Dataset Generation and Machine Learning Approaches for Android Malware Detection

by

Zakeya NAMRUD

MANUSCRIPT-BASED THESIS PRESENTED TO ÉCOLE DE TECHNOLOGIE SUPÉRIEURE IN PARTIAL FULFILLMENT FOR THE DEGREE OF DOCTOR OF PHILOSOPHY
Ph.D.

MONTREAL, APRIL 30, 2022

ÉCOLE DE TECHNOLOGIE SUPÉRIEURE
UNIVERSITÉ DU QUÉBEC

Zakeya NAMRUD, 2022

This Creative Commons license allows readers to download this work and share it with others as long as the author is credited. The content of this work cannot be modified in any way or used commercially.

BOARD OF EXAMINERS

THIS THESIS HAS BEEN EVALUATED

BY THE FOLLOWING BOARD OF EXAMINERS

Mr. Sègla Kpodjedo, Thesis Supervisor
Department of Software and IT Engineering, École de technologie supérieure

Mr. Chamseddine Talhi, Co-supervisor
Department of Software and IT Engineering, École de technologie supérieure

Mr. Georges Kaddoum, President of the Board of Examiners
Department of Electrical Engineering, École de technologie supérieure

Mrs. Latifa Guerrouj, Member of the jury
Department of Software and IT Engineering, École de technologie supérieure

Mr. Hamid Mcheick, External Independent Examiner
Software engineering and distributed applications, Université du Québec à Chicoutimi

THIS THESIS WAS PRESENTED AND DEFENDED

IN THE PRESENCE OF A BOARD OF EXAMINERS AND THE PUBLIC

ON "APRIL 01, 2022"

AT ÉCOLE DE TECHNOLOGIE SUPÉRIEURE

FOREWORD

The dissertation is organized into five chapters, three of which being articles published or submitted. No changes were applied to these journal submissions or publications. In the first chapter, I discuss my research objectives, contributions, and the methodologies that we used in this research. Next, I did a comprehensive literature review, and background discussion. Chapters 3,4 and 5 are the published journal papers. The thesis is concluded with a discussion of the results and future research directions.

ACKNOWLEDGEMENTS

Many people helped make this thesis feasible. I would want to thank everyone who helped make this Ph.D thesis feasible. I am especially grateful to my family without whom it would not be possible to complete the work.

Génération de jeux de données et approches basées sur l'apprentissage automatique pour la détection des logiciels malveillants sur Android

Zakeya NAMRUD

RÉSUMÉ

Les logiciels malveillants sont de plus en plus complexes et nombreux sur Android. Face à cette complexité et présence grandissantes, les méthodes d'apprentissage automatique sont de plus en plus utilisées pour aider les systèmes Android à détecter ces logiciels. Cet apprentissage machine se construit sur des modèles de comportements dynamiques et/ou de caractéristiques statiques des apps Android. La présente thèse s'articule autour de l'analyse statique d'apps Android pour l'extraction de métriques pertinentes pour la détection de logiciels malveillants par apprentissage machine.

L'accès à des bancs d'essais de qualité peut constituer un frein à la proposition d'approches d'apprentissage machine efficaces. En premier abord, le travail présenté dans ce document a donc consisté en la proposition de scripts qui appliquent diverses analyses statiques sur une app et en extraient une suite de métriques inspirées de divers travaux de la littérature. De plus, nous proposons sur cette base un banc d'essai de plus de 17000 apps pour l'évaluation d'approches d'apprentissage machine.

Le présent travail comprend également des expériences d'apprentissage automatique menées en utilisant des stratégies de classification qui définissent les caractéristiques statiques légitimes des applications bénignes en opposition à celles des applications malveillantes. En général, les applications bénignes partagent des caractéristiques similaires, tandis que les applications malveillantes présentent des caractéristiques anormales. En nous appuyant sur les jeux de données développés, nous proposons et testons les performances de divers modèles de classification dans la détection des applications malveillantes.

Les modèles testés incluent des classificateurs courants, ainsi que des modèles plus avancés de Support Vector Machine et Deep Learning, dont les hyperparamètres ont été réglés pour améliorer la précision et l'efficacité de la détection des logiciels malveillants. Enfin, nous avons examiné sur la base des permissions Android, les divergences possibles de patterns d'utilisation entre les applications bénignes et les logiciels malveillants et ce, selon diverses categories d'apps.

Mots-clés: sécurité mobile, analyse statique, ingénierie inverse, informatique mobile, apprentissage machine, apprentissage profond

Dataset Generation and Machine Learning Approaches for Android Malware Detection

Zakeya NAMRUD

ABSTRACT

In recent years, Android malware has substantially increased both in incidence and developmental complexity. To address this, machine learning approaches are increasingly used to help Android systems detect such software. Such approaches are built on models and metrics encapsulating dynamic behaviors and/or static characteristics of Android apps. This thesis focuses on the static analysis of Android apps for the extraction of relevant metrics for malware detection by machine learning.

Quality benchmarks are essential to proposing effective machine learning approaches. Therefore, the work presented in this document first proposes scripts able to apply diverse static analyses on an app and extract a set of metrics inspired by various works in the literature. In addition, we propose on this basis a dataset of more than 17,000 apps for the evaluation of machine learning approaches for Android malware detection.

This thesis also includes machine learning experiments using classification strategies that define the legitimate static characteristics of benign versus malicious applications. Put trivially, benign applications will share similar characteristics, while malicious applications will exhibit anomalous characteristics that ought to be identified. Based on the developed datasets, we propose and test the performance of various classification models in detecting malicious applications.

The tested models include common classifiers, as well as more advanced Support Vector Machine and Deep Learning models, whose hyperparameters have been tuned to improve the accuracy and efficiency of malware detection. Finally, we examined on the basis of Android permissions and security risks, the possible discrepancies between permission usage patterns of benign applications versus malware across different app categories.

Keywords: mobile security, static analysis, reverse engineering, mobile computing, machine learning, deep learning

TABLE OF CONTENTS

Page

INTRODUCTION 1
1.1 Overview 1
1.2 Statement of Problem and Research Motivation 4
1.3 Requirements of a Potential approach 5
1.4 Scope of Proposed Research 6
1.5 Technical Contributions 6
1.6 Thesis Outline 8
1.7 Summary 8

CHAPTER 2 BACKGROUND AND LITERATURE REVIEW 13
2.1 Background 13
2.1.1 Android app development 13
2.1.2 Distribution and security of Android apps 16
2.1.3 Vulnerabilities in Android Apps 16
2.1.3.1 Android permissions 16
2.1.3.2 Security Smells 18
2.1.3.3 Physical attacks 18
2.1.4 Malware threats 18
2.1.4.1 Malware classification 19
2.2 Literature review 24
2.2.1 Android application datasets 24
2.2.2 Using Machine learning 26
2.2.2.1 Static Analysis 26
2.2.2.2 Dynamic Analysis 29

CHAPTER 3 PROBING ANDROVUL DATASET FOR STUDIES ON ANDROID MALWARE CLASSIFICATION 33
3.1 Abstract 33
3.2 Introduction 34
3.3 Background 36
3.3.1 Vulnerability 37
3.3.2 Dangerous Permissions 37
3.3.3 AndroBugs 37
3.3.4 Security Code Smells 38
3.4 Related Work 38
3.4.1 Dataset 38
3.4.2 Malware Classification with machine learning 40
3.5 AndroVul-T: the tool 42
3.5.1 Dangerous permissions extraction 43
3.5.2 AndroBugs extraction 43

3.5.3 Code Smell extraction ... 44
3.6 AndroVul-D: the dataset ... 44
3.6.1 Data Selection ... 45
3.6.2 Dataset Structure ... 46
3.6.3 Dataset description ... 47
3.7 Study Design ... 48
3.7.1 Identifying malwares ... 49
3.7.2 Correlation Analysis on the Androzoo data ... 50
3.7.3 Used Classifiers ... 50
3.7.4 Feature selection ... 51
3.7.5 Performance Indicators ... 52
3.7.6 Research Questions ... 52
3.7.7 Answering the research Questions ... 53
3.8 Experiments and Results ... 55
3.8.1 Results ... 55
3.8.2 Analysis and discussion of the experiments ... 58
3.9 Limitations and Threats to validity ... 59
3.10 Conclusion ... 61

CHAPTER 4 DEEP LEARNING BASED ANDROID ANOMALY DETECTION USING A COMBINATION OF VULNERABILITIES DATASET ... 65
4.1 Abstract ... 65
4.2 Introduction ... 66
4.3 Background ... 68
4.3.1 Android Vulnerabilities ... 68
4.3.1.1 Dangerous Permissions ... 69
4.3.1.2 AndroBugs Vulnerabilities ... 69
4.3.1.3 Code Smell ... 70
4.3.2 Machine Learning (ML) ... 70
4.3.2.1 Deep Learning (DL) ... 71
4.3.2.2 Support Vector Machines (SVM) ... 72
4.4 Methodology ... 72
4.4.1 Dataset ... 73
4.4.2 Feature extraction ... 73
4.4.3 General architecture of our machine learning approach ... 75
4.4.4 Android Malware Detection based on Deep Learning ... 76
4.4.5 Android Malware Detection based on support vector machine ... 81
4.5 Experiments ... 82
4.5.1 Performance Indicators ... 82
4.5.2 Experimental setup ... 83
4.5.3 Results ... 83
4.5.4 Comparison with well-known anti-virus tool ... 87

4.6 Related work 87
4.6.1 Static Analysis 88
4.6.2 Dynamic Analysis 89
4.6.3 Hybrid Analysis 90
4.7 Conclusion 92

CHAPTER 5 DEEP-LAYER CLUSTERING TO IDENTIFY PERMISSION USAGE PATTERNS OF ANDROID APP CATEGORIES 95
5.1 Abstract 95
5.2 Introduction 96
5.3 Background 98
5.3.1 Permission System 98
5.3.2 Clustering model 100
5.4 Study Objectives and data collection 103
5.4.1 Data collection 103
5.4.1.1 Feature Extraction 104
5.4.1.2 Applications Categories 107
5.5 Proposed Approach 107
5.5.1 Approach overview 107
5.5.2 Deep-layer clustering 109
5.5.3 Clusters analysis 112
5.6 Empirical study 112
5.6.1 Analysis of Cohesion 113
5.6.1.1 Analytical technique 113
5.6.1.2 Results for RQ1 115
5.6.2 Produced inferred pattern 117
5.6.2.1 Analytical technique 118
5.6.2.2 Results for RQ2 119
5.6.3 Pattern generalization evaluation 119
5.6.3.1 Analytical technique 120
5.6.3.2 Results for RQ3 123
5.7 RELATED WORK 124
5.7.1 Research related to dataset generation 124
5.7.2 Permissions based study 126
5.7.3 Category based study 128
5.8 Conclusion 131

CONCLUSION AND RECOMMENDATIONS 133
6.1 General conclusion 133
6.2 Articles in peer-reviewed journals and conferences 134

BIBLIOGRAPHY 135

LIST OF TABLES

Page

Table 2.1 The most widely used Android malware datasets 25

Table 2.2 Comparison between various state-of-art solutions 32

Table 3.1 App categories in our dataset 49

Table 3.2 Selected one Classifies form each four known Machine learning categories 51

Table 3.3 Information about the datasets' sizes and the selected thresholds for all experiments 55

Table 3.4 AUC and F1 results when considering only AndroZoo apps (Benign & Malicious) 55

Table 3.5 AUC and F1 results when considering Benign apps from (AndroZoo & Malicious apps from VirusShare) 56

Table 3.6 AUC and F1 results when considering Benign apps from AndroZoo & Malicious apps from VirusShare and Androzoo 56

Table 3.7 Comparison with related work 58

Table 3.8 Severity Level Artifacts 62

Table 3.9 Dangerous Permissions 62

Table 3.10 Dangerous Permissions 63

Table 3.11 Regular expressions used in our tool containing Smali type for code smell 64

Table 4.1 Dataset description 74

Table 4.2 Best hyper-parameters 81

Table 4.3 Comparison between the results of datasets with 11,814 samples, and 18,780 samples 84

Table 4.4 DL Confusion matrix 85

Table 4.5 The experimentation results for SVM parameters 86

Table 4.6 SVM Confusion matrix 86

Table 4.7 Comparison between DL,SVM classifiers and the Related work 87

Table 4.8 Comparison with well known anti-virus tool..... 88

Table 4.9 Comparison between state of the art research and our approach 92

Table 5.1 Level of permission protection 100

Table 5.2 Dangerous permissions and their related groups in Android 6.0 101

Table 5.3 New or changed permission groups 102

Table 5.4 The dataset contents 106

Table 5.5 The distribution of Benign & Malware App Categories in the dataset 108

Table 5.6 SOM-Kmeans average cohesiveness and summary of inferred usage patterns 117

Table 5.7 Comparison between two models (with and without MP) 124

Table 5.8 Comparison between various state-of-art solutions 129

LIST OF FIGURES

Page

Figure 1.1 Downloads of mobile applications globally from 2018 to 2024 2

Figure 1.2 First contribution 10

Figure 1.3 Second contribution 11

Figure 1.4 Third contribution 12

Figure 2.1 Android application build process 14

Figure 2.2 Mobile malwares classification 20

Figure 2.3 Global mobile malware distribution in 2019 23

Figure 2.4 2019's top Android malware distribution 23

Figure 3.1 Overview of the AndroVul tool 42

Figure 3.2 Parsing and Quantifying vulnerability data 43

Figure 3.3 Data Selection and Gathering 45

Figure 3.4 Information and vulnerabilities extracted from an Android app 47

Figure 3.5 Findings from RQ 1 59

Figure 3.6 Findings from RQ 2 59

Figure 4.1 Installation of mobile malicious packages in Android from 2017 to 2020 67

Figure 4.2 General architecture of a Deep Learning model. 71

Figure 4.3 Dataset Visualization 74

Figure 4.4 Design Methodology for malware detection in Android 77

Figure 4.5 The architecture of DL layers using Sequential neural network 80

Figure 4.6 Built DL model 81

Figure 4.7 Comparison between history models Accuracy for 11,814 samples and 18,526 samples 85

Figure 4.8 Comparison between history models loss for 11,814 samples and 18,526 samples 85

Figure 5.1 Overview of Building the dataset 105

Figure 5.2 Overview of the procedure of producing inferred pattern 109

Figure 5.3 Architecture of the SOM-Kmeans model 113

Figure 5.4 The correlation between the quality matrices 117

Figure 5.5 Overview of the quality matrices cohesion 117

Figure 5.6 PUC & Category cohesion results of the identified permissions usage patterns 119

LIST OF ALGORITHMS

Page

Algorithm 4.1 VirusShare Android apps collection. 74

Algorithm 4.2 Feature Extraction Algorithm. 76

Algorithm 4.3 Deep Learning based Model.. 80

Algorithm 4.4 Support Vector Machine based Model................................ 82

Algorithm 5.1 Feature Extraction 106

Algorithm 5.2 Clusters Analysis 112

Algorithm 5.3 A potential Malware (PM) 121

INTRODUCTION

1.1 Overview

In modern societies, smartphones are an important part of many people's everyday lives. International Data Corporation (IDC) estimates that, by 2024, the smartphone market will top 1.5 billion units, with most of the increase coming from new users in developing countries (IDC, 2021). Android phones are currently the most commonly available mobile devices, with more than three-quarters of smartphone users on an Android device (Li *et al.*, 2018a).

Google Play Store, which is the main online marketplace for Android applications (apps), lists more than 3.43 million unique apps as of January 2021 (matters, 2021). Even from these few statistics, it is clear that Android smartphones are a multi-billion-dollar industry, and growing.

Along with the increase in unit numbers of smartphones, the devices have evolved from being mobile phones to being miniature mobile computers. This change in use has effectively expanded the parameters of smartphones for both business and personal use. Available apps and those in development are geared to satisfying a broad range of everyday needs, such as social networking, online banking and gaming, along with more traditional tasks like phoning and texting. At the same time, the apps also need to be able to handle user information that may be sensitive.

Mobile ecosystems are often targeted by cybercriminals based on the type of information these systems handle. The first line of attack is usually to exploit vulnerabilities in apps. Some cybercriminals even develop apps themselves as a means to access user information. However, the majority of the attacks target the most popular apps, as gaining access to the information of millions of smartphone users can prove lucrative to black market

sellers. Additionally, non-criminal actors may target user information in order to deliver advertisements. Whether for criminal purposes or not, these ongoing attempts to illegally access user information highlight the need for protecting smartphone users' privacy and security. The best way to approach that is through the detection and mitigation of the technology's vulnerabilities

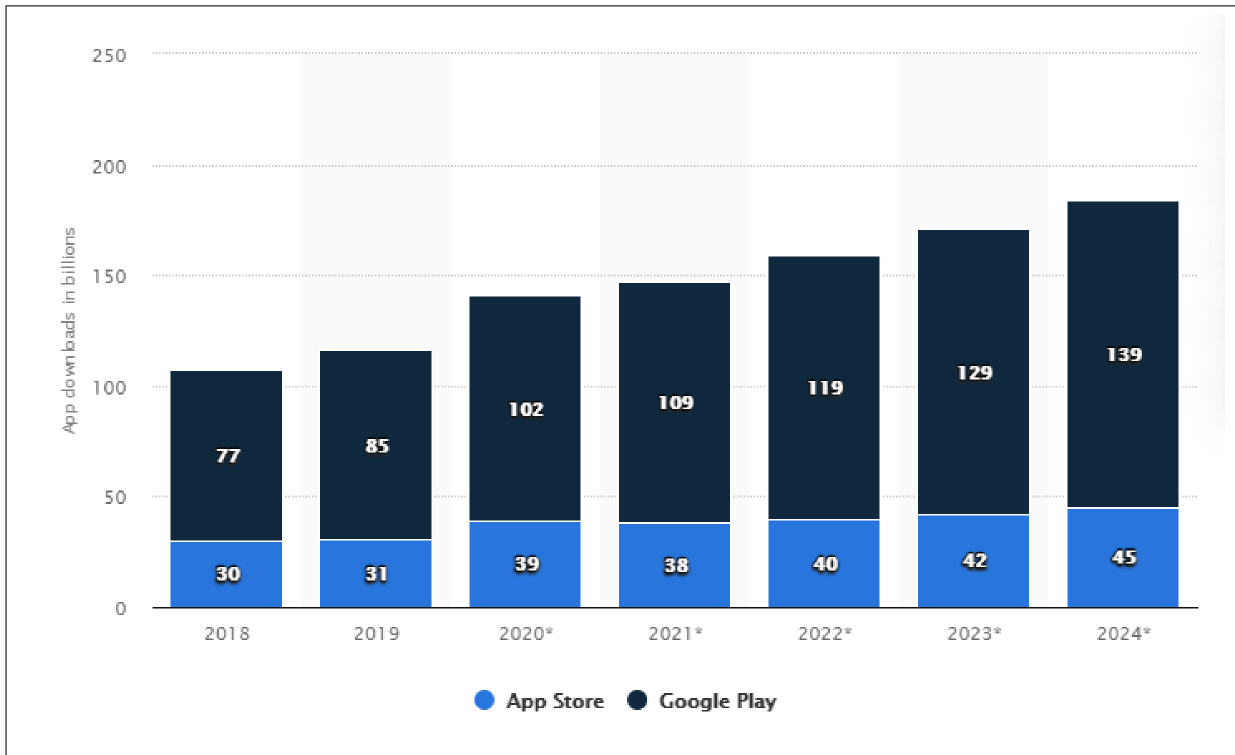


Figure 1.1 Downloads of mobile applications globally from 2018 to 2024
Taken from (Clement, 2021)

The Android ecosystem is a topic of investigation in a number of research projects. Most of these studies are looking into problems related to usability or violations of privacy and security. A brief overview of the literature reveals privacy and/or security issues. For example, (Zhou & Jiang, 2012) examined malware that is masked to appear as legitimate apps; (Fahl *et al.*, 2012) studied how invalid SSL certificates are accepted; (Drake, 2015) looked at memory corruption; (Ikram *et al.*, 2016) focused on the transmission of sensitive

information; (Calciati & Gorla, 2017; Almomani & Al Khayer, 2020; Iqbal *et al.*, 2018) studied permissions abuse. All of these research endeavours found that changes in the Android ecosystem can have a broad range of effects related to malware vulnerability, depending on the severity of the attack, the OS version, and the kind of devices that are impacted.

An area of research that is notably also expanding its studies on Android malware. The increase in the use of smartphones as everyday tools for financial transactions and control of "smart" appliances, for example, have made these tools targets of data theft. The thefts usually occur through malware that appears to be legitimate apps and are downloaded; the apps then steal the targeted data from devices or even hold users' data or devices for ransom.

In response to the massive increase in data theft, tools are being designed that are able to signal whether an app is benign or malicious. Some recent research includes frequency analysis that can determine malware-generated API calls (Aafer *et al.*, 2013; Takahashi & Ban, 2019); dangerous permissions (Arp *et al.*, 2014; Chakradeo *et al.*, 2013; Alqatawna *et al.*, 2021); call sequence (Canfora *et al.*, 2015; Vinod *et al.*, 2019); and fingerprinting (Karbab *et al.*, 2020; Canfora *et al.*, 2016; Dai *et al.*, 2013). The developed tools use program analysis, such as dynamic, static, and hybrid analysis. The results of these analyses are varied. For instance, Chen et al.(Chen *et al.*, 2016) found that the most popular tools for malware classification gave dismal performances during testing for the latest malware datasets. These test results should not be surprising, given the fast evolution of the Android OS and the malware targeting it. Malware classifiers with high performance at their creation may become obsolete within a few years or even a few months. Therefore, the design of robust malware detection tools with a long lifespan is

highly challenging, as the tools need to be replaced or modified within a relatively short time. This can lead to high costs related to retraining or replacement.

Given the above, what is urgently needed are malware classifiers designed to be robust to changes. Specifically, studies investigating issues around Android smartphone vulnerabilities, privacy and security, and detection and mitigation tools should be ongoing, considering that malicious actors are continuously changing their access and attack strategies. The present thesis intends to contribute to the ongoing research efforts in this expanding field of study.

1.2 Statement of Problem and Research Motivation

The Android ecosystem represents the latest evolution in consumer mobile communications. It is an open source mobile platform that powers hundreds of millions of smartphones and related devices, along with over a billion apps. Unfortunately, the Android's popularity has also drawn unwelcome attention to its technology and users, with the vast majority of smartphone malware targeting the Android ecosystem. Despite the fact that malware is usually distributed through third-party-operated markets, even large manufacturers cannot ensure that their listed apps are malware-free. A few of the more common Android malware threats are Spyware, Fake Installers, Bots, Phishing, Trojans, Root Exploits and SMS Fraud. Some malware, like that found in Download-Trojans apps, only download malicious code post-installation, making it difficult for compromised apps to be detected by the market vendor, even vendors as large and expert as the Google Android Market.

To date, the majority of developed strategies for malware detection have used conventional techniques, such as content signature. In that approach, a list of malware signature definitions is used as a reference, with each app then compared to a malware signature database. However, malware not included in the used database will not be detected with

this method. Chin et al.(Chin *et al.*, 2011) studied malicious patterns, concluding that signature-based strategies are consistently unable to keep up with the latest malware innovations. Therefore, despite the large number of proposed strategies in the literature aimed at detecting and analyzing malicious apps, the ever-changing Android malware always seems to be one step ahead of most available detectors and classifiers.

One promising way to keep pace with innovative malware is to develop an anomaly detection system which employs customized learning models. However, these models require high quality and plentiful data that usually exceeds dataset availability. With this in mind, the present thesis aims to develop a viable solution to the Android malware problem that includes app processing, feature extraction, and the ability to determine whether a perceived threat is benign or malicious.

1.3 Requirements of a Potential approach

In reviewing proposed approach in the relevant literature, we discovered several similar requirements that all approaches appear to have. For our proposed approach, we assume that an anomaly detection system has three main requirements, as listed below:

1. Flexibility: The proposed approach should be flexible and not restricted to limited range of applications.

2. Quality: The proposed approach should ensure data quality.

3. High performance: The proposed approach should be tuning the hyper parameters to achieve the desired performance level for the anomaly detection system.

1.4 Scope of Proposed Research

The proposed research focuses on mobile application security vulnerabilities that can be exploited to inject malicious code that threaten Android devices. Machine learning and static analysis will be employed in a scheme which identifies harmful apps to improve the security of Android users. In order to detect and identify malware in mobile apps that may cause security issues, static analysis is initially used for feature extraction and dataset development. The developed datasets will provide malware detection researchers an easy-to-use benchmark and scripts that reverse-engineer vulnerability information generation. We will then, using our developed datasets, test common classifiers as well as clustering and feature selection strategies to measure malware identification performance.

We have divided our detection process system into three general layers. In the first layer, we developed the autonomous dataset generator tool to create a dataset with information on the apps (benign and malicious) vulnerabilities as features. In the second layer, we investigate the impact of those features in terms of distinguishing between malware and benign apps by deploying several machine learning techniques. While in the third layer we focus on risks that occur when permissions grant access to sensitive data.

1.5 Technical Contributions

Our research introduces a security vulnerabilities repository known as "AndroVul" (short for Android Vulnerabilities) for identifying hidden malware in mobile apps that may result in security threats. AndroVul utilizes a dataset with 16,180 apps that can be downloaded either at Google Play Store or third-party venues. In our early experiments, we used four classifiers for assessing the vulnerability information's predictive power. When we used our dataset in various combinations, the test results were overall very good, showing that our approach is able to outperform earlier strategies in the literature. We found that our

method gives malware researchers important insights on the choice and calibration of possible classifiers for use in malware detection. The main contributions of the present research are as follows:

1. The development of a technique that employs reverse-engineering tools to simplify vulnerability information generation regarding aspects such as dangerous permissions, code smell, and vulnerabilities in AndroBugs in a wide range of applications.

2. The compilation of vulnerability data using 16,180 Android apps as a random sample. The apps, which were downloaded from the dataset AndroZoo, were further extended in our work by the addition of 3,978 malwares sourced from the VirusShare repository.

3. The publication of an in-depth study giving insights into the latest vulnerability information prediction mined using our developed tool. Classifiers as well as data labelling decisions are shown in this work to give better performance using our information.

4. Developing a malware detection model based on deep learning and we investigated several node architectures in hidden layers in order to get the highest possible performance. The proposed model outperforms the state-of-the-art.

5. Developing a malware detection model based on SVM and investigated different parameter settings to identify which were the best for our malware detection task.

6. Providing comparison of the performance of our DL and SVM classifiers, with respect to state-of-the-art approaches and even some commercial anti-viruses and results show that our classifiers are the most effective in identifying malicious applications. As such, our models establish a new, important reference point in the current state-of-the-art when it comes to malware detection.

7. Using an adapted combination of deep learning and the K-means clustering algorithm, we provide a novel strategy for mining deep-layers permission usage patterns.

8. Assessing our approach's efficacy by examining the coherence and the identified patterns' generalizability. The results reveal that our method was able to discover a greater number of usage patterns at various degrees of usage cohesiveness.

