INSTITUTO UNIVERSITÁRIO DE LISBOA

Malware detection methods for Android mobile applications

João Pedro Lapa da Silva Lopes

Master in Telecommunications and Computer Engeneering

Supervisors:
Professor Carlos Serrão, Professor Auxiliar,
ISCTE-IUL

Professor Luís Nunes, Professor Auxiliar,
ISCTE-IUL

September, 2020

Acknowledgements

This thesis would not have been possible without the manifested support throughout its writing and my academic journey as a whole.

Firstly, I would like to thank my supervisors Carlos Serrão and Luís Nunes, for their availability to help whenever I had any doubts and for the knowledge they transmitted to me. Their dedication and support were paramount to the conclusion of this stage of my academic journey.

I would also like to thank my family for all the love, support, and most importantly, for allowing me to pursue this academic career. I would also like give a special thanks to my sister Sílvia, for always wanting the best for me and encouraging me through all the tough times.

A very special thanks to my girlfriend, Inês Sousa, for teaching me that I should always aim higher and give my best effort in everything that I do. I will be forever grateful for her constant love, care, and support because I think I could not have finished this journey without it.

Lastly, I would like to thank my friends for their friendship and support.

This work is part of the AppSentinel project, co-funded by Lisboa2020/Portugal2020/EU in the context of the Portuguese Sistema de Incentivos à I&DT - Projetos em Copromoção (project 33953). The authors also would like to acknowledge the FCT Project UIDB/MULTI/04466/2020 (ISTAR-IUL) and UIBD/EEA/50008/2020 (Instituto de Telecomunicações). The authors also appreciate Aptoide's collaboration for providing support and data that was used in this study.

Abstract

Advancements in mobile computing are attracting traditional device users to transition toward mobile platforms to fulfil their data processing needs. Among these, the Android platform is the most popular, holding the majority of the market share due to its open-source policy and ability to install applications from different application stores. This fact, coupled with the amount of sensitive data these devices now store, makes it attractive for malware authors to attack the Android platform, causing a large influx of malicious applications in the ecosystem. Traditional malware detection methods cannot effectively control and prevent this influx, demanding an automatic and intelligent approach such as machine learning. In this thesis, three machine learning algorithms, XGBoost, SVM and K-NN were trained with several features, with a focus on Android permissions and applications' static features, to measure the effectiveness of applying machine learning techniques to combat the proliferation of malware. Given the dataset's goodware to malware ratio of 99/1, four experiments with an under-sampled version of the dataset with a ratio of 70/30 were conducted to test different subsets of the feature space as well as feature elimination and aggregation before training the algorithms with the full set of features using feature normalization across two distinct scenarios. This approach showed promising results, with XGBoost, SVM and K-NN distinguishing between malware and goodware with a score of 90 % (Area Under the Receiver Operating Curve values).

Keywords: malware detection, machine learning, Android, security, mobile

Resumo

Os avanços na computação móvel estão a atrair utilizadores de dispositivos tradicionais a transitar para as plataformas móveis para atender às suas necessidades de processamento de dados. Entre estas, a plataforma Android é a mais popular, detendo a maioria da quota de mercado devido à sua política open-source e capacidade de instalar aplicações através de várias lojas de aplicações. Este facto, conjuntamente com a quantidade de dados sensíveis que estes dispositivos agora armazenam, torna o ataque à plataforma Android atraente para os autores de malware, causando um grande fluxo de aplicações maliciosas no ecossistema. Os métodos tradicionais de deteção de malware não conseguem controlar e prevenir este fluxo eficazmente, exigindo uma abordagem automática e inteligente, como a aprendizagem automática. Nesta tese, três algoritmos de aprendizagem automática, XGBoost, SVM e K-NN, foram treinados com diversas características, focando-se nas permissões Android e características estáticas das aplicações, para medir a eficácia da aplicação de técnicas de aprendizagem automática no combate à proliferação de malware. Dado o rácio de goodware para malware de 99/1 do conjunto de dados, realizaram-se quatro experiências com uma versão subamostrada do mesmo com um rácio de 70/30 para testar diferentes subconjuntos do espaço de características bem como eliminação e agregação de características antes de treinar os algoritmos com o conjunto completo de características usando normalização de características em dois cenários. Esta abordagem apresentou resultados promissores, com XGBoost, SVM e K-NN distinguindo entre malware e goodware com um *score* de 90 % (valores *Area Under the Receiver Operating Curve*).

Palavras-chave: deteção de malware, aprendizagem automática, Android, segurança, móvel

Table of Contents

List of Figures

List of Tables

List of Abbreviations

AAPT – Android Asset Packaging Tool

ANN – Artificial Neural Networks

API – Application Programming Interface

APK – Application Package

ART – Android Runtime

AUC – Area Under the ROC Curve

BN – Bayesian Network

CFG – Control Flow Graph

CRISP-DM – Cross-Industry Standard Process for Data Mining

DL – Deep Learning

DT – Decision Tree

FPR – False Positive Rate

GBT – Gradient Boosted Trees

IDE – Integrated Development Environment

IG – Information Gain

K-NN – K-Nearest Neighbours

LD – Linear Discriminant

LR – Logistic Regression

NB – Naïve Bayes

NFC – Near Field Communication

OS – Operating System

OWASP – Open Web Application Security Project

PC – Principal Component

PCA – Principal Component Analysis

RF – Random Forest

ROC – Receiver Operating Characteristic

RT – Random Tree

SMO – Sequential Minimal Optimization

SVM – Support Vector Machine

TPR – True Positive Rate

t-SNE - t-distributed Stochastic Neighbour Embedding

UI – User Interface

UID – Unique User ID

URL – Uniform Resource Locator

VM – Virtual Machine

XGBoost – eXtreme Gradient Boosting

XML – Extensible Markup Language

1. Introduction

1.1. Motivation

Malicious software, or malware, has been a part of computing ever since the first ever documented computer virus, albeit experimental, in 1970. From this point forward, malware has not only grown from a diversity standpoint – having the need to categorize malware applications based on their proliferation method and/or how it affects its victim such as viruses, worms, trojans, spyware, ransomware, etc – but also from a volume standpoint, with AV-TEST registering 1081.61 million total malware as of August 13th 2020 [1]. The former is due to the never-ending battle between security professionals which are constantly creating new forms of malware protection and prevention, and malware developers which in turn innovate through the discovery of new attack vectors and proliferation methods, whereas the latter is due to technological advances such as the Internet, the emergence of smartphones, etc. As the Internet became more accessible to the general public [2], it also became an important tool to perform a variety of tasks remotely such as home banking, communication via e-mail/social media, etc. These tasks involve sensitive information that is very attractive to attackers. The Internet is also a very effective vehicle for malware developers to disseminate malicious software to obtain such information [3].

Although the majority of attacks target desktop computers, the emergence of smartphones and their subsequent rise in popularity quickly turned these mobile devices into an appealing attack vector for malware developers [3]. They are so popular in fact, that Statista forecasts the number of mobile and smartphone users worldwide will reach 7.33 billion in 2023 and 3.8 billion in 2021 [4] [5]. This is due to their portability, high computational power, ability to connect ubiquitously to the Internet and to acquire additional functionality through the installation of applications provided by application stores.

Among the most used mobile platforms (Android and iOS), Android is the most popular, holding 74.6% of the mobile Operating System (OS) market share due to its open-source approach, providing a free Integrated Development Environment (IDE), without any hardware restrictions as well as an application approval policy that is more lenient than its main competitor iOS, which in turn is very strict and extensive [6]. Furthermore, to publish iOS applications, it is necessary to pay a yearly subscription fee, creating an additional barrier of entry which also contributes to Android’s popularity. The Android OS also allows the installation of applications from third-party application stores and unverified sources.

These factors make Android an appealing platform for malware authors to instrument their attacks, resulting in the rapid growth of Android malware, both in quantity and sophistication [7]. This lowers the effectiveness of Android application stores' malware detection methods because they cannot cope with the volume of malware that is being developed.

These stores usually have an application review process in order to determine if a submitted application is accepted into the store. For example, the Google Play Store' application review process consists of reviewing if it achieves Android base-level security, followed up by an automated review and a manual review in addition to having developer policies that application developers must adhere to if they want their application to be distributed. If these policies are violated, the application is not published, and the developer is notified with information about the violation and after it addresses these issues it can be resubmitted for review. The application can also be suspended due to violations of the developer policies. Repeated violations, such as malware, can also lead to the termination of the accounts that are owned by the developer. Once the review process is completed and the application is published into the Google Play Store, it is verified continuously through Google Play Protect, which uses machine learning techniques to detect malicious applications and activities such as impersonation and fraud while also protecting Android devices [8] [9].

Given that malware detection is an important deciding factor of Android application stores' application approval process, it is important to research methods that could enable the development of intelligent and automated Android malware detection methods, namely using machine learning techniques, in order to combat the rampant proliferation of malware.

To address this problem, Aptoide – a reputable and popular open-source third-party Android application store without geo-restrictions which provides an user-generated content platform where every user can create their own application stores – in partnership with ISCTE-IUL proposed the AppSentinel project – which this thesis is integrated on – with the aim of researching and developing a cloud-based malware detection system using machine learning techniques that will be implemented in its current security system [10]. Using both static and dynamic analysis, this system will use applications from multiple sources to discover their patterns and test future applications that are introduced to its store. Additionally, the extraction of static and dynamic characteristics will be used to aid the comprehension of an applications' vulnerabilities in accordance with the Open Web Application Security Project's (OWASP) mobile top 10 mobile risks [11]. Afterwards, the combination of these two components are used to create a profile for each application and together with user feedback a threat level will be determined. The system also aids the developer by sending good software practices according to the vulnerabilities found during the analysis.

In this thesis, several machine learning algorithms that are fed with features that are extracted through the employment of static analysis to a large quantity of Android applications provided by Aptoide will be tested, in order to conclude if this approach can help mitigate Android malware proliferation in application stores. Additionally, Aptoide shared valuable insights and knowledge about the malware detection and machine learning domains throughout this thesis.

1.2. Research Questions

This work aims to answer the following research question:

- How effective can machine learning techniques be, using only static data, when applied to malware detection in a real-world scenario?

1.3. Objectives

The main objective of this thesis is to determine if the usage of machine learning techniques can be an effective approach for Android malware detection, contributing to the improvement of Android malware detection. In contrast to the majority of the machine learning-based Android malware detection methods found in the literature, this research is conducted using real, current data on a large scale.

Another important objective is the development of a malware detection software prototype, which implements the most effective malware detection methods in an automated manner found by researching the literature. Additionally, the prototype will be used to provide experimental proof regarding the effectiveness of the chosen malware detection methods and to compare the obtained results against the ones found in the literature.

1.4. Structure

The remainder of this thesis is structured as follows: Chapter 2 will present an overview of the mobile malware detection landscape with a focus on the Android platform, followed by an overview of the Android platform's application components and security, and finally, the literature review of static, dynamic and hybrid-analysis-based machine learning detection approaches. Chapter 3 will explain the CRISP-DM standard – the methodology chosen to conduct the experimental procedures in this thesis –, the reason behind its adoption and present the outputs of each phase that are applicable to all four experimental procedures. Chapter 4 will focus on the presentation and discussion of the results of each experimental procedure. Chapter 5 concludes this thesis and proposes ideas for future work.

1.5. Contributions

This thesis' contributions threefold: (i) to show a proof of concept for the application of machine learning algorithms fed with features obtained through static analysis with a focus on Android permissions using real data; (ii) a systematic study on the adequacy of classification techniques in the field of Android malware detection and (iii) an analysis on the importance of Android permissions to Android malware detection as well as a proposal for feature engineering.

2. Literature Review

Mobile computing technology has been advancing rapidly, and with the emergence of smartphones, due to their ability to not only perform data processing tasks that are employed by desktop computers, but also provide a mobile platform to do so, their adoption does not show signs of stopping, given how attractive these features are for end-users [4]. These tasks – messaging, health and fitness, productivity, home banking, payments, etc are provided in the form of mobile applications, pieces of software that are mainly distributed through application stores. Thus, mobile devices store a wide variety of data, most of which contain personal and sensitive information, becoming an attractive target for malware authors to instrument their attacks on [12]. The Android platform in particular suffers the majority of these attacks in the form of premium-rate SMS trojans, spyware, botnets, aggressive adware, ransomware, etc, due to the fact that is it the most popular out of the two most used mobile platforms (Android and iOS), making up 74.6 % of the mobile OS market share given that its open-source policy enables mobile device manufacturers to use it as the base of their respective OS versions [6] [7]. In addition, Android's official application store, Google Play Store, is not the only location from where users can download applications. Users can download applications from third-party stores or directly from the web, giving malware authors multiple paths to distribute malicious applications, especially those which are disguised as benign applications [12]. The combination of these factors contributed to the exponential increase of Android malware, and traditional signature-based detection methods lose their effectiveness when faced with this increase in volume. Therefore, is it imperative to develop automatic and intelligent malware detection methods to prevent the proliferation of Android malware, and machine learning techniques could be leveraged to achieve this objective.

2.1. Android platform application components and security

Android is an open-source platform providing a Linux-based operative system for mobile devices, and its major platform architecture components are shown in Figure 2.1 [13]. Additionally, it provides an application environment in order to install developer-made applications which are composed of four main components: the AndroidManifest.xml file, activities, services and broadcast receivers [14]. These applications are then compiled into a file called the Android application package (APK), which contains the application's code in the form of files with a .dex extension, resources, assets and the AndroidManifest.xml file, as shown in Figure 2.2 [15].

The AndroidManifest.xml file describes essential information about a given application to the Android build tools, Google Play, but most importantly, to the Android OS [16]. This file instructs the system about how to use an application's activities, services, services and content providers, and describes which permissions are required to execute it [14].

Activities can be thought of as a single screen within an application, providing it with a window where it can draw its User Interface (UI) on. An application can have as many activities as it needs. Therefore, activities enable user interaction with an application [17] [18].

Services are application components that can perform background tasks (such as network operations) that are transparent to the user and can run indefinitely. Services can also perform foreground tasks (such as playing music while the user interacts with another application), but they must notify the user whenever it is being executed [17] [19].

Receivers are messages that an application is interested in receiving in order to perform an operation afterwards. When a specific event occurs (such as the completion of the download of a file), the application that triggered it sends a broadcast message that notifies every application that specified that it desires to receive such message [17] [20].

Security-wise, the Android platform provides user-based protection by assigning a Unique User ID (UID) to each application and forcing them to run in their own separate process. An application is also unable to interact with others and has limited access to the OS by running it within an application sandbox [21]. The purpose of this security mechanism is twofold: protect applications and OS from malicious applications [21]. Because of this characteristic, applications need to share resources and data with each other in a deliberate manner. This is achieved using the concept of permissions, where an application declares the need of a given permission to access resources and device features that are outside its sandbox [22]. In addition, it also uses a secure inter-process communication feature to allow applications which are executed in different processes to communicate with each other [23].

Permissions are an integral element to the security of an Android user, more specifically, its privacy. Permissions control which user data (such as contacts and emails) and system features (such as the camera and Near Field Communication (NFC)) a given Android application can access and use. They are so important to the security of the Android platform that no application has permission to perform actions that could impact a user (such as reading and writing its private data), other applications or the operating system negatively by default [22].

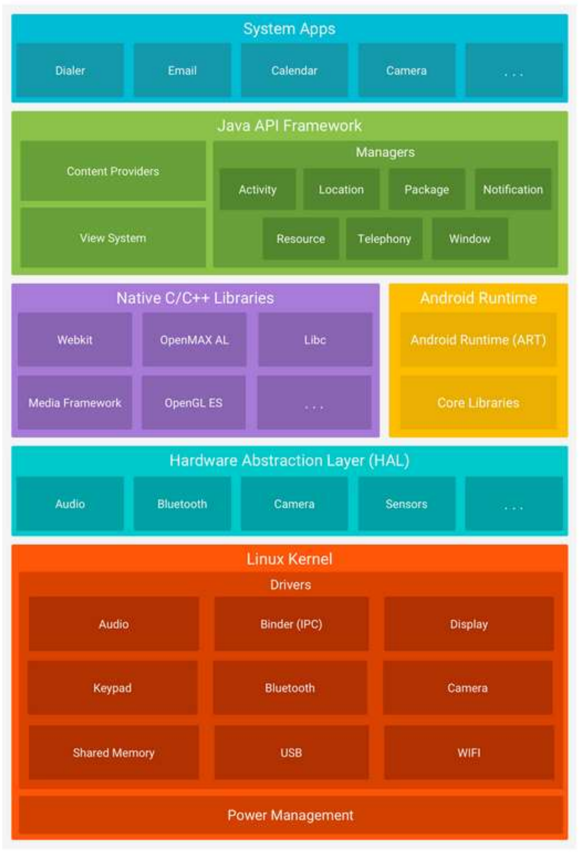


Figure 2.1 – Android platform main components (source: [13]*)*

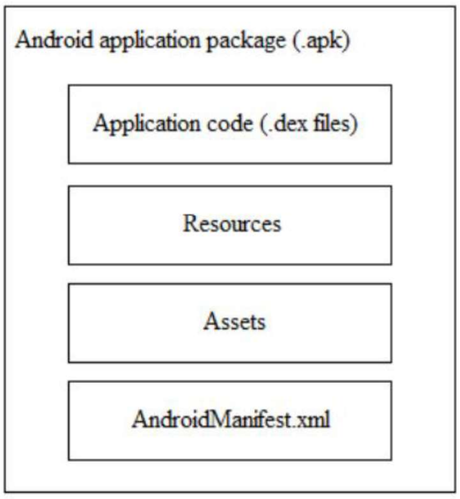

Android application package (.apk)
Application code (.dex files)
Resources
Assets
AndroidManifest.xml

Figure 2.2 – Android application package contents (adapted from [24]*)*

The following sub-chapters encompass the study of the state of the art of machine learning-based Android malware detection. These methods can fall into one of three categories, depending on how it analyses software to extract features in order to train the machine learning algorithms: static analysis-based, dynamic analysis-based and hybrid analysis-based.

