysis approach. It was developed in Python and Androguard. The latter enables the extraction of the features from Android app files. Thus, this module extracts static features from an Android app's APK file. The features sought for extraction were related to permissions, classes, methods, intents, activities, services, receivers, providers, software, and hardware. These features were preferred since they are often found to be the most relevant features obtained via

FS in the analysed datasets and frequently mentioned in the literature in the context of static analysis.

The mapping between the extracted features and the features deemed more indicative (obtained in the data pre-processing stage using FS)of the presence of malware in Android apps provides the input data to the model, which can then classify/predict the Android application as benign or malicious.

Android applications

Basic Android applications, shown on the bottom right-hand side of Figure 3.4, were developed in the Kotlin programming language to allow for an assessment of the developed prototype of the proposed approach with real-world apps.

3.3 Software tools

In this section, a brief introduction will be given to the software tools used in developing and implementing the proposed approach.

Orange

Orange[1] is a framework with a front-end for visual programming that allows for data analysis and interactive visualisation, ML, and data mining. It is open-source, has many functionalities, and is a good visualisation tool, making data analyses easier. It was mainly used to verify feature statistics, correlations, distributions, and if the datasets were imbalanced. The version used was 3.31.0.

Python

Python[2] is a high-level, general-purpose programming language. It is lightweight, open-source, versatile, and supports a variety of frameworks and libraries. It is a leading programming language in the ML field due to the extensive collection of specialised libraries, such as scikit-learn and TensorFlow. The version used was 3.10.2.

scikit-learn

scikit-learn[3] (also known as sklearn) is a free software ML library for the Python programming language. It provides a wide variety of techniques and algorithms, with all the data pre-processing (except for FS), data splitting techniques, evaluation metrics, classifiers and the function to perform hyperparameter tuning used in the proposed approach implementation belonging to this library. The 1.1.3 version was used.

[1] https://orangedatamining.com/
[2] https://www.python.org/
[3] https://scikit-learn.org/stable/

NumPy

NumPy[4] is an open-source and cross-platform library for the Python programming language. It supports large, multi-dimensional arrays and matrices and an extensive collection of high-level mathematical functions [50]. This library was mainly used for its ease of dealing with arrays of numerical data and for the operations it allows to perform. The version used was 1.23.5.

pandas

pandas[5] is a library written for the Python programming language for data manipulation and analysis. It also allows importing data from various file formats. It is built upon the NumPy library and is also free and cross-platform. As it is deemed helpful for importing and manipulating data, it was used for that effect. The version used was 1.5.1.

PyCharm

PyCharm[6] is an Integrated Development Environment (IDE) used for programming in Python. It is cross-platform and available in a free, open-source version (Community version). As such, it was used to implement the proposed approach. The 2021.3.1 version was used.

Androguard

Androguard [10] is a Python library that enables interaction with Android files. It is a tool for reverse engineering applications and features many functions for automated analysis. Thus, it is used to implement the feature extraction module of the proposed approach, allowing the extraction of features. The version used is 3.3.5.

Kotlin

Kotlin[7] is a cross-platform, concise, general-purpose, high-level programming language. It was used to develop Android applications. The version used is 1.7.20.

Android Studio

Android Studio[8] is the official IDE for the Android OS and is explicitly designed for the development of Android applications. The 2021.3.1 version was used.

[4] https://numpy.org/
[5] https://pandas.pydata.org/
[6] https://www.jetbrains.com/pycharm/
[7] https://kotlinlang.org/
[8] https://developer.android.com/studio

Alternative tools

Although the presented tools were the ones used in the development of the proposed approach, other alternatives, given more time, could be explored, such as Jupyter[9] or Spyder[10], as alternatives to the PyCharm IDE.

[9] https://jupyter.org/
[10] https://www.spyder-ide.org/

4

Experimental Evaluation

This section describes the experiments performed and exhibits the results obtained. From these, conclusions are drawn and considered for the improvement of the ML model, contributing to the development of the prototype of the proposed approach and its assessment.

Section 4.1 describes the testing environment in which the experiments were conducted.

Section 4.2 exhibits some analysis of the used datasets.

Section 4.3 presents experiments regarding the baseline models.

Section 4.4 displays the experimental results obtained after applying different data preprocessing techniques.

Section 4.5 exhibits the results obtained after applying FS.

Section 4.6 displays the results obtained via CV and by performing hyperparameter tuning.

Section 4.7 depicts a comparative analysis of results, comparing the obtained results with ones from the literature.

Finally, in Section 4.8, real-world Android applications are used to assess the prototype of the proposed approach.

4.1 Testing environment

The characteristics of the computational environment where the experiments were performed are detailed in Table 4.1.

Table 4.1: Characteristics of the computational environment where the proposed approach was assessed.

Hardware	Software
RAM size 16 GB	Windows 11
Processor Intel core i7	Orange v3.35.0
	Python v3.10.2
	scikit-learn v1.1.3
	NumPy v1.23.5
	pandas v1.5.1
	Pycharm Community v2021.3.1
	Androguard v3.3.5
	Kotlin v1.7.20
	Android Studio v2021.3.1

To evaluate the ML model's performance, the evaluation metrics described in Section 2.6 were used. All of them are available in the scikit-learn library.

Regarding challenges found throughout the development of this thesis, aside from the lack of experience with the Android system and the Kotlin and Android Studio software tools, the training time for the ML models sometimes led to a "training time bottleneck". This is a concern when dealing with large datasets, complex models, or limited computational resources. These long training times lead to delays in the model development and experimentation, making it sometimes challenging to perform the experimental evaluation efficiently.

4.2 Dataset analysis

The datasets used were Debrin, CICAndMal2017, AM and AMSF, available in [25], [22], [11] and [12], respectively. The datasets were retrieved from the Kaggle[1] website in the CSV file format.

Since the proposed approach is based on static analysis, only static features are considered. Thus, dynamic features needed to be removed if a dataset contained both. The

[1] https://www.kaggle.com/

Debrin and AM datasets only possessed static features. However, the CICAndMal2017 dataset contained 110 static features and 73 dynamic features from a total of 183 features. The removal process was facilitated by the authors of the dataset, who identified each feature name with an '(S)' if it was static and a '(D)' if it was dynamic. The AMSF dataset also possessed static and dynamic features. Given that it was divided into six datasets, each affiliated with a different group of features, only the ones containing static features were merged into the single dataset that was then used. Subsequently, the number of instances and features in each of the used datasets are depicted in Table 4.2.

Table 4.2: Number of instances (n) and features (d) for each dataset and their online reference.

Dataset	n, Number of instances	d, Number of features	Available in
Drebin	15036	215	[25]
CICAndMal2017	29999	110	[22]
AM	11476	182	[11]
AMSF	5019	966	[12]

The Orange framework contributed to the dataset analysis. The workflow created, using Orange's widgets is depicted in Figure 4.1.

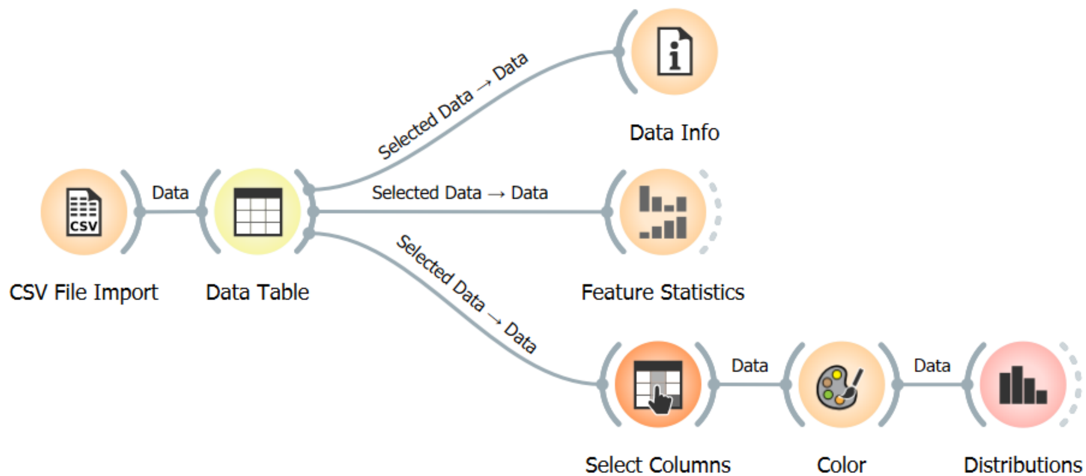

Figure 4.1: Orange workflow for dataset analysis.

With the usage of the 'Distributions' widget, the class distribution in the datasets was analysed to evaluate if there were cases of imbalanced data. Figure 4.2 depicts the number of instances per class label for each dataset. All datasets have binary class labels, with the CICAndMal2017 and AMSF datasets identifying malware as a positive and a benign app as a negative, coinciding with the proposed approach. Meanwhile,

the Drebin and AM had a categorical class label, with 'B' labelling benign instances and 'S' malicious instances in the Drebin dataset, and 'benignware' marking benign apps and 'malware' malicious apps in the AM dataset.

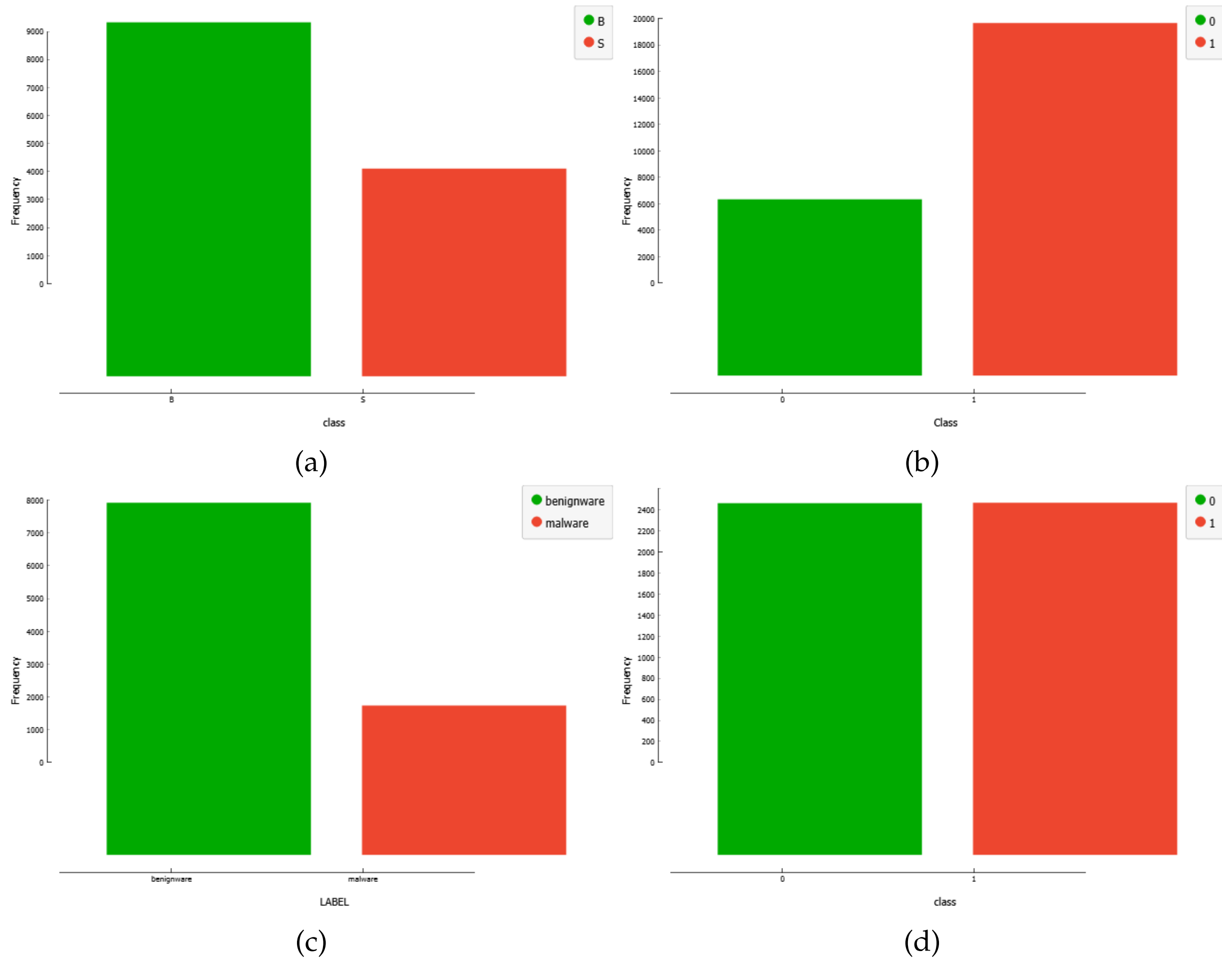

Figure 4.2: Class distribution ('benign' in green and 'malicious' in red) obtained via the 'Distributions' widget for the Drebin, CICAndMal2017, AM and AMSF datasets, in (a), (b), (c) and (d), respectively.

Both Drebin and CICAndMal2017 datasets in (a) and (b) present a ratio of approximately one-third between class labels. Thus, both datasets are not perfectly balanced but can not be considered imbalanced. The AM dataset in (c) is the most unbalanced among the chosen datasets, with the malicious class label being less than half of the benign class label. The AMSF dataset in (d) presents a balanced ratio between class labels. Table 4.3 synthesises the number and percentage of benign and malicious instances in each dataset.