1.6 Thesis Outline

This chapter provides a general introduction to the thesis problem statement and research motivation. As well, the chapter presents the study's main goals, objectives, proposed methodology, and contributions. Chapter 2 presents a review of the relevant literature as a means to frame the research problem and discuss gaps in the latest schemes. Chapters 3, 4, and 5 represent three published and submitted articles that focus on the stated research problem:

1. Probing AndroVul dataset for studies on Android malware classification.

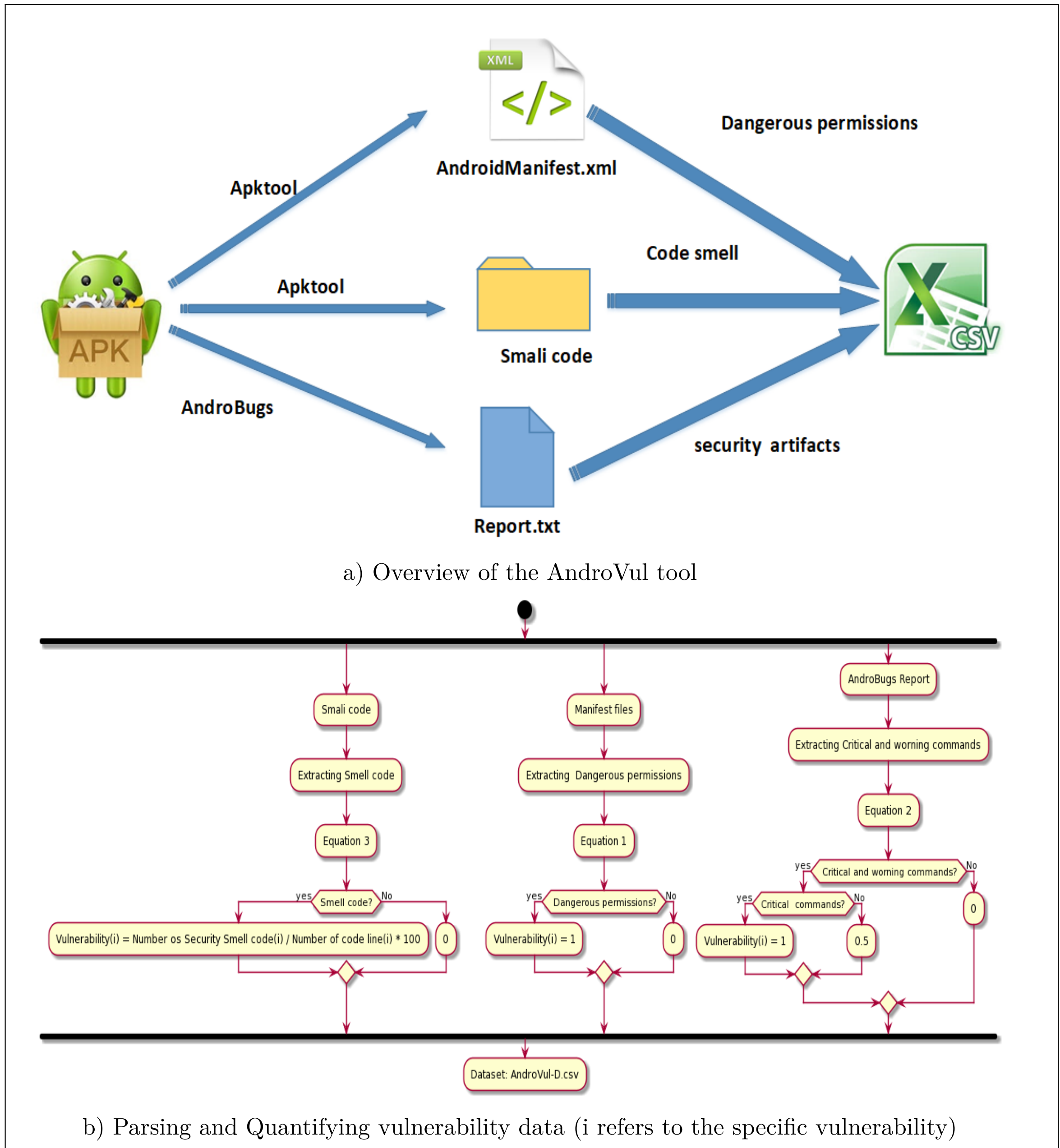
2. Deep learning Based Android Anomaly detection using a combination of vulnerabilities dataset.

3. Deep-layers Clustering to identify permission usage patterns of Android apps' categories.

1.7 Summary

The presented figures provide a summary of our contributions in this work. Figure 1.2 (a) gives an overview for the AndroVul tool, while figure 1.2 (b) depicts how our datasets were created using data from parsing/quantifying vulnerabilities. The primary objective is

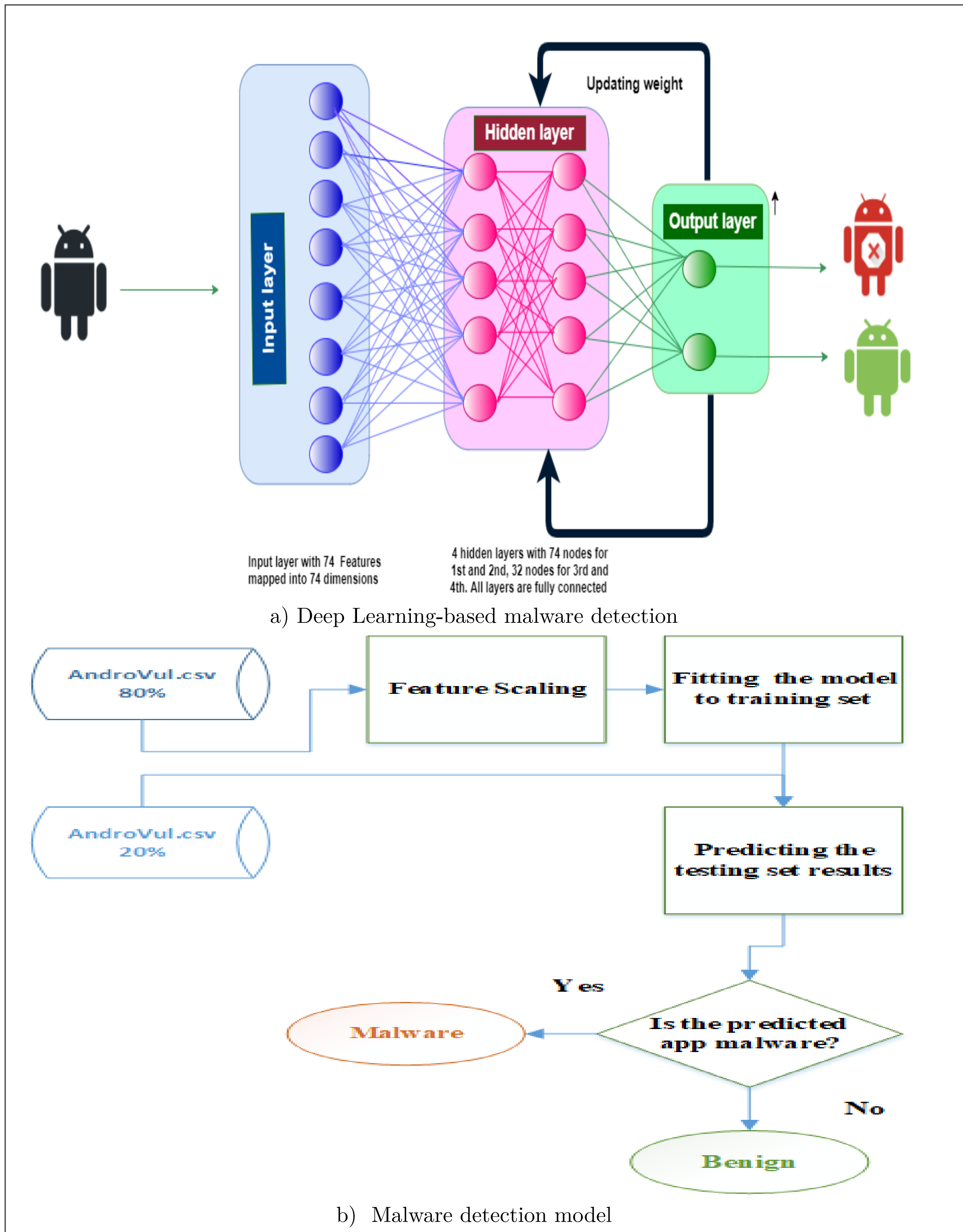
extracting vulnerabilities from both malicious and benign apps, and generating a dataset. As well, in order to detect malicious code, our developed machine learning models are trained and tested. In figure 1.3 (a) the model for deep learning layers in a sequential neural network is illustrated, along with our design methodology of Android malware detection. In figure 1.3 (b), the Support Vector Machine (SVM) model is presented, showing its training and testing phases. Our primary aim here is adding scalability as a feature in our scheme in order to better detect and discern between benign and malicious apps. In figure 1.4, a model combining SOM and K-means clustering based on a clustering validity test is built using permission as features and the self-organizing map (SOM). Our purpose is to describe the picture or pattern of how applications in a particular category behave by optimizing our model.



a) Overview of the AndroVul tool

b) Parsing and Quantifying vulnerability data (i refers to the specific vulnerability)

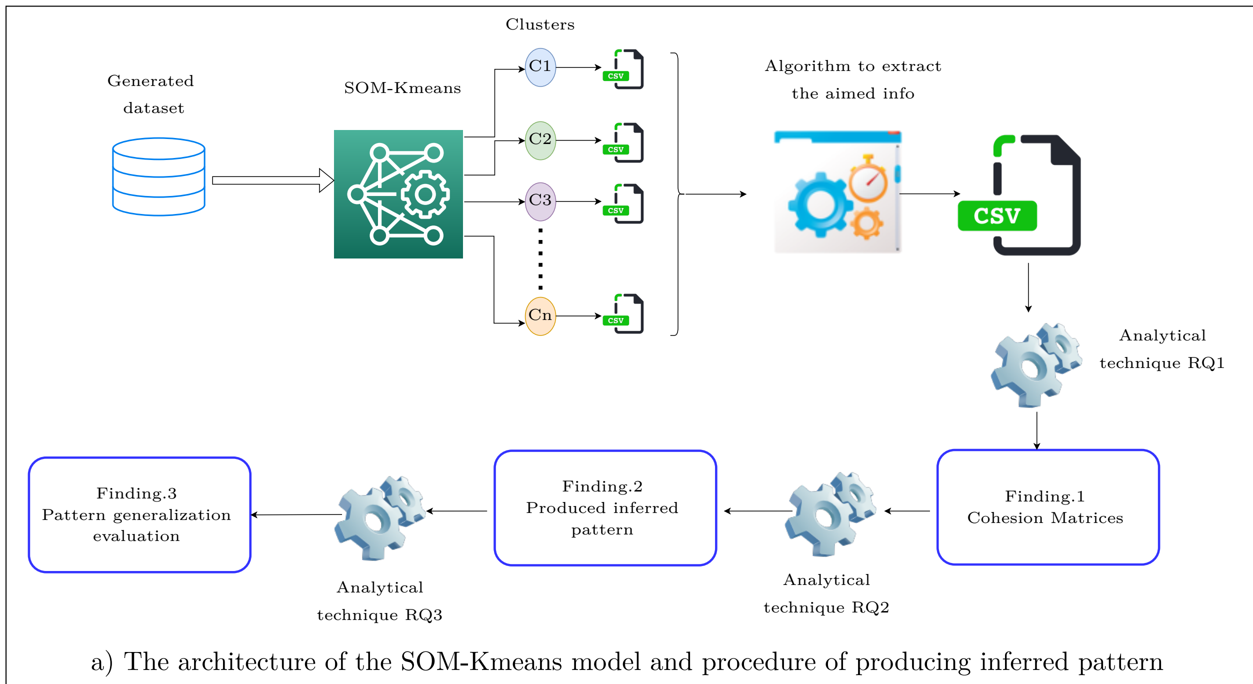
Figure 1.2 First contribution



a) Deep Learning-based malware detection

b) Malware detection model

Figure 1.3 Second contribution



a) The architecture of the SOM-Kmeans model and procedure of producing inferred pattern

Figure 1.4 Third contribution

CHAPTER 2

BACKGROUND AND LITERATURE REVIEW

2.1 Background

This chapter presents a high-level overview of Android app development and security mechanisms, followed by a review of the relevant literature. Additionally, we discuss the framework for our analysis, focusing on the structure and functioning of Android applications.

2.1.1 Android app development

In Android devices, apps are distributed through files called Android Packages (APK), which are ZIP files. APK files hold bytecodes for the apps, as well as data such as resources and third-party libraries. The APK files also includes a manifest in file format that provides information on the capabilities of the app. In figure 2.1, a simplified process is illustrated showing Java source code projects being translated into APK files (Herong, 2021). As a way to boost security, apps function within a sandboxed environment. Specifically, during the installation process, the Android OS gives the apps a Linux user ID that is unique, and file permissions are set such that only a specific app can access them. In this way, every app receives a unique Virtual Machine (VM), effectively isolating that code from all other apps.

Several different components are utilized in building Android apps. The best source for information regarding Android app components is official documentation.

1. **Activities**

   Activities denote a single screen featuring a user interface. Apps typically have several different activities for different purposes. For example, a music player may

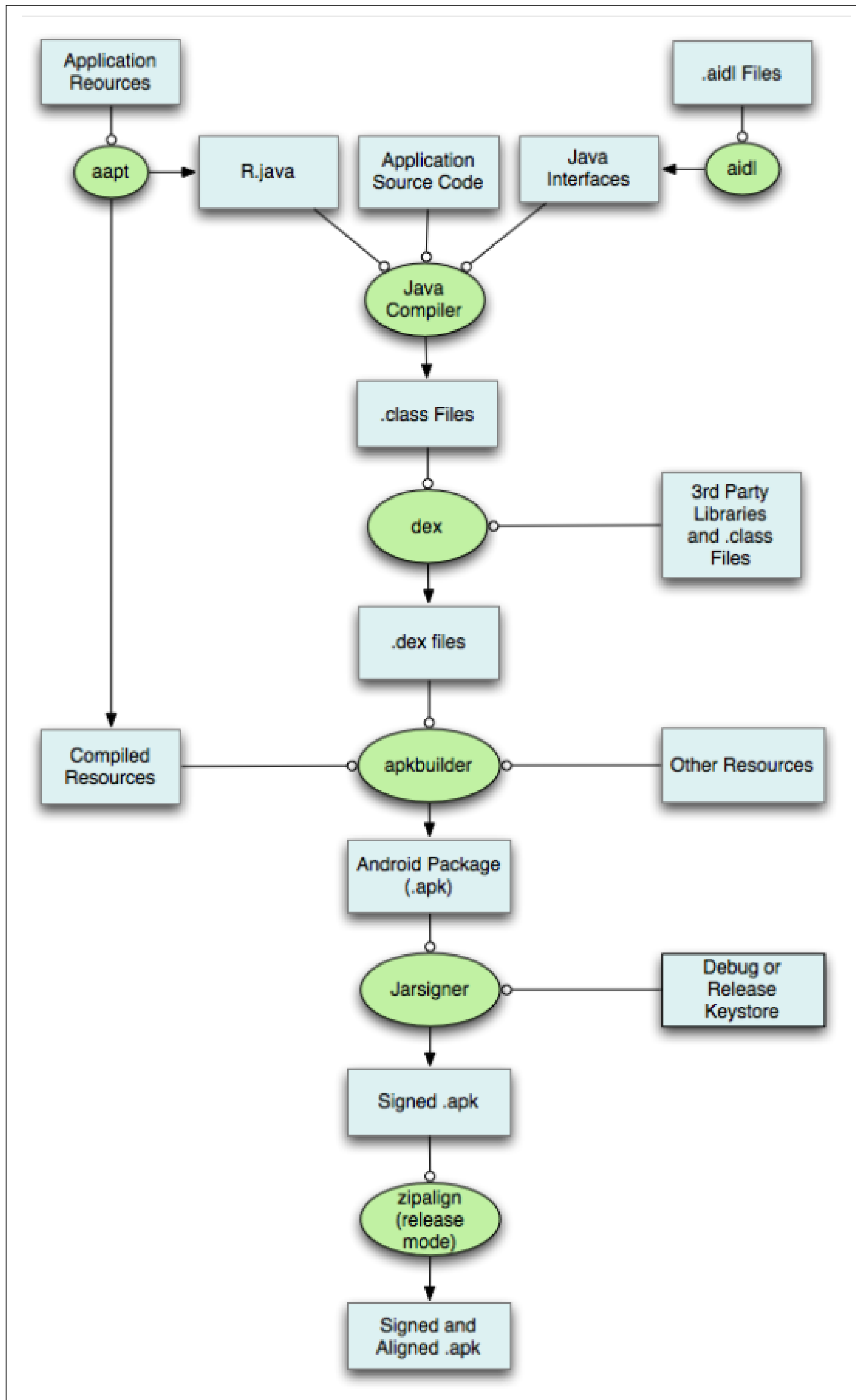


Figure 2.1 Android application build process
Taken from (Herong, 2021)

feature an activity displaying available albums in list format, along with other activities displaying the song currently playing and buttons for pausing the music, fast-forwarding, shuffling song options, etc. All of these activities are independent from each other and may be started by other apps, if the app grants permission. In this case, for instance, an e-mail user would be able to launch an app's "play" activity in an audio file.

2. **Services**

   Services represent background components which perform operations that have no user interface. In music apps, for instance, music might be able to be played in the background when a user is on another app. Various app components (broadcast receiver, activity, etc.) can launch services.

3. **Content providers**

   Multiple applications can share data through content providers. For example, one app can store information that could be accessed and queried by other apps. In a music player, the content provider stores information on the song currently playing. This information may then be shared with, for instance, a social media application, updating the "current listening" information of the user.

4. **Broadcast receivers**

   The task of broadcast receivers is to monitor broadcast announcements and respond appropriately to their message. For example, a broadcast might announce a low battery, completion of a reboot process, an incoming text message, etc. There are no user interfaces for broadcast receivers, as they generally only serve as component gateways between components.

5. **Intents**

   Intents are asynchronous messages that activate broadcast receivers, services, and activities. In broadcast receivers, intents define any announcement currently in broadcast. Using an incoming SMS text message as an example, an additional data

field may contain the message content as well as the phone number of the message sender. In services and activities, intents define any action to be performed, such as "send" or "view". The intents could also involve other data that indicates what should be acted upon. So, for instance, a music player application might send a "view" intent to a browser which then launches a webpage showing information about the artist performing the music.

2.1.2 Distribution and security of Android apps

Users of Android devices are able to install free or paid apps through Google Play Store, which is the main application distribution platform for Android. Malicious apps are constantly being screened through Google Bouncer, which is an automated anti-virus system developed by Google. When installing apps from other sources outside Google Play, users have to enable the settings option for "unknown sources" on the device and accept any risk that may occur from the installation. For external app sources, users are able to choose web-downloaded APK files or third-party markets.

2.1.3 Vulnerabilities in Android Apps

Vulnerabilities in Android apps are a growing problem. The main issues arise from Google Play Store's open format as well as from users' ability to side-load apps. These options limit oversight on the apps' safety. At the same time, Android apps usually have some form of limitation and flaws that leave them vulnerable to the actions of malicious players. The following subsections provide a general overview of important Android vulnerabilities.

2.1.3.1 Android permissions

Application permissions in Android devices serve as controls for increasing the awareness of users as well as limiting an app's access to data considered "sensitive", such as passwords or bank account information (Source.android.com, 2021). During installation, a package

manager grants apps permissions, with the application framework responsible for enforcing system permissions.

However, because apps, by their nature, require interactive access to other apps or system services, they occasionally have to move beyond the sandbox via an Android permission model that enforces restrictions only for certain operations. Permissions occur at the four following protection levels (Developer.android.com, 2021b):

1. Normal: This protection level designates a permission that is low-risk, which means it will permit apps access to API calls such as SET_ WALLPAPER. This form of access will not harm users (Developer.android.com, 2021b).

2. Dangerous: This protection level designates a permission that is high-risk, meaning that it will permit apps access to API calls that are potentially harmful, such as READ_CONTACTS. This form of access could result in the leak of sensitive user data or even the loss of control of the Android device. Permissions designated as "dangerous", also referred to as "runtime" permissions, must be explicitly approved by the device user prior at run-time. In other words, it is up to the user to grant a permission or deny the runtime permission (Developer.android.com, 2021b).

3. Signature: This protection level designates a permission granted when the requesting app has been signed using the same certificate as the app (being requested) uses. The signature in this case constitutes permission granted (Developer.android.com, 2021b).

4. Signature-or-System: This protection level designates a permission granted only when the requesting app is either signed using the same certificate as the app (see "Signature" protection level above) or exists on the same Android system (Developer.android.com, 2021b).

2.1.3.2 Security Smells

"Code smells", in computer programming, is a term used to describe elements within a code which may potentially be problematic. In a publication on code refactoring, Fowler (Fowler, 2018) investigated some of the more common code smells. Research on code smells in Android has thus far mainly looked at issues around energy consumption (Gottschalk *et al.*, 2012), performance (Hecht *et al.*, 2016), and maintainability (Hecht *et al.*, 2015; Palomba *et al.*, 2017).

In the present research work, we explore the concept of security code smells in relation to security vulnerabilities. It is worth noting that security code smells refer to aspects of a code which indicate a potential security vulnerability, while security vulnerabilities refer to security problems which compromise the privacy and security of users. Similar to conventional code smells, security code smells often concern a technical component, albeit one directly or indirectly related to security.

2.1.3.3 Physical attacks

Due in part to their ease of portability, smartphones have become a repository for many users' day-to-day private information, ranging from contact lists to bank-related passwords. This makes smartphones even more of a target for theft than wallets. Stolen mobile devices, as well as lost devices, are an increasing security threat (Dunham, 2008; Murray, 2021).

2.1.4 Malware threats

The word "malware" is a relatively new term that was formed by combining the first syllable of "malicious" and the last syllable of "software". As its name implies, malware represents malicious code that is developed for the sole purpose of attacking a target with malignant and abusive behavior, such as disrupting system operation, executing unwanted programs and commands, gaining unauthorized access to system resources, and

collecting sensitive information without permission. Most malware exploits vulnerabilities that already exist in systems (Chen & Peikari, 2008).

The targeting of smartphones by malware is increasing exponentially, despite extensive efforts to thwart it. According to Aittokallio (Aittokallio, 2021), upwards of 15 million mobile devices have been infected by malware globally, and more than half of the malware-infected devices are Android smartphones.

Many researchers focus their studies on Android OS, as nearly 99% of detected malware (as of 2013) targets Android systems (Chebyshev & Unuchek, 2014). The targeting of Android is likely due to three main reasons: 1) Android's general popularity: 2) Android's architecture makes it vulnerable to malicious interventions; and 3) Android offers app developers extensive assistance and resources for developing and uploading apps in their online stores. Moreover, Android malware developers have made great strides since the first malware was detected on Android in 2010. Specifically, they have enhanced their abilities both to distribute the malware and avoid being detected. For example, the China-originating Ginmaster trojan was released in 2011. It was distributed via legitimate apps through the injection of malicious code and easily resisted detection by, among other tactics, encrypting URLs and changing class names (Chebyshev & Unuchek, 2014).

2.1.4.1 Malware classification

Both smartphone malware classifications and those of computer malware share many similarities, with the different classifications being made according to specific characteristics. To determine the malware classification, a range of questions can be asked about the malware, such as:

1. Is the malware hiding either itself or its malicious function?

2. Can the malware be described as “non-cloning” or “self-cloning”?

3. Based on its manifestations, does the malware appear to be a part of a program (i.e., a piece of a code) or is it a stand-alone program?

Figure 2.2 depicts the main classifications for malware (Chen & Peikari, 2008).

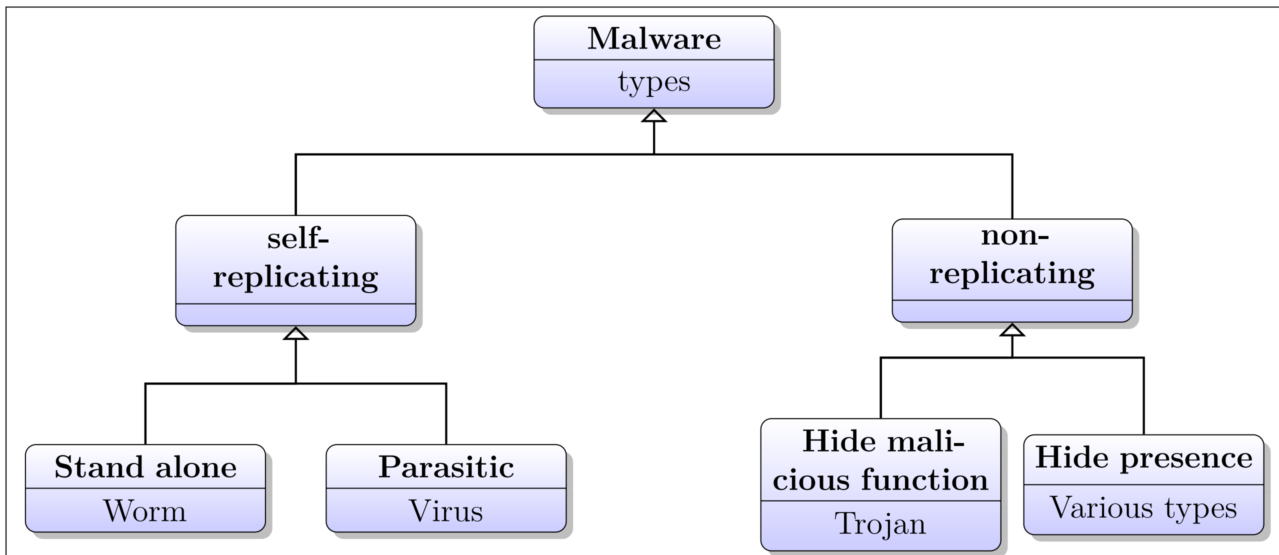


Figure 2.2 Mobile malwares classification

1. **Virus:** A virus is malware which spreads through the insertion of a self-copy, enabling it in this way to become part of other programs. Viruses proliferate from device to device, infecting each device before moving to the next. Nearly all viruses have an executable file, meaning that the virus can only become active on a system if a user opens or runs the malicious program or host file. Executing the host code also executes the viral code. Host programs continue to function even when infected. In fact, for most viruses, the aim is to cloak their existence so that they can spread "invisibly" from one device to the next through file-sharing, infected email attachments, etc. (Chen & Peikari, 2008; Security, 2021).

2. **Worm:** A worm refers to standalone software which does not need a host program or human intervention to spread. Instead, a worm will exploit a system's vulnerabilities or trick targeted users into downloading the software. Worms gain access to a system through a vulnerability and then use the system's information-transport or file-transport features. This enables worms to move unaided. Some worms, to outsmart detectors, use technologies such as encryption and ransomware to launch an attack (Chen & Peikari, 2008; Security, 2021).

3. **Trojan:** A trojan describes a kind of malware that gets its name from the infamous wooden horse used by the Greeks to get inside the city gates of Troy without the Trojans knowing. To all appearances, a trojan looks like legitimate and benign software. This benign appearance tricks the targeted users into loading and executing the software. Then, when activated, the trojan starts launching attacks. These can be as mundane as constantly presenting pop-up windows or as severely damaging as deleting files or stealing sensitive data. Trojan are also well-known for creating backdoors that provide other malicious players access to the targeted user's system. As mentioned above, both viruses and worms reproduce in a system by self-replicating or infecting other files. However, trojans spread via user interaction, which may include downloading and running an Internet file or opening an attachment on an email (Chen & Peikari, 2008; Security, 2021).