2.2. Static analysis-based malware detection

Static analysis, which is the focus of this thesis, consists of analysing an application's source code without having to execute it [25]. In the Android platform, this implies the analysis of the contents of the APK file [26]. The advantage of this type of analysis is that it is fast and low on resource consumption. However, it is vulnerable to both code obfuscation techniques and dynamically loaded code [26] [27].

Sanz et al. [28] developed a static malware detection method that leverages the contents of the AndroidManifest.xml file. To extract this file from the APK, it uses a tool named Android Asset Packaging Tool (AAPT) [29]. Two specific fields from this file were used as features: uses-permission, which lists every permission that the application needs to operate correctly and uses-feature, which declares hardware and software features the application needs (for instance, the compass sensor) [16]. These features were used to train the following algorithms: Logistic Regression (LR), Naive Bayes (NB), Bayesian Network (BN), Sequential Minimal Optimization (SMO), an implementation of K-Nearest Neighbours (K-NN) named IBk, Decision Tree (J48), Random Tree (RT) and Random Forest (RF). To train these algorithms, it was used a dataset comprised of 249 malware samples and 357 benign samples achieving the best performance (Area Under the Curve (AUC)) of 0.920 with the RF algorithm.

Peiravian and Zhu [30] developed a malware detection framework using permissions and Application Programming Interface (API) calls as features. This information is obtained using the tool Apktool [31] to reverse engineer a given APK, extracting its AndroidManifest.xml file and class files. For a given application, the permissions are extracted from the AndroidManifest.xml file and are embedded in a binary vector P, where $P_i = 1$ if the ith permission is requested in its AndroidManifest.xml file, otherwise, $P_i = 0$. The API calls are extracted from the class files, and as a result, every application is represented by a single binary vector of permissions and API calls in addition to a benign or malicious class label. These features were used to train the following algorithms: Support Vector Machine (SVM), Decision Tree (DT) and Bagging [32]. The used dataset is comprised of 610 malware samples and 1250 benign samples. Three experiments were conducted using different feature combinations. Using permissions, the best performing algorithm was Bagging with an AUC (defined in Chapter 3.4) of 0.956. Using API calls, SVM achieved the best performance with an AUC of 0.957. Using both permissions and API calls, the best performing algorithm achieved an AUC of 0.991.

D.Arp, M. Spreitzenbarth, M. Hübner et al. [33] developed a malware detection method in the form of an application that is installed on an Android smartphone .Given that the malware detection process occurs in the device itself, it needs to be lightweight. Therefore, the static features that are used to train the machine learning algorithms need to be extracted efficiently. To achieve this, these features are extracted from two specific locations: the AndroidManifest.xml file using the tool AAPT [29] and from the application's Dalvik bytecode using a self-developed disassembler to minimize the feature extraction time, resulting in eight sets of features. Four of those sets were extracted from the AndroidManifest.xml file, namely: requested hardware components (e.g. GPS, camera access etc), requested permissions, application components (activities, services, content providers and broadcast receivers) and filtered intents. The remaining four sets were extracted from the disassembled Dalvik bytecode, namely: restricted API calls, used permissions, suspicious API calls and network addresses. Restricted API calls are, as the name implies, a set of sensitive API calls that the Android permission system restricts access to. These API calls are useful as features because using them without requesting the corresponding permission could mean that a given application is using privilege escalation exploits. The used permissions set is created by matching the restricted API call set with the requested permissions in order to determine which permissions are requested and used. The suspicious API call set contains API calls that allow access to sensitive data or resources given that in most cases, they can lead to malicious behaviour. Finally, the net address set include IP addresses, hostnames and Uniform Resource Locators (URL's). These feature sets are merged into a single feature set S containing approximately 545,000 features, which are then embedded in a binary S-dimensional vector where each dimension s has a value of 1 if a given application contains that feature or 0 otherwise. This process is employed for every sample in the dataset. The dataset contains 131,611 samples comprising applications from various Android application markets, including every sample from the Android Malware Genome Project [34]. In order to determine if a given sample is malicious or benign, every sample is scanned using the VirusTotal service, using ten anti-virus scanners (AntiVir, AVG, Bit- Defender, ClamAV, ESET, F-Secure, Kaspersky, McAfee, Panda and Sophos) [35]. Every sample that is classified as malicious by at least two scanners is declared as malicious in the dataset, otherwise, it is declared as benign. Furthermore, every sample that is classified as adware is removed from the dataset since this type of application is in a grey area between malicious and benign. After this labelling process, the final dataset is comprised of 123,453 benign samples and 5,560 malicious applications, that are fed to an SVM algorithm for training, achieving an AUC of 0.939. Another feature of Drebin is the explanation of the detection results it yields. When an application is scanned, Drebin presents a screen with a detecting score representing how confident the classification is as well as the top k features indexed by their weights, which are the features that contributed the most in classifying it as malicious or benign along with a description of the functionality of each top feature.

C. Zhao, W. Zheng, L. Gong et al. [36] used a subset of API calls as features to train an ensemble of DT and K-NN as the base classifiers to detect Android malware. This subset is generated by extracting the API calls of a given application using the tool Androguard [37] to decompile the `class.dex` file and applying a regular expression pattern to get all the methods from it. Afterwards, a sensitivity score is computed for each extracted API call representing the correlation between each API call and its appearance in malicious applications. Given that there exists numerous API calls, tests were conducted using the true positive rate as the metric to narrow down the number of API calls to use as features to train the algorithms. Observing the results of those tests illustrated in pp. 145, figure 5, [36], the optimal number of API calls to use is 20, therefore the top 20 most sensitive API calls were chosen. The name of the chosen API's as well as their respective sensitivity scores are illustrated in pp. 145, table 1, [36]. To determine the number of neighbours k to use when training the K-NN classifier, the algorithm was trained using a dataset of 450 benign samples and 450 malicious samples and tested with 100 samples using various values for k. Based on the results, which are illustrated in pp. 147, figure 4, [36], the optimal number for k is 5. Finally, the ensemble was trained using a dataset comprised of 516 benign samples and 528 malicious samples. This paper also explored the impact of using an ensemble model rather than standalone classifiers, the effect of different classifiers and the effects of different weights of the classifiers within the ensemble module. In the first experiment, it concluded that using an ensemble improves both the accuracy as well as the false positive rate (FPR), achieving an average accuracy above 90 %, as shown in pp. 147, figure 6, [36]. The second experiment concluded that using the accuracy and the true positive rate (TPR) as performance metrics, using K-NN and DT as the classifiers of the ensemble yielded the best results, as shown in pp. 148, figure 8, [36]. The final experiment concluded that the ensemble performs the best when K-NN and DT have a weight of 0.4 and 0.6 respectively, achieving an accuracy of approximately 90 % as shown in pp. 148, figure 9, [36].

M. Kumaran and W. Li [38] developed a lightweight malware detection method using the information provided by the AndroidManifest.xml file, namely permissions and intent-filters, in order to learn if it proves to be enough in order to classify applications as benign or malicious. To extract these features, each application is decompiled using the tool Apktool [31] to gain access to AndroidManifest.xml file. Afterwards, this file is fed to a Python Extensible Markup Language (XML) API named ElementTree to extract both intents and permissions. This process results in 183 features that can belong in three categories: requested permissions (permissions to access phone functionality like location, camera, etc.), declared permissions (permissions that are created by a given application in order to protect itself from others that try to access data in it) and intent filters, which are used to tell which intents an application can use. These features are used to train the following algorithms using a dataset comprised of 500 malicious applications and 500 benign applications, using 10-fold cross-validation: Linear Discriminant (LD), Cubic SVM, Weighted K-NN, Complex Tree (DT), Linear SVM and Course K-NN. Cubic SVM proved to be the best performing algorithm with an accuracy of 91.7 %, as shown in pp. 2, figure 2, [38]. Additionally, this paper concluded that solely using intent-filters as features yields poor classification results and that using both permissions and intent-filters leads to the best performance, as shown in pp. 2, figure 1, [38].

K. Allix, T. Bissyandé, Q. Jérome et al. [39] developed a malware detection method based on features extracted from control flow graphs (CFG's), using SVM, RF, the RIPPER rule-learning algorithm and the tree-based C4.5 algorithm as classifiers. The feature extraction process begins with using the tool Androguard [37] to perform static analysis on a given Android application's bytecode and extract a representation of its CFG. The CFG is then represented as character strings using a method developed by Pouik et al. [40] , which holds information about the application's code structure while discarding information that is less useful such as variable names or register numbers. This representation allows it to be protected against obfuscation given that two malware variants can have the same CFG while having different bytecode. With this alternative representation of an application's CFG, all basic blocks – sequences of instructions of the CFG with only an entry point and an exit point – are extracted from it. An important property of these basic blocks is that they represent the smallest piece of the application that is always executed collectively. If a basic block is noted as BB_i then BB_{all} can be defined as the set of the n basic blocks found in at least one application, as seen in (1):

$$BB_{all} = \{BB_1, BB_2, \dots, BB_n\} \qquad (1)$$

And thus, every application is represented by a list of binary values that encode all the blocks in BB_{all}, where if a basic block is present the corresponding element has a value of 1, otherwise it has the value 0. This paper establishes two scenarios of malware detection: in the lab and in the wild. The "in-the-lab scenario" is characterized by using a dataset comprised of a few thousand samples at most, as well as employing 10-fold cross validation to test the machine learning algorithms. An "in-the-wild" scenario is a real-world malware detection scenario. Using a dataset comprised of over 50 000 applications, two sets are created: $Set_{\propto}$, containing all known malware and a randomly chosen subset of Google Play's applications dataset, which are labelled as goodware, and a second set, Set_{∂}, which is comprised of the remaining subset of Google Play's applications dataset. Set_{∂} is always used for testing and $Set_{\propto}$ can be used as a training set in an in the wild scenario or as a combined training and testing dataset. With the datasets for each scenario created, the next step is feature evaluation and selection, where the InfoGain feature evaluation implemented by the Weka software [41] is computed for every feature. In the feature selection step, every feature that had a null InfoGain score is discarded, which comprised of over 99% of the features (over 2.5 million). The steps of the overall system are illustrated in pp. 9, figure 1, [39]. In the "in-the-lab scenario", various experiments were conducted in order to assess the performance of the developed malware detection method, the impact of class imbalance, the sensitivity to the number of used features and the performance of each classifier. In the performance assessment experiment, which was comprised of 960 10-fold cross-validation experiments with all combinations of possible parameter values (10 repetitions per algorithm × 4 goodware to malware ratios × 6 values for number of features × 4 algorithms), the distribution of performance was as follows: the majority of the classifiers achieved very high precision rates with a median of 0.94, as well as high recall and F1-scores (defined in Chapter 3.4) with a median of 0.91. The class imbalance experiment consisted of using various goodware to malware ratios: 1/2, 1, 2 and 3, corresponding to 620, 1257, 2500 and 3500 goodware applications. This experiment showed that the classifiers perform the best when the goodware to malware ratio is in favour of goodware, as shown in pp. 13, figure 3, [39]. The feature number experiment showed that with a range of 50, 250, 500, 1000, 1500 and 5000 features, the classifier performs better as the number of features increase, as shown in pp. 14, figure 4, [39]. Regarding the performance of the different classifiers, RF, the RIPPER rule-learning algorithm and C4.5 showed high F1-scores while SVM had an overall lower F1-score, as shown in pp. 14, figure 5, [39]. In the "in-the-wild scenario", the classifiers drop abruptly in performance, achieving a distribution of precision values with a median of 0.11, as well as recall and F1-score values of approximately 0 as shown in pp. 16, figure 7, [39]. The variation of goodware to malware ratio yielded the same trend as the in the lab experiment. However, as the number of features increase, the performance drops in contrast to the in the lab scenario, as shown in pp. 18, figure 11, [39]. This paper also highlights the importance of using datasets that are large and of good quality (that contain goodware samples that aren't in fact unknown malware applications) in order to improve the performance of classifiers in a real world scenario, as shown in pp. 19, figure 12, [39].

Table 2.1 displays the performance of the previously mentioned static analysis-based Android malware detection methods that used the AUC as their performance measure.

Table 2.1 – Performance of malware detection methods based on static analysis-obtained features that use AUC as its performance metric

References	Features	AUC (Technique used)
[28]	Permissions and Used Features	0.890 (LR)
		0.780 (NB)
		0.790 (BN)
		0.860 (SMO)
		0.900 (IBK)
		0.860 (J48)
		0.850 (RT)
		0.920 (RF)
[30]	Permissions	0.917 (J48)
		0.920 (SVM)
		0.956 (Bagging)
	API Calls	0.918 (J48)
		0.957 (SVM)
		0.956 (Bagging)
	Permissions and API Calls	0.936 (J48)
		0.963 (SMV)
		0.991 (Bagging)
[33]	Hardware Components, Requested Permissions, Application Components, Filtered Intents, Restricted API Calls, Used Permissions and Suspicious API Calls	0.939 (SVM)

Table 2.2 displays the performance of the static analysis-based Android malware detection methods previously mentioned that used only Accuracy as their performance metric.

Table 2.2 – Performance of malware detection methods based on static analysis-obtained features that use Accuracy as its performance metric

References	Features	Accuracy (%) (Technique Used)
[14]	Sensitive API calls	~90 (K-NN and DT Ensemble)
[16]	Permissions and Intent filters	91.7 (Cubic SVM)

2.3. Dynamic analysis-based malware detection

Dynamic analysis, in contrast to static analysis, consists of executing a given application in a sandbox environment to monitor its behaviour. However, it is more time consuming than static analysis methods [7].

Singh and Hofmann [25] developed a malware detection method using the frequency of system calls as features. The first stage of this process consists of executing each application of the sample set, which is comprised of 216 malicious samples and 278 benign samples in an emulator using the tool Monkey [42]. This tool generates pseudorandom user actions (clicks, touches, gestures and system-level events) [16]. As every application is being executed, a total of 337 Linux system calls are monitored, resulting in a feature vector of 337 elements, where each element represents how many times that specific system call was invoked during runtime. In the next stage, every system call that has zero variance is removed from the feature set, resulting in a final feature vector of 43 attributes, excluding the class label. These features are used to train the following algorithms: DT, RF, Gradient Boosted Trees (GBT), K-NN, SVM, Artificial Neural Networks (ANN) and Deep Learning (DL). In order to improve the performance of the algorithms, three feature weighing techniques were also applied before training and testing the algorithms once more, namely, Information Gain (IG), Chi-square statistic and correlation.

Bhatia and Kaushal [43] also used frequency of invoked system calls at runtime as features. Using a dataset comprised of 50 malicious samples and 50 benign samples, every application is executed in an Android Virtual Machine (VM) using the Monkey tool for one minute, generating 500 gestures with a 500 millisecond delay between each event, while the Linux command strace is executed in parallel to extract the frequencies of every invoked system call during that period [42]. This information is aggregated in a single matrix where each row represents the frequency of the system calls of a given application and each column represents the frequency of a given system call for every application. The algorithms that were chosen were J48 and RF.

Afonso, de Amorim, Grégio, Junquera, and de Geus [44] developed a malware detection system using the frequency of both API and system calls that are invoked at runtime. In order to extract the API calls, the tool APIMonitor [45] is executed for five minutes while it is being executed on an emulator using the tool MonkeyRunner [46]. Furthermore, the file that handles the collection of API calls contained in this tool was modified in order to monitor additional API calls related to network access, process execution, string and file manipulation and information reading. The Linux command strace is also used during this period in order to extract the system calls. This information is aggregated into a vector of 74 API calls and 90 system calls, amounting to a total of 164 features, each one representing how many times that particular API or system call was invoked. Using a dataset of 2295 malicious samples and 1485 benign samples, the following algorithms were trained in order to determine which one will be used by the proposed method: RF, J48, LR, NB, BN, SMO, and IBk. RF achieved the best performance with an F1-score of 0.96 using the dataset mentioned above, therefore it was tested afterwards using a dataset comprised of 2257 malware samples and 1483 benign samples.

Table 2.3 displays the performance of the Android malware detection methods mentioned above, using F1-Score as the performance metric considering that it is the one that is shared among every study.