Table 4.3: Number and percentage of benign and malicious instances (n) in each dataset.

Dataset	n (Benign)	n (Malicious)	n (Total)
Drebin	9476 (63.02%)	5560 (36.98%)	15036
CICAndMal2017	9999 (33.33%)	20000 (66.67%)	29999
AM	8058 (70.22%)	3418 (29.78%)	11476
AMSF	2508 (49.97%)	2511 (50.03%)	5019

Regarding the data type of the features, Table 4.4 presents the number of features (d) of categorical and numerical nature in each dataset, with the numerical features in each dataset being mainly binary.

Table 4.4: Number of categorical and numerical features (d) in each dataset.

Dataset	d (Categorical)	d (Numerical)	d (Total)
Drebin	1	214	215
CICAndMal2017	5	105	110
AM	12	170	182
AMSF	0	966	966

Concerning the number of missing values, Table 4.5 exhibits the number of occurrences in each dataset.

Table 4.5: Number of missing values in each dataset.

Dataset	Number of missing values
Drebin	0
CICAndMal2017	204
AM	19888
AMSF	0

Throughout this dataset analysis, it can be concluded that the AM dataset, among the used datasets, requires more data pre-processing since it is the most unbalanced dataset with the most categorical features and a high number of missing values. Meanwhile, the Drebin and AMSF datasets require fewer data pre-processing since they possess no missing values and their features are essentially all numerical.

4.3 Experimental Results: Baseline

This section presents the baseline results and how they were obtained. The ML classifiers, RF, SVM, KNN, NB and MLP, described in Section 2.5, were applied. Two significant issues were addressed to perform the first experiments: categorical features and missing values, since some classifiers can't deal with these. As a first approach, all categorical features were converted to numerical via label encoding. Missing values were dealt with by removing the instances that contained them, except if all instances of a feature were missing, in which case the feature was removed.

Initially, no CV was applied, no validation set was used, and no hyperparameter tuning was performed. Training and testing sets were obtained via a random stratified sampling with a 70-30 ratio, respectively.

Figure 4.3 summarises the obtained results for each dataset and classifier regarding the accuracy metric. The complete experimental results of all the evaluation metrics can be found in Table A.1 from Section A.1 of Appendix A.

With the Drebin dataset, the best result was obtained with the MLP classifier, closely followed by the RF and SVM classifiers. Overall, all classifiers presented good results regarding this dataset, with the worst result being obtained with NB with an accuracy of 92.66%, nevertheless a good result.

With the CICAndMal2017 dataset, the worst results, overall, were obtained. The best accuracy was 79.14% with the RF classifier, and the worst accuracy was 62.01% with the KNN classifier.

The RF classifier obtained the best accuracy value (93.96%) with the AM dataset. The other classifiers showed more unsatisfying results, with the lowest accuracy being 64.91% with the NB classifier.

The AMSF dataset was the dataset with which the best results were obtained overall. The highest accuracy was 99.33% with the RF classifier, and the lowest was 96.81% with the SVM classifier.

Regarding the overall performance of the classifiers, the RF classifier performed the best, with high accuracies, only once being the second-best ranked classifier instead of the first. The MLP classifier presents promising results, especially in the Drebin and AMSF datasets. However, it was solely tested to assess how a DL algorithm would perform regarding this problem. As previously said, this thesis focuses on ML and not DL.

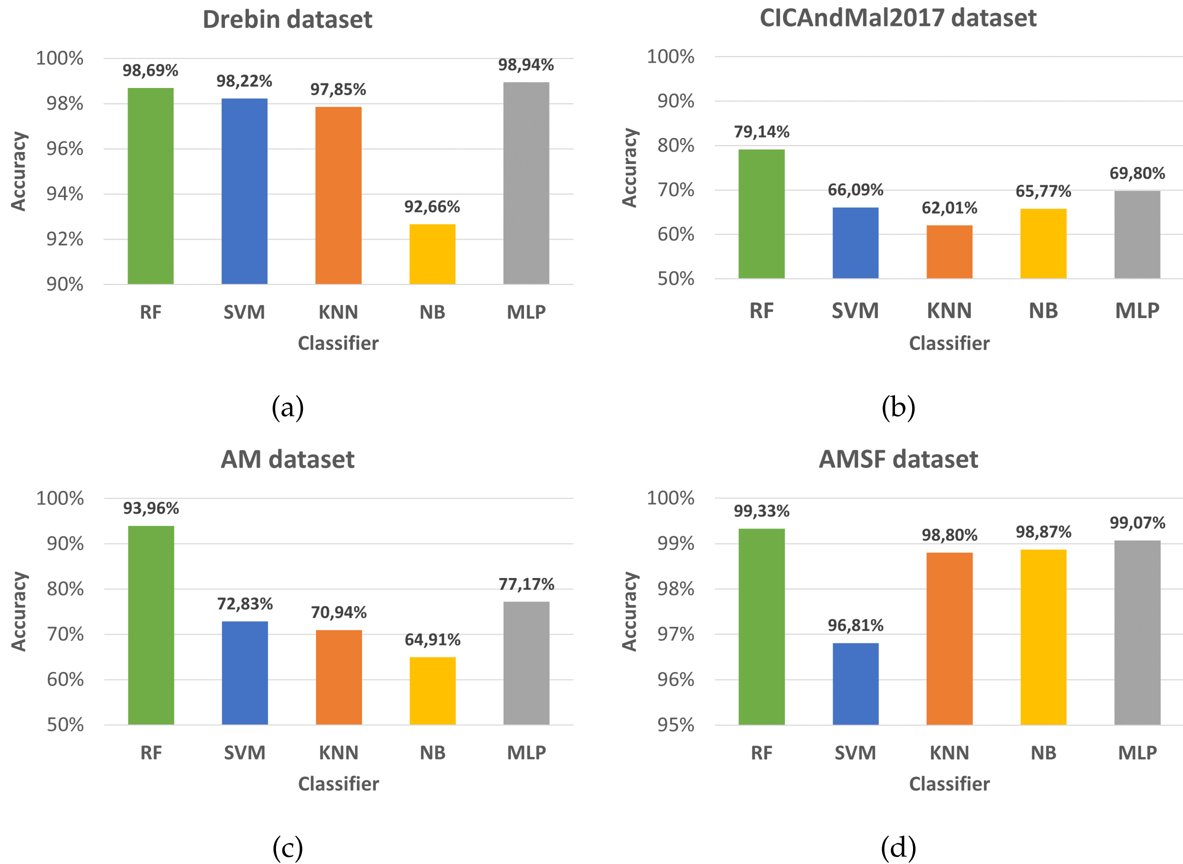

Figure 4.3: Accuracy (%) obtained with each classifier (RF, SVM, KNN, NB and MLP) for each dataset, Drebin, CICAndMal2017, AM and AMSF, in (a), (b), (c) and (d), respectively.

From these baseline experimental results, it can be concluded which classifiers performed overall better and, thus, which ones are used in subsequent experiments. MLP is not considered since the focus is not on DL, thus, the RF and SVM were the classifiers considered. With SVM presenting good accuracy values, often being the third-best classifier among the tested ones. RF and SVM are also the most popular ML classifiers among the literature for this problem.

4.4 Experimental Results: Data Pre-processing

In this section, the results regarding the usage of different data pre-processing techniques are presented and compared with the previous baseline results.

4.4.1 Handling missing values

Initially, removing instances containing missing values was the method applied to deal with missing values. However, it yields data loss. Thus, experiments with different methods to deal with missing values were performed. Namely, the following methods were tested: removing features with missing values, imputing the missing values by the respective feature mean, median or mode and removing instances with missing values (considered as baseline).

The experiments regarding missing values didn't influence the results obtained when using the Drebin and AMSF datasets since they possess no missing values. Thus, only the results obtained with the CICAndMal2017 and AM datasets were considered. The complete experimental results of all the evaluation metrics regarding the handling of missing values can be found in Section A.2 of Appendix A. Figure 4.4 exhibits the accuracies obtained when applying different methods to deal with missing values to the AM dataset, which possesses the higher amount of missing values among the used datasets.

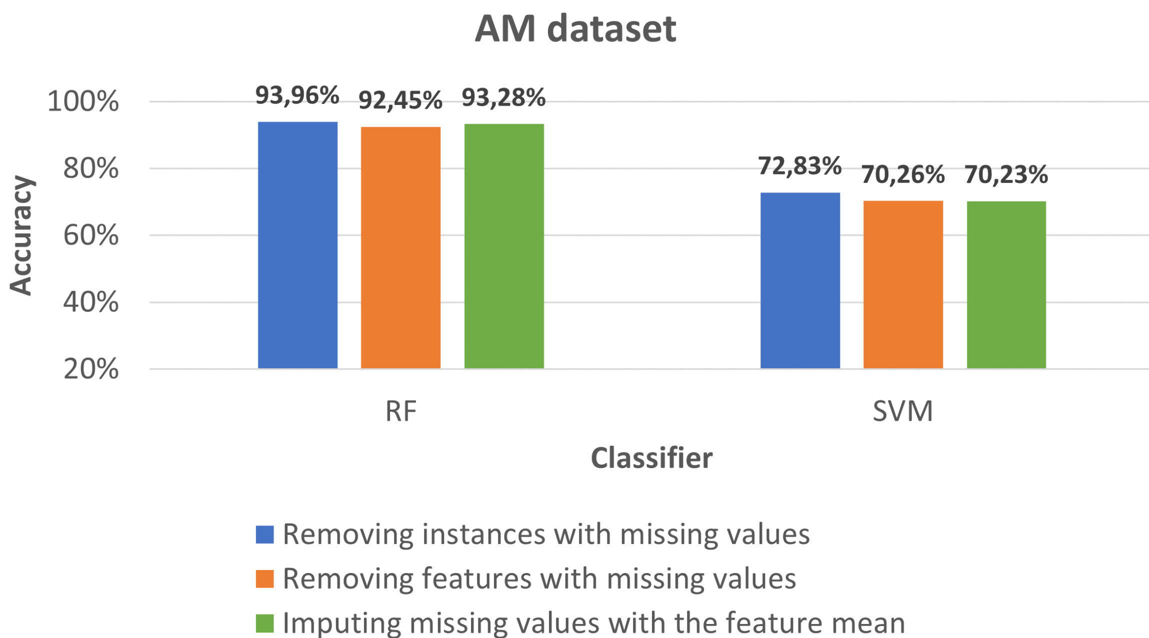

Figure 4.4: Accuracy (%) obtained, with the RF and SVM classifiers, for the AM dataset after applying different methods to deal with missing values.

The accuracy results obtained with the different methods to deal with missing values do not differ significantly. The same was verified with the remaining evaluation

metrics. With both RF and SVM, removing instances containing missing values provided the best results in terms of accuracy. The results for the CICAndMal2017 dataset also did not vary substantially, with removing features containing missing values and imputing missing values with the feature mean achieving the best results among the tested methods.

The results obtained by removing instances or features (containing missing values) do not differ significantly from the ones where the missing values are imputed with the estimated value based on the feature information. This indicates that the CICAndMal2017 and AM datasets possess irrelevant, maybe even harmful, data for the model's training. Thus, it is adequate to perform dimensionality reduction by, for example, using a FS technique, which is further explored in Section 4.5.

Given that the results are similar between the different tested methods, the preferred approach was imputing missing values with the feature mean since it does not yield data loss and is the most straightforward approach to maintain the data distribution.

4.4.2 Normalisation

The conversion of categorical features to numerical via label encoding can introduce large differences in the scales of features, mainly when applied to categorical features with many distinct values. Additionally, algorithms that rely on distance calculations, such as SVM, tend to be sensitive to feature scales. Normalising features can improve the model performance and faster convergence since normalised features are often more interpretable by algorithms. Min-max normalisation was applied to accommodate all the values of both datasets between zero and one while maintaining the original data distribution.

Table 4.6 presents the obtained results, in terms of accuracy, F1-score and AUC-ROC with and without the usage of Min-max normalisation, for the RF and SVM classifiers and for the datasets with most categorical features, the CICAndMal2017 and AM datasets. The complete experimental results of the usage of the Min-max normalisation for each dataset and all the evaluation metrics can be found in Table A.6 in Section A.2 of Appendix A.

With the RF classifier, overall, the results do not differ significantly, most likely because the RF algorithm does not rely on distance calculations and, thus, is generally robust to significant differences in the scales of features better.

Regarding the SVM classifier, with the Drebin dataset, the results barely differ, and

Table 4.6: Experimental results, in terms of accuracy (Acc), F1-score and AUC-ROC for the CICAndMal2017 and AM datasets and the RF and SVM classifiers with and without the use of Min-Max normalisation.