4. **Spyware:** Spyware installs on a targeted user's device as a program. The two primary purposes of spyware are to monitor the activities of the device user, and to collect protected personal information, like passwords, usernames, and contact lists. Once collected, the information is transmitted by the spyware to a party that typically uses it for either law enforcement or advertising purposes. The installation of spyware usually happens when the targeted user accepts an offer of a "free" trial software and downloads it. Spyware may even be installed simply by visiting a web page (Chen & Peikari, 2008; Security, 2021). A recent example of common spyware on an Android device is Nickispy, which is able to record phone calls when installed (Jiang, 2021).

5. **A botnet:** Bots are self-propagating malware. They attack smartphones by first infecting the system and then connecting back to a server. The infected server, in turn, serves as a command and control center that controls a whole network of systems that have been infected. This network is called a "botnet". Bots are able to perform tasks such as collect passwords, register keystrokes, and launch Denial of Service attacks. Bots can also open back doors within infected devices, allowing unauthorized access to other malicious players (Security, 2021).

6. **Trackware:** As a program, trackware collects information which identifies a device or a specific user of a specific device in relation to an app, such as one which gives the general or even exact location of the device being used (Johnson, 2021).

7. **Adware:** Adware refers to apps where ads play during execution. Adware's sole purpose is generating revenue for the app developer. However, some adware is more intrusive, changing settings on the system or leaking the device user's private information (Johnson, 2021). Figures 2.3 and 2.4 show malware classifications (Johnson, 2021). Some smartphone adware, such as RiskTool and AdWare, has been developed for more nefarious purposes, like stealing a user's private data.

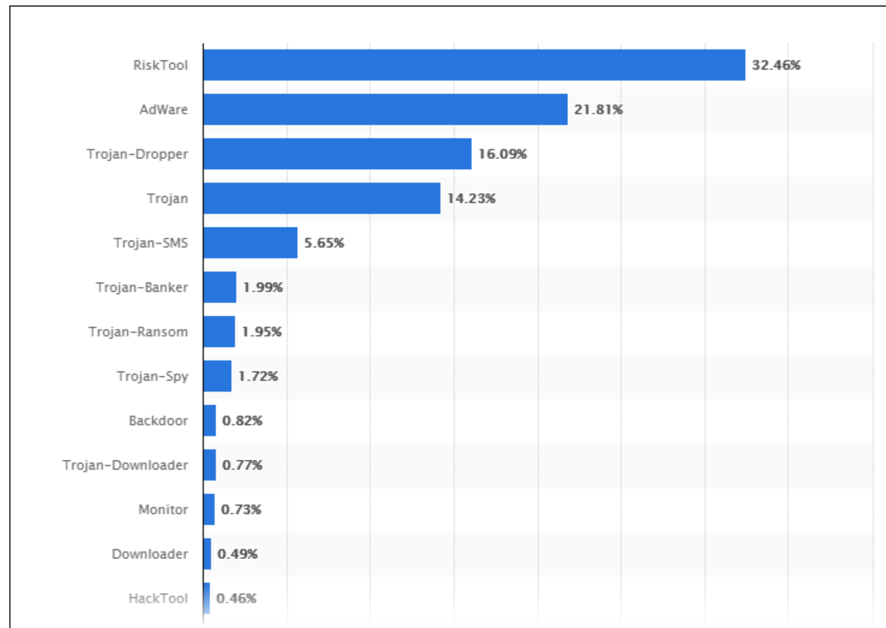


Figure 2.3 Global mobile malware distribution in 2019
Taken from (Johnson, 2021)

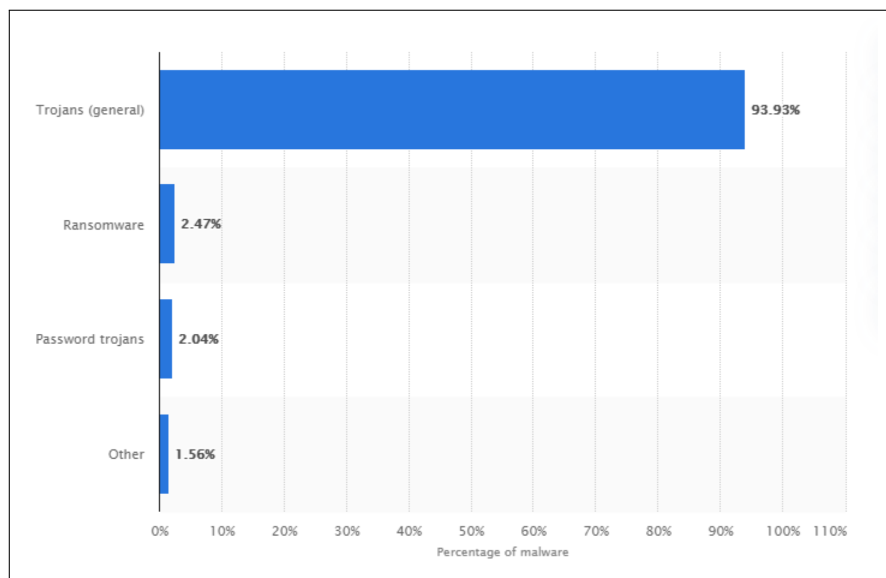


Figure 2.4 2019's top Android malware distribution
Taken from (Johnson, 2021)

2.2 Literature review

Research focusing on security is a hot topic, mainly owing to the large-scale distribution of Android devices. In particular, numerous studies on mobile device security are being conducted, but these are not simple endeavours, considering that this type of security has several different actors involved. The present work examines Android-related literature on vulnerabilities in security, with an emphasis on how datasets and machine learning can identify malware. The review begins by presenting various analyses on how vulnerabilities can be identified. This chapter also reviews the state-of-the-art in malware detection. Because the purpose of this study is to improve on existing detection and mitigation methods, it is useful to review the latest literature in the field, along with some earlier work that laid the foundation for malware detection and mitigation.

2.2.1 Android application datasets

Various datasets have been collected and studied in order to understand the behavior of one particular application or apps in general. These datasets strongly influence both the accuracy and execution performance of malware detection strategies. The data are extracted at different levels, namely the operating system (OS), the app, and the device's hardware. For a few years after the Android system's introduction over a decade ago, very few datasets representing labeled malware for Androids were available to the public (Bose *et al.*, 2008; Enck *et al.*, 2009; Shabtai *et al.*, 2012; Yamaguchi *et al.*, 2011; Zhou & Jiang, 2012). Then, in 2012, one of the first labeled datasets (Android Malware Genome Project) was released (Zhou & Jiang, 2012). This dataset featured over 1,200 samples of Android malware. A few years later, the Drebin dataset was publicly released (Arp *et al.*, 2014). It contained substantially more samples of Android malware, but it was also soon outdated (Wei *et al.*, 2017). So, another dataset had its debut in 2017, showcasing over 24,000 malware samples that had been collected from 2010 to 2016 (Wei *et al.*, 2017). However, within the Drebin and the later dataset, there was clear evidence of data drift (Arp *et al.*, 2014). The Drebin dataset in particular was found to be inadequate for training in light of

the ever-evolving complexity of malware threats (Grano *et al.*, 2017). In the later dataset were about 400 apps from the F-Droid repository, and among these were approximately 600 versions for each application. As well, 8 unique code smells were extracted (Wang *et al.*, 2019).

From 2014 to 2018, (Wang *et al.*, 2019) developed a dataset of Android malware from Google Play's application maintenance results, using VirusTotal for app labelling. In addition to collecting and labeling malicious apps, other information was also collected for each sample, such as the name of the app and its package name, the malware family name, the name of the developer, a description of the app, user ratings, the type of installation, the category of the app, and how many times it was flagged on VirusTotal (Wang *et al.*, 2019). Another recent dataset of apps was made publicly available by AndroZoo (Allix *et al.*, 2016). It contains over 14 million Android applications that are available on Google Play Store, as well as on some smaller markets and repositories. The purpose of AndroZoo is to develop a collection of apps for research purposes that is comprehensive and up-to-date.

Table 2.1 presents a listing of the Android malware datasets that are most commonly consulted today, including details on their main features and size.

Table 2.1 The most widely used Android malware datasets

Dataset	Size	Features	Year	Metadata
Genome (Zhou & Jiang, 2012)	1,200	Permissions	2010-2011	No
Drebin (Arp *et al.*, 2014)	5,560	Permissions API calls	2014	No
Piggybacking (Li *et al.*, 2017)	1,497	Permissions API calls	2016	No
Krutz et al. (Krutz *et al.*, 2015)	4,416	Permissions intents	2015	No
Grano et al. (Grano *et al.*, 2017)	600	Code smells user reviews	2017	No
Wang et al. (Wang *et al.*, 2019)	9,133	No	2014-2018	yes

2.2.2 Using Machine learning

The next subsection provides an overview of research work on malware detection for the Android environment. The two main detection techniques are dynamic analysis and static analysis. In dynamic analysis, apps are investigated within a run-time environment. The researchers look at the system's responses to the apps' dynamic behavior, with dynamic features such as system calls, resources usage and network connections being monitored. Static analysis, on the other hand, investigates source code in apps for detecting malicious patterns. However, instead of executing the source code, executable apps are disassembled down to the files for the source code. There, any number of features can be extracted, including broadcast receivers, intents, permissions, APIs, data and control flow, and hardware components. In both dynamic and static analysis, data are collected as a means to train the classifiers for machine learning, which will then create separation modeling between the apps' malicious and benign characteristics. The next subsections present some recent important research studies focusing on dynamic and static analysis in Android app malware detection.

2.2.2.1 Static Analysis

In (Arp *et al.*, 2014) published a study proposing DERBIN. This is a static analysis framework which extracts from the AndroidManifest.xml features (e.g., filtered intents,requested permissions, hardware components, app components) for generating joint vector space (Arp *et al.*, 2014). The authors also employed Support Vector Machine (SVM) in their dataset as a way to separate benign and malicious app classes. They then tested their proposed system using 123,453 benign apps and 5,560 malware apps showing a 94% successful detection rate for malware and a 1% false positive rate (Arp *et al.*, 2014). As well, the proposed system provides the user with guidance on identifying malicious patterns and suspicious properties (Arp *et al.*, 2014).

A few years earlier, (Sanz *et al.*, 2012) had proposed a strategy for detecting malicious Android apps using machine learning. The method analyzes extracted permissions found within the app. For this approach, the classification features include the following: the uses-feature, which applies to the device used by the app; the total permissions for the app; and tag uses-permission, which refers to each permission an app uses in order to function. Moreover, (Sanz *et al.*, 2012) adopted supervised machine learning methods for the classification of Android apps as being either benign or malicious. They gathered 820 applications. To detect benign apps, the authors chose apps from different categories on Google Play. These apps included: Web apps that were mostly developed using JavaScript, HTMS and CSS; native applications that were developed using Android SDK; and widgets that were relatively simplistic apps that are similar to Web apps and are developed for Android Desktop.

In (Huang *et al.*, 2013) study, the authors investigated using classification learning to detect malicious apps on Android devices. The machine learning algorithms used in the research were SVM, AdaBoost, Decision Tree (C4.5), and Naïve Bayes. A total of 20 features were extracted during the research, such as requested permissions and permissions that were actually used. The selected features' values were then stored under the classification of feature vector and represented by comma-separated values displayed sequentially (Huang *et al.*, 2013). In the work, the authors demonstrated that the best result for their classifiers could detect upwards of 81% of malicious apps. Hence, by combining the results from several classifiers, a filter could be quickly developed for identifying even more suspicious apps (Huang *et al.*, 2013). Overall, 480 malicious and 124,769 benign apps were collected as a dataset and then utilized in the study's experiments (Huang *et al.*, 2013).

In (Zarni Aung, 2013) research, a framework was introduced for developing machine learning-based malware detection in Android devices. The authors aimed to employ their framework both for detecting malicious apps and for enhancing users' privacy and security when using Android smartphones. The proposed system is able to extract permission

as features from APK files and then analyse them with machine learning classifiers in order to determine if the app is malicious or benign. Examples of the features include requested permissions like CHANGE_CONFIGURATION, WRITE_SMS, SEND_SMS, CALL_PHONE, etc. The authors experiment on a dataset of 500 Android apps that have 160 different features, and the best result is 91.6% in precision. (Zarni Aung, 2013).

In (Su & Chang, 2014) conducted research to find out if an app can be classified as malware by investigating the app's announced permission combination. The authors employed two weighted methods in their study for adjusting the permissions' weights. The two approaches use permission occurrences for both of the samples, combined with frequency gap between them. Some metrics determined by previous studies were then used for determining risk scores, with a higher risk score denoting a high user risk and the likelihood that the app is malware (Su & Chang, 2014).

In a study conducted by (Sanz *et al.*, 2012), a machine learning technique was proposed for malware detection and automatic categorization of Android apps. This strategy functions by analyzing various sets of app features that have been extracted from source code files (regarding printable string occurrence frequency), the Android market (regarding rating, number of ratings, and permissions), and the AndroidManifest.xml file. In the study, the authors employed the following five machine learning algorithms: Support Vector Machines (SVM), K-Nearest Neighbour (KNN), Decision Trees (DT), Random Forest (RF), and Bayesian Networks (BN). Overall, the study results showed BN as being the best classifier of the five. RF was found to be the second-best, while DT was found to be the worst (Sanz *et al.*, 2012). In later related work, Sun et al.(Sun *et al.*, 2017) used SVM in their extraction strategy for detecting malware in Android devices. Based on Keywords Correlation Distance (KCD), the approach combined features to form keyword feature vectors. SVM was then employed for detecting novel malware as well as malicious variants, giving an accuracy rate up to 88% (Sun *et al.*, 2017).

In the proposed and developed IntelliAV, (Ahmadi *et al.*, 2017) introduced a malware detection system that is built into the Android device. IntelliAV, which utilizes static analysis and machine learning, is currently (as of the time of the present thesis' research) available on Google Play Store. In (Ahmadi *et al.*, 2017) study, the technique trained and validated 19,722 VirusTotal apps, of which 9,664 were malicious. According to the results of their experiments, the authors had a 92.5% true positive rate and a 4.2% false positive one, based on 1,000 attributes that had been generated in training. Notably, this is substantially smaller than those found in static analysis approaches that have machine learning off-device. While the study's results are not as successful as some dynamic analysis and monitoring strategies, the available solution may potentially be used as a complementary addition to other methods (Ahmadi *et al.*, 2017).

2.2.2.2 Dynamic Analysis

In (Shabtai *et al.*, 2012) study explored the use of an Artificial Neural Network (ANN)-based system for detecting Android malware by analyzing system calls and permissions in an app. In this work, the authors employed two kinds of ANNs: Feedforward Neural Networks (FNN) and Recurrent Neural Networks (RNNs). The FFNs trained the proposed model by using requested permissions for creating patterns that could distinguish between malware and benign apps. The RNNs trained the proposed model using system calls from the execution behaviours of benign apps (Shabtai *et al.*, 2012).

In earlier research, (Zhao *et al.*, 2011) developed detection engines called Antimaldroid for detecting malware by utilizing either heuristics-based or permission-based methods. The proposed Antimaldroid features two distinct engines that are either heuristic-based or footprint-based. The heuristic-based detection engine essentially is used for detecting zero-day malware, with the heuristics-based filtering automatically monitoring specific malicious actions. These could include native Linux system calls or dynamic loading of new code. Heuristic-based detection engines also analyze logged system calls in order to find any malicious behaviors. On the other hand, the footprint-based one employs

behavioral footprint matching and permission-based filtering for detecting malware. The behavioral footprint matching then attempts to match an app (based on its behaviors) to other malware listed in AndroidManifest.xml according to its APIs, structural layout, byte code, etc.(Zhao *et al.*, 2011).

In (Wu & Hung, 2014) research into malware detection developed DroidDolphin. This approach has been widely citepd in the literature and has succeeded in setting the standard for machine learning-based dynamic detection of malware. Although DroidDolphin's performances are relatively low, with detection rates between 86% and 93%, it includes 64,000 app samples, of which around 32,000 are found to be malicious. The authors' study was conducted using an emulator, further, The accuracy of detection increases dramatically as the dataset grows, making this technology attractive. (Wu & Hung, 2014). Another important detection method is (Amos *et al.*, 2013) STREAM, which (Wu & Hung, 2014) DroidDolphin is often compared to. STREAM is typically citepd in reference to machine learning-based dynamic analysis. ADroid (Bhatia & Kaushal, 2017) and SafeGuard (Jeong *et al.*, 2017) both use dynamic analysis and monitoring of real environments to detect malware. However, as neither of these two approaches need root access, they cannot trace kernel calls and so cannot detect privilege escalations. (Saracino *et al.*, 2016) proposed a malware detection approach called MADAM that can be launched directly on an Android smartphone device. It uses multi-level architecture featuring machine learning and dynamic analysis. Although this technique is highly promising, users need to install the MADAM apk, as well as Xposed for hooks and events and SuperUser for root access (Saracino *et al.*, 2016).

In related research, (Bhatia & Kaushal, 2017) develop a dynamic analysis strategy that uses strace-based feature extraction, Monkey 5 exploration, and virtual machine installation. The same year, (Yalew *et al.*, 2017) published their study on the proposed T2Droid method, which partly is run within ARM TrustZone's secure zone.

Various latest-generation smartphones come with ARM TrustZone as a trusted execution system for launching the system and apps. Furthermore, ARM TrustZone provides a system overview and is guaranteed secure, which many antivirus softwares are not. On the other hand, because this approach needs access to secure areas and full control of hardware for software deployment, it is challenging to include it on mobile devices. Hence, In (Yalew *et al.*, 2017) study offers innovations which could be useful to manufacturers and other players in the industry, though not necessarily the end user. Also in (Fan *et al.*, 2017) published a study on a dynamic analysis approach called DroidInjector, which is based on injecting a library (via ptrace_attach) into the processes of apps that are potentially malicious. The main purpose is to monitor these apps by gathering information on their behaviour and then sending it to a remote server, where the information would be analyzed. Information that could be recovered during this process might be, according to (Fan *et al.*, 2017) even more relevant than information retrieved using dynamic analysis strategies that are based on ptrace or strace, e.g., DroidTrace (Zheng *et al.*, 2014) . Furthermore, the authors propose that their developed technique functions equally well on an emulator as on a physical device (Fan *et al.*, 2017). Although they do not offer either quantitative results or an implementable app in their study, they mention their intentions to deploy their proposed malware detection strategy on Android markets (Fan *et al.*, 2017).

Table 2.2 Comparison between various state-of-art solutions

Reference	Generated Dataset	Metadata	Feature Selection	Machine Learning	Dataset Size	Features	Limitations
(Arp *et al.*, 2014)	✓	✗	✗	✓	5,560	permissions API calls	Ancient dataset
(Sanz *et al.*, 2012)	✗	✗	✓	✓	820	permissions	Very small dataset Ancient dataset
(Huang *et al.*, 2013)	✗	✗	✗	✓	124,769	permissions	Ancient dataset lack in accuracy
(Zarni Aung, 2013)	✗	✗	✗	✓	500	permissions	Ancient dataset Very small dataset
(Su & Chang, 2014)	✓	✓	✗	✗	535	permissions	Ancient dataset Very small dataset
(Ahmadi *et al.*, 2017)	✓	✗	✗	✓	19,722	permissions APIs	lack in accuracy
(Shabtai *et al.*, 2012)	✓	✗	✗	✓	44	system calls	Ancient dataset Very small dataset
(Wu & Hung, 2014)	✓	✗	✗	✓	64,000	API calls	Consume resources
(Bhatia & Kaushal, 2017)	✗	✗	✗	✓	100	system calls	Over-fitting issues
(Saracino *et al.*, 2016)	✗	✓	✗	✓	2,800	system calls APIs	Manually configured
(Yalew *et al.*, 2017)	✗	✓	✗	✓	160	system calls API calla	Emulator Used Avoid threats
(Fan *et al.*, 2017)	✗	✗	✗	✗	230	API calls	Emulator Used Manually configured
Our work	✓	✓	✓	✓	18,526 15,893	Permissions Code Smell AndroBugs Vulnerabilities	

CHAPTER 3

PROBING ANDROVUL DATASET FOR STUDIES ON ANDROID MALWARE CLASSIFICATION

Zakeya Namrud[a], Sègla Kpodjedo[a], Chamseddine Talhi[a], Alvine Boaye Belle[b]

[a] Department of Software and IT Engineering, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[b] Department of Electrical Engineering and Computer Science, York University, Toronto, ON M2J 4A6, Canada

Paper published in Journal of King Saud University-Computer and Information Sciences, September 2021

3.1 Abstract

Security issues in mobile apps are increasingly relevant as this software has become part of the daily life of billions of people. As the dominant OS, Android is a primary target for ill-intentioned programmers willing to exploit its vulnerabilities by spreading malwares. Significant research has been devoted to the identification of these malwares. The current paper is an extension of our previous effort to contribute to research with a new benchmark of Android vulnerabilities. We proposed AndroVul, a repository for Android security vulnerabilities, that builds on AndroZoo – a well-known Android app dataset – and contains data on vulnerabilities for a representative sample of about 16,000 Android apps. The present paper adds confirmed malwares from the VirusShare dataset and explores more thoroughly the effectiveness of different machine learning techniques, with respect to the classification of malicious apps. We investigated different classifiers and feature selection techniques as well as different combinations for our input data. Our results suggest that the classifier MPL is the leading classifier, with competitive results that favorably compare to recent malware detection work. Additionally, we investigate how to classify (as benign or malicious) AndroZoo apps based on the number of antivirus flags they are tagged with. We found that different thresholds only marginally affect the

machine learning classifier results and that the strictest choice (i.e. one flag) performs best on the confirmed malwares from VirusShare.

3.2 Introduction

With a market share of 73% (Statcounter, 2021), Android is undoubtedly the leading mobile Operating System (OS) of our times. Besides, thanks to its openness, it is well-positioned to become the default OS for new applications centered on the Internet of Things (IoT). Security aspects are thus increasingly relevant as there are more incentives for malware developers to target Android devices. Android security has accordingly been extensively researched and that effort has been helped by the AndroZoo(Allix *et al.*, 2016) dataset put together by Allix et al.(Allix *et al.*, 2016) in 2016. The AndroZoo dataset is a very useful resource for Android researchers in general but security researchers still have many hurdles to pass from the moment they discover AndroZoo to the moment they can effectively get actionable data from it.

In (Namrud *et al.*, 2019b), we proposed AndroVul, a repository aiming to provide researchers working on anomaly detection of Android applications with: i) a benchmark readily usable (to test hypotheses), and ii) tool that will jumpstart their data collection. Our dataset includes data on 16,180 applications from Google Play store as well as third party stores (Allix *et al.*, 2016). Our tool extracted from these apps' binaries called APKs (Android Package Kits): 1) permissions classified as dangerous by the Android permission system; 2) data collected via AndroBugs, a popular Android vulnerability scanner[1]; and 3) security code smells, as recently defined by (Gadient *et al.*, 2017). Furthermore, we proposed preliminary experiments aiming at probing the predictive power of these various sources of vulnerability and found that it was preferable to include all of them, with dangerous permissions data being the best input.

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

The present paper builds on that previous work with an extensive investigation of different classifiers and feature selection techniques applied to various setups of our benchmark.

A key question that many researchers face when using AndroZoo for malware detection/ classification purposes relates to the labeling of the apps: which ones should be considered malwares? AndroZoo only gives the number of antivirus flags an app got on the VirusTotal(virustotal, 2021) website and it is up to the researchers to know how to use that information.

(Arp *et al.*, 2014) — an influential paper published in 2016 — answered that question by classifying as a malware any app with at least 2 flags from a set of ten selected antivirus products. Since then, the set of antiviruses from VirusTotal has grown so the relevance and completeness of that initial set of ten products is questionable. More importantly, the number of antivirus flags is the information that is readily available from the main AndroZoo file; there are no information on which antiviruses flagged an app. Typically, papers who consider flags from a preselected group of antiviruses from VirusTotal are not based on AndroZoo data. Papers based on AndroZoo usually settle on a threshold for the number of antivirus flags as a way to decide whether an app should be considered a malware or not. A standard threshold is 2: any app with at least 2 antivirus flags is considered a malware (Arp *et al.*, 2014). However, the literature also includes papers using thresholds such as 1 (Li *et al.*, 2017), 28 (Li *et al.*, 2017), 8 (Li *et al.*, 2016), or half the antiviruses (Wei *et al.*, 2017). Hence, there is no firm consensus on the threshold of antivirus flags starting from which an app in the AndroZoo dataset should be considered a malware.

In any case, this is an important question, with which we struggled as malware researchers and which the current paper investigates through various setups of our dataset, which we extended, since (Namrud *et al.*, 2019b), with confirmed Android malwares from VirusShare, a prominent repository of malware samples. In particular, the addition of

malwares from VirusShare serves as an interesting addition in the way of testing the effectiveness of classifiers built using our dataset and different malware thresholds.

Moreover, we take interest in assessing possible performance differences between well-known machine learning techniques (e.g., JRIP, Naive Bayes, MLP, and J48). We also included feature selection options for a finer granularity of analysis as this — sometimes overlooked — step has proven to be a significant addition in some studies (Bhattacharya & Goswami, 2018). In short, while our first effort focused on proposing AndroVul and investigating the input data (the various vulnerability metrics), the current study expands the previous investigation on two fronts: i) the labeling (as benign or malware) of Androzoo apps; and ii) the effectiveness of the treatment (with machine learning) of the input.

The current paper aims to be self-contained and as such, it includes elements from (Namrud *et al.*, 2019b). The rest of the paper is organized as follows: Sections 3.3 and 3.4 propose relevant background notions and related work on app datasets and malware classification. Section 3.5 and Section 3.6 present the tool and datasets in AndroVul. In Section 3.7, we lay out our study from preliminary statistical analyses to the definition of research questions and investigations aiming at answering them. Section 3.8 presents and discusses our results and findings. We then discuss some of the limitations and threats to validity to the present work in Section 3.9. Finally, Section 3.10 concludes the paper and outlines some future work.

3.3 Background

In this section, we briefly present the possible security vulnerabilities our dataset focuses on: dangerous permissions (as defined by the Android system), troubling code attributes (as collected by AndroBugs) and security code smells (as proposed in (Gadient *et al.*, 2017)).

3.3.1 Vulnerability

A vulnerability is a soft spot in a system, that places it in danger of an attack by hackers, viruses or any other event that will lead to breaking the security of any system (Mansourov & Campara, 2010). Mansourov and Campara (Mansourov & Campara, 2010) define it as "a certain unit of knowledge about a fault in a system that allows exploiting this system in unauthorized and possibly even malicious ways". A fault in a system can be triggered by several factors such as human error, poor specifications of requirements, the use of processes that are poorly developed, the use of technologies that evolve rapidly, or even the poor understanding of threats. Malicious software also known as malware are a means to exploit faults in a system. Malware are usually referred to as viruses, worms, trojan horses, backdoors, keystroke loggers, rootkits, or spyware.

3.3.2 Dangerous Permissions

A key security mechanism of Android is its permission system, which controls the privileges of apps, that apps must request specific permissions in order to perform specific functions. This mechanism requires that app developers declare which sensitive resources will be used by their apps. App users have to agree with the requests when installing or using the apps. Android defines several categories of permissions, among which"dangerous" ones, deemed more critical and privacy sensitive since they provide access to system features such as the camera, Internet, personal contacts, SMS, etc. Table 3.10 in the appendix proposes a list of the various dangerous permissions as defined by the official Android developer resource(Developer.android.com, 2021b).