Table 2.3 – Performance of malware detection methods based on dynamic analysis-obtained features

References	Features	F1-score (Technique used)
[25]	System Calls, no feature weighing	0.946 (RF)
		0.943 (SVM)
		0.973 (DT)
		0.976 (GBT)
		0.901 (K-NN)
		0.912 (ANN)
		0.937 (DL)
	System Calls, using IG	0.939 (RF)
		0.966 (SVM)
		0.972 (DT)
		0.981 (GBT)
		0.961 (K-NN)
		0.952 (ANN)
		0.977 (DL)
	System Calls, using Chi-square statistic	0.946 (RF)
		0.966 (SVM)
		0.967 (DT)
		0.981 (GBT)
		0.960 (K-NN)
		0.946 (ANN)
		0.965 (DL)
	System Calls, using correlation	0.961 (RF)
		0.969 (SVM)
		0.972 (DT)
		0.991 (GBT)
		0.986 (K-NN)
		0.920 (ANN)
		0.968 (DL

[43]	System Calls	0.850 (J48)
		0.885 (RF)
[44]	System Calls and API Calls	0.968 (RF)

2.4. Hybrid-analysis-based malware detection

Hybrid analysis methods consist of using both static and dynamic analysis methods in order to overcome their respective limitations [27].

Zhao, Xu and Zhang [47] developed a system that extracts permissions and API calls as static features and runtime behaviour as dynamic features in order to classify applications. In the static analysis process, the tool Androguard [37] is used to extract the permissions from the AndroidManifest.xml file, resulting in a permission feature set that is further optimized in order to remove features that are rarely present. This results in a binary permission feature vector of 45 dimensions, representing the presence of each permission in each application. Additionally, the API calls of applications from various sample sets are extracted through the analysis of their respective classes.dex files, using both Androguard and the reverse-engineering tool baksmali [48]. In order to optimize the obtained API feature vector, the filter feature selection algorithm Relief [49] is used, resulting in a final API call feature set of 22 dimensions where each dimension represents an API call. In the dynamic analysis process, every application is installed and executed on an emulator. In order to extract runtime behaviours as features, the tool Monkey [42] is executed while the tool DroidBox [50] monitors the runtime behaviour to determine whether a given application exhibits malicious behaviour such as automatic network connection, malicious SMS sending, private information logging, among others. Additionally, the number of occurrences of each behaviour is registered and the Relief algorithm is used to remove irrelevant features, resulting in a final feature vector of 20 dimensions such as battery usage, user activity, network features, among others. Afterwards, this information is aggregated into a single feature vector with 87 dimensions. Using a dataset comprised of 359 malware samples and 500 benign samples, 150 malicious samples and 150 benign samples were chosen randomly to form training and testing datasets, which were used by the following algorithms: SVM, K-NN, NB, DT and RF. Using features that were extracted from static analysis, the best performing algorithm was RF with an accuracy of 92.07 %. Using both static and dynamic analysis derived features, the best performing algorithm was RF with an accuracy of 94.89 %.

Liu, Zhang, Li and Chen [51] developed a method that employs static analysis or dynamic analysis depending on the result of the APK extraction process. Using the tool Apktool [31], if it can successfully decompile a given application, it proceeds to the static analysis stage. However, if it does not produce useful information (for instance, if code obfuscation techniques were used) it employs dynamic analysis. In the static analysis stage, the AndroidManifest.xml file is extracted from each application and every permission is mapped to a feature vector of 151 dimensions. Additionally, every API call is extracted using the tool baksmali [48] and is mapped to a feature vector of 3262 dimensions. Afterwards, both feature vectors are merged, resulting in a final feature vector of 3413 dimensions. In the dynamic analysis phase, a system call feature vector of 345 dimensions is created where each dimension represents the frequency of the invoked system calls. To extract these features, the ADB (Android Debug Bridge) tool [52] is used. Afterwards, the application is executed using the Monkey [42] tool and the invoked system calls are monitored using the Linux command strace. Using a dataset comprised of 500 malicious samples and 500 benign samples, the following algorithms were trained: K-NN, SVM and NB. Using permissions as the feature set, the best performing algorithm was SVM with an accuracy of 96.53 %. Using API calls, the best performing algorithm was also SVM with an accuracy of 99.07 %. Using both permissions and API calls, SVM performed the best with an accuracy of 99.28 %. Finally, using system calls as features, the best performing algorithm was NB with an accuracy of 90 %.

Arshad et al. [27] developed a hybrid malware detection model where the static analysis phase is carried out on a remote server and the dynamic analysis phase is employed on the device. This model is composed of two major components: a client application in which the dynamic analysis process occurs, and a remote server that handles the static analysis process as well as the training and testing of the machine learning algorithms. The client application is developed in order to let the user dynamically analyse an application of his/her choice. Once it does, the client application hooks the Linux command "strace" with that application which monitors its invoked system calls. The client application was programmed to monitor the frequency of 10 specific system calls related to file operations and network access. Afterwards, a system call log file is generated and sent to the remote server. In the static analysis phase, the remote server receives the application identifier through the client application, and the server queries its database to check if it was not previously classified. If so, a report is sent back to the user, otherwise the server downloads the application and employs static analysis. This process consists of the extraction of several features such as requested hardware components, requested permissions, application components (services, broadcast receivers and content providers), intent filters, suspicious API calls and restricted API calls. The first four are extracted from the application's AndroidManifest.xml file using the AAPT tool [29]. The last two are extracted from disassembling the application code from the classes.dex file using the baksmali [48] tool. Afterwards, the remote server generates both static and dynamic binary feature vectors to train the following algorithms: SVM, RF, DT and NB. The mentioned algorithms were evaluated using the Drebin dataset [33], which is comprised of 5,560 malware samples and 123,453 benign samples. Using only the features extracted from static analysis, the best performing algorithm was RF with an accuracy of 99.07 %. Using dynamic analysis-derived features, the best performing algorithms were both RF and SVM with an accuracy of 82.76 %.

Table 2.4 displays the performance of the Android malware detection methods previously mentioned, using Accuracy as the performance metric.

Table 2.4 – Performance of malware detection methods based on hybrid analysis-obtained features

References	Features	Accuracy (%)
[47]	Static	85.74 (NB)
		88.19 (J48)
		92.07 (RF)
		91.27 (SVM)
		84.56 (K-NN)
	Hybrid	84.52 (NB)
		89.34 (J48)
		94.89 (RF)
		93.66 (SVM)
		86.71 (K-NN)
[51]	Permissions	93.33 (NB)
		96.52 (SVM)
		95.58 (K-NN)
	API Calls	94.23 (NB)
		99.07 (SVM)
		98.42 (K-NN)
	Permissions and API Calls	94.41 (NB)
		99.28 (SVM)
		98.66 (K-NN)
	System Calls	90.00 (NB)
		85.75 (SVM)
		87.92 (K-NN)
[27]	Static	91.60 (NB)
		99.07 (RF)
		98.97 (SVM)
		98.56 (DT)
	Dynamic	62.07 (NB)
		82.76 (RF)
		82.76 (SVM)
		72.41 (DT)

Observing Table 2.1, Table 2.2, Table 2.3 and Table 2.4, and, although the majority of the malware detection methods deliver high accuracy rates, there are some concerns regarding how they would perform in a realistic scenario due to: i) using datasets with goodware to malware ratios above 90 % [27] [33] and ii) using controlled datasets with a low number of total samples, the lowest being 100 total samples [25] [28] [36] [38] [43] [47] [51]. The latter is particularly highlighted in [39], where using small controlled datasets and using 10-fold cross-validation lead to high performances. However, when those classifiers were used in a realistic scenario, their performances plummeted [39].

In this thesis, a large dataset with 55378 samples provided by Aptoide, a well-known third-party Android application market will be used, therefore the trained algorithms will produce results that we have a high degree of confidence on regarding their performance given that it represents a realistic malware detection scenario.

3. Experimental Methodology

To answer the research question defined in Chapter 1, we need to employ machine learning techniques to address a classification problem. In other words, given a set of Android applications, we want to be able to distinguish between benign and malicious applications. To achieve this, the CRISP-DM reference model [53] will be adopted to train and test several machine learning classifiers that can distinguish goodware from malware. Given that the AppSentinel project follows this standard since it is a widely approved blueprint for data mining and machine learning projects in a business context, it is appropriate to adopt this methodology in this thesis, given that it is integrated in the project.

Figure 3.1 illustrates the phases of the CRISP-DM reference model. This model represents the lifecycle of a data mining/machine learning project, which is composed of a set of phases and their respective tasks as well as how these tasks relate to each other, albeit it cannot capture every possible relationship because they change based on which project they are integrated in [53].

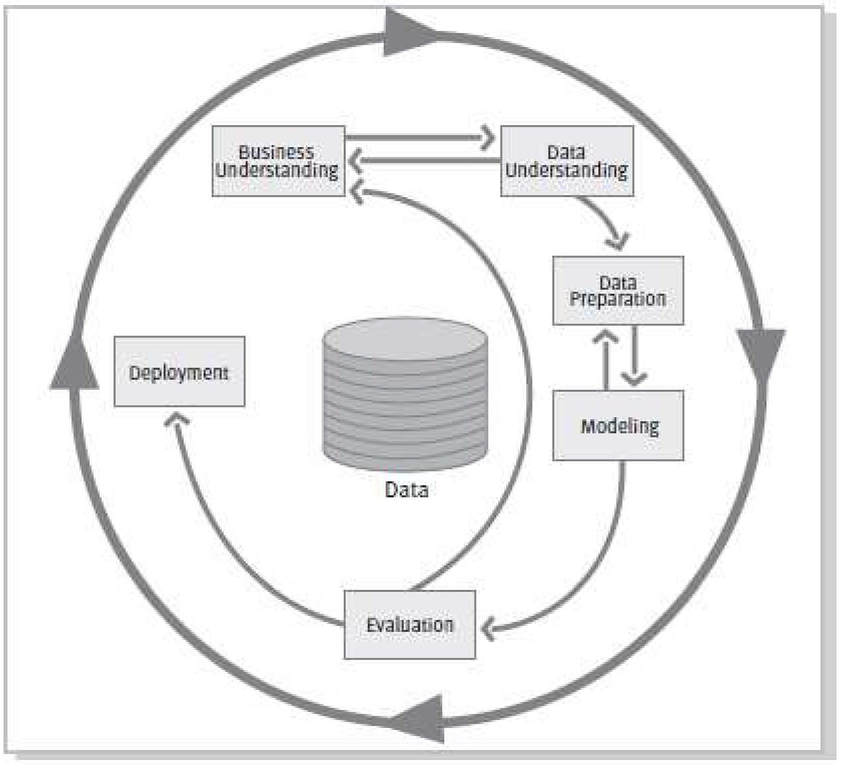


Figure 3.1 - CRISP-DM reference model [53]

There is a total of 6 phases in the CRISP-DM reference model, and there is not a pre-defined order at which a given phase is performed. More specifically, the result of a given phase will dictate which phase or task of a given phase should be performed next. The inner arrows indicate the major dependencies between phases. The outer circle represents the idea that the data mining process (and in this case, the machine learning process) is cyclical. This means that the deployment of a solution does not end the process. Instead, what is learned from a given cycle can be used to improve the solution in the consequent cycle [53].

3.1. Business Understanding

The main purpose of this phase is to comprehend the project's goals and requirements from a business perspective and derive a machine learning problem definition from them as well as a plan to accomplish the project's objectives [53]. The output of this phase can be found in Chapter 1 section 1.

3.2. Data Understanding

This phase is characterised by an initial data collection process, followed by data exploration tasks in order to become acquainted with the data's structure, examine superficial and obvious properties of the data such as the number of records, attributes and attribute data types (nominal, ordinal or continuous) and the existence of missing values or irregularities in order to analyse the quality of the data [53].

Observing Table 3.1, the dataset is composed of 55378 APK's provided by Aptoide across 29 days of the month of July of 2019, with a peak in number of samples at 2266 on the 8th of July. In contrast to the majority of papers that were studied in the Chapter 2, this dataset has the advantage of having a substantially larger sample size.

Table 3.1 - General statistics of the dataset

Number of samples	55378
Number of days	29
Day with the most samples	2019-07-08
Minimum number of samples per day	1371
Maximum number of samples per day	2266

Observing Table 3.2 and Figure 3.2, the dataset is heavily unbalanced, containing much more goodware samples. More specifically, this dataset has a goodware to malware sample ratio of approximately 99/1. Furthermore, this discrepancy between the number of goodware samples versus malware samples is maintained throughout the days. Given that the dataset was provided by Aptoide, a reputable Android application market, it is natural for it to have such a low ratio of malware samples to goodware samples given that it is a store with the purpose of distributing applications worldwide. Most importantly, this dataset represents a realistic scenario, meaning that the results will be more reliable in contrast to the majority of papers that try to tackle this problem [39].

Table 3.2 – Daily goodware and malware distribution

Date	**Goodware**	**Malware**	**Total**
2019-07-01	1734	16	1750
2019-07-02	1759	20	1779
2019-07-03	1920	25	1945
2019-07-04	1903	17	1920
2019-07-05	2148	24	2172
2019-07-06	1673	22	1695
2019-07-07	1560	20	1580
2019-07-08	2241	25	2266
2019-07-09	2040	24	2064
2019-07-10	2232	15	2247
2019-07-11	1968	9	1978
2019-07-12	2075	11	2086
2019-07-13	1825	19	1844
2019-07-14	1607	28	1635
2019-07-15	1542	11	1553
2019-07-16	2014	37	2051
2019-07-17	2100	55	2155
2019-07-18	2234	15	2249
2019-07-19	2136	28	2164
2019-07-20	1971	27	1998
2019-07-21	1371	10	1381
2019-07-22	1824	11	1835
2019-07-23	1775	14	1789
2019-07-24	2113	11	2124
2019-07-25	1766	15	1781
2019-07-26	2091	16	2107
2019-07-27	1752	6	1758
2019-07-28	1651	5	1656
2019-07-29	1814	1	1816
Total	54839	537	55376

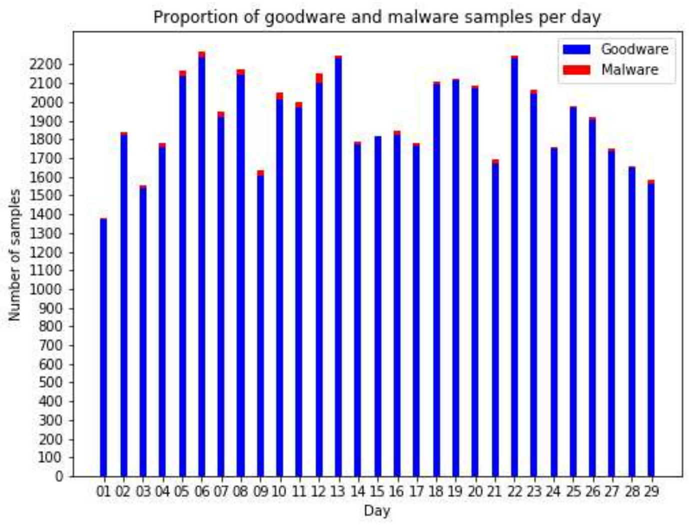


Figure 3.2 – Proportion of goodware and malware samples per day in the dataset

Given that the feature space is reasonably large (162 dimensions), the pair-wise correlation between features was computed and plotted into a heatmap for better visualization in order to investigate the heatmap it can be shortened in later experiments. However, this large feature space means that the heatmap will be too large to be readable; therefore, the features that achieved pair-wise correlations above 0.80 are shown in a smaller heatmap in Appendix A. The full heatmap also revealed permissions that are not requested by any application, as shown in Table 3.3. Table 3.4 shows feature pairs that achieved a correlation equal or higher than 0.85.

Table 3.3 – Unrequested permissions in the dataset

Feature
ACCEPT_HANDOVER
BIND_AUTOFILL_SERVICE
BIND_CALL_REDIRECTION_SERVICE
BIND_CARRIER_MESSAGING_CLIENT_SERVICE
CALL_COMPANION_APP
READ_VOICEMAIL
SMS_FINANCIAL_TRANSACTIONS
START_VIEW_PERMISSION_USAGE
WRITE_VOICEMAIL

Table 3.4 – Feature pairs that achieved a correlation equal or higher than 0.85

Feature pair		Correlation
READ_CALENDAR	WRITE_CALENDAR	0.85
READ_SYNC_SETTINGS	WRITE_SYNC_SETTINGS	0.90
BIND_CARRIER_MESSAGING_SERVICE	BIND_CARRIER_SERVICES	0.91
REQUEST_COMPANION_RUN_IN_BACKGROUND	REQUEST_COMPANION_USE_DATA_IN_BACKGROUND	0.97
BIND_CONDITION_PROVIDER_SERVICE	BIND_TV_INPUT	1
BIND_CONDITION_PROVIDER_SERVICE	BIND_VR_LISTENER_SERVICE	1
BIND_VR_LISTENER_SERVICE	BIND_TV_INPUT	1
SET_ALWAYS_FINISH	BIND_VOICE_INTERACTION	1

In the first experiment, the entirety of the feature space will be used to train the machine learning algorithms to establish a baseline scenario. In the third experiment, the features shown in Table 3.3 and Table 3.4 will be explored further in order to make an educated decision regarding their possible elimination and aggregation and its impact on algorithm performance.

The final step was to visualize the dataset to examine the distribution of its data points and investigate the existence of obvious groups of goodware or malware applications and other interesting findings. Given the high dimensionality of the dataset, two dimensionality reduction techniques were employed to generate 2D and 3D scatter plots of the dataset: Principal Component Analysis (PCA) and t-distributed Stochastic Neighbour Embedding (t-SNE).