Classifier	Dataset	Min-max Normalisation	Acc (%)	F1-Score (%)	AUC-ROC (%)
RF	CICAndMal2017	✗	79.81	84.82	77.40
		✓	79.62	84.72	77.06
RF	AM	✗	93.28	88.02	90.28
		✓	93.28	88.05	90.33
SVM	CICAndMal2017	✗	**66.13**	**78.96**	**51.51**
		✓	**70.81**	**79.71**	**63.22**
SVM	AM	✗	**70.23**	**00.00**	**50.00**
		✓	**90.88**	**82.77**	**85.89**

with the AMSF, there is a slight improvement across all the evaluation metrics. However, the results significantly improved regarding the CICAndMal2017 and AM datasets, which previously contained more categorical values. Namely, these results highlight how the accuracy metric can be misleading. Without Min-max normalisation, the SVM classifier achieved an accuracy of 66.13% with the CICAndMal2017. However, the AUC-ROC metric presented a value of 51.51%, implying a random classifier. With the Min-max normalisation, the AUC-ROC improved from 51.51% to 63.22%, and the accuracy improved from 66.13% to 70.81%. The result was even more impactful with the AM dataset, with the accuracy improving by approximately 20%. The AUC-ROC value previously was 50% (indicating a random classifier) improved to 85.89% after Min-max normalisation. The F1-score, Precision and Recall metrics were 0.0%, with the TP value from the confusion matrix being zero. Thus, the model failed to identify any of the actual positive instances. After applying min-max normalisation, the Precision, Recall, and F1-score values improved to 94.60%, 73.56% and 82.77%, respectively.

4.4.3 Numerosity balancing

To further improve the model, numerosity balancing techniques, such as random undersampling, random oversampling and SMOTE, presented in Section 2.4.1, were applied to deal with data imbalance.

Table 4.7 presents the results, in terms of accuracy and Recall, obtained with the RF and SVM classifiers with the different numerosity balancing approaches for the AM dataset, the most imbalanced dataset among the used datasets. The complete experimental results, with all the metrics and for each dataset, can be found in Table A.7 in Section A.2 of Appendix A.

The results obtained with the AM dataset improved significantly, with both RF and

Table 4.7: Accuracy (Acc) and Recall (Rec) values, obtained with the RF and SVM classifiers with the different numerosity balancing approaches for the AM dataset.

Classifier	Numerosity balancing method	Acc (%)	Rec (%)
RF	None	93.28	83.02
	Random undersampling	91.36	86.44
	Random oversampling	**96.28**	**96.07**
	SMOTE	94.06	91.39
SVM	None	70.81	**86.00**
	Random undersampling	89.18	82.44
	Random oversampling	**89.47**	82.75
	SMOTE	88.81	81.42

SVM classifiers, namely, with the usage of random oversampling. To highlight the improvement of the SVM classifier with the AM dataset in terms of accuracy and AUC-ROC, improving from 70.81% and 63.22%, respectively, to 89.47%.

With the CICAndMal2017 dataset, the results also improved, especially with random oversampling with the RF classifier. With the Drebin dataset, the results did not vary significantly between different numerosity balancing methods. The same was verified for the AMSF, which was already a balanced dataset.

Overall, the best results were obtained using random oversampling, closely followed by SMOTE and random undersampling. The latter yields information loss, resulting in fewer instances to train the model. SMOTE and random oversampling were the methods that often presented the best results, with the difference in results only differing slightly. Random oversampling is more straightforward than SMOTE but can lead to overfitting, which SMOTE can reduce. However, since the minority class is moderately imbalanced in the chosen datasets, random oversampling is effective. Thus, random oversampling was the preferred approach to attain numerosity balancing.

4.5 Experimental Results: Feature Selection

This section exhibits the experimental results obtained by applying FS, namely, RRFS, mentioned in Section 2.4.1. Different relevance measures were tested, namely the supervised relevance measure FR and the unsupervised relevance measure MM. The redundancy measure used was the AC with a maximum allowed similarity between consecutive pairs of features (M_s) of 0.3, with one representing redundant features and zero a complete lack of redundancy.

Table 4.8 presents the accuracy values obtained with the SVM classifier for each dataset

by applying RRFS with MM or FR, or not applying FS. The complete experimental results, with all the metrics for each dataset and the RF and SVM classifiers, can be found in Table A.8 in Section A.2 of Appendix A.

Table 4.8: Accuracy (Acc) obtained with the SVM classifier for each dataset, by not applying FS, applying RRFS with MM or RRFS with FR.

Dataset	FS	Acc (%)
Drebin	None	98.50
	RRFS (MM)	94.71
	RRFS (FR)	96.66
CICAndMal2017	None	71.69
	RRFS (MM)	60.04
	RRFS (FR)	68.52
AM	None	89.47
	RRFS (MM)	86.99
	RRFS (FR)	84.55
AMSF	None	99.53
	RRFS (MM)	99.87
	RRFS (FR)	98.41

Comparing the relevance measures, the results generally do not differ significantly, but the FR supervised relevance metric provided slightly better results than the MM unsupervised relevance measure. Overall, the results worsen slightly after applying RRFS. However, these slight drops in accuracy are arguably compensated by the reduction in the number of features. The original numbers of features versus the number of features after applying the RRFS approach with different relevance measures for the Drebin, CICAndMal2017, AM and AMSF datasets are represented in (a), (b), (c) and (d), respectively, of Figure 4.5.

Regardless of the relevance measure, the RRFS approach significantly reduced the number of features in each dataset. With the supervised relevance measure FR, a more considerable reduction in dimensionality was obtained when compared with the unsupervised relevance measure MM. The number of reduced features combined with the evaluation metrics results indicate that, overall, the FR relevance measure outperforms MM. Thus, the class label is relevant to this problem.

With the FR measure, a subset of the most relevant features is obtained. The RRFS approach continues by removing redundant features from this subset to obtain the best feature subset [5], which consists of the most relevant and non-redundant features.

The redundancy measure applied was the AC where the cosine between two features is zero (orthogonal) if they are maximally different features or one if they are colinear

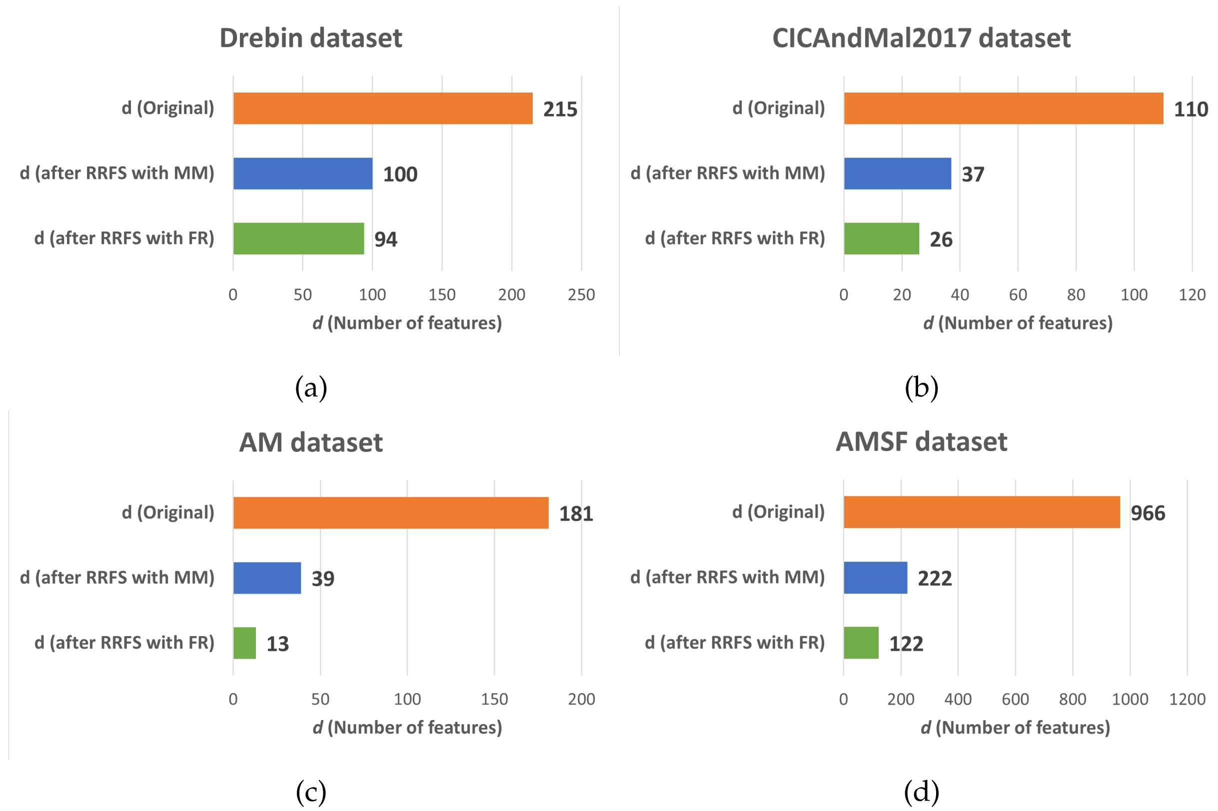

Figure 4.5: The original numbers of features versus the number of features after applying RRFS with different relevance measures for the Drebin, CICAndMal2017, AM and AMSF datasets presented in (a), (b), (c) and (d), respectively.

(redundant) features. The M_s value defines the maximum allowed similarity between pairs of features. Other values of M_s were tested to improve the balance between the number of reduced features and the evaluation metrics results. Experiments were conducted with values of M_s: 0.2, 0.3 and 0.4.

The results in terms of evaluation metrics obtained with different M_s were similar. However, typically, M_s=0.4 would provide the best results, closely followed by M_s=0.3 and then M_s=0.2, with some exceptions, for example, with the RF classifier and CICAndMal2017 dataset M_s=0.2 provided the best results across the different evaluation metrics. The complete experimental results, with all the metrics and for each dataset, can be found in Table A.10 in Section A.3 of Appendix A.

Increasing the M_s value makes the selection less strict regarding redundancy between features. Thus, more features are kept, and by decreasing the M_s value, the more rigorous the selection is, and fewer are kept. This can be seen in Figure 4.6, which presents the number of features kept for the Drebin, CICAndMal2017, AM and AMSF datasets represented in (a), (b), (c) and (d), respectively, for the different values of M_s tested with RRFS (FR). Based on these results, also exhibited in Table A.10 in Section A.3 of

Appendix A, to better accommodate both the reduction of features and maintaining good performance, M_s=0.3 was the preferred approach.

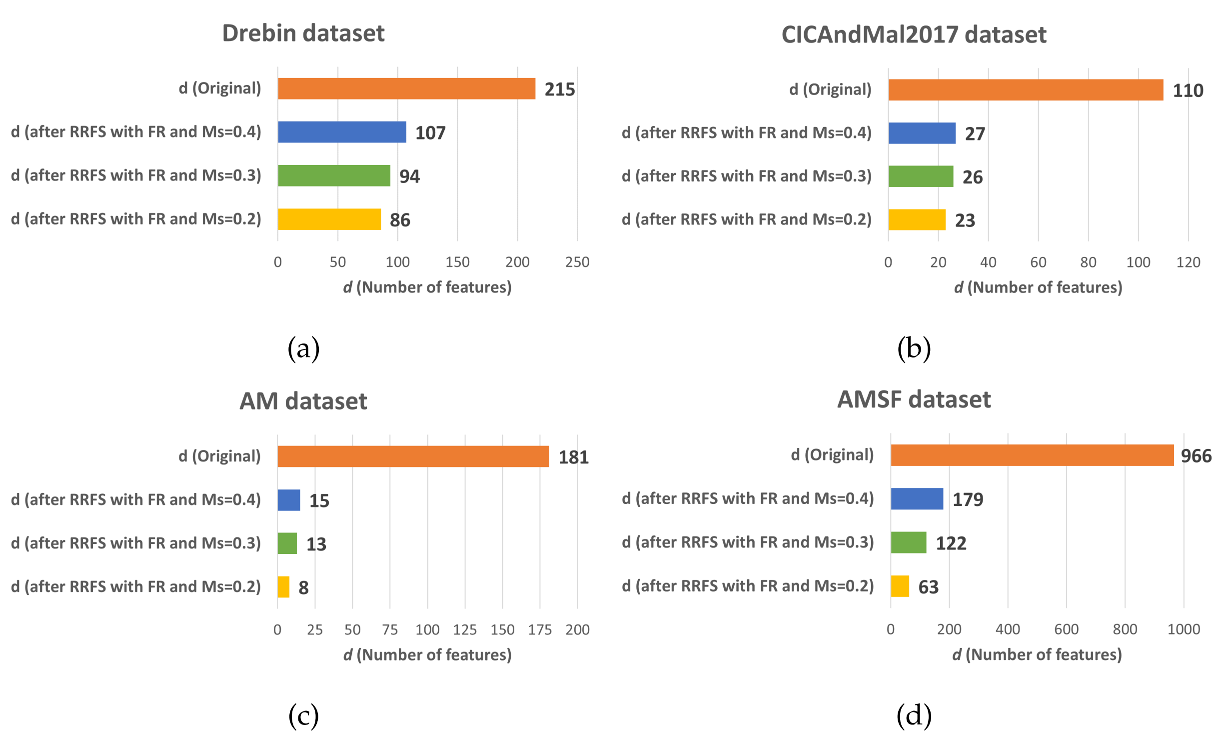

Figure 4.6: Numbers of features in the Drebin, CICAndMal2017, AM and AMSF datasets presented in (a), (b), (c) and (d), respectively, before RRFS and after RRFS with FR for different values of M_s.