3.3.3 AndroBugs

Androbugs is a popular security testing tool that checks Android apps for vulnerabilities and potentially critical security issues. The tool reverse engineers APKs and looks for various issues, from failures to adhere to best practices to the use of dangerous shell

commands or exposure to vulnerabilities from third party libraries. AndroBugs has a proven track record of discovering security vulnerabilities in some of the most popular apps or SDKs. It is a command line tool that issues reports with four severity levels: Critical Confirmed vulnerability that should be solved (except for testing code), Warning possible vulnerability that should be checked by developers, Notice Low priority issue and Info.

3.3.4 Security Code Smells

“Code smells” refer to code source elements that may indicate deeper problems (Shezan *et al.*, 2017). In (Gadient *et al.*, 2017), Ghafari et al. introduced security code smells in Android apps as "symptoms in the code that signal the prospect of a security vulnerability". After reviewing the literature, they identified 28 security code smells (Gadient *et al.*, 2017) that they regrouped into five categories, such as Insufficient Attack Protection, Security Invalidation, Broken Access Control, Sensitive Data Exposure, and Lax Input Validation.

3.4 Related Work

3.4.1 Dataset

Dataset availability is a key issue when it comes to getting insights about a topic or evaluating approaches or hypotheses. We briefly present below some of the most notable efforts related to this issue in the context of Android apps and more specifically their possible security concerns.

A number of repositories have been proposed over the years for the study of mobile apps. F-Droid(F-droid, 2018) is such an effort; it is a repository of free open source Android apps that have been used in an impressive number of studies. Recently, Allix et al. (Allix *et al.*, 2016) have proposed and continued to maintain AndroZoo, certainly the largest Android app repository, with millions of apps (and APKs) from the Google Play store and

other third party markets. Even more recently, Geiger et al. (Geiger *et al.*, 2018) made available a graph–based database with information (metadata, commit and code history) on 8,431 open-source Android apps available on GitHub and the Google Play Store. Also notable, although slightly older, is Krutz et al. (Krutz *et al.*, 2015), with a public dataset centered on the lifecycle of 1,179 Android apps from F-Droid. Complementary to these research initiatives, there are a number of websites such as AppAnnie and Koodous that gather Android apps and perform various types of analyses, including downloads over time and advertising analytics.

When it comes to security aspects, there have been a number of papers investigating large numbers of Android apps but few propose publicly available data-sets. Among those, we can citep Munaiah et al. (Munaiah *et al.*, 2016) which propose data (e.g., app category, permissions) on reverse engineered benign applications from Google Play store and malware applications from several sources.

Our dataset on vulnerabilities of Android apps shares some similarity with the work of Gkortzis et al. (Gkortzis *et al.*, 2018), which also proposes a dataset of security vulnerabilities but for open source systems (8,694). Similar to (Krutz *et al.*, 2015), we propose a subset of a well-known mobile app repository; we start with AndroZoo while (Krutz *et al.*, 2015) builds on F-Droid. Similar to (Krutz *et al.*, 2015), we also propose tool that interface with well-known reverse engineering and static analysis tools, but we do so with a focus on security vulnerabilities and use a different set of tools. Overall, our dataset and tool propose a unique offering for Android security researchers.

As touched upon in the introduction, malware research using AndroZoo involve deciding on which of the apps present in the benchmark can be considered malwares. Different approaches are present in the literature. They fall mostly into two categories based on whether they rely on a preselected subset of antivirus results from VirusTotal or a given number of antivirus flags as reported by AndroZoo data.

Zheng et al. (Zheng *et al.*, 2012) focused on the top ten anti-virus products from VirusTotal that flagged apps as malware. Their results were subsequently used by Shen et al. (Shen *et al.*, 2014) to evaluate the effectiveness of antivirus tools against malware obfuscation. In (Yousefi-Azar *et al.*, 2018), Yousefi-Azar et al. considered nineteen of the most well-known antiviruses, including Kaspersky, Symantec, Avast, McAfee, AVG, Malwarebytes, etc.

Also notable is the work of Ma et al. (Ma *et al.*, 2019), which considered, as malwares, apps that were flagged by four well-established anti-viruses (i.e. McAfee, 360 Security Guard, Kingsoft Antivirus, Norton). Some other approaches based their analysis on the number of antivirus that flagged an app as possible malware. Li et al. (Li *et al.*, 2017) considered that even one flag was enough to classify an app as malware. Other studies were more lenient, with Li et al. (Li *et al.*, 2016) needing 8 antivirus flags before deciding that an app is a malware, and Wei et al. (Wei *et al.*, 2017) needing flags from at least 50% of the anti-viruses of VirusTotal before recognizing an app as a malware. Different from these works, our paper attempts to evaluate the impact of different thresholds for malware labelling in a machine learning context.

3.4.2 Malware Classification with machine learning

A primary intended use of our dataset will be as input for malware detection techniques. Malware detection approaches generally use some form of machine learning to classify candidate apps as malicious or not. The existing literature is quite extensive on that subject. In this section, we will focus on the work that is the most recent and the closest to our experiments.

Permissions, especially dangerous ones, have been used in lots of studies as inputs to various classification approaches. A particular recent work of interest is the one of Bhattacharya et al. (Bhattacharya & Goswami, 2017), which proposed a framework that gathered permissions from apps' manifest files and applied advanced feature selection techniques. The features obtained from such process were organised into four groups

and used as input for fifteen different machine learning classifiers (including JRip, J48, MPL, and NB) from Weka. The authors evaluated their approach on a sample of 170 apps and reported the highest accuracy to be 77.13%. Many other research work investigated features other than permissions for malware detection purposes. For instance, Sharma et al. (Sharma & Sahay, 2018) tried to leverage Dalvik[2] opcode occurrences for malware classification purposes. They selected 5531 android malwares from the DREBIN repository (Arp *et al.*, 2014) and 2691 benign apps from the Google Play Store. They applied different machine learning techniques and reported the best detection accuracy obtained to be 79.27%. Also of interest is the work of Sachdeva et al. (Sharma & Sahay, 2018), which focused on features extracted through Mobile Security Framework, an open source tool dedicated to mobile app security(Abraham, 2020). The approach proposed in (Sharma & Sahay, 2018) aimed at classifying apps with respect to three levels (Safe, Suspicious, Highly Suspicious). The reported experiments involved many refined machine learning classifiers applied on a corpus of 13,850 Android apps, with accuracy results up to 81.80% when considering the three proposed levels and up to 93.63% when considering a binary benign / malicious decision.

DroidDeepLearner is a weighted malware detection technique proposed by (Li *et al.*, 2018b). The method employs both dangerous API calls and risky permission combinations as features in order to build a Deep Belief Network model capable of automatically distinguishing malware from benign ones. Their method achieves over 90% accuracy with 237 features on the Drebin dataset, according to the findings.

Authors (Lee *et al.*, 2020) investigated whether the dangerous permissions are a key component of detection when determining whether an app is malicious or benign. They used a total of 10,818 malicious and benign apps. To determine the accuracy of the detection, they used four separate deep learning algorithms and measured them using the confusion matrix. The selected features resulted in about 90% accuracy. We are different then both (Li *et al.*, 2018b; Lee *et al.*, 2020), they focus only in dangerous permissions

[2] Dalvik is a now discontinued process virtual machine in Google's Android OS

while we investigated the use of vulnerabilities in different levels including dangerous permissions.

3.5 AndroVul-T: the tool

Our repository proposes a tool that allows, given a directory of APKs, the automatic generation of a CSV file with information on the apps vulnerabilities. To accommodate statistical analysis, each vulnerability corresponds to a column, to which we attach some quantitative data indicating its presence (for dangerous permissions), the certainty behind it (for AndroBugs vulnerabilities), or its weight (for security code smells).

Figure 5.1 proposes an overview of the inner workings of our tool, which makes use of very well-known tools to reverse engineer any APK. The tool APKtool[3] is applied on a given APK to reverse engineer its manifest file and Smali code, which is basically a human readable description of the binary code (contained in a .dex file). Similarly, via our tool, an APK can be given to the AndroBugs tool in order to generate a report on its potential security vulnerabilities. Once we get these three artefacts from AndroBugs and

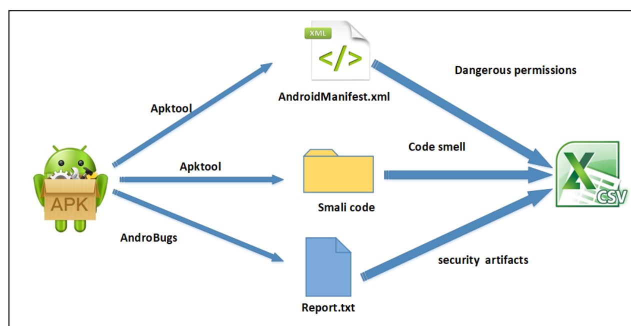


Figure 3.1 Overview of the AndroVul tool

Apktool, our tool proceeds on parsing dangerous permissions from the Android Manifest, various vulnerabilities from AndroBugs and security code smells from the Smali code. Our

[3] https://ibotpeaches.github.io/Apktool/

treatment of the extracted information is illustrated in Figure 3.2 and further detailed in the subsections below.

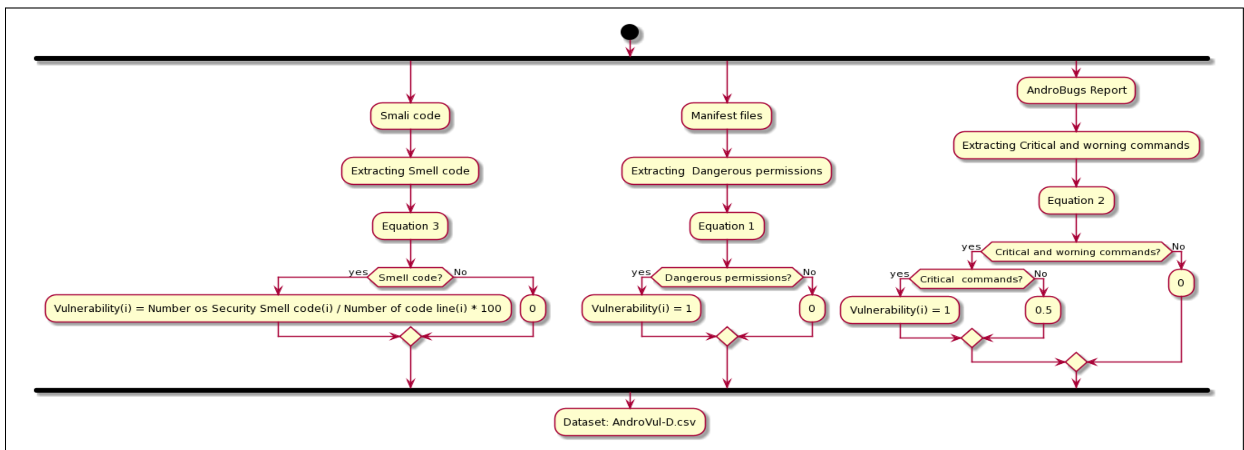


Figure 3.2 Parsing and Quantifying vulnerability data (i refers to the specific vulnerability)

3.5.1 Dangerous permissions extraction

Extracting dangerous permissions is a relatively straightforward process, after which we fill in the csv file information about the presence (1) or absence (0) of any dangerous permission. This is summed up in Equation 3.1, with V_i standing for the inherent vulnerability coming with the granting of a given dangerous permission i[4].

$$V_{i=1..24} = \left\{ \begin{array}{ll} 1 & \text{if } the\, permission\, is\, requested \\ 0 & otherwise \end{array} \right\} \tag{3.1}$$

3.5.2 AndroBugs extraction

As for the AndroBugs report, we parse it to extract vulnerabilities tagged Critical or Warning (see Section 3.3.3). To quantify the collected information for every vulnerabil-

[4] i is the index assigned to a given possible vulnerability in our dataset of vulnerabilities

ity(V), we give a weight of 1 to Critical, 0.5 to Warning, and 0 otherwise. This is summed up in Equation 3.2, with V_i standing for Critical or Warning-level security vulnerabilities in our dataset.

$$V_{i=1..41} = \left\{ \begin{array}{cc} 1 & \text{if } V_i is present and Critical \\ 0.5 & \text{if } V_i is present and a Warning \\ 0 & \text{if } V_i \; is \; not \; present \end{array} \right\} \tag{3.2}$$

3.5.3 Code Smell extraction

We used regular expressions to parse the Smali code and extract security code smells defined in (Gadient *et al.*, 2017) and used successfully in (Habchi *et al.*, 2019; Gadient *et al.*, 2019). After which, we compute (see Equation 3.3) for each vulnerability posed by a security code smell, a ratio indicating the relative presence of that vulnerability; said ratio is obtained by dividing the number of identified instances of the code smell (NS_i) by the number of lines of code in the Smali format (LOC_{SMALI}).

$$V_{i=1to9} = \frac{NS_i}{LOC_{SMALI}} * 100 \tag{3.3}$$

3.6 AndroVul-D: the dataset

Androvul-D is our dataset of 78 vulnerability metrics collected on a sample of Android apps from AndroZoo. Figure 3.4 illustrates the process through which AndroVul-D was generated and can serve as a blueprint for other researchers willing to generate vulnerability datasets for their own sample of AndroZoo apps. The figure starts with a researcher (carefully) selecting the Android apps she wants in her study or preliminary tests, and follows up with the application of the scripts of AndroVul-T to generate csv files filled with vulnerability metrics about each app of the dataset. Furthermore, since

AndroZoo does not have all the information related to the apps it archived, the researcher may, as we did, have to go fetch some metadata (e.g., category) about an app from its store. Additionally, a researcher may have to add known malwares from other sources.

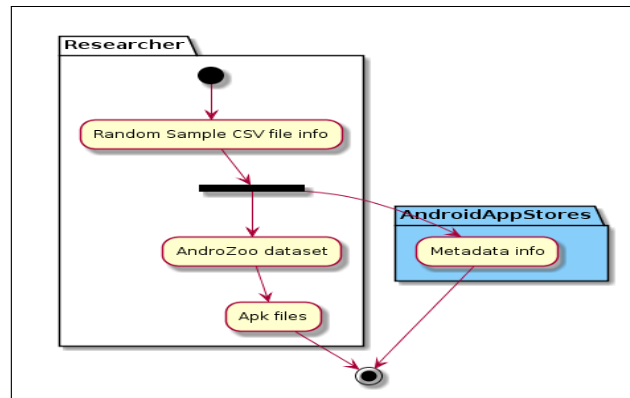


Figure 3.3 Data Selection and Gathering

3.6.1 Data Selection

For our data selection, we resorted to the AndroZoo data-set which contained 5,848,157 apps when we started our investigation. The AndroZoo dataset proposes data on the APKs it archived in a main CSV file containing important information for each application, including hash keys (such as sha256, sha1, md5), size information (for APKs and DEX), date of the binary, package name, version code, market place as well as information about how well the app fared on the VirusTotal website (number of antiviruses that flag the app as a malware, scan date). Using a sample size calculator[5], we computed that to get a representative sample with very high confidence level (99%) and confidence interval (1%), we ought to consider 16,586 apps. After removing duplicates and some entries that are not actual apps, we ended up with 16,180 APKs that were downloaded and used as input for the AndroVul-T scripts.

[5] https://www.surveysystem.com/sscalc.htm

To complement the above data, we resorted to data from VirusShare(Forensics, 2021) as a way to obtain malware data that has been validated as such. VirusShare is a website that collects virus data, whether from desktop or mobile software, from a variety of sources (Zhu *et al.*, 2018b). The data is offered in big archive files with malwares for desktop or mobile environments. We used two uploaded archive files (dating from 2019-08-08 & 2019-06-02), which comprises around 127K desktop or mobile apps, of which we were able to recover 3,978 Android APK files. These files come with no information other than a hash for file integrity.

3.6.2 Dataset Structure

The dataset we propose consists of CSV files containing information (as illustrated in Figure 3.4) about the 16,180 apps from AndroZoo and the 3,978 apps from VirusShare (Forensics, 2021), one app per line. There are 78 columns in the file, each with a header clearly indicating the information it provides. There are four types of information in the CSV:

1. Information from the AndroZoo dataset, if applicable[6], as described in Section 3.6.1
2. The nine (9) code smells extracted from the reverse engineered Smali code (see Table 3.11 in the Appendix)
3. The twenty-four (24) dangerous permissions, as parsed from the app's manifest file
4. The forty (40) metrics derived from the six types of vulnerabilities provided by AndroBugs

Overall, the file contains 78 metrics about info from AndroZoo, dangerous permissions, Smali code smells and AndroBugs-tagged vulnerabilities.

---

[6] Malicious apps from VirusShare do not have any info other than a hash.

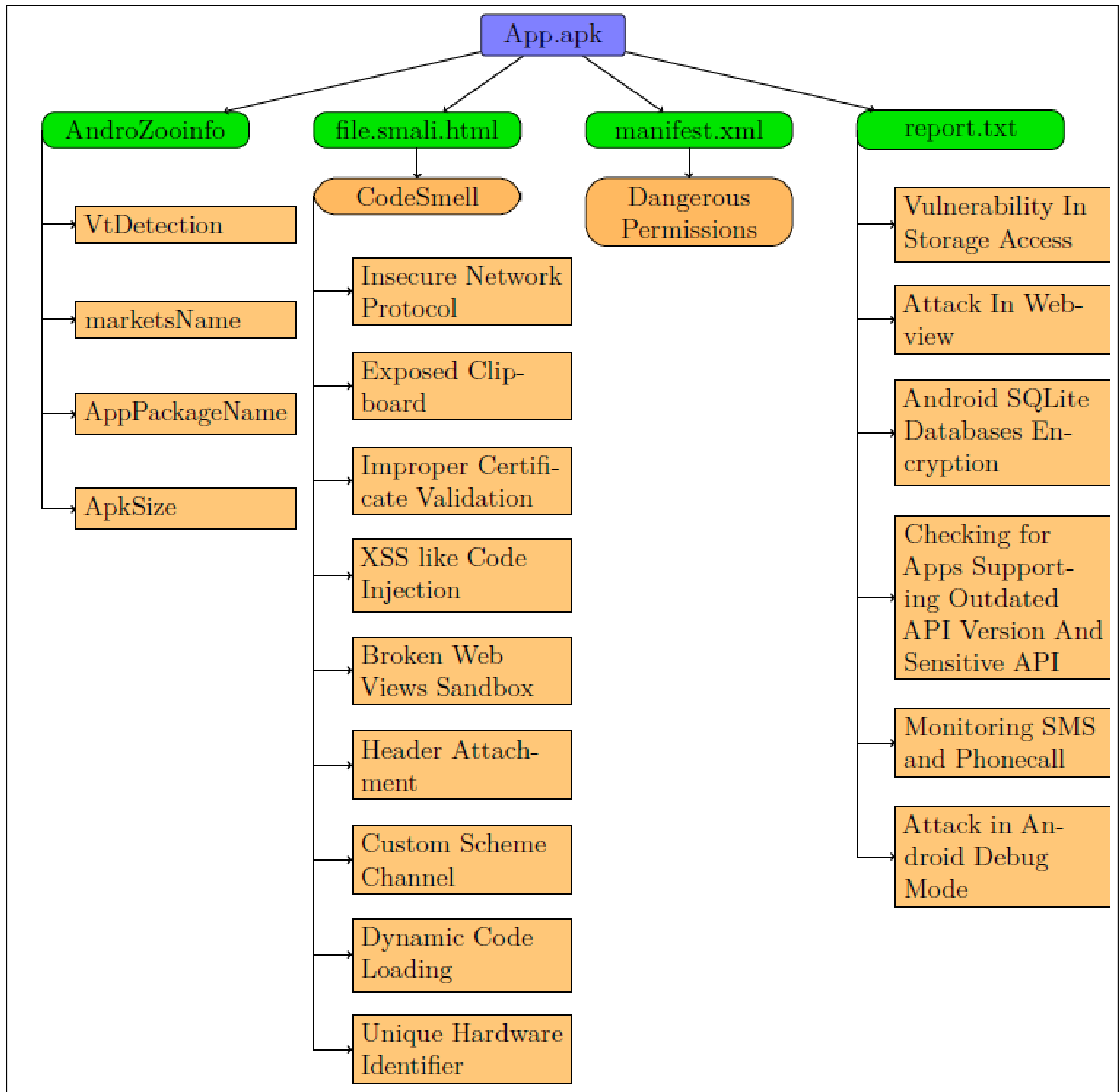


Figure 3.4 Information and vulnerabilities extracted from an Android app

3.6.3 Dataset description

Apps from the AndroZoo dataset: here, we provide some descriptive statistics on our data-sets, relatively to the date, the category, the store, APK size and number of antivirus flags. With respect to the binary dates, 3.37% of the apps display an unreliable date (1980). About 1 out of 4 apps are from 2016 to 2018, 2 out of 3 apps are from 2014 to 2018.

The APK sizes range from 7 KB to 330 MB, with an average of 9 MB and a standard deviation of 12 MB. Marketplace–wise, the most dominant stores are the Google Play Store (74%), appchina (10%), mi.com (2%) and anzhi (1%). When it comes to information from the antiviruses of VirusTotal(virustotal, 2021), the apps in the dataset have between 0 (74% of the apps) and 40 flags, out of the 63 antiviruses; the average is 2 flags and the standard deviation is 5.11. We also collected data related to the apps' categories. A sizable part of the apps (about 43%) could not be mapped to a category, mainly because they are no longer available on the market stores. Another 14% of the apps are only available in Chinese markets. Overall, we could find the category information for only 43% of the apps. Table 3.1 lists all the categories that make for at least 1% of the dataset.

Apps from the VirusShare dataset: there are no additional information coming with the APKs in this dataset. We took interest in getting an estimate about how many of these malicious apps would be classified as such, using a threshold of 3 antivirus flags from VirusTotal, threshold that we used in our previous work (Namrud *et al.*, 2019b). There were issues in automating the scanning process from VirusTotal, so we proceeded with a small random sample of 94 apps, giving us a confidence level of 95% with an interval of 20% (as computed from https://www.surveysystem.com/sscalc.htm). We found that 72% of these apps were flagged three times or more (This means we can be 95% confident that between 52% and 92% of the malicious Android apps from VirusShare(Forensics, 2021) have 3 or more antivirus flags from VirusTotal).

3.7 Study Design

In this section, we lay out the design of our study, from preliminary sanity checks to research questions and experimental design.

Table 3.1 App categories in our dataset

CATEGORY	Proportion
GAME	1231 (7.61%)
EDUCATION	571 (3.53%)
LIFESTYLE	540 (3.34%)
BUSINESS	437 (2.70%)
ENTERTAINMENT	423 (2.62%)
TOOLS	416 (2.57%)
PERSONALIZATION	397 (2.45%)
BOOKS AND REFERENCE	373 (2.31%)
TRAVEL AND LOCAL	307 (1.90%)
MUSIC AND AUDIO	282 (1.74%)
NEWS AND MAGAZINES	254 (1.57%)
PRODUCTIVITY	245 (1.52%)
HEALTH AND FITNESS	211 (1.31%)
FINANCE	193 (1.19%)
COMMUNICATION	169 (1.05%)
SPORTS	165 (1.02%)
SOCIAL	161 (1.00%)

3.7.1 Identifying malwares

A first point to be addressed is the identification of malwares in the AndroZoo dataset. Androzoo does not explicitly tag apps as malwares. Rather, it provides the number of antiviruses from VirusTotal that flagged the app, with malware researchers left to decide which number of antivirus flags is enough to label an app from AndroZoo as a malware. In addition to AndroZoo, we gathered malware apps from the VirusShare (Forensics, 2021) repository to strengthen the generalisability of our experiments.

Overall, we took into account three malware datasets:

1. AZM: AndroZoo apps flagged as possible malwares.

2. VSM: malware dataset from VirusShare.

3. MM: dataset mixing malwares from AndroZoo and VirusShare.

3.7.2 Correlation Analysis on the Androzoo data

To get a quick initial sense of how much the collected metrics can contribute to malware detection, we first proceeded to some statistical correlation analysis of the metrics to: i) the number of antivirus flags; and ii) a binary value representing the benign / malicious classification: 0 for benign or 1 for malware (with the threshold of 3 flags used in our previous work (Namrud *et al.*, 2019b) to label an app as a malware). We computed Pearson correlations for all metrics and found some interesting values in all 3 categories:

1. for permissions, *READ_PHONE_STATE* returns the highest correlations with respectively 0.35 for the number of antivirus flags and 0.38 for a benign/malware decision;

2. for code smells, the *Dynamic Code Loading* metric yields 0.4 for the number of flags and 0.38 for the binary decision;

3. and for Androbugs, the vulnerability *Using critical function* returns 0.34 (number of flags) and 0.31 (binary decision) as correlation values.

All three metrics mentioned above returned p-values significantly lower than 0.05 (the commonly accepted statistical significance threshold), as is the case for all but a few metrics in the dataset.

3.7.3 Used Classifiers

We selected four classifiers representing four types of machine learning algorithms commonly used in the Android malware research community. More precisely, we used the well known machine learning software Weka and selected NaiveBayes (NB) from its *bayes* category, MLP classifier from its *function* category, JRip from its *rules* category, and J48

from its *tree* category as shown in table 3.2. Using these classifiers, we proceeded to the commonly used statistical method that is the K-fold cross-validation. In short, it consists in splitting, after random shuffling, the dataset in K groups; after which, each group is used as a test group while the other K-1 groups are used for training. More specifically, we chose, in accordance to many similar studies (e.g., (Bhattacharya & Goswami, 2017)), K = 10 for a 10-fold cross validation study, in which 90% of the data is used for training and 10% for testing (prediction).

Table 3.2 Selected one Classifies form each four known Machine learning categories

Classifier	**Category**
MLP	Function
NaiveBayes	Bayes
JRip	Rules
J48	Tree

3.7.4 Feature selection

The features (vulnerability metrics in our case) extracted from our data constitute a relatively large set that is likely to contain some duplication. To tackle this, along with reducing risks of overfitting, feature selection is to be considered. It allows identifying the best features and excluding the least important features. To do so, it generally relies on assessing the information gained or lost by adding or removing a particular feature. Various techniques have been proposed and implemented in tools for this purpose. In this work, we relied on the well-established open source machine learning software Weka and selected the following three attribute evaluators: ChiSquaredAttributeEval (CS), InfoGainAttributeEval (IG), and ReliefFAttributeEval (RF).

3.7.5 Performance Indicators

The application of classifier results in decisions about individual apps that can be quantitatively evaluated through various measures. As it relates to the detection of malwares, we refer to True Positive (TP) as the number of malwares actually classified as such, True Negative (TN) as the number of benign apps classified as such, False Positive (FP) as the number of benign apps wrongly classified as malwares and finally False Negative (FN) the number of malwares wrongly classified as benign. From these basic measures are derived more insightful measures, commonly used in malware detection research work, such as:

1. **Precision:** It is the ratio of actual malwares in the set of apps classified as such: TP/(TP+FP)
2. **Recall:** It is the ratio of malwares that were detected as such: TP/(TP+FN)
3. **Accuracy:** It is the percentage of correctly classified apps: (TP+TN)/(TP+TN+FP+FN).
4. **F1-Measure:** It is a performance indicator that takes into account both precision and recall of the obtained classification: 2 * (Recall * Precision) / (Recall + Precision).
5. **Area under ROC Curve (AUC):** It is a measure of the predictive power of the classifier that basically informs on how much the model is capable of distinguishing between classes (here benign apps vs malwares).