PCA is a multivariate analysis technique with the aim of reducing the dimensionality of a dataset by computing new, uncorrelated variables, that are linear combinations of the original variables named principal components (PC's) while retaining as much variance as possible in such a way that the first PC holds the highest possible variance, the second PC the second highest possible variance while also being orthogonal to the first PC etc, depending on how many dimensions the user wants to reduce a given dataset to [54] [55].

Similarly, t-SNE is also a dimensionality reduction technique with a focus on representing highly dimensional datasets accurately in 2D and 3D space. A particularity of this algorithm is a tuneable parameter named perplexity, which can be interpreted as an approximate measure of how many close neighbours each data point has. It also has a learning rate parameter, which can be used to speed up the algorithm. When using this algorithm, it is common to use a perplexity value between 5 and 50 [56] [57]. To produce the best graphical representation possible, several t-SNE projections were created using various learning rate and perplexity value pairs, with the best pair highlighted in bold, as shown in Table 3.5.

Table 3.5 – Distribution of learning rate and perplexity values used to create t-SNE 2D and 3D projections

Learning rate	[**50**, 150, 250, 500]
Perplexity	[10, **20**, 50]

Observing Figures Figure 3.3 and Figure 3.4, 2D and 3D PCA projections do not give information about the dataset's underlying structure besides the existence of seven outliers, two being goodware samples and five being malware samples, illustrated by a black circle. Additionally, there is not a clear separation between malware and goodware samples in these dimensions, and within those classes the data points are concentrated into one large group. There is also a large quantity of lost information when reducing to two and three dimensions, given that the cumulative explained variance of two and three principal components is 15.48 % and 19.42 % respectively.

Observing Figures Figure 3.5 and Figure 3.6, t-SNE projections show that goodware samples are compressed together forming a ball-like shape. Furthermore, and similarly to PCA projections, there does not exist a clear separation between goodware and malware samples in these dimensions. Therefore, conclusions about the structure of these data points cannot be drawn. On the other hand, there exists a small number of malware sample groups.

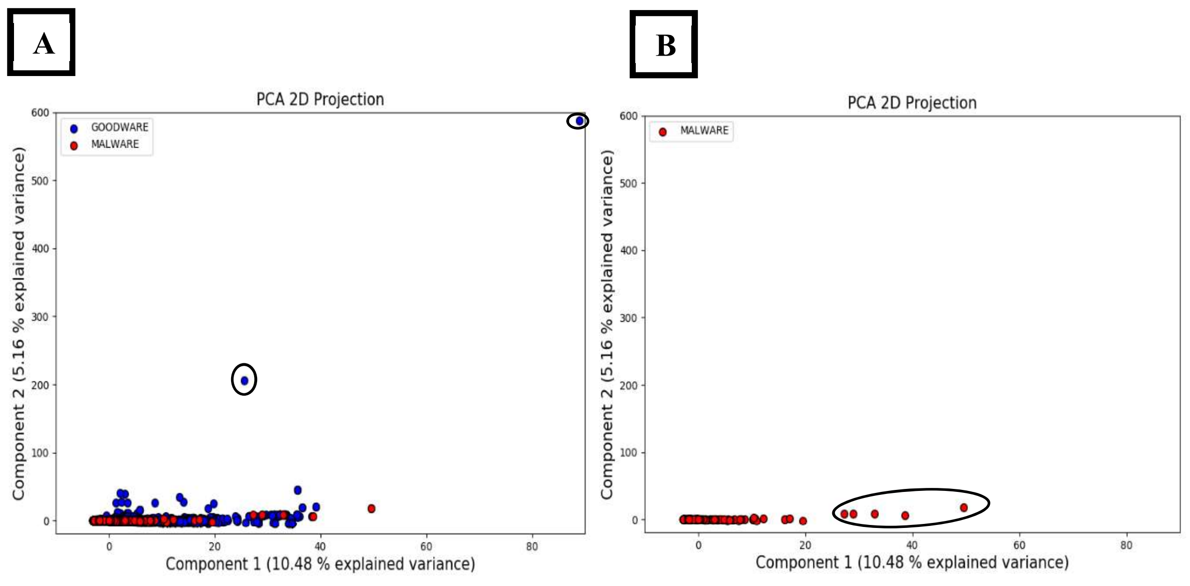


Figure 3.3 – First 2 PCA principal components, (A) goodware and malware, (B) malware. The outliers are circled in black

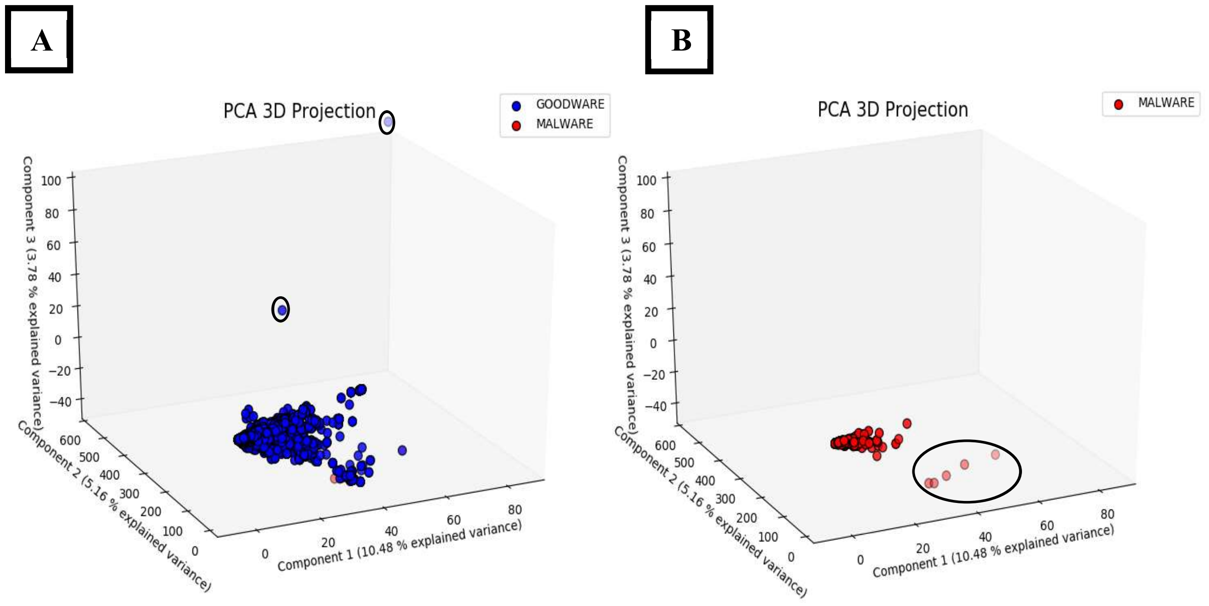


Figure 3.4 – First 3 PCA principal components, (A) goodware and malware, (B) malware. The outliers are circled in black

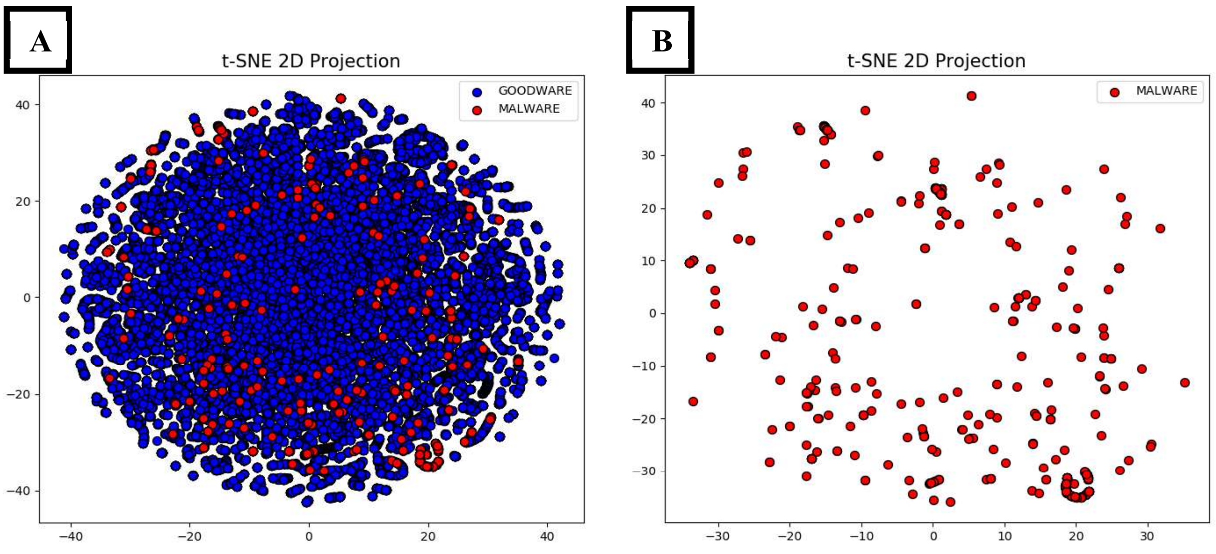


Figure 3.5 – 2-dimensional t-SNE projection of the dataset with a perplexity of 20 and a learning rate of 50, (A) goodware and malware, (B) malware

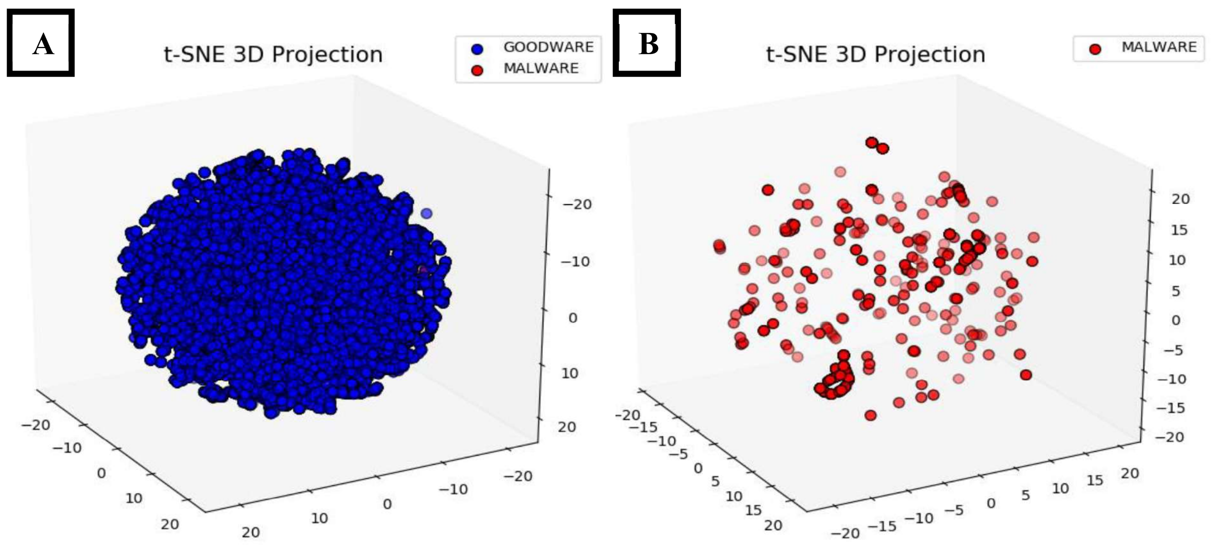


Figure 3.6 – 3-dimensional t-SNE projection of the dataset with a perplexity of 20 and a learning rate of 50, (A) goodware and malware, (B) malware

Across all experiments, the following features were used: Android permissions up to API level 29 (Android version 10.0) [58], application size, number of activities, number of services and the number of receivers.

Regarding Android permissions, it was used the official list developed by Android [58]. Each permission has the value of 1 if a given application requests it, otherwise it is NaN.

The 'Size' feature represents the size of each sample in bytes, with a range of values of $]0, \infty[$. Observing Table 3.6, the mean size of the APK's is approximately 28 MB. Given that the third quartile has a value of 39.51892 MB and the maximum size is 1668.079 MB, these values along with the standard deviation indicates that the dataset might contain outliers. A total of 1521 samples have a size of NaN.

Table 3.6 – General statistics of the feature 'Size' of the dataset

	Size (MB)
Count	53857
Mean	27.96009
Minimum	0.009746
First quartile	6.395092
Second quartile	16.84694
Third quartile	39.51892
Maximum	1668.079367

The ‘Activities’ feature represents the number of activities of each sample, with a range of values of $[0, \infty[$.

The ‘Services’ feature represents the number of services of each sample, with a range of values of $[0, \infty[$.

Lastly, the ‘Receivers’ feature represents the number of receivers of each sample, with a range of values of $[0, \infty[$.

It is important to note that the ‘Activities’, ‘Services’ and ‘Receivers’ features did not present NaN values.

Together, these last four features represent an estimation of an application’s complexity. This notion of complexity is important, because malware applications usually have a small number of these components, in contrast to benign applications such as games or productivity applications. The reason behind the usage of these features in particular is twofold: they are features that are easily extractable given that they are present in the AndroidManifest.xml file, but also because they are aligned with Aptoide’s business model, which demands the system to be lightweight on resource consumption. In total, the feature space is composed of 162 features.

3.2.1. Labels

Each application in the dataset can have one of six labels: *GOODWARE*, *UNKNOWN*, *MDUAL*, *MFAKE*, *MVIRUS* and *WHITELIST*. Although this field is not a feature, there is a need to explain the meaning of each label.

A *GOODWARE* application is, as the name implies, a benign application. An *UNKNOWN* application is a goodware application that has passed Aptoide’s security systems, but it is labelled this way because is it related to Aptoide’s business logic and other internal aspects. A *MDUAL* application is a malware application that was detected by Aptoide’s anti-virus and static rule systems. A *MVIRUS* application is a malware application that was detected by Aptoide’s anti-virus system. An *MFAKE* application was detected as an application that is disguising itself as another, therefore being a ‘fake’. A *WHITELIST* application has some type of detection in Aptoide’s anti-virus system. However, because it comes from its partners and placed in their respective stores, it is part of a whitelist.

3.3. Data Preparation

The data preparation phase encompasses every task that is related with the transformation of the raw initial data into a new dataset in order to become usable by the chosen models. Tasks such as selection or exclusion of specific sections of data, data cleaning, data reformation and data merging belong to this phase [53].

3.3.1. Data cleaning

The first data cleaning step was to replace every NaN value with the value 0. This means that the permissions will be represented by a binary value of 1 if it is requested by a given application. However, if it is not requested it will have a value of 0.

Additionally, comparing the total number of samples in Table 3.1 and Table 3.2, there are two samples that don’t have a label, possibly due to an error during data collection. These samples were removed, given that without a label, they will not prove useful in the classification process.

Furthermore, given that the provided dataset had 1521 samples with the ‘Size’ feature having a NaN value, and it could not be determined if it was a data collection problem, these samples were removed from the dataset.

Lastly, the labels *GOODWARE*, *UNKNOWN*, *MDUAL*, *MFAKE*, *MVIRUS* and *WHITELIST* were merged into *MALWARE* and *GOODWARE*. This process is necessary given that the objective of this thesis is solely to train algorithms that can classify applications as malware or goodware. The labels *MDUAL*, *MFAKE*, *MVIRUS* and *WHITELIST* will be merged into *MALWARE*, because they are samples that were detected by Aptoide’s malware detection systems. The labels *UNKNOWN* and *GOODWARE* will be merged into *GOODWARE* given that *UNKNOWN* applications passed Aptoide’s malware detection systems successfully. After merging, the label field will be a binary value, where *GOODWARE* is represented by the value 0 and *MALWARE* is represented by the value 1.

After the cleaning process, the sample size of the dataset decreased from 55378 samples to 53855 samples as shown in Table 3.7.