Table A.12 from Section A.3 of Appendix A displays the results before applying RRFS and after applying it with FR and AC with a M_s of 0.3. Overall, the results with the SVM seem to vary more with the usage of FS than the results obtained with the RF classifier, with the latter being more robust to irrelevant features. Thus not being significantly impacted by FS. Meanwhile, the results with the SVM classifier suffered more influence with the usage of FS, with a tendency to get slightly worse. This could be because of the removal of too many features, which may oversimplify the model (underfitting). However, the slightly worse results in terms of evaluation metrics are compensated by the reductions achieved in the dataset's dimensionality. With a reduction of 56% for the Drebin dataset, 76% for the CICAndMal2017 dataset, 92% for the AM dataset and 87% for the AMSF dataset.

Besides dimensionality reduction, RRFS enables the recognition of the most relevant features for malware detection in Android applications, a key factor for the culmination of the proposed approach. To better understand if the most relevant features follow a pattern or are the same among the different datasets, each dataset's ten most relevant features, selected by RRFS (FR) with M_s=0.3, are enumerated next.

Ten most relevant features in the Drebin dataset selected by RRFS (FR) with M_s=0.3:

1. `transact`
2. `SEND_SMS`
3. `Ljava.lang.Class.getCanonicalName`
4. `android.telephony.SmsManager`
5. `Ljava.lang.Class.getField`
6. `RECEIVE_SMS`
7. `Ljavax.crypto.spec.SecretKeySpec`
8. `WRITE_SMS`
9. `READ_SYNC_SETTINGS`
10. `TelephonyManager.getSubscriberId`

Ten most relevant features in the CICAndMal2017 dataset selected by RRFS (FR) with M_s=0.3:

1. `Category`
2. `Price`
3. `Network communication : view network state (S)`
4. `Your location : access extra location provider commands (S)`
5. `System tools : set wallpaper (S)`
6. `Description`
7. `Number of ratings`
8. `Related apps`
9. `System tools : automatically start at boot (S)`
10. `System tools : send sticky broadcast (S)`

Ten most relevant features in the AM dataset selected by RRFS (FR) with M_s=0.3:

1. `com.android.launcher.permission.UNINSTALL_SHORTCUT`
2. `android.permission.VIBRATE`
3. `android.permission.ACCESS_FINE_LOCATION`
4. `name`
5. `android.permission.BLUETOOTH_ADMIN`
6. `android.permission.WAKE_LOCK`
7. `android.permission.READ_EXTERNAL_STORAGE`
8. `android.permission.RECORD_AUDIO`
9. `android.permission.ACCESS_NETWORK_STATE`
10. `android.permission.CAMERA`

Ten most relevant features in the AMSF dataset selected by RRFS (FR) with M_s=0.3:

1. `androidpermissionSEND_SMS`
2. `android.telephony.SmsManager.sendTextMessage`
3. `float-to-int`
4. `android.telephony.SmsManager`
5. `android.support.v4.widget`

6. `android.intent.action.DATA_SMS_RECEIVED`
7. `or-int/2addr`
8. `com.software.CHECKER`
9. `android.content.pm`
10. `android.widget.Button.startAnimation`

The most relevant features in the Drebin and AMSF datasets are permissions, classes and methods. Permissions are the most relevant features in the AM dataset. In the CICAndMAl2017 dataset, the most relevant features are permissions and meta information. **Overall, permissions seem to have a prevalent presence among the most relevant features for Android malware detection**. The complete list of features, by relevance order, for each dataset is enumerated in Section A.3 of Appendix A.

4.6 Experimental Results: CV & Hyperparameter tuning

This section exhibits the experimental results obtained after performing hyperparameter tuning to the RF and SVM classifiers and the usage of CV. Initially, a random stratified split was applied to the datasets with a ratio of 70-30 for training and testing, respectively, with no validation set considered and no hyperparameter tuning performed.

To perform hyperparameter tuning of the RF and SVM classifiers, the function GridSearchCV [58] of the scikit-learn library was applied. This function performs an exhaustive search over specified parameter values for an estimator. The parameters of the estimator are optimised by CV. The training set is provided to the function, which splits it into training and validation sets. By default, the CV splitting strategy is stratified 5-fold CV. This function also enables the specification of the hyperparameters aimed to be optimised and in what range of values.

The hyperparameters to optimise were the ones deemed more impactful for each ML algorithm. For the RF classifier, we considered:

- the number of trees in the range [100, 1000] with steps of 100.
- the maximum tree depth with the values: 3, 5, 7, and None. The latter means the nodes are expanded until all leaves are pure or until all leaves contain less than the minimum number of samples required to split an internal node.
- the split quality measure as Gini, Entropy, or Log Loss.

For the SVM classifier, we considered:

- the regularisation parameter (C) in the range [1, 20] with steps of 1.
- the kernel type to be used in the algorithm: the RBF kernel, the polynomial kernel, the linear kernel, and the Sigmoid kernel.
- the kernel coefficient (gamma) for the previous kernel types (except the linear kernel).

In Section A.4 of Appendix A, the hyperparameters of the RF and SVM classifiers, optimised for each dataset, are exhibited.

Table A.13 from Section A.4 of Appendix A displays the results obtained by tuning or not the classifier's hyperparameters. Overall, the results improved across all evaluation metrics. However, this improvement did not rise above 2%, thus only slightly improving the ML model's performance.

To also perform CV with the training and testing sets, an outer loop for CV was added. Thus, creating a nested CV considering the CV performed by the GridSearchCV function with the training and validation sets. For the outer loop, 10-fold CV and LOOCV were applied.

Here, the training time for the ML models frequently led to a "training time bottleneck" due to the limited computational resources, the number of iterations combined, and the number of hyperparameter combinations being tested. These long training times lead to delays in the model development and experimentation, making it sometimes challenging to perform the experimental evaluation efficiently. This was an even more significant issue with LOOCV, whose amount of iterations matches the number of instances of the used dataset.

As an attempt to contour this issue, the number of hyperparameter combinations in the GridSearchCV function was reduced by considering the values more often chosen in the optimisation for each of the used datasets. However, some results still could not be obtained, namely, with LOOCV, which is much more delayed than 10-fold CV. Although it takes longer, its results are more stable and reliable than 10-fold CV since it uses more training samples and iterations.

With 10-fold CV, some results were obtained, namely, in the form of the mean and standard deviation measures for each evaluation metric. Overall, the results were satisfying, with the mean values not differing substantially from those obtained after performing hyperparameter tuning, and the standard deviation obtained throughout the different evaluation metrics was low, indicating that the results are clustered around the mean. Thus, more stable and reliable.

4.7 Comparative Analysis of Results

In this section, some of the experimental results are compared to ones from the literature. However, this comparison is not straightforward, often due to one or two main reasons: the datasets used are not available, and the ML techniques used, namely, in the data pre-processing stage, are not fully described in the existing studies, with the source code also not being available for analysis.

Since two of the datasets herein used, the Drebin and CICAndMal2017 datasets, are also used by Alkahtani and Aldhyani [6], the results obtained are briefly compared with theirs. The authors performed a random split with 70% for training and 30% for testing. Regarding data pre-processing, they only mention Min-max normalisation. Aside from this, no other pre-processing methods or tuning of hyperparameters are mentioned. Thus, the methodology with which the results were obtained differs from ours. Since the authors did not use the RF classifier, we will compare only the accuracy results regarding SVM. Table 4.9 summarises these results.

Table 4.9: Comparison of the experimental results, in terms of accuracy (%), obtained by Alkahtani and Aldhyani [6] with the SVM classifier, with the ones obtained with the proposed approach using the same classifier.

Dataset	Alkahtani and Aldhyani	Proposed
Drebin	80.71	**97.47**
CICAndMal2017	**100.00**	73.22

The proposed approach presented better accuracy on the Drebin dataset, achieving an accuracy of 97.47% compared to the 80.71% reported by Alkahtani and Aldhyani [6]. However, regarding the CICAndMal2017 dataset, the proposed approach only achieved 73.22% compared to the accuracy of 100% claimed by the authors. This disparity in the obtained results between the two studies using the same datasets lies in the different approaches in the pre-processing applied, further emphasising its importance since it significantly impacts the obtained results.

Regarding the most relevant features for malware detection in Android applications, Keyvanpour *et al.* [35] applied FS with effective and high weight FS and reported the most relevant features on the Drebin dataset. Two features deemed more relevant to classify malware were `SEND_SMS` and `android.telephony.SmsManager`. These coincide with the most relevant features to classify malware obtained with the RRFS (with FR and M_s=0.3) approach on the Debrin dataset where `SEND_SMS` ranked second

and `android.telephony.SmsManager` ranked fourth and on the AMSF dataset where they ranked first and fourth, respectively.

4.8 Evaluating the model with real-world applications

In this section, the developed prototype of the proposed approach is assessed with real-world applications. On these experiments, the RF classifier was preferred since, throughout the previous experiments, it outperformed the SVM classifier.

Recalling the proposed approach, a ML model was achieved and improved using different techniques. At the same time, knowledge regarding the most relevant features for malware detection in Android applications was attained. With this, the feature extraction module was developed. Similar to the ML model module, it was developed with the Python programming language and Androguard, a tool and Python library to interact with Android Files, which enables the extraction of the features from the Android applications files.

The features sought for extraction were permissions, classes, methods, intents, activities, services, receivers, providers, and software and hardware features. These features were preferred to be extracted since they are often found in the analysed datasets, the most relevant features obtained via FS, and frequently mentioned in the literature in the context of static analysis.

Here, a significant challenge emerged since the names of the features throughout the datasets are not standardised. For example, when extracting the names of the permissions required by the app, the feature `android.permission.SEND_SMS` is obtained. However, this feature in the Drebin dataset corresponds to `SEND_SMS` and in the AMSF dataset to `androidpermissionSEND_SMS`. This complicates the association/mapping between the dataset features and the features extracted from the APK file. An approach based on string similarity was preferred to improve the feature extraction module on this issue. With this, although the feature extraction module was not able to identify/map correctly all features, its mapping improved.

Real-world Android apps were used to evaluate the ML model and the feature extraction module performance, overall assessing the performance of the prototype of the proposed approach. First, some basic Android apps, shown in Figure 4.7, with only one activity and no actual functionality, were developed. These are briefly described next.

- 'App1', shown in Figure 4.7 (a) and (b), will try to, unknowingly to the user, send

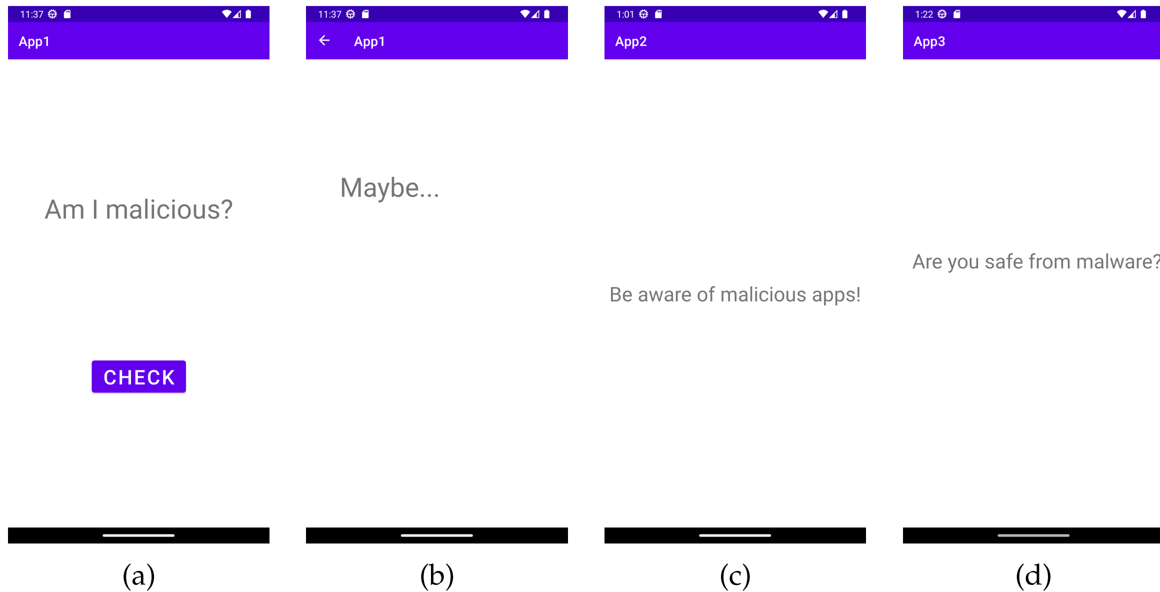

Figure 4.7: Developed Android applications: 'App1', 'App2' and 'App3' in (a) and (b), (c) and (d), respectively.

an SMS message when the user clicks on the button in the app.

Requests permissions regarding SMS, namely, `android.permission.SEND_SMS`, `android.permission.RECEIVE_SMS`, and `android.permission.WRITE_SMS` and uses the class `android.telephony.SmsManager`, all of these included in the top ten most relevant features in the Drebin dataset previously presented, with `android.permission.SEND_SMS` and `android.telephony.SmsManager` also present in the AMSF dataset.

The user might be asked to grant these permissions when installing or updating the app. As is often the case, we assume that the user doesn't dispense adequate awareness of the permissions he/she grants while using the app. Although it was just a code attempting to send an SMS, the user could face one of the malware examples mentioned in Section 2.2.