For all these measures, the higher, the better, with 1 being the perfect value.

3.7.6 Research Questions

Our research questions flow from the above considerations (as laid out in the previous subsections) and aim at answering the following research questions:

1. **RQ1**: Which classifiers and feature selection techniques perform the best?

2. **RQ2**: Relatively to Androzoo, which subset of apps should be labeled as malwares? More specifically, how many antivirus flags from VirusTotal are enough to label an app from the AndroZoo dataset as a malware.

To answer these questions, we propose the following experiments based on different slicing of the AndroVul data.

3.7.7 Answering the research Questions

Our research questions are designed around two key elements pertaining to malware classification: i) the input (the "best" dataset to use) and ii) the treatment (the "best" machine learning technique to choose).

The first point is an important one when it comes to using AndroZoo for malware classification. Given that AndroZoo only provides the number of antivirus flags (from VirusTotal) for a given app, researchers typically have to decide which threshold to use to consider a given app malicious or benign. The choice of a threshold is somewhat arbitrary and rarely motivated so, in the following, we propose experiments to probe the choice of a good threshold through the lens of its contribution to effective malware classification, especially when it comes to classifying correctly confirmed malwares (elements from VirusShare). We mainly focus on a single threshold below which apps are considered benign: thresholds of 1, 2, 3, 5, 10 and 20 but also consider one experiment with two thresholds: i) benign apps being those apps with 1 or zero antivirus flags, and ii) malware apps being those apps with 10 or more flags.

The choices outlined above delineate subsets of our benchmark that ought to be clarified. Taking as an example the threshold 1, meaning apps with one or more flags are considered as malwares, we define three (3) malware datasets:

1. AZM_1: the set of AndroZoo apps with 1 or more antivirus flags,

2. VSM: the set of malwares from VirusShare (Forensics, 2021), and

3. MM_1: the mixed set of apps from AZM_1 and VSM.

In single threshold experiments, the choice of a threshold also defines which apps from AndroZoo should be considered benign; in this case, these are the apps with 0 antivirus flags: AZB_0. This means that, with a threshold of 1 for malware decision, the complete datasets we use as input for malware classification are the following:

1. AZB_0 (benign apps) $\cup$ AZM_1 (malicious apps),
2. AZB_0 (benign apps) $\cup$ VSM (malicious apps), and
3. AZB_0 (benign apps) $\cup$ MM_1 (malicious apps).

Therefore, our experiments explore respectively the antivirus flag thresholds 1 (AZB_0 and AZM_1), 2 (AZB_1 and AZM_2), 3 (AZB_2 and AZM_3), 5 (AZB_4 and AZM_5), 10 (AZB_9 and AZM_{10}), and 20 (AZB_{19} and AZM_{20}) as well as a double threshold 1 and 10 (AZB_1 and AZM_{10}) [7]

Throughout these experiments, we computed and analyzed the effectiveness of our selected classifiers (JRIP, NB, MPL, and J48) and feature selection techniques (CS, IG, and RF). To assess the results of the experiments, we relied mostly on the F1-score and the AUC measure, which are standard metrics recognized as more robust than, for instance, the accuracy measure.

The answers to our two research questions come from the analysis and comparisons of these numbers intra-experiment (RQ1: best classifiers and feature selection techniques) and inter-experiment (RQ2: best input data).

[7] Some of these configurations resulted in imbalanced data; so we used a SpreadSubsample instance filter as an imbalance reduction technique.

3.8 Experiments and Results

The experiments defined in Section 3.7.7 are summarised in Table 3.3, which display, for each experiment (Exp), data about its AndroZoo benign apps (antivirus flags and data size), its AndroZoo malwares (antivirus flags and data size), its VirusShare malwares (data size), and the combined set of AndroZoo and VirusShare malwares (data size). The following sections present and discuss the performance metrics for AUC & F1 obtained from these experiments.

Table 3.3 Information about the datasets' sizes and the selected thresholds for all experiments

	AZ Benign	AZ Malware	VS Malware	All Malware
	flag| Size	**flag**|Size	Size	Size
Exp 0	**0**|11,928	**1+** | 4,201	3,978	8,179
Exp 1	**0 - 1** | 13,086	**2+** | 3,042	3,978	7,020
Exp 2	**0 - 2** | 13,553	**3+** | 2,621	3,978	6,599
Exp 3	**0 - 4** | 13,933	**5+** | 2,195	3,978	6,173
Exp 4	**0 - 9** | 14,713	**10+** | 1,397	3,978	5,375
Exp 5	**0 - 19** | 15,722	**20+** | 406	3,978	4,487
Exp 6	**0 - 1** | 13,086	**10+** | 1,397	3,978	5,375

3.8.1 Results

Table 3.4 AUC and F1 results when considering only AndroZoo apps (Benign & Malicious)

AUC	*JRIP*	*NB*	*MPL*	*J48*
Exp 0	0.81	0.77	0.81	**0.85**
Exp 1	0.84	0.82	**0.86**	0.83
Exp 2	0.85	0.83	**0.87**	0.83
Exp 3	0.84	0.85	**0.89**	0.83
Exp 4	0.86	0.86	**0.89**	0.82
Exp 5	0.85	0.91	**0.92**	0.85
Exp 6	0.89	0.90	**0.94**	0.88

F1	*JRIP*	*NB*	*MPL*	*J48*
Exp 0	0.78	0.72	0.77	**0.80**
Exp 1	**0.82**	0.76	0.80	**0.82**
Exp 2	0.81	0.76	0.81	**0.82**
Exp 3	**0.83**	0.78	**0.83**	**0.83**
Exp 4	**0.83**	0.80	**0.83**	0.82
Exp 5	0.82	0.83	**0.85**	**0.85**
Exp 6	0.86	0.82	**0.88**	0.87

Table 3.5 AUC and F1 results when considering Benign apps from (AndroZoo & Malicious apps from VirusShare)

AUC	*JRIP*	*NB*	*MPL*	*J48*
Exp 0	0.92	0.88	**0.94**	0.90
Exp 1	0.92	0.88	**0.93**	0.90
Exp 2	**0.92**	0.87	**0.92**	0.90
Exp 3	0.91	0.89	**0.93**	0.89
Exp 4	0.91	0.86	**0.92**	0.88
Exp 5	**0.92**	0.85	0.91	0.88
Exp 6	0.92	0.88	**0.93**	0.91

F1	*JRIP*	*NB*	*MPL*	*J48*
Exp 0	**0.90**	0.77	0.88	0.87
Exp 1	**0.88**	0.77	**0.88**	0.86
Exp 2	**0.88**	0.74	**0.88**	0.86
Exp 3	**0.88**	0.85	0.87	0.85
Exp 4	**0.88**	0.74	0.86	0.85
Exp 5	**0.88**	0.71	0.86	0.79
Exp 6	**0.89**	0.77	0.87	0.86

Table 3.6 AUC and F1 results when considering Benign apps from AndroZoo & Malicious apps from VirusShare and Androzoo

AUC	*JRIP*	*NB*	*MPL*	*J48*
Exp 0	0.85	0.81	**0.87**	0.82
Exp 1	0.86	0.83	**0.89**	0.76
Exp 2	0.86	0.83	**0.90**	0.79
Exp 3	0.87	0.84	**0.90**	0.86
Exp 4	0.87	0.84	**0.90**	0.85
Exp 5	0.89	0.85	**0.91**	0.75
Exp 6	0.89	0.87	**0.92**	0.87

F1	*JRIP*	*NB*	*MPL*	*J48*
Exp 0	**0.81**	0.69	0.80	0.75
Exp 1	0.82	0.72	**0.83**	0.79
Exp 2	0.82	0.71	**0.83**	0.79
Exp 3	0.83	0.72	**0.83**	0.82
Exp 4	**0.84**	0.72	0.83	0.82
Exp 5	**0.85**	0.71	**0.85**	0.79
Exp 6	**0.86**	0.76	**0.86**	0.84

Tables 3.4, 3.5, and 3.6 present the results of all experiments with no feature selection. A first very clear output of our experiments is that the various feature selection techniques only very marginally affected the results. We thus decided, for simplicity's sake, against displaying them in the summary tables.

Table 3.4 presents the experiments using **AndroZoo data only**. Below are the main observations we can draw from it.

1. Experiment 0 posts the worst results (AUC: 0.77 - 0.85, F1: 0.72 - 0.80)

2. Experiments 1 & 2 (AUC: 0.82 - 0.87, F1: 0.76 - 0.82) propose very similar results, with values within 0.01 from one experiment to the other, suggesting unsurprisingly

that there is not much difference between using two or three as the number of anti-virus flags needed to label an app as a malware.

3. The same goes for Experiments 3 and 4 (AUC: 0.82-0.89, F1: 0.78-0.83), which provide results at most 0.02 from one another. Additionally, aside from the classifier NB, results from experiments 3 and 4 are relatively close to those from Experiments 1 & 2.

4. Experiment 5 (AUC: 0.85 - 0.92, F1: 0.82 - 0.85) and Experiment 6 (AUC: 0.88 - 0.94, F1: 0.82 - 0.88) provide results that are distinct but suggest that higher malware thresholds translate into better results for the classifiers. It should be noted, however, for Experiment 6, which proposes the best performance measures that it leaves out a lot of apps (any app having between 3 and 15 flags)

5. Overall, when it comes to classifier performance, MPL is the leading classifier, with J48 coming in second.

Table 3.5 presents the experiments using **AndroZoo data for benign apps and VirusShare for malwares**. The main insights are as follows:

1. Results are generally better than those involving only AndroZoo apps and around 0.90 in most cases, indicating the effectiveness of the vulnerability metrics in correctly classifying VirusShare malwares.

2. MPL and JRIP are the leading classifiers and exhibit remarkable consistency over the range of experiments, with their results always within 0.03 from one experiment to the other. Similar (but somewhat lower) consistency can be observed with J48, whereas the classifier NB proposes results that fall into a broader and generally lower range, depending on the experiment, and especially for the F1 measure (0.71-0.85).

3. There's no trend in the effectiveness of the classifiers as the malware thresholds get higher, although the results from Experiment 0 (one antivirus flag is enough to classify an AndroZoo app as a malware) are almost always the best ones.

Table 3.6 presents the experiments using **only AndroZoo data for benign and AndroZoo + VirusShare for malwares**. The results propose a pattern very similar to that of Table 3.4 (AndroZoo data only): worst results for Experiment 0, Experiments 1 & 2 as well as 3 & 4 being very close, better results as malware thresholds rise, etc.

3.8.2 Analysis and discussion of the experiments

MPL is clearly the leading classifier. It was the best (or close) classifier in all the experiments. Furthermore, looking beyond the AUC and F1 measures, it provided balanced performance for precision and recall, with their values almost always above 0.80 (and mostly above 0.85) and within 0.02 from one another meaning that the classifier provides a classification with roughly equal precision and recall. JRIP is a close second; while it visibly trails MPL on the AUC measure, it is as good as (and possibly slightly better than) MPL on the F1 measure. The worst classifier is consistently the Naive Bayes (NB) one.

Table 3.7 compares the results we obtained against those from recent papers. To represent our experiments, we used the average of values obtained for the experiments involving benign AndroZoo apps and malwares from VirusShare (Table 3.5). The table shows that the results we obtained compare mostly favorably to the ones from previous studies.

Table 3.7 Comparison with related work

Papers	Tool	Accuracy	F1	AUC
Bhattacharya et al. (Bhattacharya & Goswami, 2017)	Weka	0.77	0.86	0.82
Sharma et al. (Sharma & Sahay, 2018)	Weka	0.79	/	/
Sachdeva et al. (Sachdeva *et al.*, 2018)	Weka	0.93	/	/
AndroVul	Weka	0.92	0.87	0.92

Figures 3.5 and 3.6 summarize the findings from RQ 1 and findings from RQ 2.

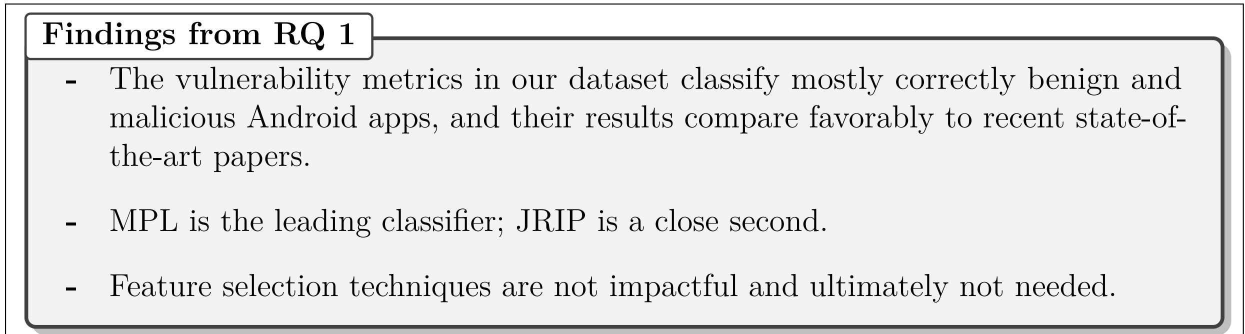


Figure 3.5 Findings from RQ 1

As for the best dataset, the emerging picture is mixed: the strictest approach (only apps with no antivirus flags are considered benign) performed slightly better on VirusShare dataset (confirmed malwares) but marginally worse on dataset configurations involving AndroZoo apps labeled as malwares. On the other hand, the laxer approaches (highest thresholds) performed equally or slightly better than mid-range thresholds. Still, the main takeaway is that the differences between the results obtained from these thresholds are small, suggesting that the choice of the threshold may not matter that much.

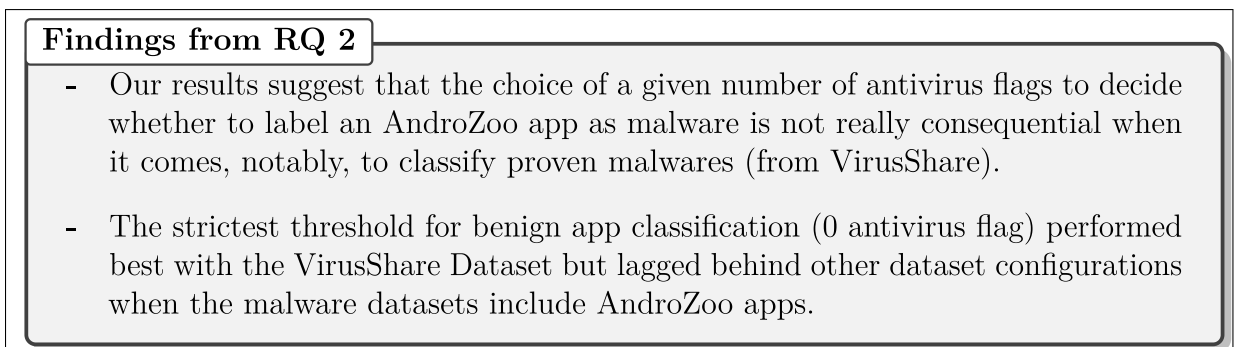


Figure 3.6 Findings from RQ 2

3.9 Limitations and Threats to validity

The present paper extends our previous paper (Namrud *et al.*, 2019b) and set out to provide a more complete benchmark for malware classification, along with empirical data

about the effectiveness of different classifiers and the impact of possible data labeling decisions. As any work, it comes with limitations and some validity threats.

The main limitation of our dataset is that many of the apps do not have complete metadata (category, ratings, etc.). AndroZoo does not store that info so there is a need to retrieve it from the different stores. That retrieval may be hampered by the following scenarios: i) the apps may no longer be available in those stores; and ii) the stores themselves do not offer simple ways to automatically get the info. In these cases, HTML parsers have to be written for webpages that could be in other languages or have structures that can (and do) change. We plan future work to complement the data through the parsing of various app store clones that keep app metadata even after their deletion from the main market places.

As it relates to our study, some threats to validity are worth noting. An external threat to validity of our results is that we mostly used a sample of AndroZoo, which itself does not account for all Android Apps. However, AndroZoo is a widely used repository and we took pains to extract a random sample large enough to be reasonably representative of AndroZoo. That sample covers several different domains (see Table 3.1): game, education, sports, and tools, to name a few. Furthermore, different from our previous paper, we retrieved additional malware dataset, which we believe help mitigate these external validity concerns.

As for threats to internal validity, it is worth noting that our investigation of different classifiers and feature selection techniques relied mostly on default settings from the widely used statistical tool Weka. Obviously, it is possible that our results and the ranking of these classifiers could be altered with different settings or more elaborate versions of these classifiers. However, to the best of our knowledge and based on the existing literature, malware researchers generally do not engage in complex parameter tuning of the classifiers they use. Furthermore, such parameter tuning may depend on a research expertise with a given classifier and may introduce additional variability. Overall, we believe that our

results provide malware researchers with an informed perspective about which classifiers to choose or avoid, especially if they do not intend to dedicate a significant amount of time to tune the parameters used by these classifiers.

Overall, we believe that the provided dataset and the accompanying experiments can be used to guide research ideas for anomaly detection, mining of safe / dangerous patterns, etc. In addition, the accompanying scripts that we provide offer options for researchers needing their own datasets. By relying on the instructions available on the AndroVul GitHub repository[8], they can add their APKs (in the apks folder) and run the provided scripts.

3.10 Conclusion

The ubiquity of smartphones, and their growing use make the security of these devices as important as that of standard computers. In this paper, we proposed a repository for Android vulnerabilities and experiments on classifier performances for different benchmarks (taken from the repository) to better support the research community engaged with anomaly detection and security issues for Android apps. Our contributions are threefold. First, we proposed a tool that harnesses well-known reverse engineering tools and greatly simplifies the generation of diverse vulnerability information (i.e. dangerous permissions, vulnerabilities from AndroBugs, and code smells in Smali code) for any app. Second, we proposed vulnerability data on a random sample of 16,180 Android apps downloaded from the well-established AndroZoo dataset, which we extended with 3,978 malwares retrieved from the VirusShare repository. Our tool and data make it so that an Android app researcher can start applying statistic analysis and machine learning experiments right away on our benchmark or right after downloading his/her own set of APKs. Third, we proposed detailed studies that provide: i) insights into the very good predictive power of the vulnerability information mined by our tool; and ii) information into which classifiers and data labelling decisions perform better. Our tool and data samples are available on

[8] https://github.com/Zakeya/AndroVul

GitHub and we intend to build and extend on that repository, notably by working on recovering more completely apps metadata.

Table 3.8 Severity Level Artifacts

Severity Level	Description
Critical	Confirmed security vulnerability (except for testing code)
Warning	AndroBugs Framework is not sure if this is a security vulnerability. Developers need to manually confirm.
Notice	Low priority issue or AndroBugs Framework tries to let you know some additional information.
Info	No security issue detected.

Table 3.9 Dangerous Permissions
(Definitions from Android) Taken from (Developer.android.com, 2021b)

Permission	Description
USE_SIP	Allows an application to use SIP service.
PROCESS_OUTGOING_CALLS	Allows an application to see the number being dialed during an outgoing call with the option to redirect the call to a different number or abort the call altogether.
BODY_SENSORS	Allows applications to discover and pair bluetooth devices.
SEND_SMS	Allows an application to send SMS messages.
RECEIVE_SMS	Allows an application to receive SMS messages.
READ_SMS	Allows an application to read SMS messages.
RECEIVE_WAP_PUSH	Allows an application to receive WAP push messages.
RECEIVE_MMS	Allows an application to monitor incoming MMS messages.
READ_EXTERNAL_STORAGE	Allows an application to read from external storage.
WRITE_EXTERNAL_STORAGE	Allows an application to write to external storage.

Table 3.10 Dangerous Permissions
(Definitions from Android) Taken from (Developer.android.com, 2021b)

Permission	Description
READ_CALENDAR	Allows an application to read the user's calendar data.
WRITE_CALENDAR	Allows an application to write the user's calendar data.
CAMERA	Required to be able to access the camera device.
READ_CONTACTS	Allows an application to read the user's contacts data .
WRITE_CONTACTS	Allows an application to write the user's contacts data.
GET_ACCOUNTS	Allows access to the list of accounts in the Accounts Service.
ACCESS_FINE_LOCATION	Allows an app to access precise location.
ACCESS_COARSE_LOCATION	Allows an app to access approximate location.
RECORD_AUDIO	Allows an application to record audio.
READ_PHONE_STATE	Allows read only access to phone state, including the phone number of the device.
READ_PHONE_NUMBERS	Allows read access to the device's phone number(s).
CALL_PHONE	Allows an application to initiate a phone call without going through the Dialer user interface.
ANSWER_PHONE_CALLS	Allows the app to answer an incoming phone call.
READ_CALL_LOG	Allows an application to read the user's call log.
WRITE_CALL_LOG	Allows an application to write the user's call log data.
ADD_VOICEMAIL	Allows an application to add voicemails into the system.

Table 3.11 Regular expressions used in our tool containing Smali type for code smell Taken from (Ghafari *et al.*, 2017)

Smell Code	Symptom in Smali Code and Corresponding Escaped Regexes
Custom Scheme Channel	Scheme registration code exists Lorg/apache/http/conn/scheme/SchemeRegistry;-> , register*Lorg/apache/http/conn/scheme/Scheme*; , Lorg/apache/http/conn/scheme/Scheme
Header Attachment	Header attachment code exists Lorg/apache/http/client/methods/HttpGet;-> , addHeader*Ljava/lang/String*;*Ljava/lang/String*;
Unique Hardware Identifier	Hardware identifier access code for MACs and IMEI exists Landroid/telephony/TelephonyManager;-\\ , getDeviceId\(\) , Ljava/lang/String Landroid/bluetooth/BluetoothAdapter;-> , getAddress\(\) \, Ljava/lang/String Landroid/net/wifi/WifiInfo;-> , getMacAddress\(\) , Ljava/lang/String
Exposed Clipboard	Clipboard manipulation code exists Landroid/content/ClipboardManager;-> , getPrimaryClip\(\) , Landroid/content/ClipData Landroid/content/ClipboardManager;-> , setPrimaryClip*Landroid/content/ClipData*;
Insecure Network Protocol	Http connection establishment code exists Ljava/net/HttpURLConnection;- > , *Ljava/net/URL*;
Improper Certificate Validation	Customised certificate validation code exists .implements Ljavax/net/ssl/X509TrustManager;
Dynamic Code Loading	Dynamic code loading mechanism exists Landroid/content/Context;-> , createPackageContext*Ljava/lang/String*;*I* , Landroid/content/Context
XSS-like Code Injection	WebView JavaScript setting code exists Landroid/webkit/WebSettings;-> , setJavaScriptEnabled*Z*
Broken WebView's Sandbox	WebView Java interface code exists Landroid/webkit/WebView;-> , addJavascriptInterface\(Ljava/lang/Object; , Ljava/lang/String;\)

CHAPTER 4

DEEP LEARNING BASED ANDROID ANOMALY DETECTION USING A COMBINATION OF VULNERABILITIES DATASET

Zakeya Namrud[a], Sègla Kpodjedo[a], Chamseddine Talhi[a], Ahmed Bali[a], Alvine Boaye Belle[b]

[a]Department of Software and IT Engineering, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3
[b] Department of Electrical Engineering and Computer Science, York University, Toronto, ON M2J 4A6, Canada

Paper published in Journal of Applied Sciences, August 2021

4.1 Abstract

As the leading mobile phone operating system, Android is an attractive target for malicious applications trying to exploit the system's security vulnerabilities. Although several approaches have been proposed in the research literature for the detection of Android malwares, many of them suffer from issues such as small training datasets, few features (most studies are limited to permissions) that ultimately affect their performance. In order to address these issues, we propose an approach combining advanced machine learning techniques and Android vulnerabilities taken from the AndroVul dataset, which contains a novel combination of features for three different vulnerability levels, including dangerous permissions, code smells, and AndroBugs vulnerabilities. Our approach relies on that dataset to train Deep Learning (DL) and Support Vector Machine (SVM) models for the detection of Android malware. Our results show that both models are capable of detecting malware encoded in Android APK files with about 99% accuracy, which is better than the current state-of-the-art approaches.

4.2 Introduction

The adoption of mobile applications in a wide range of domains has made many activities, from banking to education or gaming, simpler, faster, or more convenient. The dominant mobile operating system is Android, thanks in part, to the high number of freely available apps accessible through its official market (Google Play store[9]). The reach of the Android system goes even beyond that official market since the open source OS allows users to install unofficial (e.g., third-party) apps. A key security feature of Android is its permission system; permissions sought by an Android application must be granted manually by the user of the mobile device before the app is installed (on older OS versions) or before the app can perform some operations (on newer OS versions). However, users are generally uneducated about the risks of the permissions they can be asked to grant. They may grant permissions allowing malicious apps to exploit security breaches (Sirisha *et al.*, 2019) and to monitor a mobile device without the user's consent (Sabhadiya *et al.*, 2019). These malwares can cause severe malfunction, steal sensitive personal information (e.g., banking information, passwords), corrupt files, display unwanted advertisement, and even lock the device unless a ransom is paid.

According to Haystack(Zink, 2021), 70% of mobile apps fetch users' personal data and hand it over to third-party companies. Furthermore, a report published by AV-TEST security Institute(GmbH, 2020) states that there is an exponential increase of new malicious program (malware) samples every year. In 2020, Kaspersky (CHEBYSHEV, 2021) detected around 5.7 million malicious installation packages for mobile devices, which was an increase of 2.1 million over 2019 (see Figure 4.1). Given this increasing influx of new malwares, typical signature-based malware detection approaches, which, in short, rely on databases of specific characteristics of known malwares are not up to the task of effectively safe-guarding Android devices from the malware threats. Malwares may go undiscovered if their signature is not identified in the database, and the databases must be continuously updated to stay relevant.

[9] https://play.google.com/store/apps

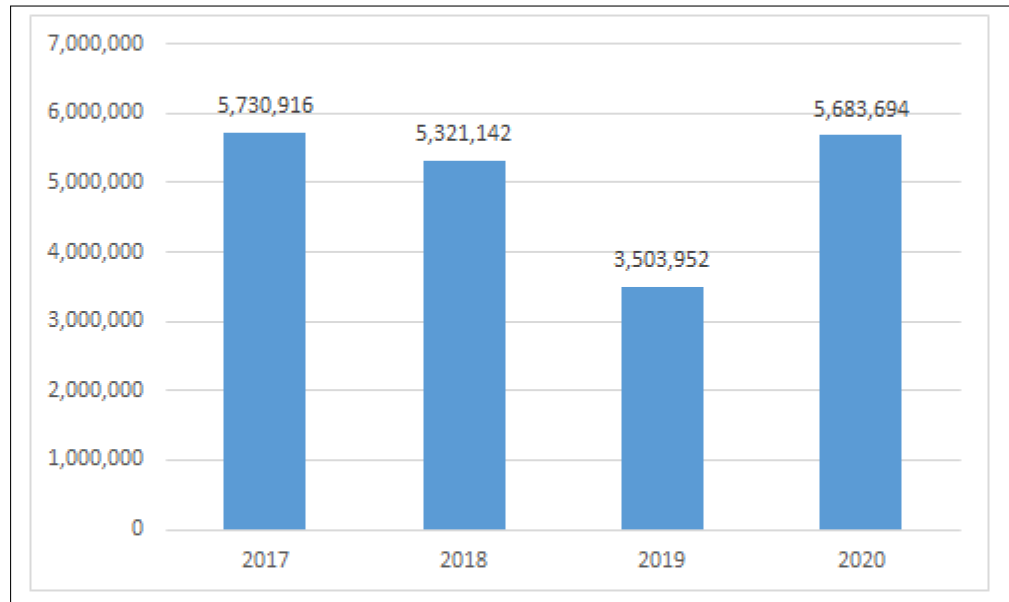


Figure 4.1 Installation of mobile malicious packages in Android from 2017 to 2020

Research literature on malware detection (e.g., (Kumaran & Li, 2016), (Sirisha *et al.*, 2019), (Li *et al.*, 2018b)) includes advanced proposals using machine learning techniques to detect with a higher accuracy unknown Android malware embedded in APK files. Such work typically extracts features (e.g., permissions, and API calls in the code) from known benign apps and malware, then uses machine learning algorithms (e.g., decision tree, Random Forest) to uncover ways to detect malicious apps.