Table 3.7 – Dataset sizes before and after data cleaning

Before data cleaning	**After data cleaning**
55378	53855

3.3.2. Undersampling

Given that the number of malware and goodware is so discrepant, it is impossible to achieve the intended goodware to malware ratio of 70/30. The reason for choosing this ratio is to retain the notion that the original dataset has a much higher number of goodware samples than malware samples. To achieve this ratio, it is necessary to under-sample the dataset. The under-sampling algorithm was developed with the constraint that we will extract 1000 samples per day with a goodware to malware ratio of 70/30, meaning that it would be required that each day has 700 goodware samples and 300 malware samples. This algorithm also needs to contemplate four different scenarios:

- **The number of goodware (N_g) is the limiting factor in a given day, meaning that there is not enough goodware samples to make up the 70% ratio, in this case 700 samples:** Although this situation never happens in this dataset, fictitious sample numbers will be used to exemplify this scenario. Because the number of goodware is the limiting factor, it is necessary to compute the total number of samples (malware and goodware) to satisfy the proportion of goodware to malware given the shortage of goodware samples (T_g), as shown in (2). Afterwards, computing the number of malware samples (N_m) is trivial, as shown in (3). Once the number of goodware and malware samples are computed, they are extracted randomly using the method sample() from the Python library Pandas. Equations (4) and (5) shows that after under-sampling, the ratio of goodware to malware ratio is preserved to approximately 70/30 as intended.

$$T_g = \frac{N_g}{Percentage\ of\ goodware} = \frac{600}{0.7} \cong 857 \tag{2}$$

$$N_m = T_g - N_g = 857 - 600 = 257 \tag{3}$$

$$Goodware\ sample\ ratio = \frac{N_g}{T_g} = \frac{600}{857} = 0.7001 \cong 0.7 \times 100 = 70\% \tag{4}$$

$$Malware\ sample\ ratio = \frac{N_m}{T_g} = \frac{257}{857} = 0.2998 \cong 0.3 \times 100 = 30\% \tag{5}$$

- **The number of malware (N_m) is the limiting factor in a given day, meaning that there is not enough malware samples to make up the 30% ratio, in this case 300 samples:** this is the only scenario that is present in the dataset, therefore it will be exemplified using the number of samples of 01/07/2019 shown in Table 3.2. In this scenario, there is a shortage of malware samples to make up the necessary 300, therefore we need to apply a similar principle as the previous scenario. Firstly, the total number of samples in proportion to the shortage of malware samples is computed (T_m), as shown in (6). Afterwards, N_g is computed, as shown in (7). Equations (8) and (9) show that the desired proportion of goodware to malware is preserved successfully.

$$T_m = \frac{N_m}{Percentage\ of\ malware} = \frac{16}{0.3} \cong 53 \tag{6}$$

$$N_g = T_m - N_m = 53 - 16 = 37 \tag{7}$$

$$Goodware\ sample\ ratio = \frac{N_g}{T_m} = \frac{37}{53} = 0.6981 \cong 0.7 \times 100 = 70\% \tag{8}$$

$$Malware\ sample\ ratio = \frac{N_m}{T_m} = \frac{16}{53} = 0.3018 \cong 0.3 \times 100 = 30\% \tag{9}$$

- **There is a shortage of goodware and malware samples in a given day to accommodate the goodware to malware ratio of 70/30 (700 goodware samples and 300 malware samples):** because there is a shortage of both goodware and malware, we need to verify which of those is more limiting. If the number of goodware samples is the most limiting of the two, we use the procedure of the first scenario. Otherwise, we use the procedure of the second scenario. Because this scenario is not present in the dataset and the procedure is identical to the first two scenarios depending on which class has less samples, it will not be exemplified.
- **There are no shortages of goodware and malware samples in a given day:** given that there are no shortages, the number of goodware (700) and malware (300) is extracted using the `sample()` method from the Python library `Pandas`.

Observing Table 3.8, while the dataset is drastically smaller after under-sampling, it is a compromise that was taken in order to have enough data given the low quantity of malware present in the original dataset. In addition, having a dataset with a goodware to malware ratio of 70/30 after under-sampling is significantly more adequate to train the algorithms than the original dataset, given that it had a goodware to malware ratio of approximately 99/1.

Table 3.8 – Total number of samples before and after the under-sampling process

Before under-sampling	After under-sampling
53855	1713

3.4. Modelling

The modelling phase consists of defining which algorithms will be used, to train the models in order to determine their optimal hyper-parameter configurations and the metrics that will be used to assess their performance and validity, depending on the type of problem it is trying to be solved. It is also necessary to define how the final dataset will be split into training and testing sets [53].

This sub-section will provide insight into the modelling setup that was used in every experiment. However, additional modelling steps were taken in the fourth experiment and are detailed in its respective chapter. The algorithms that are going to be trained are: eXtreme Gradient Boosting (XGBoost), K-NN and SVM.

XGBoost is an implementation of gradient boosted trees with a focus on computation speed, efficiency, and scalability [59]. Using decision trees as an example, boosting is an approach where weak decision tree models are created in a sequential manner, with each subsequent tree being fitted using the previous tree's residuals and added to the overall model, improving its performance [60]. It is important to note that the first learner is trained using a weighted version of the original dataset, so that the next iterations can modify the weights on the examples in order to focus on correcting the examples that were misclassified by the majority of the earlier weak learners. This is done by increasing the weights of the incorrect decisions and decreasing the weights of the correct decisions of the weak learner in the current boosting iteration [61]. Gradient boosting expands on the boosting approach by representing the residual as a gradient, where the goal is to add the tree that has the maximum negative gradient, which is the one that will minimize the loss function the most [62]. XGBoost improves regular gradient boosted tree algorithms by improving the regularized objective to further prevent overfitting and making the learning algorithm easier to parallelize. Additionally, rather than using an exact greedy algorithm to find the best split, which is impossible to use when the data is too large to store in memory and when using in a distributed computing context, XGBoost uses an approximate algorithm that not only is modified to be able to handle weighted datasets but can also be used in this context. Lastly, it implements sparsity-aware split finding to be able to handle real-word datasets which in most scenarios has either missing values, zero entries or products of feature engineering, and cache aware access to optimize split finding speed [59].

SVM belongs to the family of discriminant-based models and is heavily inspired by the maximum margin classifier [63]. This algorithm is based on distributing the data in space and computing hyperplanes – subspaces that have one less dimension than the original space, that perfectly separate the data – and selecting the optimal separating hyperplane afterwards [60].

To select the most optimal hyperplane, the margin – the perpendicular distance from the hyperplane to the closest observations on either side – is computed for each of the found separating hyperplanes and the one which has the largest margin is the optimal separating hyperplane [60] [63]. The reason behind this choice is to maximize model generalization, to prevent misclassification due to noise [63]. The margin also contains the support vectors, training observations that are alongside the optimal separating hyperplane and equidistant from it [60]. It is also important to note that only the support vectors carry information, given that they are the only points that if shifted would cause the optimal separating hyperplane to shift accordingly unless it is a point that surpasses the margin hyperplane [60] [63]. Using a two class classification example where the targets values are {-1,1}, in order for SVM to classify a test observation, the optimal separating hyperplane's coefficients are replaced by the observations' feature measurements and depending on the sign of the result, it will appear on one of the sides of the optimal separating hyperplane, and the farthest it is from a given side the more confidence we have that it was correctly classified [60].

K-NN belongs to the family of instance-based learners, storing instances of training data rather than creating an internal model [64]. This algorithm works by assigning a test observation x to the class that has the most observations among its K nearest neighbours [63]. This classifiers' performance changes drastically depending on the chosen value for K, and as its value rises the model becomes stiffer, sacrificing variance for bias given that it uses a larger number of training data points to make predictions [60].

Random search will be used to find the best hyper-parameter configuration for each algorithm as opposed to grid search and manual search, because it is more effective than testing each configuration individually, given that it is a technique where, given a set of hyper-parameters and their value distribution, it chooses a random hyper-parameter value combination rather than testing them one by one as grid search does [65] [66]. For each algorithm, the number of random search iterations is set to 750. It is important to note that given K-NN's lower hyper-parameter space, it is only possible to do 120 iterations of Random Search, instead of the intended 750. Additionally, the dataset will be divided into two sets: 80 % will be used to train and validate the algorithms using 10-fold cross-validation, and 20 % to create the testing set. This division is stratified, meaning that it preserves the goodware and malware ratios [67].

K-fold cross-validation is an evaluation technique where the data is divided into k equal or partially equal parts (folds). Afterwards, k iterations of training and validation are done where k-1 parts are used for training and the remaining part is used for validation. The process stops when every fold has been used for validation [68]. This process is also stratified. It is important to note that 10-fold cross-validation was used across all experiments.

The evaluation metrics that will be used are the AUC of the ROC curve, F1-score, FNR and FPR given that the algorithms are being trained to solve a binary classification problem.

The ROC curve is a graphical representation of the TPR (true positive rate) versus FPR (false positive rate) for different TPR thresholds, and for each threshold a (TPR, FPR) pair is obtained. The best-case scenario is a classifier that has a (1,0) pair. Additionally, given that the TPR is on the y axis and the FPR is on the x axis, the closer a classifier's curve is to the upper-left corner, the better it is [63] [69]. Figure 3.7 shows an example ROC curve.

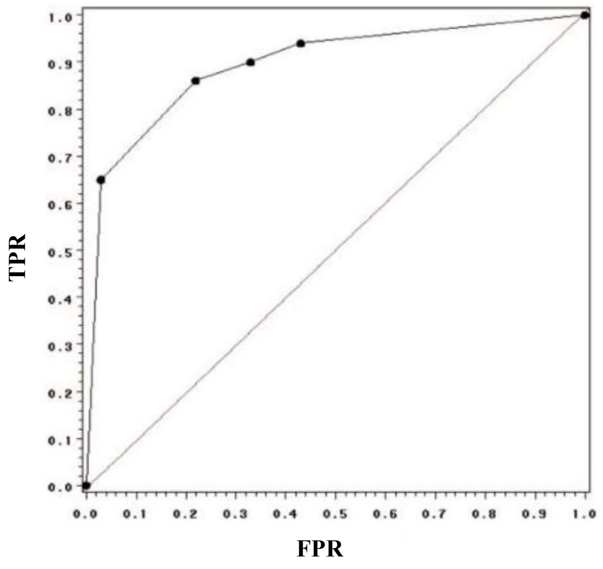


Figure 3.7 – Example ROC curve (adapted from [69]*)*

The AUC of the ROC curve is a numerical value that summarizes the ROC curve over all thresholds. This value has a range of [0,1], where an AUC of 1 corresponds to the best-case scenario mentioned earlier for the ROC curve, and an AUC of 0 represents a classifier that is completely inaccurate. An AUC of 0.5 represents a classifier that cannot distinguish between positive and negative samples [60] [63] [69].

The F1-score (F_1) represents the harmonic mean of the precision (P) and recall (R), as shown in (10). The precision is given by (11) and recall is given by (12), where TP is the number of true positives, FP is the number of false positives and FN is the number of false negatives [70].

$$F_1 = 2 \times \frac{P \times R}{P + R} \tag{10}$$

$$P = \frac{TP}{TP + FP} \tag{11}$$

$$R = \frac{TP}{TP + FN} \tag{12}$$

The FPR is given by (13) and represents the proportion of negative samples that are wrongly classified as positive samples, where FP is the number of false positives and N is the number of negative samples [63].

$$FPR = \frac{FP}{N} \tag{13}$$

The FNR is given by (14) and represents the proportion of positive samples that are wrongly classified as negative samples, where FN is the number of false negatives and P is the number of positive samples.

$$FNR = \frac{FN}{P} \tag{14}$$

Given that the dataset does not have an equal ratio of goodware samples to malware samples (70 % to 30 % respectively), the accuracy is not a good metric to ascertain which algorithm performs the best. This is true for every experiment given that this ratio does not change. The modelling setup is identical across the first, second and third experiments, the only aspects that change are the hyper-parameters chosen by random search. Therefore, the modelling phase of the fourth experiment is detailed in Chapter 4.4.

3.5. Evaluation

The evaluation phase consists of assessing whether the selected models in the previous phase achieved the business objectives that were set in the business understanding phase and reviewing the process thoroughly in order to check for disregarded tasks. Depending on the result of the assessment, a decision about whether to perform a new iteration, create a new project or proceed to the deployment phase should be made [53]

3.6. Deployment

When a project moves forward to the deployment phase, it usually means integrating the models into an organization's decision-making processes – in this case, the permission or prohibition of an Android application in an application marketplace. This can be as straightforward as solely elaborating a final report or as complex as implementing a cyclical data mining/machine learning process pipeline. Regardless of the purpose of the models, it is necessary to elaborate a plan which describes the steps required to deploy the models and how to perform them. Additionally, it should be outlined how the models should be maintained and analysed during their operation. Another output of this phase is a final report, which could be in the form of a written report that integrates the deliverables of the previous phases and summarizes the empirical results, or in the form of a comprehensive presentation in order to exhibit the obtained results. Lastly, the project should be reviewed in order to report which aspects went wrong, which aspects need to be improved in future projects of similar nature and which aspects were employed well [53].

4. Tests and Validation

This chapter will focus on presenting and discussing the results of the four experiments conducted, in addition to the specification of the hyper-parameters of the models in each experiment. Any additional data understanding, data preparation and modelling tasks that are employed in a particular experiment are also detailed in this chapter.

4.1. Baseline Scenario

The objective of the first experiment is to establish a baseline scenario from which additional data preparation and modelling tasks can be performed in further experiments in order to explore their impact on model performance. This experiment will use all the 162 features, and only 3.18 % of all the data (1713 samples with a ratio of 70 % goodware to 30 % malware).

Modelling

Table 4.1, Table 4.2 and Table 4.3 show the value distribution of hyper-parameters for each algorithm. The hyper-parameter combination that yielded the best performance is shown in bold.

Table 4.1 – XGBoost's hyper-parameter value distribution in the first experiment

Hyper-parameter	Values
min-child-weight	**1**, 5, 10
gamma	0.1, 0.3, 0.5, **0.7**, 1, 1.3, 1.5, 1.7, 2, 5
subsample	0.1, 0.3, 0.6, 0.8, **1.0**
colsample_bytree	0.1, 0.3, **0.6**, 0.8, 1.0
max_depth	3, 5, 6, 10, **13**, 15, 17, 20, 25, 30
learning_rate	0.01, **0.05**, 0.1, 0.3, 0.5
n_estimators	100, **250**, 500, 750, 1000
early_stopping_rounds=50	20, 30, 50, **70**

Table 4.2 – SVM's hyper-parameter value distribution in the first experiment

Hyper-parameter	Values
kernel	**rbf**
gamma	[60 values from 0.01 to 2] (**0.01**)
C	[20 values from 0.1 to 2] (**0.5**)

Table 4.3 – K-NN's hyper-parameter value distribution in the first experiment

Hyper-parameter	**Values**
algorithm	**ball_tree**, kd_tree
n_neighbors	1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, **14**, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30
weights	uniform, **distance**

Evaluation

Observing Table 4.4, XGBoost is the highest performing algorithm, given that it outmatches the remaining algorithms in every metric, with a F1-score of 0.89 and an AUC ROC of 0.911. Additionally, it has the lowest FPR and FNR rates of 0.038 and 0.279 respectively, which are very important factors in a malware detection context. The SVM algorithm yielded anomalous results, given that it achieved a FPR of 0 and a FNR of 0.952. This is because the feature "Size" has a different value range than the remaining features coupled with the fact that it is in Bytes and the mean size is approximately 28 MB. This hinders its ability to compute the optimal separating hyperplane and consequently, to classify the samples. This issue will be investigated in the second experiment by removing this feature from the feature space. Lastly, K-NN's performance makes it unfeasible for classification, especially with a FNR of 40.4 %.

Table 4.4 – Algorithm performance of the first experiment

Algorithm	**F1-score**	**AUC ROC**	**FPR**	**FNR**
XGBoost	0.890	0.911	0.038	0.279
SVM	0.610	0.588	0	0.952
K-NN	0.770	0.772	0.155	0.404

Observing Table 4.5, XGBoost and SVM took two hours, 19 minutes, and 54 seconds and two hours, 19 minutes and 23 seconds to train respectively. K-NN took 35 seconds to store the training instances.

Table 4.5 – Time taken to train each algorithm in the first experiment (XGBoost and SVM) and for K-NN to store the training instances

Algorithm	**Time taken**
XGBoost	2 hours, 19 minutes, and 54 seconds
SVM	2 hours, 19 minutes, and 23 seconds
K-NN	35 seconds

Using 10-fold cross validation, the first experiment demonstrates how superior XGBoost is in comparison to the remaining machine learning algorithms, having a 91.1 % change of distinguishing between goodware and malware correctly. However, having a 27.9 % probability of classifying a malware application as goodware is problematic. Additionally, SVM's showed abnormal results and the next experiment will focus on correcting this issue in order to better compare these three algorithms.

4.2. Experiment with the removal of the 'Size' feature

This experiment's objective is to test if the removal of the 'Size' feature from the feature space will improve SVM's results without interfering with other algorithms. This experiment will use 161 features, and only 3.18 % of all the data (1713 samples with a ratio of 70 % goodware to 30 % malware).

Data Preparation

In addition to the data preparation steps of the first experiment, the 'Size' feature was removed from the dataset.

Modelling

Table 4.6, Table 4.7 and Table 4.8, show the hyper-parameters that yielded the best performance for the XGBoost, SVM and K-NN algorithms respectively.

Table 4.6 – XGBoost's best model hyper-parameters in the second experiment

Hyper-parameter	Values
min-child-weight	1
gamma	0.1
subsample	1.0
colsample_bytree	0.6
max_depth	10
learning_rate	0.01
n_estimators	500
early_stopping_rounds=50	50

Table 4.7 – SVM's best model hyper-parameters in the second experiment

Hyper-parameter	Values
kernel	rbf
gamma	0.01
C	1.9

Table 4.8 – K-NN's best model hyper-parameters in the second experiment

Hyper-parameter	Values
algorithm	kd_tree
n_neighbors	30
weights	distance

Evaluation

Observing Table 4.9, removing the 'Size' feature solved SVM's abnormal results in the first experiment and improved K-NN's performance across all metrics. Given that K-NN is a distance-based algorithm and the 'Size' feature has a value range that is extremely different than the remaining features, this performance improvement is expected after its removal from the feature space. However, XGBoost is still the best performing algorithm with a F1-Score of 0.890, a FNR of 0.017, an FPR of 0.298 and an AUC ROC of 0.911, corresponding to a FPR and FNR improvement of 0.021 and 0.019 respectively in comparison to the first experiment.

Observing Table 4.10, XGBoost and SVM took two hours, 33 minutes and 44 seconds, and one hour, 41 minutes and 18 seconds to train respectively. Comparing to the first experiment, SVM took less time to train with the removal of the 'Size' feature, while XGBoost had its training time increased. K-NN took more time to store the instances than the previous experiment, specifically, one minute and 21 seconds.