- 'App2', shown in Figure 4.7 (c), doesn't request/use any unnecessary features, thus is a benign app.

- 'App3', shown in Figure 4.7 (d), requests the following permissions: `android.permission.CAMERA`, `android.permission.INSTALL_PACKAGES`, and `android.permission.READ_HISTORY_BOOKMARKS` although it doesn't require any of them for any functionality. These permissions are present

among the most relevant features selected by RRFS in the Drebin dataset and `android.permission.CAMERA` is also present among the most relevant, selected by RRFS in the AM dataset.

The expected classifications for the apps 'App1', 'App2' and 'App3' were malicious, benign, and benign, respectively. Different predictions were obtained depending on the dataset used to train the ML model. The full results regarding the experiments with real-world apps are presented in Table A.17 in Section A.5 of Appendix A

The proposed approach classified 'App1' as malicious, with three (Drebin, CICAndMal2017 and AMSF) out of the four datasets used. Most predictions were 'malicious', which was the expected result, with features considered the most relevant in malware detection present in the app's APK.

With three (Drebin, AM and AMSF) out of the four datasets used, the proposed approach classified 'App2' and 'App3' as benign. Most predictions were 'benign', which was the expected result for both, although 'App3' had a few features included in the most relevant features for malware detection present.

To further test the developed approach, APK found online were used. As benign samples, APK of known apps were obtained from APKPure[2]. The benign samples used were the APK files 'WhatsAppMessenger' and 'Amazon Shopping', and in both cases, the predictions were correct when using the Drebin and AM datasets.

In contrast, examples of malicious APK were obtained from a website [49] that presents a collection entitled 'android-malware-samples' of interesting and diverse Android malware samples. Three APK were used, each briefly described next.

- A app that was an SMS stealer was classified by the ML model as benign, in most cases, thus not corresponding to the expected prediction.
- A ransomware disguised as a simple screen locker app. The ML model classified it as benign when trained with half of the datasets (Drebin and AM) and correctly classified it as malicious when trained with the other half (CICAndMal2017 and AMSF).
- A app which makes unwanted calls and has some clever obfuscation techniques. Similarly to the previous case, the proposed approach correctly identified the presence of malware with models trained with half of the datasets.

[2] https://m.apkpure.com/app

The proposed approach could not correctly identify malware in all cases, which was expected. The issue of feature mapping that greatly influences/impacts the performance, often not identifying or misidentifying the features, should be taken into account. Additionally, some of the malware samples tested used obfuscation techniques, known to be a weakness of static analysis, the type of analysis considered in the proposed approach. Furthermore, the datasets used also significantly impact the obtained prediction, with the final prediction depending greatly on the data used/considered.

5

Conclusions

This chapter provides an overview (Section 5.1) of this study and contemplates its conclusions. Possible improvements and alternatives resulting from self-assessment are presented in Section 5.2.

5.1 Overview

Malware in Android applications affects millions of users worldwide and is constantly evolving. Thus, its detection is a current and relevant problem. In the past few years, ML approaches have been proposed to mitigate malware in mobile applications.

In this thesis, a prototype that resorts to ML techniques to detect malware in Android applications was developed. The problem was formulated as a **binary classification problem**.

There are a variety of datasets for Android malware detection. However, many are not up-to-date or easy to access. Four public domain datasets, the **Drebin, CICAndMal2017, AM and AMSF datasets**, were used throughout the experiments. Concerning the type of analysis, the proposed approach follows **static analysis**, the most straightforward and used throughout the literature.

Experiments were performed with the RF, SVM, KNN, and NB classifiers. Based on the surveyed works, the RF and SVM classifiers are the most popular. The experimental results show that the RF and SVM classifiers are the most suited for this problem.

Experiments were also performed with the MLP classifier to assess the performance of DL algorithms, which are shown to be effective.

Data pre-processing techniques were explored to improve the ML model. The disclosure of the approach taken in the data pre-processing stage is absent in many existing studies. Often, authors do not specify any technique for data pre-processing, making it difficult to reproduce the experiments and compare results accurately. Thus, this thesis emphasised the data pre-processing stage and its impact on the final solution. Techniques to handle missing values and perform numerosity balancing were explored. Min-max normalisation was also applied, and it was shown to significantly improve the results in cases of significant differences in the scale of features. Often, the choices made in data pre-processing provided a better result for one dataset but a worse outcome for another. Thus, there is no ideal solution for all datasets.

Emphasis was given to the use of FS by applying the RRFS approach [5] to obtain the most relevant and non-redundant subset of features. Experiments were conducted to conclude which relevance measure provided better results: the supervised FR relevance measure and the unsupervised MM relevance measure—culminating with FR yielding the best results. Thus, the class label is relevant for malware detection in Android applications. Regarding the redundancy of the features, the AC measure was applied, and different experiments with different strictness regarding the similarity between features were performed, directly impacting the resulting subset of features.

Although RRFS provided slightly worse results regarding the evaluation metrics, these were arguably compensated by the dimensionality reduction achieved in each of the used datasets. A reduction of 56% was achieved for the Drebin dataset, 76% for the CICAndMal2017 dataset, 92% for the AM dataset and 87% for the AMSF dataset.

Aside from the reduction in dimensionality and possible improvement to the ML model, RRFS selected the most relevant subset of features to identify the presence of malware. Knowledge about these enabled the development of the feature extraction module that integrates the prototype of the proposed approach. Furthermore, it culminated in conclusions regarding that, among the different datasets, the features that are better suited to identify the presence of malware. **Overall, permissions have a prevalent presence among the most relevant features for Android malware detection**.

Nested CV was used to evaluate the trained model better and to tune the ML algorithms hyperparameters, improving the final ML model. In the outer loop, two alternative CV techniques were applied, Stratified 10-fold CV and LOOCV, providing a better assessment of the ML model.

As for evaluation metrics, the accuracy metric, commonly the most used throughout

the literature, was used to evaluate the ML model. However, accuracy can be misleading. Therefore, other evaluation metrics such as confusion matrix, precision, recall, F1-score, and AUC-ROC were also applied to further assess the ML model.

Lastly, the prototype of the proposed approach was assessed using real-world applications. The feature extraction module extracted features from the APK file and mapped them according to the most relevant subset of features selected by RRFS. These were then provided to the ML model as input data to be classified as benign or malicious. Overall, the results were negatively impacted by the non-standardization of the dataset's feature names, which prevented an accurate mapping between the extracted features and the most relevant subset of features.

The proposed approach can identify the most decisive features to classify an app as malware and greatly reduce the data dimensionality while achieving good results in identifying malware in Android applications across the various evaluation metrics.

5.2 Future Work

Performing a self-assessment of the developed work, the results obtained and comparing this study with other existing ones, some significant improvements to the proposed approach and alternatives that could be further explored emerge. Some of these are the following:

- The use of more up-to-date datasets, but since they are often not easy to obtain, creating a new dataset could be interesting.
- Expand the proposed approach to hybrid analysis.
- Further explore DL approaches and others that present good results in mitigating this problem.
- Address this problem with a multiclass approach instead of a binary one, where malware could be classified in levels according to its impact.
- Aim to use datasets whose feature names are more standardised, namely, feature names that align with the names of the features extracted from the APK file.

References

[1] Abdurrahman Pektaş, Mahmut Çavdar, and Tankut Acarman, "Android Malware Classification by Applying Online Machine Learning", in *Computer and Information Sciences*, Tadeusz Czachórski, Erol Gelenbe, Krzysztof Grochla, and Ricardo Lent, Eds., Cham: Springer International Publishing, 2016, pages 72–80.

[2] *1.17. neural network models (supervised) — scikit-learn 1.3.0 documentation*, https://scikit-learn.org/stable/modules/neural_networks_supervised.html, (Accessed on 25/10/2023).

[3] *2021 mobile malware statistics*, https://securityforeveryone.com/blog/2021-mobile-malware-statistics, (Accessed on 25/10/2023).

[4] Olawale Surajudeen Adebayo and Normaziah Abdul Aziz, "Improved malware detection model with apriori association rule and particle swarm optimization", *Security and Communication Networks*, vol. 2019, 2019. DOI: https://doi.org/10.1155/2019/2850932.

[5] Artur Ferreira and Mário Figueiredo, "Efficient feature selection filters for high-dimensional data", *Pattern Recognition Letters*, vol. 33, no. 13, pages 1794 –1804, 2012, ISSN: 0167-8655. DOI: http://dx.doi.org/10.1016/j.patrec.2012.05.019.

[6] Hasan Alkahtani and Theyazn HH Aldhyani, "Artificial intelligence algorithms for malware detection in Android-operated mobile devices", *Sensors*, vol. 22, no. 6, page 2268, 2022.

[7] Hani AlOmari, Qussai M. Yaseen, and Mohammed Azmi Al-Betar, "A Comparative Analysis of Machine Learning Algorithms for Android Malware Detection", *Procedia Computer Science*, vol. 220, pages 763–768, 2023, The 14th International Conference on Ambient Systems, Networks and Technologies Networks (ANT

2022) and The 6th International Conference on Emerging Data and Industry 4.0 (EDI40), ISSN: 1877-0509. DOI: https://doi.org/10.1016/j.procs.2023.03.101. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1877050923006361.

[8] Ebtesam J Alqahtani, Rachid Zagrouba, and Abdullah Almuhaideb, "A Survey on Android Malware Detection Techniques Using Machine Learning Algorithms.", in *2019 Sixth International Conference on Software Defined Systems (SDS)*, IEEE, 2019, pages 110–117.

[9] Muhammad Amin, Tamleek Ali Tanveer, Mohammad Tehseen, Murad Khan, Fakhri Alam Khan, and Sajid Anwar, "Static malware detection and attribution in Android byte-code through an end-to-end deep system", *Future generation computer systems*, vol. 102, pages 112–126, 2020.

[10] *Welcome to androguard's documentation! — androguard 3.4.0 documentation*, https://androguard.readthedocs.io/en/latest/index.html, (Accessed on 25/10/2023).

[11] *Android malware dataset | kaggle*, https://www.kaggle.com/datasets/saurabhshahane/android-malware-dataset, (Accessed on 25/10/2023).

[12] *Android malware static feature dataset(6 datasets) | kaggle*, https://www.kaggle.com/datasets/laxman1216/android-static-features-datasets6-features, (Accessed on 25/10/2023).

[13] *Apk (file format) - wikipedia*, https://en.wikipedia.org/wiki/Apk_(file_format), (Accessed on 25/10/2023).

[14] *App Manifest Overview | Android Developers*, https://developer.android.com/guide/topics/manifest/manifest-intro, (Accessed on 25/10/2023).

[15] *How Many Apps In Google Play Store? (Sep 2023)*, https://www.bankmycell.com/blog/number-of-google-play-store-apps/, (Accessed on 25/10/2023), Sep. 2023.

[16] *Are iphones more secure than android devices? | techtarget*, https://www.techtarget.com/searchmobilecomputing/tip/Are-iPhones-more-secure-than-Android-devices, (Accessed on 25/10/2023).

[17] Catarina Palma, Artur Ferreira, and Mário Figueiredo, "On the use of machine learning techniques to detect malware in mobile applications", Sep. 2023, Simpósio em Informática (INForum), Porto, Portugal.

[18] Catarina Palma, Artur Ferreira, and Mário Figueiredo, "A study on the role of feature selection for malware detection on Android applications", Oct. 2023, Portuguese Conference on Pattern Recognition (RECPAD), Coimbra, Portugal.

[19] *Bayes' theorem: What it is, formula, and examples*, https://www.investopedia.com/terms/b/bayes-theorem.asp, (Accessed on 25/10/2023).

[20] Catarina Palma, *Android malware detection with machine learning*, https://github.com/CatarinaPalma-325/Android-Malware-Detection-with-Machine-Learning, Available, 2023.

[21] *Chamois: The big botnet you didn't hear about | decipher*, https://duo.com/decipher/chamois-the-big-botnet-you-didnt-hear-about, (Accessed on 25/10/2023).

[22] *Android Permission Dataset | Kaggle*, https://www.kaggle.com/datasets/saurabhshahane/android-permission-dataset, (Accessed on 25/10/2023).

[23] Witten, H. I., Frank, E., Hall, M. A., and Pal, C. J, *Data Mining – Practical Machine Learning Tools and Techniques*, 4th ed. Morgan-Kaufmann, 2016.

[24] *Data Preprocessing in Machine Learning [Steps & Techniques]*, https://www.v7labs.com/blog/data-preprocessing-guide, (Accessed on 25/10/2023).

[25] *Android Malware Dataset for Machine Learning | Kaggle*, https://www.kaggle.com/datasets/shashwatwork/android-malware-dataset-for-machine-learning, (Accessed on 25/10/2023).

[26] R. Duda, P. Hart, and D. Stork, *Pattern classification*, 2nd edition. John Wiley & Sons, 2001.

[27] Meenu Ganesh, Priyanka Pednekar, Pooja Prabhuswamy, Divyashri Sreedharan Nair, Younghee Park, and Hyeran Jeon, "CNN-based Android malware detection", in *2017 International conference on software security and assurance (ICSSA)*, IEEE, 2017, pages 60–65.