The present work builds on AndroVul(Namrud *et al.*, 2019b), our previous research work centered on the proposal of a dataset of vulnerability features of Android apps. Our current study, not only adds around 6K apps to that dataset, but most importantly explored the use of advanced techniques such as Deep Learning (DL) and Support Vector Machine (SVM) to achieve the highest possible malware classification performances. Overall, we started from reverse-engineering an Android application APK file into a set of vulnerabilities features that can be used to reflect the application's behaviors. As a result, we obtain a dataset of more than 18K apps (about 6K more than in the original paper)

and 74 vulnerabilities features, which we use to experiment on DL and SVM models. Thus, our contributions can be summarized as follows:

1. We developed a malware detection model based on deep learning and we investigated several node architectures in hidden layers in order to get the highest possible performance. The proposed model outperforms the state-of-the-art.
2. We developed a malware detection model based on SVM and investigated different parameter settings to identify which were the best for our malware detection task.
3. We provide comparison of the performance of our DL and SVM classifiers, with respect to state-of-the-art approaches and even some commercial anti-viruses and results show that our classifiers are the most effective in identifying malicious applications. As such, our models establish a new, important reference point in the current state-of-the-art when it comes to malware detection.

The remainder of this paper is organized as follows: Section 5.3 introduces some background concepts. Section 4.4 describes the overall design of our Android malware detection system and how it operates. Section 4.5 shows the experimental results obtained when assessing the performance of our models. Section 4.6 presents related work. Section 4.7 concludes the paper and outlines future work.

4.3 Background

In this section, we define some concepts needed to better grasp our approach.

4.3.1 Android Vulnerabilities

Vulnerabilities, commonly referred to as security-sensitive defects, can be found statically using rules that describe vulnerable code patterns. They are typically diverse in terms of the components involved, the attack vector necessary for exploitation, and so forth. We

focus on common vulnerabilities that have a severity level that warrants their inclusion in security reports and earlier Android security research in this study. We briefly list the vulnerabilities that we have taken into consideration as features in our work.

4.3.1.1 Dangerous Permissions

The permission system in Android is a critical security feature since it regulates the rights granted to apps, requiring them to request particular permissions in order to execute specific operations. This approach necessitates the declaration by app developers of which sensitive resources will be utilized by their applications. When installing or using the apps, app users must consent to the requests made by the developers. According to Android, there are several categories of permissions, among which are "dangerous" ones, which are deemed more critical and privacy sensitive because they grant access to system features such as cameras and internet access as well as personal contact information and SMS messages, among other things (Tchakounté & Hayata, 2017).

4.3.1.2 AndroBugs Vulnerabilities

AndroBugs is a well-known security testing tool for Android applications, and it is used to evaluate them for vulnerabilities and possibly critical security issues. APKs are reverse engineered using the tool, which searches for a variety of concerns, ranging from a lack of adherence to best practices to the usage of potentially dangerous shell commands or the exposure to vulnerabilities via third-party libraries. It has a demonstrated track record of uncovering security flaws in some of the most popular applications and software development kits (SDKs). It is run as a command line utility and generates reports with four severity levels: Critical is confirmed vulnerability that should be solved (except for testing code). Warning is possible vulnerability that should be checked by developers. Notice is a low priority issue, and Info is no security issue detected.

4.3.1.3 Code Smell

Code smells refer to code source items that may suggest more serious issues in the code (Verbraeken *et al.*, 2020). The term "security code smells" refers to "symptoms in the code that signal the possibility of a security vulnerability" in Android applications, according to Ghafari et al. (Gadient *et al.*, 2017). Following a review of the literature, they identified 28 security code smells (Gadient *et al.*, 2017) that they categorized into five categories, including Insufficient Attack Protection, Security Invalidation, Broken Access control, Sensitive Data Exposure, and Lax Input Validation.

4.3.2 Machine Learning (ML)

Machine Learning (ML) refers to a class of methods for automatically creating models from data. These methods allow solving complex problems such as anomaly detection, classification, clustering, and regression (Verbraeken *et al.*, 2020). As Verbraeken et al. (Verbraeken *et al.*, 2020) point out, a problem can be solved with ML through two phases: training and prediction. The training phase results in a trained model, after which the trained model is deployed in practice at the prediction phase. During that phase, the trained model is fed with new data and generates predictions by inferring these new data. Different ML algorithms (e.g., supervised, unsupervised, classification, regression) have been proposed depending on the kind of feedback that the algorithm receives while learning (Verbraeken *et al.*, 2020). Machine learning techniques have been deployed in related proposals by some other security researchers in articles such as (Catak *et al.*, 2020), (Catak *et al.*, 2021). In the current work, we investigated two of the most powerful families of machine learning techniques: Deep Learning (DL) and Support Vector Machines (SVM).

4.3.2.1 Deep Learning (DL)

Deep learning (Naway & Li, 2018) is a subfield of Artificial Neural Networks (ANNs) and ML. The DL approach is rapidly gaining traction and is widely utilized in computer vision, speech recognition, and natural language processing. At the same time, DL-based malware detection for Android has become a major trend. A typical DL model for data processing is an extremely deep neural network with numerous hidden layers of many linked neurons. Each layer consists of several different neurons, each with its own weights and likely activation mechanism. When data are fed into a neural network, the loss function computes the prediction error. The optimizer is used to progressively change the weights in order to reduce the loss function error and increase the accuracy. It trains the data and assesses its accuracy on the test set. One of the dominant models in deep learning is ANNs (Li *et al.*, 2018c) which have been widely used for image recognition and have shown promising results in contextual categorization in DL. An ANN algorithm can learn hidden patterns from data on its own, combine them, and create much more powerful decision rules (Li *et al.*, 2018b). Figure 4.2 shows the overall DL definition, which is composed of three layers, namely the input layer, the hidden layer, and the output layer.

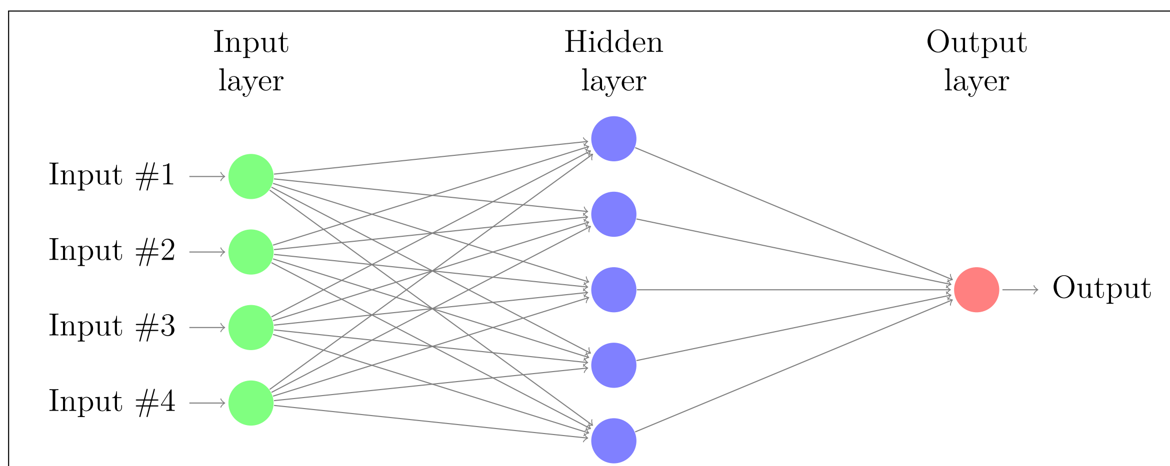


Figure 4.2 General architecture of a Deep Learning model.

4.3.2.2 Support Vector Machines (SVM)

A Support Vector Machine (SVM) is a machine learning technique that, in its most basic version, consists in finding a line that separates two classes of (training) data points, in such a way that future data points can be accurately classified, depending on which side of the line they end up on. In most cases, the line will be an hyperplane because the data will often be in an N-dimensional (N denotes the number of features) space (Verma & Sharan, 2017). Additionally, among the possible hyperplanes that could separate the data into two classes, the hyperplane which is the furthest from the two data classes it is separating is preferred because it reduces risks of overfitting the model to the current training data. Furthermore, it is not always possible (or even desirable) to neatly separate all the data into two perfectly clean classes. A certain amount of mis-classification can be allowed in order to account for outliers or erroneous data. Finally, the boundary between the two classes of data may not be linear, in which case, there is a need to involve mathematical kernels that can handle those situations. In our study, we use a nonlinear SVM with a Gaussian radial basis function (rbf) kernel, which is a well established and robust version of SVM. When constructing such classifier, two parameters must be passed as arguments. The parameter C accounts for the intolerance to mis-classification of the training data; the higher it is, the more the training data points will have to be correctly classified. The other parameter gamma can be understood as controlling the influence of a single training data point; the higher it is, the lower the reach of a single data point. The parameters C and gamma work together and have to be carefully chosen.

4.4 Methodology

In this section, we present the sample of apps on which we performed our experiments, the features we extract from a given apk, and information about the machine learning techniques we selected.

4.4.1 Dataset

Our evaluation was carried out with the AndroVul (Namrud *et al.*, 2019b) dataset, which core is a sample of 18,780 Android apps collected from the AndroZoo[10] repository. The Androzoo project proposes along with the apks of its apps, metadata that includes the number of antiviruses from the website Virus Total ((virustotal, 2021)) that flagged the app as a malware. For our study, and consistent with (Namrud *et al.*, 2019b), we considered as benign apps the apps with zero flags and as malicious apps, the apps with two or more antivirus flags.

To this core set of apps, we added malwares gathered from VirusShare(Forensics, 2021), a malware repository intended to help security analysts and malware researchers. However, the VirusShare repository is not dedicated to Android malwares and does not propose any mechanisms or metadata to quickly identify which programs are apks and which are not. It simply hosts a variety of files without even a specified extension. To recover the Android malwares in that repository, we had to download Giga Bytes worth of potentially harmful files and figure out a simple procedure to identify which files were Android apks. As seen in Algorithm 4.1, we renamed all the files by adding the extension ".apk", then tried to apply our reverse engineering scripts and tools. Files that return empty folders after the reverse engineering are discarded; the others are saved as apks.

Table 4.1 contains a description of the datasets, and Figure 5 depicts a visualisation of the datasets.

4.4.2 Feature extraction

We introduced the AndroVul dataset in (Namrud *et al.*, 2019b) and the interested reader can find full details in that publication. In this section, we propose a brief overview of the feature extraction process and output as applied to an app. In short, we used

[10] https://AndroZoo.uni.lu/

Algorithm 4.1 VirusShare Android apps collection.

Input: Execution files
1 **for** *all files in folder* **do**
2 $file \leftarrow rename$(file.apk) ▷considering all files Android apps
3 $APK_{file} \leftarrow Open$(file) ▷Reverse engineering APK with Android tools
4 $Package \leftarrow Get$(files) **if** $(Package \leftarrow empty)$ **then**
5 $Package \leftarrow delete_it$
6 **else**
7 $Package \leftarrow App_android$
8 **end**
9 ▷ App_android save it in Android folder
10 **end**

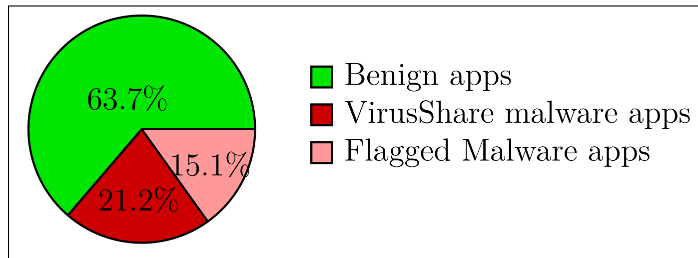


Figure 4.3 Dataset Visualization

well-known static analysis tools (Apktool, AngroBugs) to extract three kinds of features, i.e., dangerous permissions from the app's manifest file, code smells from the app's Smali code representation, and AndroBugs vulnerabilities from the APK files. In Algorithm 4.2, we describe the general process for extracting the vulnerability features from an apk file. It starts with the reverse engineering process (Line 2), followed by the extraction of

Table 4.1 Dataset description

App	**Samples**
Benign	11,971
Flagged Malware	2,831
VirusShare Malware	3,978
Total	18,780

the desired features from their respective files (lines 3 -5). The extracted vulnerabilities features are then mapped to values and written into a single csv file. The values mapped to the features are determined as shown in the equations below:

$$permissions \rightarrow \begin{cases} Requested_permission & 1 \\ Other_permissions & 0 \end{cases}$$

$$CodeSmells \rightarrow \begin{cases} Weight(security\,code\,smell) \end{cases}$$

$$AngroBugs_vulnerabilities \rightarrow \begin{cases} Critical & 1 \\ Worning & 0.5 \\ Other & 0 \end{cases}$$

4.4.3 General architecture of our machine learning approach

Figure 4.4 presents the general architecture of our Android malware detection approach, which is divided into three stages: pre-processing, training, and detection.

In the preprocessing phase, the original feature dataset is standardized by reducing the mean and scaling to unit variance. The following formula is used to compute the standard score of sample x:

$$z = (x - u)/s \tag{4.1}$$

Where u denotes the mean of the training samples, and s denotes the standard deviation of the training samples.

Algorithm 4.2 Feature Extraction Algorithm.

```
   Input: Apk files; apps
   Output: Dataset in CSV_file
1  for all apps in Dataset do
2  |  APK_file ← Open(file)  ▷ Reverse engineering APK
3  |  Permissions_list ← Get_Distinct_Permissions(manifest_File)  ▷ Extracting
   |   permissions from manifest file
4  |  Code_Smell_list ← Get_CodeSmell(Smali_Files)  ▷ Extracting code smell
   |   from Smali files
5  |  AngroBugs_vulnerabilities_list ←
   |   Get_AngroBugs_vulnerabilities(AndroBugs_report)  ▷ Extracting
   |   AngroBugs_ vulnerabilities from AndroBugs_report
6  |  foreach app do
7  |  |  Permission ← App[i].Permission  ▷ Mapping permissions
8  |  |  Code_Smell ← App[i].CodeSmell  ▷ Mapping code smells
9  |  |  AndroBugs_vulnerabilities ← App[i].AngroBugsvulnerabilities  ▷
   |  |   Mapping AndroBugs_ vulnerabilities
10 |  end
11 end
12 CSV_(file) ← Append(CSV_(file),Concat(Vector_(Permission),
    Vector_(Code_Smell),Vector_(AngroBugs_vulnerabilities)))  ▷ Concatenating all
    vulnerabilities features in CSV file
13 return (CSV_(file))
```

As for training and testing, we opted for K-fold cross-validation. This validation approach consists in splitting, after random shuffling, the dataset into K groups, after which each group is used as a test group, while the other K-1 groups are used for training. More specifically, we chose, in accordance to many similar studies (e.g., (Bhattacharya & Goswami, 2017)), K = 20 for a fold cross-validation study, in which 80% of the data is used for training and 20% for testing (prediction).

4.4.4 Android Malware Detection based on Deep Learning

Figure 4.5 presents our system architecture for Android malware detection using Deep Learning.

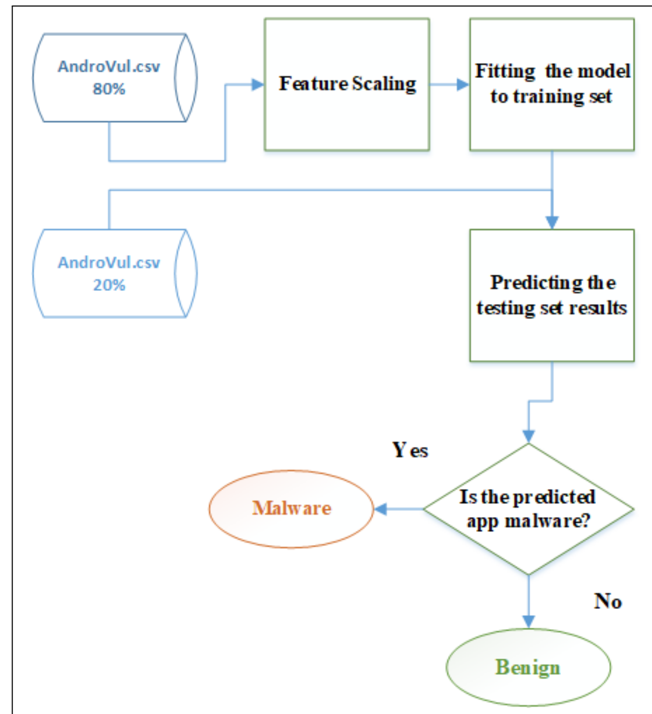


Figure 4.4 Design Methodology for malware detection in Android

In the training phase, the malware and benign behavior patterns are learned by the ANN (Xu *et al.*, 2017). The DL model was designed for learning the pattern with four hidden layers, a single input layer and a single output layer. Fully connected feed-forward deep neural network architecture with four hidden layers was utilized to implement the suggested approach. The rationale for limiting the number of hidden layers to four is the complexity of the design. Figure 4.5 shows the architecture of the DL algorithm and the number of nodes in each layer, and Figure 4.4 shows the overall model design. The description of mapping layers is as follows. Seventy-four neurons are used in the input layer to read the 74 features. The neurons are then linked to the first hidden layer, known as the dense layer, where a mathematical computation is performed using activation functions (Xu *et al.*, 2017). The ReLU activation mechanism was used in this case. Another non-linear activation feature that has become common in the field of DL is the ReLU function. ReLU stands for Rectified Linear Unit. The key benefit of using the ReLU mechanism is that it does not simultaneously activate all neurons. The neurons are

therefore disabled only if there is less than 0 in the output of the linear transformation. In addition, the 74 features were also mapped into 74 dimensions, and 74 dimensions are mapped into 32 dimensions in the second hidden layer. Thirty-two dimensions were mapped in 32 dimensions of the third hidden layer. Finally, their values were mapped to a single-dimensional output layer. Equations 4.2 and 4.3 define the activation function Y.

$$Y = \max(0, x) + bias \tag{4.2}$$

where Y denotes the output, x denotes the data, and bias is used to train the neural network on malware and benign patterns.

$$Weighted_{sum} = \sum_i W_i X_i \tag{4.3}$$

where wi denotes the weight applied to each input node and xi denotes the input applied to each node.

At the final output layer, the sigmoid function is applied to provide output values ranging from 0 to 1. A function of activation is defined by equation 4.4.

$$Sigmoid = \left(\frac{1}{1 + e^{-x}}\right) \tag{4.4}$$

Binary Cross Entropy loss function was used to compile the neural network model.

$$BinaryCrossEntropy = Error\,(y, f(X)) \tag{4.5}$$

where y=actual values, f(x) = predicted values. Furthermore, the weights are changed using the gradient descent optimizer (Huang *et al.*, 2017). It changes the parameters in such a way that the loss function can be reduced using Equation 4.6.

$$X = X - \alpha\left(\frac{\sigma}{dX}j(X)\right) \tag{4.6}$$

where X is the new updated weight, α denotes the rate of learning, and j(X) denotes the cost function, which is a quadratic equation based on the 74 features extracted from Android applications. There are two primary hyperparameters that govern the network's architecture or topology, which are the number of layers and nodes found within each hidden layer. Systematic experimentation allows configuring these hyperparameters when solving a specific predictive modeling problem. We have increased the number of epochs until the model was able to correctly classify the inputs. In the test phase, the DL model is tested using 20% of the dataset. After training the model, we tested it using the remaining 3,706 samples. The trained neural network model determines whether the provided APK file is malicious or benign based on the pattern. In the first experiment, the training phase results were significantly higher than the testing phase's, causing overfitting in the model. However, when we increased the number of samples in the dataset for both benign samples and malware samples, the overfitting was solved and the performance in the training phase was almost the same as that in the testing phase.

In Algorithm 4.3, we describe the general process of our classifier generation phases namely, data processing (lines 1-3), model building (Line 4), and model fitting (Line 5) as well as using the model for prediction (lines 6-13). For the complexity discussion, we focus on the prediction phase because once the model is trained, it can be reused as much as there is a need. The time and space complexity of the model's prediction phase is $O(p \times n_{l1} + ... + n_{li-1} \times n_{li} + ... + n_{ln-1} \times o)$, where p is the number of features, n_{li} is the number of neurons at layer i in a neural network, and o is the number of outputs. Therefore, the complexity is asymptotically quadratic, $O(\max(n_{li-1} \times n_{li}))$, in the size of the network layers. The architecture and parameters of our DL model were determined experimentally and tuned for best performance; they are described in Table 4.2 and Figure 4.6.

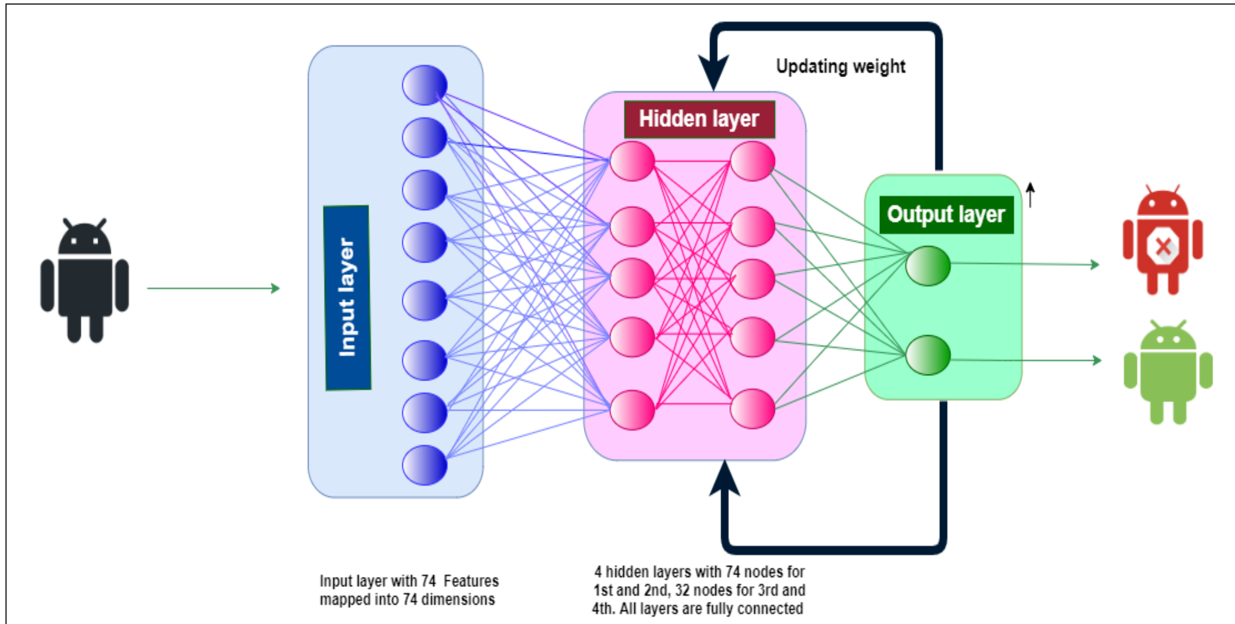


Figure 4.5 The architecture of DL layers using Sequential neural network

Algorithm 4.3 Deep Learning based Model.

Input: $X : apps_features, Y : labels$ ▷ label is benign or malware
$Building_params_{list} = (units, activation_function, input_dim, dpoint)$
$fitting_params_{list} = (X_{train}, Y_{train}, batch_size, epochs)$
$dpoint : decision_point$ ▷ 0.5 by default
Output: $predicted_app$ ▷benign or malware app

1. $X_{train}, X_{test}, Y_{train}, Y_{test}$ ▷ Splitting the dataset
2. $X_{train} = sc.fit_{transform(X_{train})}$ ▷Feature Scaling using StandardScaler (sc)
3. $X_{test} = sc.transform(X_{test})$
4. $Model = build_ANN_model_architecture(Building_params)$ ▷ANN model Building
5. $Model \leftarrow ANN_model_fit(fitting_params)$ ▷ANN model fitting
6. $y_{pred} = Model.predict(X_{test})$
7. **for** $app.pred \in y_{pred}$ **do**
8.   **if** $(app.pred > dpoint)$ **then**
9. $app \leftarrow Malware$;
10.   **else**
11. $app \leftarrow Benign$;
12.   **end**
13. **end**

```
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 74)                5550
_________________________________________________________________
dense_1 (Dense)              (None, 74)                5550
_________________________________________________________________
dropout (Dropout)            (None, 74)                0
_________________________________________________________________
dense_2 (Dense)              (None, 74)                5550
_________________________________________________________________
dense_3 (Dense)              (None, 32)                2400
_________________________________________________________________
dense_4 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 33
=================================================================
Total params: 20,139
Trainable params: 20,139
Non-trainable params: 0
_________________________________________________________________
..
```

Figure 4.6 Built DL model

Table 4.2 Best hyper-parameters

Parameters	Value
Number of units	74-74-74-32-32-1
Number of layers	one input, 4 hidden, one output
Activation function	relu,sigmoid
Kernel initializer	uniform
Dropout	0.2
optimizer	adam
epochs	1000
batch_size	200
loss	binary_crossentropy

4.4.5 Android Malware Detection based on support vector machine

We opted for a (non linear) Radial Basis Function (RBF) kernel SVM. In Algorithm 4.4, we describe the general process of our SVM classifier, from the data processing (lines 1-3), to the model building (Line 4), the model fitting (Line 5) as well as the prediction phase (lines 6-9). The complexity of the training phase is polynomial, $O(n_{sv}^2 \times p + p^3)$, in the size of the model parameters, where p is number of features and n_{sv} is the number of support vectors). That relatively high complexity of the training model is compensated

by the low complexity of the prediction model, which is of only $O(n_{sv} \times p)$ and can be reused several times once the model is well trained.

Table 4.5 shows the best hyper-parameters for SVM model.

Algorithm 4.4 Support Vector Machine based Model.