Table 4.9 – Algorithm performance for the second experiment

Algorithm	F1-score	AUC ROC	FPR	FNR
XGBoost	0.890	0.911	0.017	0.298
SVM	0.850	0.835	0.042	0.385
K-NN	0.860	0.856	0.063	0.298

Table 4.10 – Time taken to train each algorithm in the second experiment (XGBoost and SVM) and for K-NN to store the training samples

Algorithm	Time taken
XGBoost	2 hours, 33 minutes, and 44 seconds
SVM	1 hours, 41 minutes, and 18 seconds
K-NN	1 minute and 21 seconds

Despite correcting SVM's abnormal results and improving K-NN's performance in relation to the first experiment, XGBoost remains the best performing algorithm. However, it also increased XGBoost's FNR by 1.9 %.

4.3. Experiment applying feature elimination and aggregation

This experiment will explore how eliminating and aggregating certain features will affect the algorithm's performance. Feature reduction is also important to evaluate the possibility of reducing training times without sacrificing performance. It is important to note that similarly to the second experiment, the 'Size' feature will not be used in order to maintain normal SVM results. This experiment will use 132 features, and only 3.18 % of all the data (1713 samples with a ratio of 70 % goodware to 30 % malware).

Data Understanding

Feature aggregation and elimination may lead to information loss, and consequently lower the performance of the algorithms. Therefore, these operations need to be considered carefully before they are employed. The data preparation phase will discuss which features were eliminated and aggregated and the reasoning behind such decisions. It is important to note that the only features that underwent this process were the Android permissions, given that they represent the largest portion of the feature space.

Three factors were considered to determine which features (permissions) will be eliminated: its requested percentage, XGBoost's feature importance value, and if Android classifies it as a dangerous permission.

The requested percentage of a given permission P_i represents the percentage of applications which requested it, and is given by dividing the number of samples, N_p, that request that permission (meaning it has a value of 1), by the total number of samples, N_t, as shown in (15). To obtain XGBoost's importance of each feature it is necessary to train a model first, therefore the feature importance values of the best model from the second experiment will be used. This value is computed using the average gain of splits of a given feature and represents its contribution to the increase of the model's accuracy [71] [72]. Lastly, according to Android, "dangerous permissions cover areas where the app wants data or resources that involve the user's private information, or could potentially affect the user's stored data or the operation of other apps" [73].

For a permission to be eliminated, it must have a requested percentage equal or lower than 0.009 % (which translates to a permission being requested by 5 applications or less in this dataset), a feature importance of 0 or NaN and it cannot be a dangerous permission. This requested percentage threshold was chosen to minimize the possibility of removing permissions that could be useful to detect a specific malware/goodware application while optimizing the feature space by removing permissions that might not carry useful information.

Appendix A shows the permissions that are going to be removed, and their respective values in the metrics mentioned above.

$$P_i = \frac{N_p}{N_t} \times 100 \tag{15}$$

Although Table 3.4 contains feature pairs with correlations of 0.85 and higher, pair-wise correlations alone should not be used to decide whether to aggregate those features or not, due to the possibility of losing valuable information. Therefore, the function of both features will be considered, meaning that if a pair of features has both a high correlation value and are similar in their function or allow operations that affect the same resource, that feature pair will be aggregated.

Permissions READ_CALENDAR and WRITE_CALENDAR have a correlation of 0.85 and are related to manipulating the Android calendar by reading and writing calendar data respectively [74] [75]. Similarly, READ_SYNC_SETTINGS and WRITE_SYNC_SETTINGS have a correlation of 0.90 and are related to reading and writing sync settings [76] [77]. REQUEST_COMPANION_RUN_IN_BACKGROUND and REQUEST_COMPANION_USE_DATA_IN_BACKGROUND have a correlation of 0.97 and are related to companion applications, more specifically, executing companion applications in the background and allowing them to use data in the background [78] [79]. Lastly, permissions BIND_CARRIER_MESSAGING_SERVICE and BIND_CARRIER_SERVICES have a correlation of 0.91, and Android advises the usage of the latter instead of the former given that it is deprecated since Android API level 23 [80] [81]. For these reasons, these permission pairs will be aggregated, creating a new feature as shown in Table 4.11. The aggregated feature will be assigned a value of 1 if any of the original features has a value of 1, and a value of 0 if both original features have a value of 0.

Table 4.11 – Feature pairs chosen for aggregation and the corresponding aggregated feature

Feature pair		**Aggregated Feature**
READ_CALENDAR	WRITE_CALENDAR	CALENDAR
READ_SYNC_SETTINGS	WRITE_SYNC_SETTINGS	SYNC_SETTINGS
BIND_CARRIER_MESSAGING_SERVICE	BIND_CARRIER_SERVICES	BIND_CARRIER_SERVICES
REQUEST_COMPANION_RUN_IN_BACKGROUND	REQUEST_COMPANION_USE_DATA_IN_BACKGROUND	REQUEST_COMPANION

Data Preparation

Similarly to the second experiment, the features listed in Appendix A were eliminated by removing their respective columns from the dataset. The feature aggregation process was employed as follows: for every permission pair that was chosen for aggregation, if any of those permissions are present in a given application (meaning that it has a value of 1), the aggregated feature will have a value of 1, otherwise it will have a value of 0. Afterwards, the columns of the feature pairs are removed from the dataset and the column of the aggregated feature is inserted into the dataset.

Modelling

Table 4.12, Table 4.13 and Table 4.14 show the hyper-parameters that yielded the best performance in this experiment for the XGBoost, SVM and K-NN algorithms respectively.

Table 4.12 – XGBoost's best model hyper-parameters in the third experiment

Hyper-parameter	Values
min-child-weight	1
gamma	1.3
subsample	1.0
colsample_bytree	0.8
max_depth	15
learning_rate	0.01
n_estimators	750
early_stopping_rounds=50	70

Table 4.13 – SVM's best model hyper-parameters in the third experiment

Hyper-parameter	Values
kernel	rbf
gamma	0.04372881355932204
C	1.2

Table 4.14 – K-NN's best model hyper-parameters in the third experiment

Hyper-parameter	Values
algorithm	ball_tree
n_neighbors	28
weights	distance

Evaluation

Observing Table 4.15, and comparing to the previous experiment, although eliminating and aggregating features did not impact F1-Scores, it increased the AUC ROC of every algorithm. Additionally, it decreased XGBoost's FNR to 0.260 and increased its FPR to 0.042. SVM's performance improved marginally, achieving an AUC ROC of 0.836, just 0.001 higher than the previous experiment. Likewise, K-NN achieved an AUC ROC score of 0.858, corresponding to a marginal increase of 0.002.

Observing Table 4.16, feature elimination and aggregation lowered XGBoost's and SVM's training times in comparison to the previous experiment, slightly improving training efficiency. More specifically, it reduced XGBoost's training time by 23 minutes and three seconds, and SVM's training time by 12 minutes and 52 seconds. It also reduced K-NN's time to store training instances by seven seconds.

Table 4.15 – Algorithm performance for the third experiment

Algorithm	F1-score	AUC ROC	FPR	FNR
XGBoost	0.890	0.917	0.042	0.260
SVM	0.850	0.836	0.042	0.385
K-NN	0.860	0.858	0.063	0.298

Table 4.16 – Time taken to train each algorithm in the third experiment (XGBoost and SVM) and for K-NN to store the training instances

Algorithm	Time taken
XGBoost	2 hours, 10 minutes, and 41 seconds
SVM	1 hours, 28 minutes, and 26 seconds
K-NN	1 minute and 14 seconds

This experiment's results follow the same trend as the first and second experiments: XGBoost is the best algorithm. Additionally, the combination of highly correlated features and the elimination of features that were deemed unimportant improved XGBoost's false negative rate to its lowest value yet, 26 %.

4.4. Initial exploration with all features and a small dataset

In this experiment composed of two scenarios, six new features were introduced to the dataset: operation codes, resource strings, smali strings, API packages, system commands and intents. Additionally, several feature normalization techniques will be employed to investigate how it will impact the results. Lastly, the ‘Size’ feature will be reintroduced to the feature space given that using feature normalization will compress its large value range, which was the cause of SVM’s abnormal results in the first experiment. In the first scenario, all features will be used and in the second scenario feature aggregation and elimination will be applied. The first scenario will use 168 features, and only 3.18 % of all the data (1713 samples with a ratio of 70 % goodware to 30 % malware). The second scenario will use 139 features, and only 3.18 % of all the data (1713 samples with a ratio of 70 % goodware to 30 % malware).

Data Understanding

The new features were provided by Aptoide and were extracted using a mixture of AndroGuard [37] as the base tool with additional logic by AndroPyTool [82]–[84]. Androguard is a tool that employs Android file manipulation such as disassembling DEX/ODEX bytecodes, decompiling DEX/ODEX files, etc [37]. AndroPyTool is a tool designed to employ static and dynamic feature extraction, integrating various current analysis tools such as DroidBox, FlowDroid, AndroGuard, Strace, VirusTotal and AVClass [82]–[84].

Operation codes represent the number of Dalvik bytecode operation codes from [85] in each application. Dalvik is the discontinued runtime that Android used to execute Android applications, replaced by Android Runtime (ART) [86]. This feature has a possible range of values of $]0, \infty[$.

Resource strings represent the number of string resources in each application. Resource strings are XML text resources that can be optionally stylized and formatted [87]. This feature has a possible range of values of $]0, \infty[$.

Smali strings represent the number of unique smali strings. Smali code originates from the assembler/disassembler tool named baksmali, which transforms Dalvik bytecode into a more readable syntax [48]. This feature has a possible range of values of $]0, \infty[$.

API packages represent the number of unique Java packages that are called in an application. This feature has a possible range of values of $]0, \infty[$.

System commands represent the number of Unix system commands found within the smali code of an application. This feature has a possible range of values of $]0, \infty[$.

Intents are messaging objects that describe an action to be performed in a distinct application component or application altogether [88] [89]. This feature has a possible range of values of values of $]0, \infty[$.

The dataset contained 144 NaN values in the 'Smali strings', 'API packages', 'System commands' and 'Intents' features. Upon further exploration, it was found that these values belong to 144 samples, meaning that these values are not spread randomly. This could be due to an error during the feature extraction process (e.g. due to code obfuscation).

Data Preparation

In this phase, the new features mentioned in the data understanding phase were introduced to the dataset by inserting their respective columns.

Additionally, the 144 samples that had NaN values in the 'Smali strings', 'API packages', 'System commands' and 'Intents' are goodware applications. Therefore, these values were replaced with the value 0. If these samples were a mixture of goodware and malware or solely malware, it could indicate that there could exist malware families that are characterized by having these values in these specific features and it would be necessary to adopt a different strategy.

Modelling

In comparison to the previous experiments, additional steps were introduced to the modelling approach in this experiment. Before training the algorithms, given that the range of values are not uniform across the feature space, four different feature normalization techniques are going to be employed: Z-Score, Min-Max, Quantiles Information and Unit norm. They are going to be trained without using feature normalization techniques as well in order to function as a control group to compare their impact on algorithm performance.

Additionally, each algorithm will be trained five times per feature normalization technique to obtain more confidence about the results, and 10-fold cross-validation will use a random seed in K-NN and a fixed seed in XGBoost and SVM. This way, for a given algorithm, their results will be comparable per feature normalization technique but also between different algorithms and prevent K-NN from outputting the same results each run by randomizing the partitioning of the testing sets.

The hyper-parameters per algorithm per feature normalization technique for each run can be found in Appendixes B-G. These tables are in a backslash separated format, meaning that for each normalization technique, the hyper-parameter values of each model are separated by backslashes, where the first value of a given hyper-parameter corresponds to the first run, the second value to the second run, etc, up to the fifth run (e.g. 1.0/1.1/1.2/1.3/1.4/1.5). If there is only one value in a given table cell, it means that hyper-parameter was equal on all five runs.

Evaluation

Observing Table 4.17, without employing feature elimination and aggregation, XGBoost achieves an overall better performance than both SVM and K-NN regarding F1-score and AUC ROC. Additionally, it achieved the best FPR using every feature normalization technique except Quantiles Information, where SVM achieved a value that is 0.006 lower. Regarding FNR, XGBoost always performs better than SVM, although it never achieves better results than K-NN. This is not alarming, given than XGBoost outperforms K-NN in the rest of the metrics regardless of the feature normalization technique that is used. Taking every metric into account, the version of XGBoost that achieved the best performance was using Quantiles Information normalization with an F1-score of 0.908, an accuracy of 0.912, a FPR of 0.020, a FNR of 0.244, and an AUC ROC of 0.918, outperforming the remaining XGBoost models in every metric except AUC ROC. Similarly to XGBoost's best performing version, SVM achieved its best overall performance using Quantiles Information as well, with a F1-Score of 0.890, an accuracy of 0.896, a FPR of 0.014, a FNR of 0.310, and an AUC of 0.897. This model is an improvement in every metric in relation to the first and second experiments. In addition, there does not exist a clear best performing K-NN model. However, it yielded the lowest results when the features were not normalized as well as when using Min-Max normalization.

Table 4.17 – Algorithm performance per feature normalization technique of the fourth experiment without feature elimination and aggregation

Algorithm	F1-score	AUC ROC	FPR	FNR
Without Normalization				
XGBoost	0.904	0.919	0.025	0.246
SVM	0.588	0.583	0	0.973
K-NN	0.832	0.825	0.105	0.308
With Z-score Normalization				
XGBoost	0.906	0.919	0.023	0.248
SVM	0.890	0.891	0.028	0.283
K-NN	0.870	0.897	0.086	0.240
With Min-Max Normalization				
XGBoost	0.902	0.918	0.023	0.250
SVM	0.870	0.877	0.042	0.327
K-NN	0.852	0.897	0.106	0.244
With Quantiles Information Normalization				
XGBoost	0.908	0.918	0.020	0.244
SVM	0.890	0.897	0.014	0.310
K-NN	0.866	0.909	0.089	0.237
With Unit Norm Normalization				
XGBoost	0.892	0.907	0.024	0.287
SVM	0.870	0.869	0.038	0.329
K-NN	0.870	0.887	0.077	0.248

Observing Table 4.18, feature aggregation and elimination achieved mixed results. For each algorithm, both F1-Scores and AUC ROC values either decreased or increased so marginally, that the deciding factor between which algorithm to choose is a matter of which metrics between FNR, FPR and training time are more important to who is going to implement it. However, it should be noted that feature aggregation and elimination increased AUC ROC values of every K-NN model.

Table 4.18 – Algorithm performance per feature normalization technique of the fourth experiment after feature elimination and aggregation

Algorithm	F1-score	AUC ROC	FPR	FNR
Without Normalization				
XGBoost	0.904	0.918	0.026	0.248
SVM	0.618	0.583	0	0.933
K-NN	0.834	0.826	0.095	0.313
With Z-score Normalization				
XGBoost	0.904	0.918	0.024	0.244
SVM	0.886	0.890	0.027	0.294
K-NN	0.868	0.898	0.089	0.238
With Min-Max Normalization				
XGBoost	0.908	0.917	0.023	0.240
SVM	0.870	0.877	0.040	0.327
K-NN	0.860	0.902	0.098	0.231
With Quantiles Information Normalization				
XGBoost	0.900	0.917	0.021	0.264
SVM	0.890	0.896	0.013	0.310
K-NN	0.872	0.915	0.075	0.244
With Unit Norm Normalization				
XGBoost	0.890	0.911	0.028	0.283
SVM	0.868	0.867	0.039	0.329
K-NN	0.866	0.888	0.083	0.244

Observing Table 4.19, XGBoost's best performing version took the least time to train, namely two hours, 14 minutes, and 56 seconds. Given that SVM gives abnormal results without the use of feature normalization, that model's training time will not be considered. With this in mind, the only model to achieve faster training times than in the previous experiment was when using Unit Norm normalization, taking one hour, 22 minutes and 41 seconds to train. K-NN stored the training samples slower when using feature normalization and faster without normalization in comparison to the previous experiment.

Table 4.19 – Average time taken to train each algorithm (XGBoost and SVM) and for K-NN to store the training instances across five 750 iteration-grid search runs using 10-fold cross-validation per feature normalization technique without feature elimination and aggregation

XGBoost	**SVM**	**K-NN**
Without Normalization		
2 hours, 15 minutes, 15 seconds	2 hours, 9 minutes, and 49 seconds	50 seconds
With Z-score Normalization		
2 hours 29 minutes and 20 seconds	1 hour 39 minutes 11 seconds	4 minutes and 43 seconds
With Min-Max Normalization		
2 hours, 21 minutes, and 25 seconds	1 hour, 45 minutes, and 39 seconds	5 minutes and 14 seconds
With Quantiles Information Normalization		
2 hours, 14 minutes, and 56 seconds	1 hour, 48 minutes, and 25 seconds	4 minutes and 16 seconds
With Unit-Norm Normalization		
2 hours, 36 minutes, and 13 seconds	1 hour, 22 minutes, and 41 seconds	5 minutes and 18 seconds

Observing Table 4.20, employing feature elimination and aggregation reduces training times across every algorithm, just as it occurred from the second experiment to the third. XGBoost took the least time to train when using Z-Score normalization, namely one hour and 57 minutes. SVM achieved its fastest training time of one hour, two minutes and 36 seconds when using Min-Max normalization. Lastly, K-NN achieves its fastest training instance storing time of 51 seconds when feature normalization techniques are not employed, being 1 second slower than without feature elimination and aggregation.