[28] Przemyslaw Gilski and Jacek Stefanski, "Android OS: a review", *Tem Journal*, vol. 4, no. 1, page 116, 2015.

[29] *Gootloader, from seo poisoning to multi-stage downloader*, https://blogs.blackberry.com/en/2022/07/gootloader-from-seo-poisoning-to-multi-stage-downloader, (Accessed on 25/10/2023).

[30] *How many people have smartphones? [jul 2023 update] | oberlo*, https://www.oberlo.com/statistics/how-many-people-have-smartphones, (Accessed on 25/10/2023).

[31] *Hyperparameters of random forest classifier - geeksforgeeks*, https://www.geeksforgeeks.org/hyperparameters-of-random-forest-classifier/, (Accessed on 25/10/2023).

[32] Rejwana Islam, Moinul Islam Sayed, Sajal Saha, Mohammad Jamal Hossain, and Md Abdul Masud, "Android malware classification using optimum feature selection and ensemble machine learning", *Internet of Things and Cyber-Physical Systems*, vol. 3, pages 100–111, 2023, ISSN: 2667-3452. DOI: https://doi.org/10.1016/j.iotcps.2023.03.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S2667345223000202.

[33] Umme Sumaya Jannat, Syed Md Hasnayeen, Mirza Kamrul Bashar Shuhan, and Md Sadek Ferdous, "Analysis and detection of malware in Android applications using machine learning", in *2019 International Conference on Electrical, Computer and Communication Engineering (ECCE)*, IEEE, 2019, pages 1–7.

[34] Abdullah Talha Kabakus, "What static analysis can utmost offer for Android malware detection", *Information Technology and Control*, vol. 48, no. 2, pages 235–249, 2019.

[35] Mohammad Reza Keyvanpour, Mehrnoush Barani Shirzad, and Farideh Heydarian, "Android malware detection applying feature selection techniques and machine learning", *Multimedia Tools and Applications*, vol. 82, no. 6, pages 9517–9531, 2023. DOI: https://doi.org/10.1007/s11042-022-13767-2.

[36] *K-nearest neighbors algorithm - wikipedia*, https://en.wikipedia.org/wiki/K-nearest_neighbors_algorithm, (Accessed on 25/10/2023).

[37] *What is the k-nearest neighbors algorithm? | ibm*, https://www.ibm.com/topics/knn, (Accessed on 25/10/2023).

[38] JD Koli, "RanDroid: Android malware detection using random machine learning classifiers", in *2018 Technologies for Smart-City Energy Security and Power (ICSESP)*, IEEE, 2018, pages 1–6.

[39] Vasileios Kouliaridis and Georgios Kambourakis, "A comprehensive survey on machine learning techniques for Android malware detection", *Information*, vol. 12, no. 5, page 185, 2021.

[40] Wenjia Li, Zi Wang, Juecong Cai, and Sihua Cheng, "An Android malware detection approach using weight-adjusted deep learning", in *2018 International conference on computing, networking and communications (ICNC)*, IEEE, 2018, pages 437–441.

[41] Zhuo Ma, Haoran Ge, Zhuzhu Wang, Yang Liu, and Ximeng Liu, “Droidetec: Android malware detection and malicious code localization through deep learning”, *arXiv preprint arXiv:2002.03594*, 2020.

[42] Alejandro Martín, Alejandro Calleja, Héctor D Menéndez, Juan Tapiador, and David Camacho, “ADROIT: Android malware detection using meta-information”, in *2016 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, 2016, pages 1–8.

[43] Guozhu Meng, Yinxing Xue, Zhengzi Xu, Yang Liu, Jie Zhang, and Annamalai Narayanan, “Semantic modelling of Android malware for effective malware comprehension, detection, and classification”, in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pages 306–317.

[44] *Multilayer perceptron - wikiwand*, https://www.wikiwand.com/en/Multilayer_perceptron, (Accessed on 25/10/2023).

[45] Ash Turner, *Android vs. Apple Market Share: Leading Mobile OS (2023)*, https://www.bankmycell.com/blog/android-vs-apple-market-share/, (Accessed on 25/10/2023), Sep. 2023.

[46] Ali Muzaffar, Hani Ragab Hassen, Michael A Lones, and Hind Zantout, “An in-depth review of machine learning based Android malware detection”, *Computers & Security*, page 102 833, 2022.

[47] *Naive Bayes Classifier. What is a classifier? | by Rohith Gandhi | Towards Data Science*, https://towardsdatascience.com/naive-bayes-classifier-81d512f50a7c, (Accessed on 25/10/2023).

[48] Robin Nix and Jian Zhang, “Classification of Android apps and malware using deep neural networks”, in *International joint conference on neural networks (IJCNN)*, IEEE, 2017, pages 1871–1878.

[49] *Not so boring Android malware | Android-malware-samples*, https://maldroid.github.io/android-malware-samples/, (Accessed on 25/10/2023).

[50] *Numpy - wikipedia*, https://en.wikipedia.org/wiki/NumPy, (Accessed on 25/10/2023).

[51] Lucky Onwuzurike, Enrico Mariconti, Panagiotis Andriotis, Emiliano De Cristofaro, Gordon Ross, and Gianluca Stringhini, “Mamadroid: Detecting Android malware by building Markov chains of behavioral models (extended version)”, *ACM Transactions on Privacy and Security (TOPS)*, vol. 22, no. 2, pages 1–34, 2019.

[52] *Oversampling and undersampling in data analysis - Wikipedia*, https://en.wikipedia.org/wiki/Oversampling_and_undersampling_in_data_analysis, (Accessed on 25/10/2023).

[53] *Platform Architecture | Android Developers*, https://developer.android.com/guide/platform, (Accessed on 25/10/2023).

[54] *Principal Component Analysis (PCA) Explained | Built In*, https://builtin.com/data-science/step-step-explanation-principal-component-analysis, (Accessed on 25/10/2023).

[55] *Receiver operating characteristic - wikipedia*, https://en.wikipedia.org/wiki/Receiver_operating_characteristic, (Accessed on 25/10/2023).

[56] *Regression metrics for machine learning - machinelearningmastery.com*, https://machinelearningmastery.com/regression-metrics-for-machine-learning/, (Accessed on 25/10/2023).

[57] Majid Salehi, Morteza Amini, and Bruno Crispo, "Detecting malicious applications using system services request behavior", in *Proceedings of the 16th EAI International Conference on Mobile and Ubiquitous Systems: Computing, Networking and Services*, 2019, pages 200–209.

[58] *Sklearn.model_selection.gridsearchcv — scikit-learn 1.3.1 documentation*, https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.GridSearchCV.html, (Accessed on 25/10/2023).

[59] *Support Vector Machines(SVM) — An Overview | by Rushikesh Pupale | Towards Data Science*, https://towardsdatascience.com/https-medium-com-pupalerushikesh-svm-f4b42800e989, (Accessed on 25/10/2023).

[60] *Support Vector Machine — Introduction to Machine Learning Algorithms | by Rohith Gandhi | Towards Data Science*, https://towardsdatascience.com/support-vector-machine-introduction-to-machine-learning-algorithms-934a444fca47, (Accessed on 25/10/2023).

[61] *What is a Decision Tree | IBM*, https://www.ibm.com/topics/decision-trees, (Accessed on 25/10/2023).

[62] *What is a Random Forest? | TIBCO Software*, https://www.tibco.com/reference-center/what-is-a-random-forest, (Accessed on 25/10/2023).

[63] *What is back propagation? | h2o.ai*, https://h2o.ai/wiki/backpropagation/, (Accessed on 25/10/2023).

[64] *What is Scareware? How to Identify, Prevent and Remove It*, https://www.techtarget.com/whatis/definition/scareware, (Accessed on 25/10/2023).

[65] Wikipedia, *Android (operating system) — Wikipedia, the free encyclopedia*, http://en.wikipedia.org/w/index.php?title=Android%20(operating%20system)&oldid=1140366548, [Online; accessed 23-February-2023], 2023.

[66] Qing Wu, Xueling Zhu, and Bo Liu, "A survey of Android malware static detection technology based on machine learning", *Mobile Information Systems*, 2021.

[67] Shaojie Yang, Yongjun Wang, Haoran Xu, Fangliang Xu, and Mantun Chen, "An Android Malware Detection and Classification Approach Based on Contrastive Lerning", *Computers & Security*, vol. 123, page 102 915, 2022, ISSN: 0167-4048. DOI: https://doi.org/10.1016/j.cose.2022.102915. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S016740482200308X.

[68] Yi Zhang, Yuexiang Yang, and Xiaolei Wang, "A novel Android malware detection approach based on convolutional neural network", in *Proceedings of the 2nd International conference on cryptography, security and privacy*, 2018, pages 144–149.

[69] Hanqing Zhang, Senlin Luo, Yifei Zhang, and Limin Pan, "An efficient Android malware detection system based on method-level behavioral semantic analysis", *IEEE Access*, vol. 7, pages 69 246–69 256, 2019.

Experimental Results

A.1 Experimental Results: Baseline

Table A.1: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and classifier.

Dataset	Classifier	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	98.69	2831	12	47	1621	99.27	97.18	98.21	98.38
	SVM	98.22	2819	24	56	1612	98.53	96.64	97.58	97.89
	KNN	97.85	2802	41	56	1612	97.52	96.64	97.08	97.60
	NB	92.66	2597	246	85	1583	86.55	94.89	90.53	93.13
	MLP	98.94	2825	18	30	1638	98.91	98.20	98.56	98.78
CICAndMal2017	RF	79.14	2007	966	864	4935	83.63	85.10	84.36	76.30
	SVM	66.09	1	2972	3	5796	66.10	99.95	79.58	49.99
	KNN	62.01	805	2168	1164	4635	68.13	79.93	73.56	53.50
	NB	65.77	491	2482	521	5278	68.02	91.02	77.85	53.76
	MLP	69.80	1171	1802	847	4952	73.32	85.39	78.90	62.39
AM	RF	93.96	379	6	26	119	95.19	82.07	88.14	90.25
	SVM	72.83	379	6	138	7	53.84	4.83	8.86	51.62
	KNN	70.94	329	56	98	47	45.62	32.41	37.90	58.93
	NB	64.91	254	131	55	90	40.72	62.07	49.18	64.02
	MLP	77.17	379	6	115	30	83.33	20.69	33.15	59.57
AMSF	RF	99.33	747	6	4	749	99.21	99.47	99.33	99.33
	SVM	96.81	715	38	10	743	95.13	98.67	96.87	96.81
	KNN	98.80	744	9	9	744	98.80	98.80	98.80	98.80
	NB	98.87	743	10	7	746	98.68	99.07	98.87	98.87
	MLP	99.07	748	5	9	744	99.33	98.8	99.07	99.07

A.2 Experimental Results: Data Pre-processing

A.2.1 Missing values

Table A.2: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the CICAndMal2017 dataset and the RF classifier.

Method	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Removing instances with missing values	79.14	2007	966	864	4935	83.63	85.10	84.36	76.30
Removing features with missing values	79.86	2147	853	960	5040	85.53	84.00	84.76	77.78
Imputing missing values with the feature mean	79.81	2105	895	922	5078	85.02	84.63	84.82	77.40
Imputing missing values with the feature median	79.86	2104	896	916	5084	85.02	84.73	84.87	77.42
Imputing missing values with the feature mode	79.78	2090	910	910	5090	84.83	84.83	84.83	77.25

Table A.3: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the AM dataset and the RF classifier.

Method	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Removing instances with missing values	93.96	379	6	26	119	95.19	82.07	88.14	90.25
Removing features with missing values	92.45	2339	79	181	844	91.44	82.34	86.65	89.53
Imputing missing values with the feature mean	93.28	2363	55	176	849	93.92	82.83	88.02	90.28
Imputing missing values with the feature median	93.32	2359	59	171	854	93.54	83.32	88.13	90.44
Imputing missing values with the feature mode	93.32	2361	57	173	852	93.73	83.12	88.11	90.38

Table A.4: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the CICAndMal2017 dataset and the SVM classifier.

Method	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Removing instances with missing values	66.09	1	2972	3	5796	66.10	99.95	79.58	49.99
Removing features with missing values	66.18	276	2724	320	5680	67.58	94.67	78.86	51.93
Imputing missing values with the feature mean	66.13	231	2769	279	5721	67.39	95.35	78.96	51.51
Imputing missing values with the feature median	66.13	231	2769	279	5721	67.39	95.35	78.96	51.51
Imputing missing values with the feature mode	66.13	231	2769.0	279	5721	67.39	95.35	78.96	51.51

Table A.5: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for the different methods to deal with missing values using the AM dataset and the SVM classifier.

Method	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Removing instances with missing values	72.83	379	6	138	7	53.84	4.83	8.86	51.62
Removing features with missing values	70.26	2418	0	1024	1	100.0	0.10	0.19	50.05
Imputing missing values with the feature mean	70.23	2418	0	1025	0	0.0	0.0	0.0	50.00
Imputing missing values with the feature median	70.23	2418	0	1025	0	0.0	0.0	0.0	50.00
Imputing missing values with the feature mode	70.23	2418	0	1025	0	0.0	0.0	0.0	50.00

A.2.2 Normalisation

Table A.6: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with and without the use of Min-Max normalisation.