Input: $X : apps_features; Y : labels;$ ▷ label is benign or malware
$Building_params_{list} = (C, kernel, gamma)$
$fitting_params_{list} = (X_{train}, Y_{train})$ $dpoint : decision_point$ ▷ 0.5 by default
Output: $predicted_app$ ▷benign or malware app

1 $X_{train}, X_{test}, Y_{train}, Y_{test}$ ▷ Splitting the dataset
2 $X_{train} = sc.fit_{transform(X_{train})}$ ▷Feature Scaling using StandardScaler (sc)
3 $X_{test} = sc.transform(X_{test})$
4 $Model = build_SVM_model_architecture(Building_params)$ ▷SVM model Building
5 $Model \leftarrow SVM_model_fit(fitting_params)$ ▷SVM model fitting
6 $y_{pred} = Model.predict(X_{test})$
7 **for** $app.pred \in y_{pred}$ **do**
8   **if** $(app.pred > dpoint)$ **then**
9         $app \leftarrow Malware$;
10   **else**
11         $app \leftarrow Benign$;
12   **end**
13 **end**

4.5 Experiments

4.5.1 Performance Indicators

As it relates to the detection of malwares, we refer to True Positive (TP) as the number of malwares actually classified as such, True Negative (TN) as the number of benign apps classified as such, False Positive (FP) as the number of benign apps wrongly classified as malwares, and finally False Negative (FN) as the number of malwares wrongly classified as benign. More informative measures, widely used in malware detection analysis work, are derived from these simple measures, such as:

1. **Precision:** The ratio of actual malwares in the set of apps classified as such: TP/(TP+FP)

2. **Recall:** The ratio of malwares that were detected as such: TP/(TP+FN)

3. **Accuracy:** The percentage of applications that have been appropriately categorised: (TP+TN)/(TP+TN+FP+FN)

4. **F1-Measure:** A performance indicator that takes into account both the precision and recall of the obtained classification: 2 * (Recall * Precision) / (Recall + Precision)

5. **Area under ROC Curve (AUC):** A measure of the predictive power of the classifier that basically informs on how well the model can distinguish between classes (here, benign apps vs malwares).

For all these measures, the higher, the better, with 1 being the perfect value.

4.5.2 Experimental setup

We conducted experiments with both DL and SVM models. All the experiments were carried out using the same dataset. The experiments are done using the Python programming language, and the following are the characteristics of the computer used for the experiments; Windows 10(64 bit), Intel(R) Core(TM)i7-2600 CPU@ 3.40GHZ, and 16GB RAM.

4.5.3 Results

In this section, we present obtained results. The factors below explain why our approach was able to outperform other approaches. These include hyper-parameters tuning as well as the combination of vulnerability features in our dataset.

Performance of DL model: initially, we used 11,814 apps; in this experiment, an app which has 0 flag labelled as benign, whereas an app which has 2 or more flags labelled

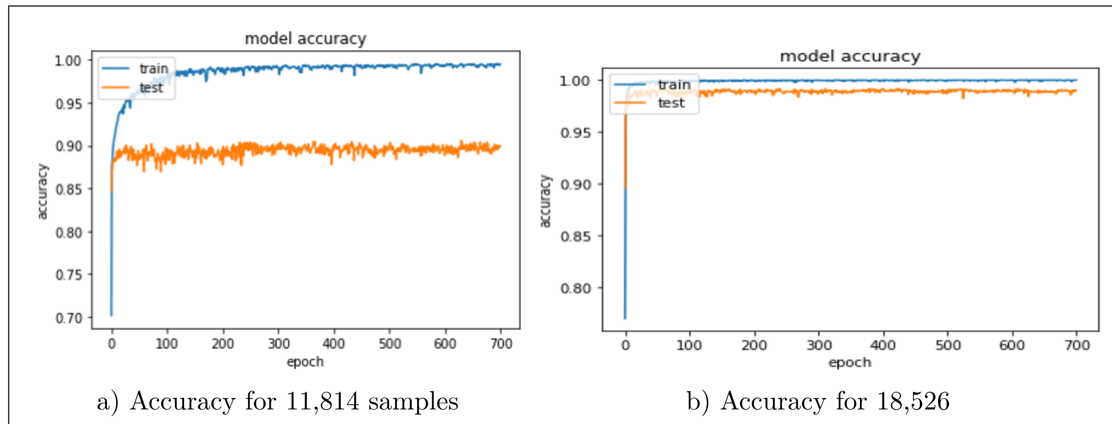
as malware, and an app with one flag are excluded. The training phase performance results were higher than the testing phase, which caused over-fitting in the model. To solve this situation, we increased the size of the dataset by adding the malware apps from VirusShare and apps with one flag as benign. Experimentally, from Table 4.3, we can observe that the size of the dataset has increased from 11,814 to 18,780 to avoid over-fitting and improve the performance.

Table 4.3 Comparison between the results of datasets with 11,814 samples, and 18,780 samples

Size of dataset	Accuracy	F1	AUC_score
11,814 samples	89%	90%	88%
18,780 samples	99.33%	99%	99.15%

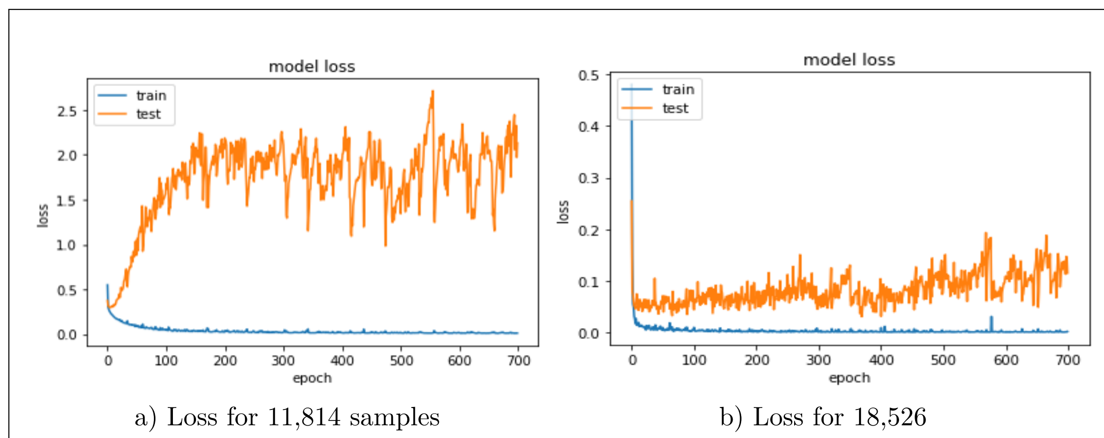
Figures 4.7 and 4.8 illustrate the history model's accuracy and loss for 11,814 and 18,526 samples, respectively. From the figures, it is clear that when we increase the size of the dataset, the accuracy and loss curve lines in the training phase are very close to the accuracy and loss curve lines in the testing phase. In the previous experiment, the difference between the accuracy and loss curve lines was huge. The performance improved when we increased the size of the dataset and the over-fitting problem was solved. If the output layer's Sigmoid result was greater than or equal to 0.5, the application was categorised as malware. Values below 0.5 were considered benign.

As Table 4.4 indicates, the performance improved when we increased the size of the dataset and the over-fitting problem was solved. If the output layer's Sigmoid result was greater than or equal to 0.5, the application was categorised as malware. Values below 0.5 were considered benign. Table 4.4 thus shows that the confusion matrix correctly classifies the 2,406 benign samples as benign and 1,275 malware samples as malware. Out of 3,706 app samples, 3,681 samples were predicted accurately and only 25 samples were wrongly predicted.



a) Accuracy for 11,814 samples b) Accuracy for 18,526

Figure 4.7 Comparison between history models Accuracy for 11,814 samples and 18,526 samples



a) Loss for 11,814 samples b) Loss for 18,526

Figure 4.8 Comparison between history models loss for 11,814 samples and 18,526 samples

Performance of the SVM model: Table 4.5 illustrates the experimental results. When we were tuning the hyperplane parameters, we noticed that when gamma is smaller than 0.01 and C is higher than 1,000, the results improve, i.e. both parameters increase the

Table 4.4 DL Confusion matrix

3706		**Predicted class**	
		Benign	Malware
Sensitivity	B(99.71%)	2406	7
Specificity	M(98.61%)	18	1275

values of AUC, F1, and the accuracy. With such parameter values, we can therefore get the correctly separating hyperplane and improve the performance of the model. Table 4.6 shows that the confusion matrix correctly classifies the 2,425 benign samples as benign and 1,235 malware samples as malware. Out of 3,706 app samples, 3,660 were predicted accurately and only 46 were wrongly predicted.

Challenges and discussion: improving the performance of the SVM classifier was challenging and involved some fine tuning with respect to the two parameters: C and gamma. Our results showed that tuning C correctly is a vital step in the use of SVMs for structural risk minimization [11]. When gamma gets smaller, the results improve.

As for the DL, the configuration of the hyperparameters (the number of layers and nodes in each hidden layer) for our specific predictive modeling problem was done via systematic experimentation. It is worth noting that the time complexity of the DL algorithm is higher than the time complexity of the SVM algorithm.

Table 4.5 The experimentation results for SVM parameters

C	Gamma	Accuracy	F1	AUC_score
10	0.1	93.98%	91.4%	94%
100	0.1	94.7%	92.4%	94.75%
1000	0.1	95.1%	93%	95.2%
10	0.01	97.95%	96.95%	97.4%
100	0.01	98.38%	97.6%	97.97%
1000	0.01	98.76%	98.2%	98.5%

Table 4.6 SVM Confusion matrix

3706		**Predicted class**	
		Benign	Malware
Sensitivity	B(99.26%)	2425	18
Specificity	M(97.78%)	28	1235

[11] In RBF kernel, both C and gamma parameters need to be optimized simultaneously. If gamma is large, the effect of C becomes negligible.

Table 4.7 shows the results (accuracy, F1 and AUC_score) obtained with our DL and SVM classifiers. It also compares these results to the ones obtained by the best state-of-the-art approach i.e. (Naway & Li, 2019). The highest accuracy for related work is 95.31%, but our models show better performances in both DL and SVM classifiers. Their accuracies are 99.33% and 98.76% respectively.

Table 4.7 Comparison between DL,SVM classifiers and the Related work

The classifier	Accuracy	F1	AUC_score
Deep Learning	99.33%	99.03%	99.15%
SVM	98.76%	98.2%	98.5%
Best result for State of Art	95.31%	95.31	N/A

4.5.4 Comparison with well-known anti-virus tool

As we mentioned previously, the dataset has two kinds of malwares: the flagged malware apps and the malware apps collected from the VirusShare repository. This repository provides access to live malware and day one malware, motivating us to upload the VirusShare malware apps to the Virus Total tool to scan them. We compared the obtained results using our model to detect VirusShare malware apps, and the obtained results using Virus Total tool to detect VirusShare malware apps (the same samples used in our approach). We observed that our model was able to detect 99.33% of the VirusShare malware apps, while Virus Total tool was able to detect only 75%. Table 4.8 shows a comparison with a well-known anti-virus tool (Virus Total).

4.6 Related work

Over the last few years, considerable effort has been devoted to the development of novel methodologies for detecting Android malware anomalies using machine learning techniques (e.g., (Baskaran & Ralescu, 2016), (Yerima *et al.*, 2014), and (Al Ali *et al.*,

Table 4.8 Comparison with well known anti-virus tool

Malware Detection	**Accuracy**
Our approach	99.33%
Virus Total	75%

2017)). In the current section, we propose an overview of the different proposals through the lens of the kind of analyses performed to obtain the features used in training the machine learnique: static analysis, dynamic analysis, hybrid analysis.

4.6.1 Static Analysis

Static analysis is the easiest and least expensive method for obtaining the features that will characterise an application. Permissions are the most commonly used features but some other elements such as intent filters, api calls, etc. have been investigated as well.

Sirisha et al. (Sirisha *et al.*, 2019) focused solely on permissions and proposed a a deep neural network model which attained an accuracy of 85%, on a dataset of 398 apps (benign and malware) and 331 features (permissions). Also focused on permissions, (Rehman *et al.*, 2018) proposed a framework that is both signature- and heuristic-based. They performed experiments using various classifiers such as SVM, Decision Tree, J48 and KNN, and used an existing dataset containing 401 apps and permissions as features. The accuracy of their approach is 85%.

Differently, Kumaran and Li (Kumaran & Li, 2016) applied different ML algorithms to features extracted from permissions and intent filters found in an app's manifest. They found that permissions performed much better than intent filters but that using both sources yielded a detection accuracy of 91.7% percent (SVM) and 91.4% percent (KNN), which outperforms the classification performance of either feature set individually. More recently, Zhu et al. (Zhu *et al.*, 2018b) proposed DroidDet, an Android malware

classification approach built on Random Forest. It utilizes various static features derived from permissions and API calls and attained an accuracy of 88.26% on a dataset of 2,130 apps. Similarly, Li et al. (Li *et al.*, 2018b) proposed a Deep Learning algorithm that achieved 90% accuracy on a dataset of 2,800 apps (benign and malware) and 237 features (permissions, API calls, and URLs). Also using deep learning, Naway et al. (Naway & Li, 2019) investigated static features (permissions, Intents, API calls, Invalid certificates) on a dataset of 1,200 apps and attained an accuracy of 95.31%.

In our previous work (Namrud *et al.*, 2019a), we proposed the AndroVul dataset and a preliminary investigation of the dataset as it relates to the detection of malwares. More precisely, we used the well-known machine learning software Weka and selected NaiveBayes (NB) from its bayes category, RBF classifier from its function category, JRip from its rules category, and J48 from its tree category. The selected machine learning approaches were applied under identical settings and with default parameters. The objective of that paper was to demonstrate the potential of the proposed features for the detection of malwares. In contrast to that work, our key objective in this research work is to propose a finely tuned machine learning appproach able to outperform existing approaches and anti-virus products. The additional work required involved tuning the hyper-parameters of the machine learning approaches, increasing the amount of malware apps, and conducting additional experiments and comparisons with existing literature and antiviruses.

4.6.2 Dynamic Analysis

Dynamic analysis takes interest into an app's behavior at run-time and may detect malicious activity on an actual execution path. As such, it is resistant to code obfuscation but on the other hand may have minimal code coverage, depending on how extensive and complete are the execution scenarios it considers.

Mas'ud et al. (Mas' ud *et al.*, 2014) proposed a malware detection system that uses dynamic analysis based on five different sets of features obtained through dynamic analysis.

It employs five separate ML classifiers in order to find the optimal combination for efficiently classifying Android malware. The experimental results showed that a multilayer perceptron classifier yielded the highest accuracy 83%. Martinelli et al. (Martinelli *et al.*, 2017) developed a method that utilizes a network of neural convolution implemented through dynamic analysis of system calls occurrences. Their work is based on a recent dataset composed of 7,100 apps. They created a number of user interface interactions and system events during the duration of the application's execution. The accuracy is 90%.

4.6.3 Hybrid Analysis

Hybrid analysis techniques (e.g., (Naway & Li, 2018), (Muttoo & Badhani, 2017)) entails the use of both static and dynamic elements. This dual perspective improve the identification's accuracy but may come with more resource consumption, especially when the analysis is done on a mobile device.

Yuan at al. (Yuan *et al.*, 2014) presented a machine learning-based method for malware detection that makes use of over 200 features collected from both static and dynamic analysis of Android apps. The comparison of modelling results reveals that the deep learning technique is particularly well-suited for Android malware detection, with a high level of 96% accuracy when applied to real-world Android application collections. their dataset contains 250 malware samples from contagio mobile[12] and 250 benign apps from Google Play Store.

In a subsequent work, Yuan at al. (Yuan *et al.*, 2016b) developed another model based on the DBN: the Droid Detector. The proposed method was validated against a broad unbalanced dataset containing 20,000 benign and malicious samples. The results showed that DBN performed well, with an accuracy of 96.76%. Around the same time, L. Xu et al. (Xu *et al.*, 2016) proposed an approach for identifying Android malware that relies on autoencoders to analyse the app's features. It then uses an SVM classifier to classify

[12] http://contagiominidump.blogspot.com/

the apps as malicious or trustworthy. They conducted experiments on a dataset of 5,888 benign and malware apps, analysing static and dynamic elements separately and found that static features outperformed dynamic features.

Some other security researchers deployed machine learning techniques to propose related approaches. For instance, in (Catak *et al.*, 2020), authors were mostly concerned with metamorphic malware. The primary objective of this research is to provide a mechanism for classifying malware based on its behaviour. They began their investigation by building a dataset of API calls performed on the Windows operating system that reflects malicious software behaviour. Long Short-Term Memory (LSTM) classifier was utilized to classify the data in this investigation. The classifier's result indicates an accuracy of up to 95% with an F1-score of 0.83. The use of machine learning to handle security vulnerabilities is similar to their approach. However, we have chosen to concentrate on Android platform security issues rather than other platforms.

Table 4.9 presents the feature, dataset, and classifier used in each related work as well as our approach. In particular, this table allows us to conclude that: 1) our work has been tested on a dataset that includes more sort of features than the ones used by other approaches; and 2) it outperforms existing approaches.

Table 4.9 Comparison between state of the art research and our approach

References	Feature used	Dataset used	Used Classifier	Accuracy
Paper (Sirisha *et al.*, 2019)	permissions	398 samples 331 features	Deep Learning	85%
Paper (Li *et al.*, 2018b)	permissions, APIs,URLs	2800 samples 237 features	Deep Learning	90%
Paper (Naway & Li, 2019)	permissions, APIs, Invalid certificate	1200 samples	Deep Learning	95.31%
Paper (Zhu *et al.*, 2018b)	permissions, APIs	2130 samples	Random Forest	88.26%
Paper (Rehman *et al.*, 2018)	permissions	401 samples	SVM	85%
Paper (Zhu *et al.*, 2018a)	permissions, APIs	2130 samples	Random Forest	89.91%.
Our previous work(Namrud *et al.*, 2019b)	Permissions, Code smell, AndroBugs vulnerabilities	1600 samples 74 features	Weka (RBF) provides best result	83%
Our approach	Permissions, Code smell, AndroBugs vulnerabilities	18,526 samples 74 features	Deep learning SVM	99.3% 98.76% Respectively

4.7 Conclusion

Android is the most popular smartphone operating system, accounting for 85 percent of the market. However, Android's widespread acceptance and openness make it an ideal target for malicious applications that take advantage of the system's security flaws. Signature-based malware detection present in most antiviruses is vulnerable to new malware, so advanced technologies such as machine learning approaches have been proposed to tackle malware detection. Our current work builds on and extends a previous

work in which we collected vulnerability features (e.g., code smells, dangerous permissions, and vulnerabilities identified by the tool AndroBugs) from Android apks and proposed a dataset of almost 12K apps from the AndroZoo repository. A first important contribution was the addition (and reverse engineering of the features) of thousands of malwares from VirusShare, a well-known virus repository. In general, the more data points, the better the prediction models, so it was important and beneficial to our experiments and the research community in general to improve the size of the dataset. The focus of the current paper is on proposing highly efficient machine learning models able to fully leverage the potential of the features we collected. To achieve that goal, we used two different advanced classifiers (Deep Learning & SVM) to learn the malware and benign patterns. We implemented these algorithms and experimented with them to get the best hyper parameters for malware detection using the features we collected. Both of our classifiers achieve an accuracy of around 99% and these results significantly outperform the state-of-art and a collection of antivirus, as proposed on the site VirusTotal.

Short term future work involves the investigation of possible trends in Android malware development (and thus detection); we plan to investigate the data on a multiple year basis to identify whether some features become more relevant in the newest malware. This is especially interesting, considering the relatively rapid pace at which the Android OS changes. Longer term, we plan to apply the lessons learned while experimenting with DL and SVM parameters on an expanded dataset of apps and features. More specifically, we plan to investigate the potential of other features, especially those that can be obtained from an app's manifest file (intent filters, xml data, etc.). Additionnally, we would like to investigate whether the category assigned to an app by a developer (whether a malicious actor or not) should be a factor in the patterns learned by advanced techniques.

CHAPTER 5

DEEP-LAYER CLUSTERING TO IDENTIFY PERMISSION USAGE PATTERNS OF ANDROID APP CATEGORIES

Zakeya Namrud[a], Sègla Kpodjedo[a], Chamseddine Talhi[a], Ahmed Bali[a]

[a] Department of Software and IT Engineering, École de Technologie Supérieure, 1100 Notre-Dame Ouest, Montréal, Québec, Canada H3C 1K3

Paper submitted for publication, September 2021

5.1 Abstract

With the increasing usage of smartphones in banks, medical services and m-commerce, and the uploading of applications from unofficial sources, security has become a major concern for smartphone users. Malicious apps can steal passwords, leak details, and generally cause havoc with users' accounts. Current anti-virus programs rely on static signatures that need to be changed periodically and cannot identify zero-day malware. The Android permission system is the central security mechanism that regulates the execution of application tasks. Although recent advances in research have provided various approaches and detection methods for finding malware apps, the available literature lacks a full analysis of this subject. We fill this gap by: 1) Systematically and automatically building a large dataset of malware and benign apps, which we have made available to the community. Our dataset has around 16K apps and 118 features. 2) We offer a novel approach for automatically identifying permission usage patterns, which are groupings of permissions that developers frequently utilise together. The approach combines SOM and K-means clustering algorithms to classify permissions according to app usage categories. The results demonstrate that the proposed methodology is able to detect most of the consistent and coherent permission usage patterns across a wide variety of application categories. To assess our strategy, we add the identified patterns as features to our dataset and then apply an SVM classifier for malware detection. Our results indicate that the identified patterns improve the performance of the classifier.

5.2 Introduction

User statistics show that Android is the most widely used operating system (OS) on mobile devices and is expected to remain the most popular OS until 2023 (O'Dea, 2021). Smartphones have been a key target for application developers who wish to exploit them for malicious purposes. Malicious tech is one of the biggest challenges with any software platform, and Android is no exception. Android apps can pose severe threats for Android users. According to Gartner, by the end of 2020, mobile applications were downloaded over 493 million times per day, generating more than $198 billion in revenue and making them popular computing tools for users worldwide. Such huge numbers are mostly driven by the Google Android mobile OS, which has an impressive smartphone market share of 82.8% (Gartner.com, 2021). This is mainly because it is open source and has a massive collection of applications in the official Android app store as well as in third-party Android app stores. However, their popularity comes at a cost: Android apps are also a vehicle for spreading vulnerabilities. A key security mechanism of Android is its permission system, which controls the privileges of applications. Under this system, apps must request access to particular permissions in order to perform certain functionalities. Moreover, the mechanism requires that app developers declare which sensitive resources will be used by their applications. Users have to agree with the requests when installing/running the applications. This constrains a given application to the resources it can request at run-time. Android has established a set of best practices designed to help developers properly define and operate permissions inside their source code. Unfortunately, there is no integrated security mechanism to guarantee that the apps only ask for the permissions they need. Moreover, developers do not always adhere to best practices guidelines (IDC, 2021), which makes the applications more sensitive to security issues.

In this paper, we explore the use of 103 permissions for around 16K apps on the Android market. First, we investigate permission use for apps in different categories. Then we present a novel methodology for mining permission usage patterns, which we refer to as

SOM+K-means. A permission use pattern is defined as a group of permissions utilised together in apps. Our strategy is based on a comparison of how permissions are used together and their correlation to apps across different categories. The patterns' permissions are dispersed over several use cohesion levels/layers. Each level indicates the frequency of co-usage of a set of permissions, while the distribution across various levels illustrates the degree of co-usage.

Our approach utilises a form of SOM+K-means, which is a commonly used clustering technique. SOM+K-means will identify probable permission usage patterns based on an investigation of its usage frequency and consistency across a number of apps within different categories. Utility permissions may be used by apps belonging to several categories. As a result, the logic behind distributing permissions in a pattern based on different levels of use cohesiveness is to distinguish between the most and least particular permissions. Additionally, our methodology is also designed to be used to find patterns associated with specific permissions that are of interest to a developer. SOM+K-means provides a pattern-recognition engine to aid developers in examining various permission usage patterns. So, we investigate the permission use for different categories of apps. Furthermore, we assess the scalability of SOM+K-means as well as the generalizability of the detected usage patterns to possible malware detection using Support Vector Machine (SVM). Our findings reveal that, across a wide range of apps in different categories, the detected usage patterns via SOM+K-means improve the malware detection model's effectiveness. The following is a brief summary of the paper's significant contributions:

1. Using an adapted combination of deep learning and the K-means clustering algorithm, we provide a novel strategy for mining deep-layer permission usage patterns.

2. We create and mine a big dataset of over 16K Android applications from the Google Play Store, investigating around 46 categories and studying their use of 103 permissions.

3. We assess our approach's efficacy by examining the coherence and generalizability of the identified patterns. The results reveal that our method was able to discover a greater number of usage patterns at various degrees of usage cohesiveness.

The remainder of this paper is structured as follow:

We begin with a brief background in Section 5.3. In Section 5.4, we describe the data gathering procedure and the study's objectives. Section 5.5 details our strategy. Section 5.7 summarises the related work. Finally, Section 5.8 concludes and outlines future work.

5.3 Background

5.3.1 Permission System

In a pessimistic scenario, all Android applications are considered to be implicitly buggy or malicious. The apps run in a process with a restricted user ID and are able to access their own files only by default. If a given application requires information or resources outside its sandbox, the permission must be explicitly requested. Permission may be granted automatically by the system, or the system may request the user to grant permission. Each Android application defines an XML-formatted file (Android Manifest.xml), which, along with other metadata such as minimal OS version requirements, contains the permission declarations to which it is requesting access (Etud.iro.umontreal, 2021). The required permission attributes are used to declare permissions in the manifest, which is supplemented by a common namespace. For Google-defined permissions, this is usually Android.permission. Applications can demand self-declared permissions, while component permissions are identified by their tag names.

The Android manifestation includes entries automatically generated by the developer environment. However, some fields must be inserted manually, particularly those relating to permission declarations (Barrera *et al.*, 2010).

Android's permissions are classified into four levels of protection, as follows:

1. Normal (lower-risk permission, which grants demanding applications access to isolated application level features).
2. Dangerous (higher-risk permission, which grants a demanding application access to control the device or private user data).
3. Signature (permission is granted only if the declaring application and requesting application have been done with the same certificate).
4. SignatureOrSystem (A permission that the system only allows to apps that are in the Android system image or are signed with the same certificate as the app that declared the permission).

At runtime, Android apps enforce permissions, but at install time, the user must accept permissions. When a new application is installed by users in Android (regardless of how the application is obtained), the application prompts users to accept or deny the permissions requested. On Android 5.1 or earlier devices, application permissions are all required or all denied, which means that users have no choice. They can either accept all permissions or refuse the application altogether. In the latter case, they cannot use the application at all, because they did not agree with certain permissions.

Since version 6.0 of Android, however, users are able to grant permissions while running applications. This means that permission is no longer required to be granted during the initial installation of an application. Version 6.0 (update) has provided users with improved functionality and control over their applications. It gives them the possibility to revoke app permissions at any time and one by one via the application's setting interface. For instance, a user might choose to grant a particular mode of transport application access to the location of their device, while rejecting access to their contact list or SMS services. Tables 5.1 and 5.2 describe permission protection levels, including dangerous permissions.

Table 5.1 Level of permission protection
Taken from (Developer.android.com, 2021a)

Level of protection	Description
Normal	A reduced risk that allows isolated application rights level features to be enabled while posing minimal danger to other applications, the system, or the user is available.
Dangerous	A higher-risk permission that A higher-risk permission that grants a requesting application access to sensitive user data or control over the device, both of which might have a detrimental impact on the user.
Signature	A permission that the system will only issue if the seeking application is signed with the same certificate as the one that declared the permission.
SignatureOrSystem	A permission that the system only allows to apps that are in the Android system image or are signed with the same certificate as the app that declared the permission.