Table 4.20 – Average time taken to train each algorithm (XGBoost and SVM) and for K-NN to store the training instances across five 750 iteration-grid search runs using 10-fold cross-validation per feature normalization technique with feature elimination and aggregation

XGBoost	**SVM**	**K-NN**
Without Normalization		
2 hours, 5 minutes, and 1 second	1 hour, 53 minutes and 33 seconds	51 seconds
With Z-score Normalization		
1 hour and 57 minutes	1 hour, 13 minutes, and 49 seconds	3 minutes and 28 seconds
With Min-Max Normalization		
1 hour, 59 minutes, and 42 seconds	1 hour, 2 minutes, and 36 seconds	3 minutes and 58 seconds
With Quantiles Information Normalization		
2 hours and 37 seconds	1 hour, 16 minutes and 10 seconds	3 minutes and 4 seconds
With Unit-Norm Normalization		
2 hours, 5 minutes, and 19 seconds	1 hour, 4 minutes, and 6 seconds	3 minutes and 35 seconds

In summary, the addition of new features and the usage of feature normalization techniques improved the overall performance of every algorithm in comparison to the third experiment. However, when all metrics are considered, XGBoost proves to be the best performing algorithm when using Quantiles Information normalization.

In addition, aggregating highly correlated features and eliminating non-important features did not produce significant changes in order to conclude whether it improves or lowers the overall performance of the trained algorithms. Therefore, more experiments with an increasing number of eliminated and aggregated features would have to be conducted to determine whether it is beneficial or detrimental to the performance of the algorithms, or if there is an ideal set of features that boost their performance.

5. Conclusions and future work

In this thesis, several machine learning algorithms were trained to measure the effectiveness of using machine learning in Android malware detection with the goal of countering the current state of rampant malware proliferation. To explore this matter, three machine learning algorithms, namely XGBoost, SVM and K-NN were trained using Android permissions, application size, activities, services, receivers, intents, operation codes, resource strings, smali strings, API packages and system commands as features across four experiments using the CRISP-DM methodology.

In the first experiment, using permissions and the number of activities, receivers, and services, XGBoost proved to be the best performing algorithm. Additionally, it was discovered that SVM yielded abnormal results therefore its performance could not be assessed.

The second experiment's objective was to test if the removal of the 'Size' feature would correct the problem that led to SVM's abnormal results. This change solved SVM's issues and improved K-NN's performance; however, XGBoost is still the best performing algorithm across every metric. Regarding training times, XGBoost was slower than the first experiment and SVM was faster. K-NN was slower at storing the training instances.

In the third experiment, several features were eliminated and aggregated to measure the impact of reducing the feature space on algorithm performance as well as the possible gains in training-time of using less features. XGBoost remains the best performing algorithm, although its FPR increased after feature aggregation and elimination. SVM and KNN showed marginal improvements over the second experiment. In comparison to the second experiment, XGBoost and SVM achieved faster training times and K-NN achieved faster training instance storage times.

The fourth and final experiment was characterized by the usage of feature normalization techniques, the reintroduction of a normalized version of the 'Size' feature and the introduction of six new features, which motivated the employment of two tests: a) using feature normalization techniques, and b) using both feature normalization techniques and feature aggregation and elimination. In the first test, XGBoost showed an improvement in F1-score, AUC ROC and FNR in comparison to the first and second experiments; however, it has a 0.003 higher FPR than the model of the second experiment. SVM also improved in every metric in comparison to its first and second model counterparts. On the other hand, none of the K-NN models performs clearly better than the ones in the first and second experiments. The second test achieved mixed results; therefore, it is not clear which algorithm performed the best. However, every K-NN model showed an improvement in their AUC ROC values.

In the first test, all XGBoost models achieved slower training times than the third experiment. SVM models also achieved slower training times except when using Unit Norm normalization. K-NN training instance storage times were slower except when normalization techniques were not used.

In the second test, all XGBoost and SVM model training times improved in comparison to the first test. K-NN training instance storage times improved except when feature normalization was employed.

In summary, the application of machine learning using static analysis-extracted features in the detection of Android malware proved to be promising by providing an automatic method that could be used in the protection of application stores in a real-word scenario. However, the fact that false negative rates never lowered past 23.1 % is a big concern, given that if this method was being applied in an application store, more than one fifth of applications would be classified as goodware when in reality they are malware, which is not acceptable. This might be due to the fact that the dataset has a small sample size of over 1700 samples after under-sampling coupled with a goodware to malware ratio of 70/30. If the dataset was larger, even with this ratio, the number of malware would be higher which would make it easier for the model to distinguish between malware and goodware given that it would have more information, leading to better classification decisions.

The fact that the models were trained using a realistic dataset provided by a reputable Android application store also give these results a higher degree of confidence, even though they achieved slightly lower performances than the best static-analysis-based malware detection methods present in the reviewed literature.

This thesis also originated a paper providing an overview of machine learning-based Android malware detection methods, which was presented in the 7th International Symposium on Digital Forensics and Security (ISDFS) in June 2019 and was published on IEEE Xplore in July 2019 [12].

Possible routes for future work include increasing the number of features to be eliminated and aggregated by increasing the permission requested and correlation thresholds chosen in this thesis (0.009 % and 85 % respectively), training the algorithms with different malware to goodware ratios to investigate its impact on their performance, exploring new features to include in the dataset, using a dataset with a larger sample size in order to increase the size of the resulting dataset after under-sampling if the goodware to malware ratio of the original dataset follows the same trend as the one used in this thesis (99/1), and dividing numerical features into sets of boolean features (e.g. the ‘Size’ feature being divided into discrete boolean features, e.g.: ‘Small’, ‘Medium’, ‘Big’).

6. References

[3] D. Gibert, C. Mateu, and J. Planes, "The rise of machine learning for detection and classification of malware: Research developments, trends and challenges," *Journal of Network and Computer Applications*, vol. 153, no. November 2019, p. 102526, Mar. 2020, doi: 10.1016/j.jnca.2019.102526.

[7] P. Faruki *et al.*, "Android Security: A Survey of Issues, Malware Penetration, and Defenses," *IEEE Communications Surveys & Tutorials*, vol. 17, no. 2, pp. 998–1022, 2015, doi: 10.1109/COMST.2014.2386139.

[8] G. Play and P. Policy, "How Google Play works," *Google*, pp. 1–49, 2019.

[12] J. Lopes, C. Serrao, L. Nunes, A. Almeida, and J. Oliveira, "Overview of machine learning methods for Android malware identification," in *2019 7th International Symposium on Digital Forensics and Security (ISDFS)*, Jun. 2019, pp. 1–6, doi: 10.1109/ISDFS.2019.8757523.

[24] A. K. Arigela and V. Bansal, "Secure and robust framework for monitoring distributed code execution on android," *International Journal of Technology and Engeneering Science*, vol. 3, no. March 2015, pp. 3217–3223, 2016.

[25] L. Singh and M. Hofmann, "Dynamic behavior analysis of android applications for malware detection," in *2017 International Conference on Intelligent Communication and Computational Techniques (ICCT)*, Dec. 2017, no. 2013, pp. 1–7, doi: 10.1109/INTELCCT.2017.8324010.

[26] B. Amro, "Malware Detection Techniques for Mobile Devices," *International Journal of Mobile Network Communications & Telematics*, vol. 7, no. 4/5/6, pp. 01–10, Dec. 2017, doi: 10.5121/ijmnct.2017.7601.

[27] S. Arshad, M. A. Shah, A. Wahid, A. Mehmood, H. Song, and H. Yu, "SAMADroid: A Novel 3-Level Hybrid Malware Detection Model for Android Operating System," *IEEE Access*, vol. 6, pp. 4321–4339, 2018, doi: 10.1109/ACCESS.2018.2792941.

[28] B. Sanz, I. Santos, C. Laorden, X. Ugarte-Pedrero, P. G. Bringas, and G. Álvarez, "PUMA: Permission Usage to Detect Malware in Android," in *Advances in Intelligent Systems and Computing*, vol. 189 AISC, 2013, pp. 289–298.

[30] N. Peiravian and X. Zhu, "Machine Learning for Android Malware Detection Using Permission and API Calls," in *2013 IEEE 25th International Conference on Tools with Artificial Intelligence*, Nov. 2013, pp. 300–305, doi: 10.1109/ICTAI.2013.53.

[32] L. Breiman, "Bagging predictors," *Machine Learning*, vol. 24, no. 2, pp. 123–140, Aug. 1996, doi: 10.1007/BF00058655.

[33] D. Arp, M. Spreitzenbarth, M. Hübner, H. Gascon, and K. Rieck, "Drebin: Effective and Explainable Detection of Android Malware in Your Pocket," in *Proceedings 2014 Network and Distributed System Security Symposium*, 2014, no. August, doi: 10.14722/ndss.2014.23247.

[34] Y. Zhou and X. Jiang, "Dissecting Android Malware: Characterization and Evolution," 2012.

[36] C. Zhao, W. Zheng, L. Gong, M. Zhang, and C. Wang, "Quick and Accurate Android Malware Detection Based on Sensitive APIs," in *2018 IEEE International Conference on Smart Internet of Things (SmartIoT)*, Aug. 2018, pp. 143–148, doi: 10.1109/SmartIoT.2018.00034.

[38] M. Kumaran and W. Li, “Lightweight malware detection based on machine learning algorithms and the android manifest file,” in *2016 IEEE MIT Undergraduate Research Technology Conference (URTC)*, Nov. 2016, vol. 2018-Janua, pp. 1–3, doi: 10.1109/URTC.2016.8284090.

[39] K. Allix, T. F. Bissyandé, Q. Jérome, J. Klein, R. State, and Y. Le Traon, “Empirical assessment of machine learning-based malware detectors for Android,” *Empirical Software Engineering*, vol. 21, no. 1, pp. 183–211, Feb. 2016, doi: 10.1007/s10664-014-9352-6.

[43] T. Bhatia and R. Kaushal, “Malware detection in android based on dynamic analysis,” *2017 International Conference on Cyber Security And Protection Of Digital Services, Cyber Security 2017*, 2017, doi: 10.1109/CyberSecPODS.2017.8074847.

[44] V. M. Afonso, M. F. de Amorim, A. R. A. Grégio, G. B. Junquera, and P. L. de Geus, “Identifying Android malware using dynamically obtained features,” *Journal of Computer Virology and Hacking Techniques*, vol. 11, no. 1, pp. 9–17, 2015, doi: 10.1007/s11416-014-0226-7.

[47] Y. Zhao, G. Xu, and Y. Zhang, *Quality, Reliability, Security and Robustness in Heterogeneous Systems*, vol. 234. Springer International Publishing, 2018.

[49] G. Chandrashekar and F. Sahin, “A survey on feature selection methods,” *Computers and Electrical Engineering*, vol. 40, no. 1, pp. 16–28, 2014, doi: 10.1016/j.compeleceng.2013.11.024.

[51] Y. Liu, Y. Zhang, H. Li, and X. Chen, “A hybrid malware detecting scheme for mobile Android applications,” *2016 IEEE International Conference on Consumer Electronics, ICCE 2016*, pp. 155–156, 2016, doi: 10.1109/ICCE.2016.7430561.

[53] C. Pete *et al.*, “Crisp-Dm 1.0,” *CRISP-DM Consortium*, p. 76, 2000.

[54] H. Abdi and L. J. Williams, “Principal component analysis,” *Wiley Interdisciplinary Reviews: Computational Statistics*, vol. 2, no. 4, pp. 433–459, Jul. 2010, doi: 10.1002/wics.101.

[55] I. T. Jolliffe and J. Cadima, “Principal component analysis: a review and recent developments,” *Philosophical Transactions of the Royal Society A: Mathematical, Physical and Engineering Sciences*, vol. 374, no. 2065, p. 20150202, Apr. 2016, doi: 10.1098/rsta.2015.0202.

[56] M. Wattenberg, F. Viégas, and I. Johnson, “How to Use t-SNE Effectively,” *Distill*, vol. 1, no. 10, Oct. 2016, doi: 10.23915/distill.00002.

[57] L. Van Der Maaten and G. Hinton, “Visualizing data using t-SNE,” *Journal of Machine Learning Research*, 2008.

[59] T. Chen and C. Guestrin, “XGBoost: A Scalable Tree Boosting System,” *Proceedings of the ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, vol. 13-17-Augu, pp. 785–794, Mar. 2016, doi: 10.1145/2939672.2939785.

[60] R. James, G., Witten, D., Hastie, T., Tibshirani, *An Introduction to Statistical Learning - with Applications in R | Gareth James | Springer*. 2013.

[61] M. V. Joshi, R. C. Agarwal, and V. Kumar, “Predicting rare classes,” in *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining - KDD ’02*, 2002, vol. 7, p. 297, doi: 10.1145/775047.775092.

[62] J. H. Friedman, “Greedy Function Approximation: A Gradient Boosting Machine,” *The*

Annals of Statistics, vol. 29, no. 5, pp. 1189–1232, Oct. 2001, doi: 10.1214/aos/1013203451.

[63] E. Alpaydin, *Introduction to Machine Learning*, 2nd ed. MIT Press, 2010.

[65] J. Bergstra and Y. Bengio, "Random search for hyper-parameter optimization," *Journal of Machine Learning Research*, vol. 13, pp. 281–305, 2012.

[69] J. N. Mandrekar, "Receiver Operating Characteristic Curve in Diagnostic Test Assessment," *Journal of Thoracic Oncology*, vol. 5, no. 9, pp. 1315–1316, Sep. 2010, doi: 10.1097/JTO.0b013e3181ec173d.

[70] H. Dalianis, "Evaluation Metrics and Evaluation," in *Clinical Text Mining*, Cham: Springer International Publishing, 2018, pp. 45–53.

[82] A. Martín, R. Lara-Cabrera, and D. Camacho, "A new tool for static and dynamic Android malware analysis," in *Data Science and Knowledge Engineering for Sensing Decision Support*, Sep. 2018, no. September, pp. 509–516, doi: 10.1142/9789813273238_0066.

[83] A. Martín, R. Lara-Cabrera, and D. Camacho, "Android malware detection through hybrid features fusion and ensemble classifiers: The AndroPyTool framework and the OmniDroid dataset," *Information Fusion*, vol. 52, no. December, pp. 128–142, Dec. 2019, doi: 10.1016/j.inffus.2018.12.006.

7. Web References

[1] AV-TEST, "Malware Statistics & Trends Report | AV-TEST." https://www.av-test.org/en/statistics/malware/ (accessed Aug. 13, 2020).

[2] J. Clement, "• Number of internet users worldwide | Statista," 2020. https://www.statista.com/statistics/273018/number-of-internet-users-worldwide/ (accessed Jul. 11, 2020).

[4] "• Forecast number of mobile users worldwide 2019-2023 | Statista." https://www.statista.com/statistics/218984/number-of-global-mobile-users-since-2010/ (accessed Sep. 24, 2019).

[5] "• Smartphone users worldwide 2020 | Statista." https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/ (accessed Feb. 08, 2020).

[6] "Mobile Operating System Market Share Worldwide | StatCounter Global Stats." https://gs.statcounter.com/os-market-share/mobile/worldwide (accessed Aug. 02, 2020).

[9] "Publish an app - Play Console Help." https://support.google.com/googleplay/android-developer/answer/6334282?hl=en (accessed Sep. 02, 2020).

[10] Aptoide, "About us: Taking App Discovery to the Next Level – Aptoide." https://en.aptoide.com/company/about-us (accessed Oct. 25, 2020).

[11] "OWASP Mobile Top 10." https://owasp.org/www-project-mobile-top-10/ (accessed Oct. 04, 2019).

[13] "Platform Architecture | Android Developers." https://developer.android.com/guide/platform (accessed Sep. 05, 2020).

[14] "Application security | Android Open Source Project." https://source.android.com/security/overview/app-security (accessed Sep. 02, 2020).

[15] "Android Platform Glossary | Android Open Source Project." https://source.android.com/setup/start/glossary (accessed Sep. 05, 2020).

[16] "App Manifest Overview | Android Developers." https://developer.android.com/guide/topics/manifest/manifest-intro (accessed Jan. 22, 2019).

[17] "Application Fundamentals | Android Developers." https://developer.android.com/guide/components/fundamentals (accessed Apr. 09, 2020).

[18] Google, "Introduction to Activities | Android Developers." https://developer.android.com/guide/components/activities/intro-activities (accessed Jan. 09, 2020).

[19] Google, "Services overview | Android Developers." https://developer.android.com/guide/components/services (accessed Jan. 06, 2020).

[20] Google, "Broadcasts overview | Android Developers." https://developer.android.com/guide/components/broadcasts (accessed Apr. 09, 2020).

[21] "Application Sandbox | Android Open Source Project." https://source.android.com/security/app-sandbox (accessed Feb. 24, 2020).

[22] Google, "Permissions overview | Android Developers."

https://developer.android.com/guide/topics/permissions/overview (accessed Jan. 09, 2020).

[23] "System and kernel security | Android Open Source Project." https://source.android.com/security/overview/kernel-security.html (accessed Sep. 02, 2020).