Classifier	Dataset	Normalisation	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
RF	Drebin	✗	98.69	2831	12	47	1621	99.27	97.18	98.21	98.38
		✓	98.58	2828	15	49	1619	99.08	97.06	98.06	98.27
RF	CICAndMal2017	✗	79.81	2105	895	922	5078	85.02	84.63	84.82	77.40
		✓	79.62	2081	919	915	5085	84.69	84.75	84.72	77.06
RF	AM	✗	93.28	2363	55	176	849	93.92	82.83	88.02	90.28
		✓	93.28	2361	57	174	851	93.72	83.02	88.05	90.33
RF	AMSF	✗	99.33	747	6	4	749	99.21	99.47	99.33	99.33
		✓	99.40	748	5	4	749	99.33	99.47	99.40	99.40
SVM	Drebin	✗	98.22	2819	24	56	1612	98.53	96.64	97.58	97.89
		✓	98.22	2819	24	56	1612	98.53	96.64	97.58	97.89
SVM	CICAndMal2017	✗	66.13	231	2769	279	5721	67.39	95.35	78.96	51.51
		✓	70.81	1213	1787	840	5160	74.28	86.00	79.71	63.22
SVM	AM	✗	70.23	2418	0	1025	0	0.0	0.0	0.0	50.00
		✓	90.88	2375	43	271	754	94.60	73.56	82.77	85.89
SVM	AMSF	✗	96.81	715	38	10	743	95.13	98.67	96.87	96.81
		✓	99.53	752	1	6	747	99.87	99.20	99.53	99.53

A.2.3 Numerosity balancing

Table A.7: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with different numerosity balancing techniques.

Dataset	Classifier	Numerosity reduction	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	None	98.58	2828	15	49	1619	99.08	97.06	98.06	98.27
		Random undersampling	98.50	1650	18	32	1636	98.91	98.08	98.49	98.50
		Random oversampling	99.26	2828	15	27	2816	99.47	99.05	99.26	99.26
		SMOTE	98.92	2828	15	46	2797	99.47	98.38	98.92	98.92
Drebin	SVM	None	98.22	2819	24	56	1612	98.53	96.64	97.58	97.89
		Random undersampling	98.02	1641	27	39	1629	98.37	97.66	98.00	98.02
		Random oversampling	98.50	2816	27	58	2785	99.03	97.96	98.50	98.50
		SMOTE	98.50	2814	29	56	2787	98.97	98.03	98.50	98.50
CICAndMal2017	RF	None	79.62	2081	919	915	5085	84.69	84.75	84.72	77.06
		Random undersampling	80.57	2599	401	765	2235	84.78	74.50	79.31	80.57
		Random oversampling	88.31	5785	215	1188	4812	95.72	80.20	87.28	88.31
		SMOTE	85.63	5337	663	1061	4939	88.16	82.32	85.14	85.63
CICAndMal2017	SVM	None	70.81	1213	1787	840	5160	74.28	86.0	79.71	63.22
		Random undersampling	70.03	2694	306	1492	1508	83.13	50.27	62.64	70.03
		Random oversampling	71.69	5456	544	2853	3147	85.26	52.44	64.95	71.69
		SMOTE	72.07	5422	578	2774	3226	84.81	53.76	65.81	72.07
AM	RF	None	93.28	2361	57	174	851	93.72	83.02	88.05	90.33
		Random undersampling	91.36	988	38	139	886	95.89	86.44	90.92	91.36
		Random oversampling	96.28	2333	85	95	2322	96.47	96.07	96.27	96.28
		SMOTE	94.06	2339	79	208	2209	96.55	91.39	93.89	94.06
AM	SVM	None	70.81	1213	1787	840	5160	74.28	86.0	79.71	63.22
		Random undersampling	89.18	984	42	180	845	95.26	82.44	88.39	89.17
		Random oversampling	89.47	2326	92	417	2000	95.60	82.75	88.71	89.47
		SMOTE	88.81	2326	92	449	1968	95.53	81.42	87.92	88.81
AMSF	RF	None	99.40	748	5	4	749	99.33	99.47	99.40	99.40
		Random undersampling	99.07	750	3	11	741	99.60	98.54	99.06	99.07
		Random oversampling	99.27	749	5	6	747	99.33	99.20	99.27	99.27
		SMOTE	99.33	750	4	6	747	99.47	99.20	99.33	99.33
AMSF	SVM	None	99.53	752	1	6	747	99.87	99.20	99.53	99.53
		Random undersampling	99.07	752	1	13	739	99.86	98.27	99.06	99.07
		Random oversampling	99.53	753	1	6	747	99.87	99.20	99.53	99.53
		SMOTE	99.53	753	1	6	747	99.87	99.20	99.53	99.53

A.3 Experimental Results: Feature Selection

Table A.8: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with RRFS with MM and FR (with M_s equal to 0.3).

Dataset	Classifier	RRFS with	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	MM	95.34	2753	90	175	2668	96.74	93.84	95.27	95.34
		FR	97.85	2793	50	72	2771	98.22	97.47	97.85	97.85
Drebin	SVM	MM	94.71	2751	92	209	2634	96.63	92.65	94.60	94.71
		FR	96.66	2778	65	125	2718	97.66	95.60	96.61	96.66
CICAndMal2017	RF	MM	86.10	5684	316	1352	4648	93.63	77.47	84.78	86.10
		FR	88.92	5765	235	1094	4906	95.43	81.77	88.07	88.92
CICAndMal2017	SVM	MM	60.04	3921	2079	2716	3284	61.23	54.73	57.80	60.04
		FR	68.52	4816	1184	2593	3407	74.21	56.77	64.34	68.52
AM	RF	MM	88.05	2253	165	413	2004	92.39	82.91	87.40	88.03
		FR	95.28	2281	137	91	2326	94.44	96.24	95.33	95.28
AM	SVM	MM	86.99	2260	158	471	1946	92.49	80.51	86.09	86.99
		FR	84.55	2270	148	599	1818	92.47	75.22	82.96	84.55
AMSF	RF	MM	99.80	754	0	3	750	100.0	99.60	99.80	99.80
		FR	99.07	751	3	11	742	99.60	98.54	99.07	99.07
AMSF	SVM	MM	99.87	754	0	2	751	100.0	99.72	99.87	99.87
		FR	98.41	754	0	24	729	100.0	96.81	98.38	98.41

Table A.9: Number of features (d) in each dataset before RRFS and after RRFS with MM and FR (with M_s equal to 0.3).

Dataset	d (Original)	d (after RRFS with MM)	d (after RRFS with FR)
Drebin	215	100	94
CICAndMal2017	110	37	26
AM	181	39	13
AMSF	966	222	122

Table A.10: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with RRFS with FR and different values of M_s.

Dataset	Classifier	M_s	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	0.4	98.80	2816	27	41	2802	99.05	98.56	98.80	98.80
		0.3	97.85	2793	50	72	2771	98.22	97.47	97.85	97.85
		0.2	97.06	2779	64	103	2740	97.72	96.38	97.04	97.06
Drebin	SVM	0.4	97.66	2802	41	92	2751	98.53	96.76	97.64	97.66
		0.3	96.66	2778	65	125	2718	97.66	95.60	96.61	96.66
		0.2	96.52	2797	46	152	2691	98.32	94.65	96.45	96.52
CICAndMal2017	RF	0.4	88.85	5752	248	1090	4910	95.19	81.83	88.01	88.85
		0.3	88.92	5765	235	1094	4906	95.43	81.77	88.07	88.92
		0.2	89.20	5765	235	1061	4939	95.46	82.32	88.40	89.20
CICAndMal2017	SVM	0.4	68.69	4763	1237	2520	3480	73.78	57.99	64.94	68.69
		0.3	68.52	4816	1184	2593	3407	74.21	56.77	64.34	68.52
		0.2	67.97	4827	1173	2669	3331	73.96	55.52	63.42	67.97
AM	RF	0.4	95.89	2307	111	87	2330	95.45	96.39	95.92	95.89
		0.3	95.28	2281	137	91	2326	94.44	96.24	95.33	95.28
		0.2	91.50	2166	252	159	2258	89.96	93.42	91.66	91.50
AM	SVM	0.4	86.16	2273	145	524	1893	92.89	78.32	84.98	86.16
		0.3	84.55	2270	148	599	1818	92.47	75.22	82.96	84.55
		0.2	83.56	2352	66	729	1688	96.24	69.84	80.94	83.55
AMSF	RF	0.4	99.00	748	6	9	744	99.20	98.80	99.00	99.00
		0.3	99.07	751	3	11	742	99.60	98.54	99.07	99.07
		0.2	99.07	749	5	9	744	99.33	98.80	99.07	99.07
AMSF	SVM	0.4	98.14	752	2	26	727	99.72	96.55	98.11	98.14
		0.3	98.41	754	0	24	729	100.0	96.81	98.38	98.41
		0.2	98.14	754	0	28	725	100.0	96.28	98.11	98.14

Table A.11: Number of features (d) in each dataset before RRFS and after RRFS with FR for different values of M_s.

Dataset	d (Original)	d (after RRFS ∣ Ms=0.4)	d (after RRFS ∣ Ms=0.3)	d (after RRFS ∣ Ms=0.2)
Drebin	215	107	94	86
CICAndMal2017	110	27	26	23
AM	181	15	13	8
AMSF	966	179	122	63

Table A.12: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers without RRFS and with RRFS with FR and M_s equal to 0.3.

Dataset	Classifier	RRFS	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	✗	99.26	2828	15	27	2816	99.47	99.05	99.26	99.26
		✓	97.85	2793	50	72	2771	98.22	97.47	97.85	97.85
Drebin	SVM	✗	98.50	2816	27	58	2785	99.03	97.96	98.50	98.50
		✓	96.66	2778	65	125	2718	97.66	95.60	96.61	96.66
CICAndMal2017	RF	✗	88.31	5785	215	1188	4812	95.72	80.20	87.28	88.31
		✓	88.92	5765	235	1094	4906	95.43	81.77	88.07	88.92
CICAndMal2017	SVM	✗	71.69	5456	544	2853	3147	85.26	52.44	64.95	71.69
		✓	68.52	4816	1184	2593	3407	74.21	56.77	64.34	68.52
AM	RF	✗	96.28	2333	85	95	2322	96.47	96.07	96.27	96.28
		✓	95.28	2281	137	91	2326	94.44	96.24	95.33	95.28
AM	SVM	✗	89.47	2326	92	417	2000	95.60	82.75	88.71	89.47
		✓	84.55	2270	148	599	1818	92.47	75.22	82.96	84.55
AMSF	RF	✗	99.27	749	5	6	747	99.33	99.20	99.27	99.27
		✓	99.07	751	3	11	742	99.60	98.54	99.07	99.07
AMSF	SVM	✗	99.53	753	1	6	747	99.87	99.20	99.53	99.53
		✓	98.41	754	0	24	729	100.0	96.81	98.38	98.41

A.3.1 Most relevant features in the Drebin dataset

Most relevant features in the Drebin dataset selected by RRFS with FR (M_s=0.3):

1. `transact`
2. `SEND_SMS`
3. `Ljava.lang.Class.getCanonicalName`
4. `android.telephony.SmsManager`
5. `Ljava.lang.Class.getField`
6. `RECEIVE_SMS`
7. `Ljavax.crypto.spec.SecretKeySpec`
8. `WRITE_SMS`
9. `READ_SYNC_SETTINGS`
10. `TelephonyManager.getSubscriberId`
11. `CAMERA`
12. `AUTHENTICATE_ACCOUNTS`
13. `Ljava.lang.Class.forName`
14. `INSTALL_PACKAGES`
15. `READ_HISTORY_BOOKMARKS`
16. `android.telephony.gsm.SmsManager`
17. `android.intent.action.PACKAGE_REPLACED`
18. `Binder`
19. `android.intent.action.SEND_MULTIPLE`
20. `abortBroadcast`
21. `URLClassLoader`
22. `ACCESS_LOCATION_EXTRA_COMMANDS`
23. `NFC`
24. `MODIFY_AUDIO_SETTINGS`
25. `WRITE_APN_SETTINGS`

26. android.intent.action.TIME_SET
27. BROADCAST_STICKY
28. BIND_REMOTEVIEWS
29. android.intent.action.PACKAGE_REMOVED
30. getCallingPid
31. READ_PROFILE
32. READ_SYNC_STATS
33. createSubprocess
34. WAKE_LOCK
35. android.intent.action.TIMEZONE_CHANGED
36. RESTART_PACKAGES
37. android.intent.action.PACKAGE_ADDED
38. chmod
39. Ljava.lang.Class.getDeclaredClasses
40. android.intent.action.ACTION_POWER_DISCONNECTED
41. TelephonyManager.getSimSerialNumber
42. PathClassLoader
43. TelephonyManager.getCallState
44. BLUETOOTH
45. READ_CALENDAR
46. READ_EXTERNAL_STORAGE
47. Runtime.load
48. TelephonyManager.getSimCountryIso
49. READ_CALL_LOG
50. SUBSCRIBED_FEEDS_WRITE
51. sendMultipartTextMessage
52. HttpPost.init
53. PackageInstaller
54. android.intent.action.ACTION_SHUTDOWN
55. remount
56. Ljava.lang.Class.getClasses
57. TelephonyManager.isNetworkRoaming
58. sendDataMessage
59. WRITE_CALENDAR
60. SUBSCRIBED_FEEDS_READ
61. chown
62. HttpUriRequest
63. CHANGE_WIFI_MULTICAST_STATE
64. MASTER_CLEAR
65. DELETE_PACKAGES
66. GET_TASKS
67. android.intent.action.PACKAGE_DATA_CLEARED
68. UPDATE_DEVICE_STATS
69. GLOBAL_SEARCH
70. WRITE_CALL_LOG
71. android.intent.action.PACKAGE_CHANGED
72. REORDER_TASKS
73. DELETE_CACHE_FILES
74. android.intent.action.NEW_OUTGOING_CALL
75. SET_WALLPAPER
76. divideMessage

77. WRITE_USER_DICTIONARY
78. BIND_INPUT_METHOD
79. Runtime.exec
80. WRITE_PROFILE
81. PROCESS_OUTGOING_CALLS
82. BIND_WALLPAPER
83. CALL_PRIVILEGED
84. BATTERY_STATS
85. READ_USER_DICTIONARY
86. ACCESS_COARSE_LOCATION
87. READ_SOCIAL_STREAM
88. RECEIVE_WAP_PUSH
89. android.intent.action.SENDTO
90. WRITE_SETTINGS
91. DUMP
92. TelephonyManager.getNetworkOperator
93. SET_TIME
94. /system/bin

A.3.2 Most relevant features in the CICAndMal2017 dataset

Most relevant features in the CICAndMal2017 dataset selected by RRFS with FR (M_s=0.3):

1. Category
2. Price
3. Network communication : view network state (S)
4. Your location : access extra location provider commands (S)
5. System tools : set wallpaper (S)
6. Description
7. Number of ratings
8. Related apps
9. System tools : automatically start at boot (S)
10. System tools : send sticky broadcast (S)
11. Dangerous permissions count
12. Default : delete applications (S)
13. Package
14. Default : bind to a wallpaper (S)
15. System tools : read sync settings (S)
16. Default : power device on or off (S)
17. Default : Install DRM content. (S)
18. Default : Access DRM content. (S)
19. Default : delete other applications' data (S)
20. System tools : read sync statistics (S)
21. Default : display unauthorized windows (S)
22. Your accounts : read Google service configuration (S)
23. Default : read frame buffer (S)
24. Default : directly install applications (S)
25. Default : modify secure system settings (S)
26. Default : interact with a device admin (S)

A.3.3 Most relevant features in the AM dataset

Most relevant features in the AM dataset selected by RRFS with FR (M_s=0.3):

1. `com.android.launcher.permission.UNINSTALL_SHORTCUT`
2. `android.permission.VIBRATE`
3. `android.permission.ACCESS_FINE_LOCATION`
4. `name`
5. `android.permission.BLUETOOTH_ADMIN`
6. `android.permission.WAKE_LOCK`
7. `android.permission.READ_EXTERNAL_STORAGE`
8. `android.permission.RECORD_AUDIO`
9. `android.permission.ACCESS_NETWORK_STATE`
10. `android.permission.CAMERA`
11. `android.permission.GET_TASKS`
12. `.//Signature`
13. `android.permission.READ_USER_DICTIONARY`

A.3.4 Most relevant features in the AMSF dataset

Most relevant features in the AMSF dataset selected by RRFS with FR (M_s=0.3):

1. `androidpermissionSEND_SMS`
2. `android.telephony.SmsManager.sendTextMessage`
3. `float-to-int`
4. `android.telephony.SmsManager`
5. `android.support.v4.widget`
6. `android.intent.action.DATA_SMS_RECEIVED`
7. `or-int/2addr`
8. `com.software.CHECKER`
9. `android.content.pm`
10. `android.widget.Button.startAnimation`
11. `android.support.v4.content`
12. `java.lang.Long.longValue`
13. `java.lang.String.replaceAll`
14. `shr-long`
15. `java.io.DataOutputStream.close`
16. `div-double`
17. `android.view.View.findViewById`
18. `java.lang.StringBuilder.delete`
19. `android.content.res.XmlResourceParser.getAttributeValue`
20. `android.location.LocationManager`
21. `or-long/2addr`
22. `add-double/2addr`
23. `shl-int/2addr`
24. `java.security.NoSuchAlgorithmException`
25. `mul-double/2addr`

26. org.json.JSONObject.get
27. com.google.android.c2dm.intent.REGISTRATION
28. android.webkit.WebView.loadUrl
29. or-int/lit8
30. com.google.firebase.INSTANCE_ID_EVENT
31. org.json.JSONObject.optString
32. android.widget.CheckBox
33. javax.microedition.khronos.egl
34. android.graphics.Canvas.drawText
35. java.io.DataOutputStream
36. sub-double/2addr
37. android.webkit.WebView.setWebChromeClient
38. android.accounts
39. android.os.Environment.getExternalStorageDirectory
40. android.app.AlarmManager.cancel
41. android.view.animation.ScaleAnimation.setDuration
42. android.app.Application
43. div-int/lit16
44. android.app.Activity.startActivity
45. aget-char
46. java.lang.Double.valueOf
47. android.app.PendingIntent.send
48. android.widget.RelativeLayout
49. fill-array-data
50. org.apache.http.params.HttpConnectionParams.setSoTimeout
51. int-to-short
52. android.database.Cursor.getInt
53. shr-int/2addr
54. android.widget.EditText.setText
55. java.lang.Process
56. RECEIVE
57. android.content.Intent.putExtras
58. java.net.URLConnection
59. android.os.Handler.sendEmptyMessage
60. java.io.DataInputStream
61. java.lang.Exception.getMessage
62. android.text.Editable
63. shr-long/2addr
64. android.widget.TextView.setLayoutParams
65. android.app.Activity.getSystemService
66. java.lang.Exception.toString
67. ushr-int
68. org.apache.http.params
69. android.view.inputmethod
70. java.lang.annotation
71. android.net.conn.CONNECTIVITY_CHANGE
72. com.google.android.gms.measurement.UPLOAD
73. java.io.BufferedWriter
74. android.content.SharedPreferences.getInt
75. java.io.BufferedReader.readLine
76. or-long

77. GETTASKS
78. RECORDAUDIO
79. android.webkit.WebView.canGoBack
80. xor-int/lit16
81. android.widget.Button.setEnabled
82. rem-int/lit16
83. android.app.Activity.getResources
84. or-int
85. move-object
86. android.telephony.TelephonyManager.getSimCountryIso
87. javax.xml.parsers
88. androidpermissionSYSTEM_ALERT_WINDOW
89. android.view.animation.Transformation
90. android.opengl
91. BROADCAST
92. mul-double
93. android.database.Cursor.isAfterLast
94. java.security.cert
95. ushr-int/2addr
96. android.content.Intent.setClass
97. android.location.LocationManager.requestLocationUpdates
98. java.io.File.getParent
99. org.apache.http.conn.ssl
100. android.preference
101. android.widget.ProgressBar.getProgress
102. java.lang.StringBuffer.append
103. android.text.util
104. android.net.wifi.WifiManager
105. and-long
106. android.speech.tts
107. android.telephony.TelephonyManager.getSimOperatorName
108. android.appwidget.action.APPWIDGET_UPDATE
109. nop
110. java.util.logging
111. android.content.pm.PackageManager.getInstalledPackages
112. neg-double
113. iput-byte
114. android.intent.action.BOOT_COMPLETED
115. int-to-char
116. android.app.ProgressDialog.dismiss
117. android.content.res.XmlResourceParser.close
118. android.preference.PreferenceManager
119. org.w3c.dom
120. filled-new-array/range
121. android.graphics.Camera.save
122. com.google.android.gms.iid.InstanceID

A.4 Experimental Results: CV & Hyperparameter tuning

Table A.13: Experimental results in the form of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers with and without tuning of the hyperparameters.

Dataset	Classifier	Hyperparameter tuning	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	✗	97.85	2793	50	72	2771	98.22	97.47	97.85	97.85
		✓	97.91	2795	48	71	2772	98.30	97.50	97.89	97.91
Drebin	SVM	✗	96.66	2778	65	125	2718	97.66	95.60	96.61	96.66
		✓	97.47	2780	63	81	2762	97.77	97.15	97.46	97.47
CICAndMal2017	RF	✗	88.92	5765	235	1094	4906	95.43	81.77	88.07	88.92
		✓	89.07	5769	231	1081	4919	95.50	81.98	88.23	89.07
CICAndMal2017	SVM	✗	68.52	4816	1184	2593	3407	74.21	56.77	64.34	68.52
		✓	73.22	5408	592	2621	3379	85.09	56.32	67.78	73.22
AM	RF	✗	95.28	2281	137	91	2326	94.44	96.24	95.33	95.28
		✓	95.37	2283	135	89	2328	94.52	96.32	95.41	95.37
AM	SVM	✗	84.55	2270	148	599	1818	92.47	75.22	82.96	84.55
		✓	86.45	2221	197	458	1959	90.86	81.05	85.68	86.45
AMSF	RF	✗	99.07	751	3	11	742	99.60	98.54	99.07	99.07
		✓	99.20	750	4	8	745	99.47	98.94	99.20	99.20
AMSF	SVM	✗	98.41	754	0	24	729	100.0	96.81	98.38	98.41
		✓	98.47	749	5	18	735	99.32	97.61	98.46	98.47

Table A.14: Hyperparameters (deemed more relevant) for the RF classifier optimised for each dataset.

Dataset	Number of trees	Max depth of the trees	Function to measure the quality of the split
Drebin	200	None	log_loss
CICAndMal2017	400	None	log_loss
AM	600	None	entropy
AMSF	500	None	log_loss

Table A.15: Hyperparameters (deemed more relevant) for the SVM classifier optimised for each dataset.

Dataset	C	kernel type	kernel coefficient
Drebin	14	RBF	scale
CICAndMal2017	19	RBF	scale
AM	18	RBF	scale
AMSF	15	RBF	scale

Table A.16: Experimental results in the form of the mean and standard deviation (std) of the evaluation metrics, accuracy (Acc), confusion matrix, precision, recall (Rec), F1-score and AUC-ROC for each dataset and the RF and SVM classifiers, with LOOCV and 10-fold CV.

Dataset	Classifier	CV	Measure	Acc (%)	TN	FP	FN	TP	Precision (%)	Rec (%)	F1-Score (%)	AUC-ROC (%)
Drebin	RF	10-Fold	mean	98.08	931.9	15.7	20.6	927.0	98.34	97.83	98.08	98.08
			std	0.44	1.69	1.88	4.49	4.49	0.48	0.73	0.45	0.44
Drebin	RF	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
Drebin	SVM	10-Fold	mean	97.74	929.6	18.0	24.8	922.8	98.09	97.38	97.72	97.74
			std	0.36	4.89	5.31	1.63	1.63	0.53	0.59	0.36	0.36
Drebin	SVM	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
CICAndMal2017	RF	10-Fold	mean	89.57	1907.7	92.3	325.0	1675.0	94.99	83.75	88.97	89.57
			std	2.07	4.96	4.96	5.43	5.43	2.07	4.45	1.96	2.07
CICAndMal2017	RF	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
CICAndMal2017	SVM	10-Fold	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
CICAndMal2017	SVM	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
AM	RF	10-Fold	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
AM	RF	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
AM	SVM	10-Fold	mean	81.08	651.4	154.4	150.4	655.4	82.06	81.34	81.45	81.08
			std	7.01	175.7	175.7	14.4	14.4	9.41	1.46	5.4	7.01
AM	SVM	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
AMSF	RF	10-Fold	mean	99.32	249.8	1.3	2.1	249.0	99.48	99.16	99.32	99.32
			std	0.43	0.47	0.81	0.81	0.94	0.50	0.49	0.43	0.43
AMSF	RF	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-
AMSF	SVM	10-Fold	mean	98.65	248.3	2.8	4.0	247.1	98.89	98.41	98.64	98.65
			std	0.62	2.44	2.49	0.94	1.41	1.02	0.82	0.62	0.62
AMSF	SVM	LOO	mean	-	-	-	-	-	-	-	-	-
			std	-	-	-	-	-	-	-	-	-

A.5 Experimental Results: Real-world Applications

Table A.17: Experimental results obtained using the RF classifier and each dataset that assess if the prototype of the proposed approach correctly predicts or not the existence of malicious content, based on the features extracted from APK files of real-world Android applications.

APK	Dataset	Correctly predicted
App1 (Benign)	Drebin	✓
	CICAndMal2017	✓
	AM	✗
	AMSF	✓
App2 (Malicious)	Drebin	✓
	CICAndMal2017	✗
	AM	✓
	AMSF	✓
App3 (Benign)	Drebin	✓
	CICAndMal2017	✗
	AM	✓
	AMSF	✓
Whatsapp (Benign)	Drebin	✓
	CICAndMal2017	✗
	AM	✓
	AMSF	✗
Amazon (Benign)	Drebin	✓
	CICAndMal2017	✗
	AM	✓
	AMSF	✗
SMS stealer (Malicious)	Drebin	✗
	CICAndMal2017	✓
	AM	✗
	AMSF	✗
Screen locker (Malicious)	Drebin	✗
	CICAndMal2017	✓
	AM	✗
	AMSF	✓
Unwanted calls (Malicious)	Drebin	✗
	CICAndMal2017	✓
	AM	✗
	AMSF	✓