[29] "AAPT2 | Android Developers." https://developer.android.com/studio/command-line/aapt2 (accessed Jan. 22, 2019).

[31] "Apktool - A tool for reverse engineering 3rd party, closed, binary Android apps." https://ibotpeaches.github.io/Apktool/ (accessed Jan. 23, 2019).

[35] "VirusTotal." https://www.virustotal.com/gui/home/upload (accessed Jan. 23, 2019).

[37] "GitHub - androguard/androguard: Reverse engineering, Malware and goodware analysis of Android applications ... and more (ninja !)." https://github.com/androguard/androguard (accessed Dec. 12, 2019).

[40]. :: "Phrack Magazine ::." http://www.phrack.org/issues/68/15.html#article (accessed May 24, 2020).

[41] "Weka 3 - Data Mining with Open Source Machine Learning Software in Java." https://www.cs.waikato.ac.nz/ml/weka/ (accessed Apr. 06, 2020).

[42] "UI/Application Exerciser Monkey | Android Developers." https://developer.android.com/studio/test/monkey (accessed Dec. 23, 2018).

[45] "droidbox/APIMonitor at master · pjlantz/droidbox · GitHub." https://github.com/pjlantz/droidbox/tree/master/APIMonitor (accessed Feb. 07, 2019).

[46] "monkeyrunner | Android Developers," 2019. https://developer.android.com/studio/test/monkeyrunner/ (accessed Feb. 07, 2019).

[48] B. Gruver, "GitHub - JesusFreke/smali: smali/baksmali." https://github.com/JesusFreke/smali (accessed Dec. 10, 2019).

[50] "GitHub - pjlantz/droidbox: Dynamic analysis of Android apps." https://github.com/pjlantz/droidbox (accessed Dec. 10, 2019).

[52] "Android Debug Bridge (adb) | Android Developers." https://developer.android.com/studio/command-line/adb (accessed Jan. 12, 2019).

[58] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission (accessed Oct. 01, 2019).

[64] "1.6. Nearest Neighbors — scikit-learn 0.23.2 documentation." https://scikit-learn.org/stable/modules/neighbors.html (accessed Feb. 07, 2020).

[66] "sklearn.model_selection.RandomizedSearchCV — scikit-learn 0.23.2 documentation." https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.RandomizedSearchCV.html (accessed Feb. 08, 2020).

[67] "sklearn.model_selection.StratifiedKFold — scikit-learn 0.23.2 documentation." https://scikit-learn.org/stable/modules/generated/sklearn.model_selection.StratifiedKFold.html (accessed Feb. 08, 2020).

[68] "Cross-Validation | SpringerLink." https://link.springer.com/referenceworkentry/10.1007%2F978-0-387-39940-9_565 (accessed Feb. 08, 2020).

[71] "Understand your dataset with XGBoost — xgboost 1.2.0-SNAPSHOT documentation." https://xgboost.readthedocs.io/en/latest/R-package/discoverYourData.html (accessed Feb. 15, 2020).

[72] "Python API Reference — xgboost 1.2.0-SNAPSHOT documentation." https://xgboost.readthedocs.io/en/latest/python/python_api.html#module-xgboost.plotting (accessed Feb. 15, 2020).

[73] "Permissions overview | Android Developers." https://developer.android.com/guide/topics/permissions/overview#dangerous_permissions (accessed Jun. 20, 2020).

[74] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#READ_CALENDAR (accessed Oct. 09, 2019).

[75] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#WRITE_CALENDAR (accessed Oct. 09, 2019).

[76] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#READ_SYNC_SETTINGS (accessed Oct. 09, 2019).

[77] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#WRITE_SYNC_SETTINGS (accessed Oct. 09, 2019).

[78] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#REQUEST_COMPANION_RUN_IN_BACKGROUND (accessed Oct. 09, 2019).

[79] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#REQUEST_COMPANION_USE_DATA_IN_BACKGROUND (accessed Oct. 09, 2019).

[80] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#BIND_CARRIER_MESSAGING_SERVICE (accessed Oct. 09, 2019).

[81] "Manifest.permission | Android Developers." https://developer.android.com/reference/android/Manifest.permission#BIND_CARRIER_SERVICES (accessed Oct. 09, 2019).

[84] "GitHub - alexMyG/AndroPyTool: A framework for automated extraction of static and dynamic features from Android applications." https://github.com/alexMyG/AndroPyTool (accessed Jul. 01, 2020).

[85] Google, "Opcodes | Android Developers." https://developer.android.com/reference/dalvik/bytecode/Opcodes (accessed Jul. 02, 2020).

[86] "Android Runtime (ART) and Dalvik | Android Open Source Project." https://source.android.com/devices/tech/dalvik (accessed Jul. 02, 2020).

[87] Google, “String resources | Android Developers.” https://developer.android.com/guide/topics/resources/string-resource (accessed Jul. 02, 2020).

[88] Google, “Intent | Android Developers.” https://developer.android.com/reference/android/content/Intent (accessed Jan. 06, 2020).

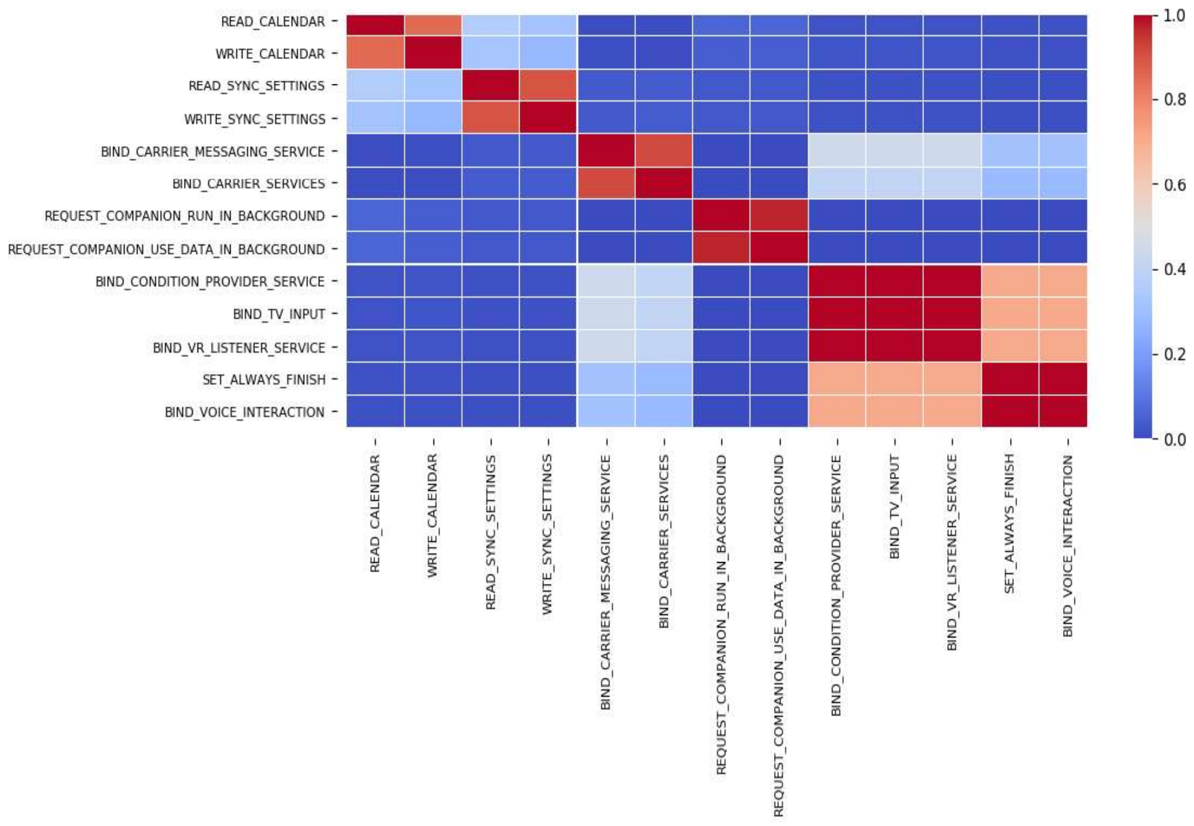
[89] Google, “Intents and Intent Filters | Android Developers.” https://developer.android.com/guide/components/intents-filters (accessed Jan. 06, 2020).

Appendixes

Appendix A

Appendix A shows the smaller version of the heatmap mentioned in Chapter 3.2, which only contains the pair-wise correlation of the permissions that achieved a value higher than 0.80. These values range from 0 to 1, where the bluer the colour of a rectangle is, the lower the correlation between a given pair of permissions is. In the same way, as the correlation increases this colour changes from blue to red and the redder it is the higher the correlation. The diagonal of the heatmap has a correlation of 1 (corresponding to the brightest red possible) given that it is the self-correlation of a given permission.

Heatmap of the permissions that achieved a pair-wise correlation higher than 0.80



Appendix B

Appendix B shows the eliminated features (permissions) in third experiment and second test of the fourth experiment and the respective values of the three conditions that had to be met for a permission to be eliminated: having a XGBoost importance value in the second experiment 0 or NaN, the percentage of samples (applications) that requested such permission and not being classified by Android as a dangerous permission.

Eliminated features (permissions) in the third experiment and second test of the fourth experiment and their respective values of the conditions to be met for elimination

Permission	XGBoost Feature Importance	Requested Percentage (%)	Dangerous Permission
BIND_AUTOFILL_SERVICE	NaN	0	No
BIND_CALL_REDIRECTION_SERVICE	NaN	0	No
BIND_CARRIER_MESSAGING_CLIENT_SERVICE	NaN	0	No
BIND_CHOOSER_TARGET_SERVICE	NaN	0.005	No
BIND_CONDITION_PROVIDER_SERVICE	NaN	0.002	No
BIND_DREAM_SERVICE	NaN	0.004	No
BIND_MIDI_DEVICE_SERVICE	NaN	0.004	No
BIND_PRINT_SERVICE	NaN	0.007	No
BIND_SCREENING_SERVICE	NaN	0.009	No
BIND_TELECOM_CONNECTION_SERVICE	NaN	0.007	No
BIND_TEXT_SERVICE	NaN	0.002	No
BIND_TV_INPUT	NaN	0.002	No
BIND_VISUAL_VOICEMAIL_SERVICE	NaN	0.005	No
BIND_VOICE_INTERACTION	NaN	0.004	No
BIND_VPN_SERVICE	NaN	0.002	No
BIND_VR_LISTENER_SERVICE	NaN	0.002	No
CALL_COMPANION_APP	NaN	0	No
FACTORY_TEST	NaN	0.004	No
READ_VOICEMAIL	NaN	0	No
REQUEST_PASSWORD_COMPLEXITY	NaN	0.002	No
SET_ALWAYS_FINISH	NaN	0.004	No
SMS_FINANCIAL_TRANSACTIONS	NaN	0	No

START_VIEW_PERMISSION_ USAGE	NaN	0	No
WRITE_GSERVICES	NaN	0.005	No
WRITE_VOICEMAIL	NaN	0	No

Appendix C

Given that five models were trained per normalization technique in the fourth experiment, Appendixes C-H are in a backslash separated format, meaning that for each normalization technique, the hyper-parameter values of each model are separated by backslashes, where the first value of a given hyper-parameter corresponds to the first run, the second value to the second run, etc, up to the fifth run (e.g. 1.0/1.1/1.2/1.3/1.4/1.5). If there is only one value in a given table cell, it means that hyper-parameter was equal on all models.

XGBoost hyper-parameter values of the first test of the fourth experiment for each feature normalization technique

Normalization / **Hyper-parameter**	Without Normalization	Z-score	Min-Max	Quantiles Information	Unit Norm
subsample	1.0	1.0	1.0	1.0	0.8/1.0 /1.0/0.8 /0.8
n_estimators	1000/750 /1000/250 /100	500/250 /500/750 /1000	1000/500 /500/250 /500	1000/1000 /100/100 /500	500/100 /500/250 /500
min_child_weight	1	1	1	1	1
max_depth	13/25/15/30 /20	17/20/17 /10/15	10/15/10 /25/17	15/30/30 /15/13	25/30/25 /15/17
learning_rate	0.05/0.01 /0.01 /0.05/0.05	0.01/0.1 /0.1/0.01 /0.1	0.05/0.01 /0.01/0.1 /0.01	0.01/0.1 /0.05/0.05 /0.01	0.01/0.05 /0.1/0.01 /0.01
gamma	1.7/0.3/1.3 /0.3	0.3/1/1/1 /1.7	1/0.7/0.1 /0.7/0.5	0.7/0.5/0.3 /1.5/1	1.7/2/1/0.1 /1.3
early_stopping_rounds=50	70/70/20/70 /20	20/20/70 /50/50	70/30/70 /20/50	50/70/20/70 /30	50/50/20 /30/70
colsample_bytree	0.6/0.3/0.6 /0.3/0.8	0.3/0.6 /0.6/0.3 /0.6	0.6/0.3 /0.3/0.3 /0.3	0.3/0.3/0.3 /0.6/0.3	1.0/0.6/1.0 /0.8/0.6

Appendix D

SVM hyper-parameter values of the first test of the fourth experiment for each feature normalization technique

Normalization / **Hyper-parameter**	Without Normalization	Z-score	Min-Max	Quantiles Information	Unit Norm
kernel	rbf	rbf	rbf	rbf	rbf
gamma	0.01	0.145 /0.145 /0.179 /0.145 /0.145	0.179	0.111/0.077 /0.111/0.077 /0.077	0.280/0.212 /0.246/0.280 /0.212
C	1.3/0.4/0.1/0.2 /0.2	2.0/2.0 /1.9/2.0 /2.0	2.0/1.8 /2.0/1.8 /2.0	2.0/2.0/1.8 /2.0/2.0	1.9/1.8/1.4 /1.9/2.0

Appendix E

K-NN hyper-parameter values of the first test of the fourth experiment for each feature normalization technique

Normalization / **Hyper-parameter**	Without Normalization	Z-score	Min-Max	Quantiles Information	Unit Norm
weights	distance	distance	distance	distance	distance
n_neighbors	9/6/10/6/7	7/8/8/7/7	6/5/5/7/11	6/10/5/7/7	8/8/9/6/6
algorithm	ball_tree/ ball_tree/ kd_tree/ ball_tree/ ball_tree/	ball_tree	ball_tree	ball_tree	ball_tree

Appendix F

XGBoost hyper-parameter values of the second test of the fourth experiment for each feature normalization technique

Normalization / **Hyper-parameter**	Without Normalization	Z-score	Min-Max	Quantiles Information	Unit Norm
subsample	1.0	1.0	1.0/1.0 /0.8/1.0 /1.0	1.0	0.8/0.8 /0.8/0.8 /1.0
n_estimators	750/250 /500/1000 /100	1000/500 /1000/500 /500	750/500 /750/500 /1000	500/1000 /750/250 /250	100/500 /1000/750 /500
min_child_weight	1	1	1	1	1
max_depth	25/17/15 /13/17	15/17/20 /20/10	15/30/15 /15/30	15/30/20 /25/25	13/17/20 /17/13
learning_rate	0.01/0.1 /0.1/0.01 /0.05	0.1/0.01 /0.01/0.05 /0.01	0.01/0.01 /0.01/0.01 /0.3	0.01/0.1 /0.01/0.05 /0.01	0.05/0.05 /0.01/0.01 /0.1
gamma	0.1/1.7/1.3 /0.7/1.3	1/0.7/1/1/1	1/1/0.1/1 /2	0.5/1.5/0.3 /1/1.3	0.3/1/1/0.7 /2
early_stopping_rounds=50	30/20/30 /50/20	50/20/20 /30/30	50/30/20 /50/30	50/20/30 /70/50	20/20/30 /70/50
colsample_bytree	0.6/0.6/0.6 /0.3/0.3	0.8/0.3 /0.3/0.3 /0.3	0.3/0.3 /0.6/0.3 /0.8	0.6/0.6 /0.3/0.6 /0.3	1.0/1.0/1.0 /1.0/0.8

Appendix G

SVM hyper-parameter values of the second test of the fourth experiment for each feature normalization technique

Normalization / **Hyper-parameter**	Without Normalization	Z-score	Min-Max	Quantiles Information	Unit Norm
kernel	rbf	rbf	rbf	rbf	rbf
gamma	0.01	0.179 /0.145 /0.179 /0.179 /0.179	0.179 /0.179 /0.212 /0.179 /0.179	0.111/0.111 /0.111/0.111 /0.077	0.246 /0.314 /0.314 /0.314 /0.314
C	1.3/0.99/0.6 /0.6/0.3	2.0/2.0 /1.8/2.0 /1.8	2.0/2.0/2.0 /2.0/1.9	2.0/2.0/2.0 /1.8/2.0	2.0

Appendix H

K-NN hyper-parameter values of the second test of the fourth experiment for each feature normalization technique

Normalization / **Hyper-parameter**	Without Normalization	Z-score	Min-Max	Quantiles Information	Unit Norm
weights	distance	distance	distance	distance	distance
n_neighbors	12/11/7/7/10	8/6/7/7/8	9/10/8/7/9	12/7/6/6/11	9/8/7/7/11
algorithm	ball_tree /kd_tree /ball_tree /ball_tree /kd_tree	ball_tree	ball_tree	ball_tree	ball_tree