5.3.2 Clustering model

Self-organizing map: SOM (Kohonen, 2001) is an unsupervised learning network architecture in the area of machine learning. It is able to map high-dimensional data onto a two-dimensional space usually defined as a map. The map is given as the set of nodes within the input space field. This mapping indicates the similarity between the input patterns as the proximity to the map. It offers an understandable methodology to capture and classify the permissions of Android apps. Each SOM node is associated with a weight vector that has the same size as the input vector. The learning algorithm repeats over the input vectors and adjusts the weight vectors in accordance with what the algorithm pulls in. For each input vector, the equivalent weight vector is chosen and manipulated to be more like the original. Further, the neighbours of the best-matched weight vector are also modified using a learning algorithm. This helps ensure convergence over several iterations.

Table 5.2 Dangerous permissions and their related groups in Android 6.0
Taken from (Developer.android.com, 2021a)

Permission Group	Permissions
CALENDAR	READ_CALENDAR WRITE_CALENDAR
CAMERA	CAMERA
CONTACTS	READ_CONTACTS WRITE_CONTACTS GET_ACCOUNTS
LOCATION	ACCESS_FINE_LOCATION ACCESS_COARSE_LOCATION
MICROPHONE	RECORD_AUDIO
PHONE	READ_PHONE_STATE CALL_PHONE READ_CALL_LOG WRITE_CALL_LOG ADD_VOICEMAIL USE_SIP PROCESS_OUTGOING_CALLS
SENSORS	BODY_SENSORS
SMS	SEND_SMS RECEIVE_SMS READ_SMS RECEIVE_WAP_PUSH RECEIVE_MMS
STORAGE	READ_EXTERNAL_STORAGE WRITE_EXTERNAL_STORAGE

In 2000, Vesanto and Alhoniemi (Bengio *et al.*, 2000) proposed using SOM for data clustering in order to achieve better results and reduce computing time. In 2013, Bacao et al. (Abaei *et al.*, 2013) reported that SOM can be utilized instead of K-means for data clustering. In recent years, study and implementation in similar fields has shown that

Table 5.3 New or changed permission groups
in Android 8 (marked with (*))
and 9
Taken from (Developer.android.com, 2021a)

Permission Group	Permissions
CALL_LOG	READ_CALL_LOG, WRITE_CALL_LOG, PROCESS_OUTGOING_CALLS
PHONE	READ_PHONE_STATE READ_PHONE_NUMBERS(*), CALL_PHONE, ANSWER_PHONE_CALLS(*), ADD_VOICEMAIL, USE_SIP, ACCEPT_HANDOVER

SOM and K-means can be merged to construct a better tool for data clustering (Yu *et al.*, 2020).

Clustering Analysis by K-means Method:

K-means is the simplest of the clustering algorithms. It employs squared error as its criterion (Kirchner *et al.*, 2010). K-means begins with a random initial partition and continues to reassign patterns to clusters on the basis of the similarities between the cluster centres and the pattern(s) until the convergence criteria are met. Patterns would not be reassigned from one cluster to another, as the squared error would then cease to decrease dramatically after a number of iterations.

Silhouette index: Silhouette index (Rousseeuw, 1987) is a highly useful indicator of cluster validity. It refers to methods for the interpretation and evaluation of consistency within clusters of data. The technique provides a sense of how well each object is categorised by displaying a clear picture of how successfully each element is classified. The silhouette value is used to determine how close an entity is to its own cluster in relation to other

clusters (separation). The silhouette varies in accuracy from (-1) to $(+1)$, where a high value means that the object is well-suited to its own cluster and is poorly matched to neighbouring clusters.

5.4 Study Objectives and data collection

Our motivation for this empirical study stems from: i) the absence of a built-in verification system to ensure that no unnecessary permissions are requested, which reduces the attack surface and makes the applications more exposed to security issues; and ii) the poor results of the Google Play Protect[13] system. Indeed, a recent evaluation of the best antivirus software for Androids, performed at the software testing laboratory AV-Test(GmbH, 2020), has reported that the Play Protect system detected 76.4% of threats in September 2020 (GmbH, 2020).

Our main goals are the following: (1) To clarify the permission system use in different categories of Android applications, and (2) to investigate the potential risk for these applications to be harmful. In order to achieve our goals, we started by collecting our dataset and labeled the data with respect to the dangerousness of the required permission and the harmfulness risk of the application. In the following, we describe how we built the dataset used in our study.

5.4.1 Data collection

For our data collection, we used the AndroZoo repository, which contained over 14,560,903 apps at the time we accessed it. The AndroZoo repository proposes data on the APKs it archived in a main CSV file containing important information for each application, including hash keys (such as sha256, sha1, md5), size information (for APKs and DEX), date of binary, package name, version code and market place, as well as information about how well the app fared on the VirusTotal website (number of antiviruses that flag the app as a malware, scan date).

[13] www.android.com/play-protect/

In this section we explain the procedure that we followed to create our two datasets. Figure 5.1 illustrates the overview of collecting and building the data. The starting point was downloading the information file for the AndroZoo repository, targeting the apps from Google Play Store from 2019 and 2020. Then we randomly selected our 16K samples. Each app from AndroZoo has its info (i.e., sha256, sha1, md5, apk size, dex size, dex date, pkg name, vercodevt detection, vt scan date, markets). Next, we deployed the information from the AndroZoo for each app to download its APK file and HTML page. However, a significant number of those apps were removed from the Google Play Store for policy reasons. This prompted us to search for it on mirror sites.

In finding the desired mirror site, however, we faced several issues, including language and difficulties downloading the html page automatically (no pattern used). To address these issues, we conducted extensive research and experiments to download the HTML pages automatically. By the end of this step, we had collected APK files and their HTML pages. The experimental dataset numbered 15,894 samples and 103 features (permissions). Clustering is an unsupervised process, so there is no need to know the class label of the samples. However, in order to check the efficiency and consistency of the clustering model, we need to know the class labels of the experimental cases, i.e., we must differentiate between "benign" and "malware" so that we can distinguish malware.

5.4.1.1 Feature Extraction

We used the info related to the 15,894 samples to download their APK files from the AndroZoo repository. We then used these files as input to Apktool[14] (reverse engineer tool) and obtain the manifest file. Next, we modified AndroVul (Namrud *et al.*, 2019b). We employed the info related to those 15,894 to download their apk files from the AndroZoo repository, after which the apk files were used as input to reverse-engineer and obtain the manifest file. To do so, we accessed Apktool, and then modified AndroVul (Namrud *et al.*,

[14] https://ibotpeaches.github.io/Apktool/

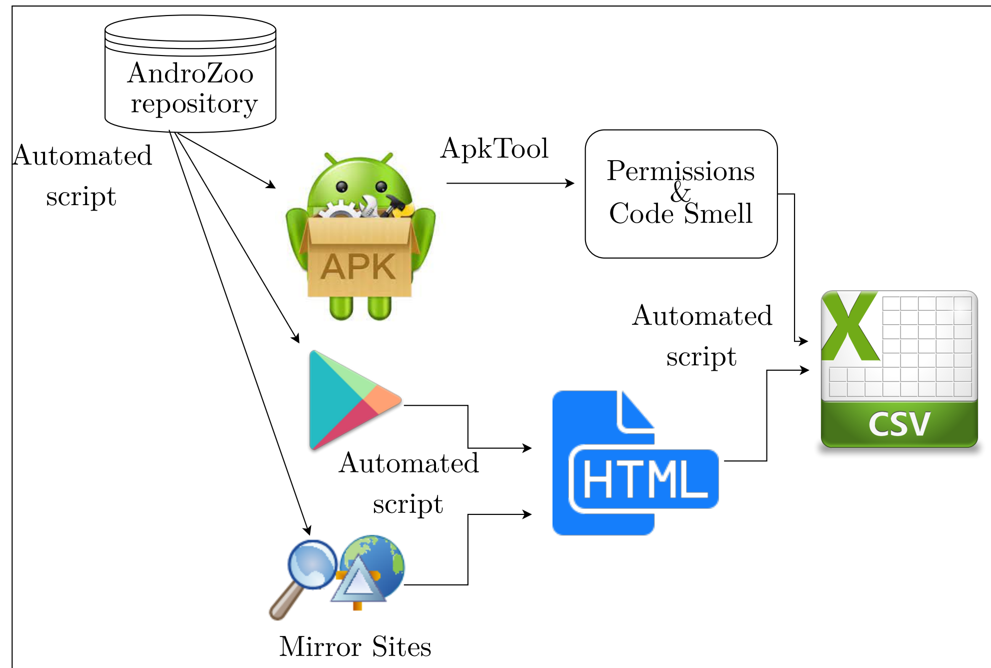


Figure 5.1 Overview of Building the dataset

2019b) to extract all the relevant features, including 103 permissions in the manifest file. In the meantime, HTML pages were deployed to parse metadata and extract the desired features, such as category, rate, date of update, number of downloads, etc. This step resulted in a dataset that contains around 16K samples in a CSV file, including 118 features from different sources. Table 5.4 explains the dataset contents. Currently, there are 103 features (permissions) for the learning system. Algorithm 5.1 explains the procedure for extracting and mapping the features in our dataset. We labeled the permission list to distinguish between Dangerous, Normal, and Signature permissions according to the protection level reported in the Android documentation. We also used different tags to distinguish between permissions giving access to hardware and those giving access to user information in order to investigate the differences in terms of permission use between different categories of applications.

Algorithm 5.1 Feature Extraction

Input : Android Applications, (Apk files & HTML pages & AndroZoo info) Dataset D
Output : A CSV file contains encpoded feature vectors for each App in the dataset

```
1  for (all f ∈ D) do
2      APK_file ← Open(f)
3      manifest_File ← ApkTool_APK_file
4      Permissions_list ← Get_Distinct_Permissions(manifest_File)
5      Metadata_list ← Get_Distinct_Metadata(HTML_File)
6      for (each_permissions ∈ Permissions) do
7          if permissions ∈ Permissions_list then
8              Vector_(Permissions) ← 1
9          end
10         else
11             Vector_(Permissions) ← 0
12         end
13     end
14 end
15 CSV_(file) ← Append(CSV_(file), Concat(Vector_(AndroZoo_info),
     Vector_(Metadata), Vector_(Permission)))
16 return (CSV_(file))
```

Table 5.4 The dataset contents

Source	Feature's name	Feature's type
AndroZoo info	Package_ name	String
	SHA256	String
	SHA1	String
	MD5	String
	Apk _Size	Integer
	VT- Detection	Integer
	Date of scan	Date
	Dex_size	Integer
	Vercode	Integer
APK file	Permissions requested 103 features	Binary
HTML page	Category'name	String
	App's installs	Integer
	Updated_date	Date

5.4.1.2 Applications Categories

When a developer releases an application on Google Play Store, he/she is required to specify the category for the application's release. Currently, Google Play Store has around 46 categories. The distribution is shown in the dataset in Table 5.5. Applications are sorted within each category depending on a range of factors, such as ratings, reviews, downloads, country of origin, etc. We have done an exhaustive analysis and found that the number of malwares is not standardised across all categories. Certain categories such as education, entertainment, games, and tools are particularly vulnerable to malware, while others such as Word, comics, and events are slightly safer from security threats. In our research, we purposefully look for ways to better leverage this knowledge.

5.5 Proposed Approach

In this section, we introduce our approach and the methodology based on mining permission usage patterns of apps from different categories. Before delving into the algorithm, we present a brief background, an overview of our method, and a description of our experiments for investigating the identified permission usage patterns. Figure 5.2 shows the overview of the procedure of producing inferred pattern.

5.5.1 Approach overview

Our technique begins with a collection of apps and a diverse range of permission schemes collected from their apk files. The output is a collection of permission usage patterns, each of which is a collection of apps arranged into distinct layers based on their frequency of co-use. We define a pattern of app co-usage as a collection of applications that are frequently used in conjunction with each other. A pattern is a collection of permissions that are dispersed over many usage cohesion layers. A cohesion layer reflects the frequency of co-use between apps. Indeed, similar permission usage patterns may exist across specific apps, and those apps are more typically classified as belonging to the same category. As

Table 5.5 The distribution of Benign & Malware App Categories in the dataset

Category	***Benign***	***Malware***	***Category***	***Benign***	***Malware***
Action	243	32	Maps	101	11
Adventure	206	10	Medical	148	29
Arcade	266	32	Music	938	165
Art	158	64	News	370	68
Auto	105	34	Not found	193	97
Beauty	63	13	Parenting	12	3
Board	104	9	Personalization	651	62
Books	755	138	Photography	237	51
Business	671	96	Productivity	422	60
Card	80	9	Puzzle	423	27
Casino	48	8	Racing	95	12
Casual	327	42	Role playing	233	26
Comics	50	5	Shopping	273	51
Communication	292	44	Simulation	269	19
Dating	27	7	Social	177	48
Education	1183	246	Sports	331	55
Entertainment	839	201	Strategy	122	14
Events	39	7	Tools	994	345
Finance	482	62	Travel	266	38
Food	201	30	Trivia	53	12
Health	243	55	Video players	115	33
House	64	23	Weather	40	12
Libraries	26	8	Word	32	1
Lifestyle	411	101			

a result, we are looking for an approach that can record co-usage relationships between permission usage patterns and app categories at various levels.

Our approach is as follows:

The input dataset is analysed to identify the various permissions that are unique to each app. Every application in the dataset is assigned a usage vector that contains information about used permissions. We aggregate the apps that are most commonly co-used by permissions using the K-means clustering algorithm based on the SOM deep learning

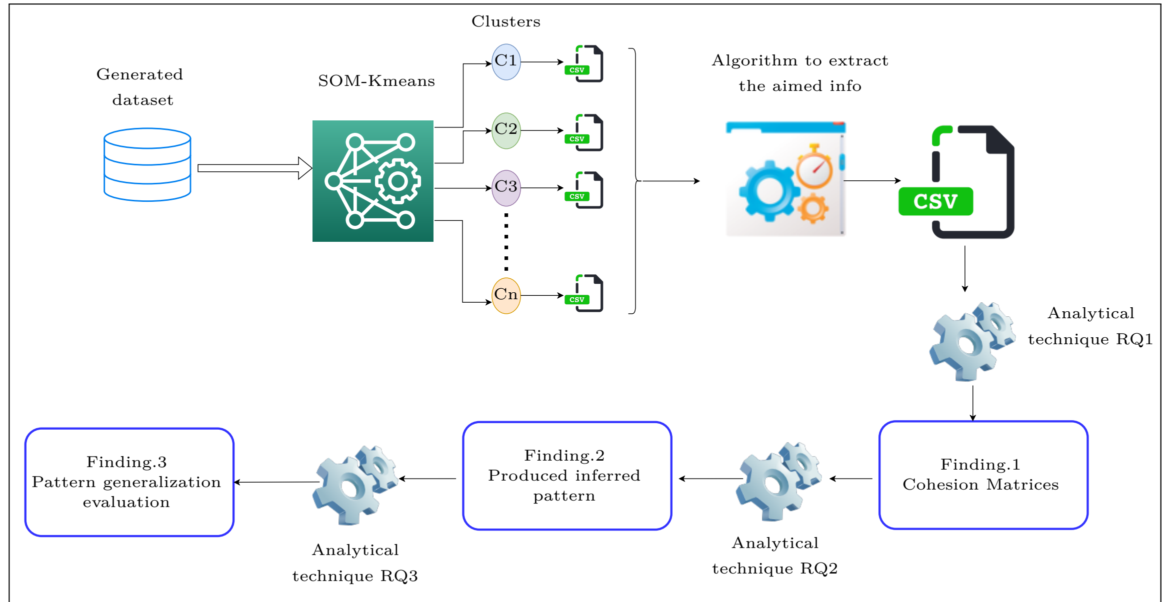


Figure 5.2 Overview of the procedure of producing inferred pattern

cluster. Permissions that are not consistently used across apps in a category are segregated and treated as noisy data.

5.5.2 Deep-layer clustering

Our study aims to investigate the use of permissions, especially dangerous ones, in Android applications and their prediction potential for risk (malware). More specifically, we seek to understand and identify the weaknesses of the Android permission model. Although the various techniques of analysis and data mining are certainly applicable, we build a cluster model that combines the clustering of SOM and K-means centred on the silhouette index, which is a cluster validity measure. The model inherits SOM's advantage (unsupervised deep learning) and K-means clustering is applied to the SOM results, addressing one of the drawbacks (nodes with questionable clustering boundaries) of SOM. Furthermore, the findings do not always yield a simple clustering due to the number of initial nodes and the order of cases. The silhouette index is used by the model to assess the validity of various clustering outcomes. As previously mentioned, we suggested a two-stage strategy clustering approach to improve grouping accuracy.

SOM is a technique for mapping high-dimensional data to a low-dimensional space for easy understanding. SOM technique relies on the following parameters:

1. Weight values are initialised with random numbers.
2. Every neuron calculates the squared Euclidean distance between the vector being processed and its weight vector, which is a measure of the difference between the input pattern and the neuron's output.
3. The winning unit is the one that best approximates the input (the best matching unit). This formula is used for distance calculation, as follows:

$$Dist = \sqrt{\sum_{i=0}^{i=n} (Vi - Wi)^2} \tag{5.1}$$

   Where V is the current input vector and W is the node's weight vector. We take a set of inputs and measure the absolute difference between them and the neuron. Then we square the difference and sum the results. The winner will be the node that yields the smallest square root.

4. A topological neighborhood of excitable neurons appears around the winning node. The topological neighborhood model looks like this:

$$Tj, I(x) = \exp\left(-S^2 j, I(x)/2\sigma^2\right) \tag{5.2}$$

   Where Sj, I is the lateral distance between two neurons $(j \& I)$, $I(x)$ is the winning neuron, and σ is the neighborhood size. The neighbourhood radius in an SOM must reduce over time and must be accomplished using an exponential formula. All excitepd neurons change their weight vectors values to align with the input patterns. The weight vectors of the winning unit are shifted closer to the input, and we change the weight vectors of the units in its neighbourhood, but to a smaller degree. The farther the unit is from the best matching unit, the less it is changed. The weight update formula used in this work is given below:

$$\Delta Wj, i = \eta(t) * Tj, I(x)(t) * (xi - wj, I) \tag{5.3}$$

Where $\eta(t)$ is the learning rate, $Tj, I(x)(t)$ is the topological neighborhood , t is an epoch, i is neuron, j is another neuron, and $I(x)$ is best matching unit; Hence, this denotes the winning neuron.

The K-means algorithm is used in the second stage for cluster analysis by assigning the correct number of (K) clusters. The goal is to identify the distinct pattern in the data to find the smallest possible difference between the attributes in the same classes. We propose integrating the SOM and K-means approaches into the SOM+K-means architecture, as shown in Figure 5.3. K-means is very commonly used in machine learning. In our study, the K-means algorithm is used to obtain the best clustering results. The key idea is to identify K centroids, one for each cluster. The basic K-means algorithm randomly selects the centroid from the application list. After that, each item is placed according to its centroid in a dataset. The K-means clustering partitions a dataset by reducing the total cost function of the squares.

$$J = \sum_{j=1}^{k} \sum_{i=1}^{x} \|Xi^{(j)} - Cj\|^2 \tag{5.4}$$

Where $\|Xi^{(j)} - Ci\|^2$is a chosen distance measure between an application $Xi^{(j)}$ and the cluster center, and Cj is a measure of the distance between applications and their cluster centroids (Ferdous *et al.*, 2009). We separate the applications into K clusters, so the application will be allocated to the one which is the smallest distance between K clusters. As a result, our SOM4+K-means builds the clusters based on improving overall average value of the silhouette index (the closer to 1, the better). Thus, we aim to increase the overall average silhouette. In order to help the SOM+K-means model succeed in its search, we tuned the K parameter in the K-means to gain a more qualitative interpretation of the acquired data. In so doing, we noted that (K = 250) led to an overall average silhouette

of 99.4% and 250 clusters. Each resulting cluster was saved as an CSV file, including identified permission usage patterns, apps, and their info from the main dataset.

5.5.3 Clusters analysis

This process generates clusters of permissions that are constantly used in conjunction with one another, as well as several noisy points that are omitted. We extract the use vectors of each generated cluster using logical disjunction in a single use vector. Each produced cluster's vector contains the name of the cluster, some statistical info, the permission usage pattern, and the number of apps per category. Algorithm 5.2 briefly explains the process of the produced results that were saved on one CSV file. This file will be exploited as a starting point to obtain the rest of the findings.

Algorithm 5.2 Clusters Analysis

Input : $Cluster_i$, $i = 1, 2, \cdots, 250$, all Clusters.
Output : CSV_{file}.
1 **for** $app \in Category_{apps}$ **do**
2 $\quad cat_i = group_category(app)$
3 $\quad Cluster_i = get_All_info_{(Cluster_i)} \oplus pattern_permissions_{Cluster_i}$
4 $\quad CSV_{(file)} \leftarrow Append(CSV_{(file)}, Concat(Vector_{Cluster_{(i)}})$
5 $\quad$ return $(CSV_{(file)})$
6 **end**

5.6 Empirical study

We describe the findings from our study of the proposed methodology of SOM+K-means in this section. Our aim is to determine whether SOM+K-means can recognise usage patterns of applications that are 1) coherent enough to provide useful information for the relevant apps, and 2) generalizable for permission usage patterns. To do so, we investigate the correlation between the resulting clusters and the permission usage patterns. We also investigate the permission patterns deployed to calculate the potential malware vector to train Support Vector Machine (SVM) and validate the enhancement of malicious

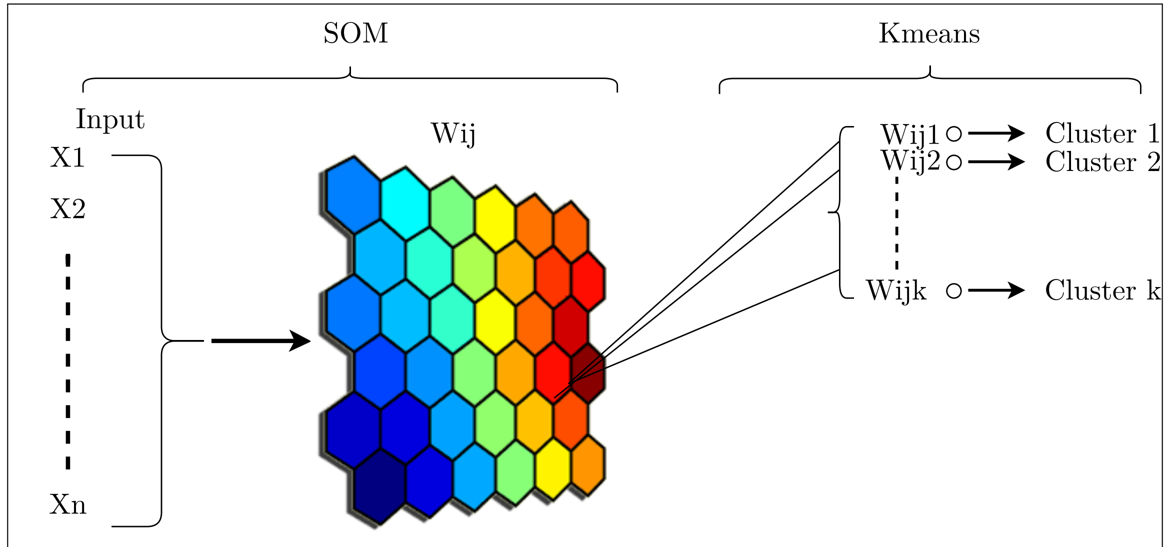


Figure 5.3 Architecture of the SOM-Kmeans model

application detection. For each experiment in this area, we present the study issues, the method used to address them, and the resulting findings.

5.6.1 Analysis of Cohesion

As an initial experiment, we assessed the cohesion of the cluster's quality identified by SOM+K-means for various matrices, including the silhouette index metric, the Pattern Usage Cohesion (PUC) metric, and the Category Cohesion (CC) metric. We intend to answer the following research question:

RQ1. What is the quality of each resulting pattern and the correlation between its apps?

5.6.1.1 Analytical technique

Firstly, the similarity between an object and its own cluster has to be measured. Thus, we utilise a cohesiveness metric, namely the silhouette index metric. The silhouette value ranges between [-1,1], with a high value indicating that the object has a high affinity for its own cluster but a low affinity for neighbouring clusters. The silhouette index is calculated as follows:

For data point $i \in Ci$ (data point i in the cluster Ci), where $d(i,j)$ is the distance between data points i and j in the cluster Ci. We are able to interpret $a(i)$ as an indicator of how successfully i is assigned to its cluster (The better the assignment, the lower the value).

$$a(i) = \frac{1}{|C_i| - 1} \sum_{j \in C_i, i \neq j} d(i,j) \tag{5.5}$$

We then define the mean dissimilarity $b(i)$ of point i to some cluster C_k as the mean of the distance between i to all points in C_k (where $C_k \neq C_i$). For each data point $i \in C_i$.

$$b(i) = min_{k \neq i} \frac{1}{|C_k|} \sum_{j \in C_k} d(i,j) \tag{5.6}$$

We now define a silhouette (value) of one data point i

$$s(i) = \frac{b(i) - a(i)}{max\{a(i), b(i)\}}, if |C_i| > 1 \tag{5.7}$$

Thus, the $s(i)$ over all the data in the entire dataset provides a measure for the data's clustering accuracy.

Next, we must determine whether the identified patterns are sufficiently coherent to reveal informative co-usage links between individual apps. As a result, we use a metric for cohesiveness called Pattern Usage Cohesion (PUC), to quantify the cohesion of the detected patterns. PUC was originally utilised for the cohesive utilisation that was inspired by Perepletchikov et al (Perepletchikov *et al.*, 2007). It assesses the uniformity of co-use of an ensemble of entities, which in our context corresponds to a number of applications in the form of a used permission model. The range of PUC values is [0,1]. The greater the PUC number, the stronger the usage cohesion, i.e., a usage pattern shows optimal usage cohesion (PUC=1) if all permission patterns are always utilised together. If p is a pattern of permission usage, then its PUC is defined as follows:

$$PUC_p = \frac{\sum_{pp} ratio_used_apps(p,pp)}{|perm(p)|} \in [0,1] \quad (5.8)$$

Where pp denotes a permission that contains the pattern p, and the ratio_used_apps(p,pp) means the ratio of permissions that include the pattern. p and are used by each app. The perm(p) defines the set of all permissions that are used in the pattern p.

The last metric, Category Cohesion (CC), measures the ratio of apps that belong to the same category in each cluster. The CC confidence interval is [0,1]. The higher the CC number, the stronger the CC for each category $Cat(i)$ in the same cluster. Thus :

$$CC_{C_i} = max_{cat_i} \frac{|Apps(Cat_i, C_i)|}{|Apps(C_i)|} \quad (5.9)$$

Where the ratio of used $Apps(Cat_i,C_i)$ denotes the number of app clusters (C_i) that belong to the same category, and $Apps(C_i)$ denotes the total apps in the cluster (C_i).

The analysis results of the three quality matrices are presented in the following subsection.

5.6.1.2 Results for RQ1

The cohesion of the three quality matrices is calculated based on the results of overall average silhouette. Table 5.6 reports the silhouette cohesion matrix measuring the quality cohesion for each cluster. From the table, we observe that an average silhouette score is (98.1%) and standard deviation value is (0.1). These values are realistic, because our clustering is based on the silhouette index, which is already high. Further, these values reflect the co-usage relationships of the apps' patterns, making them more cohesive.

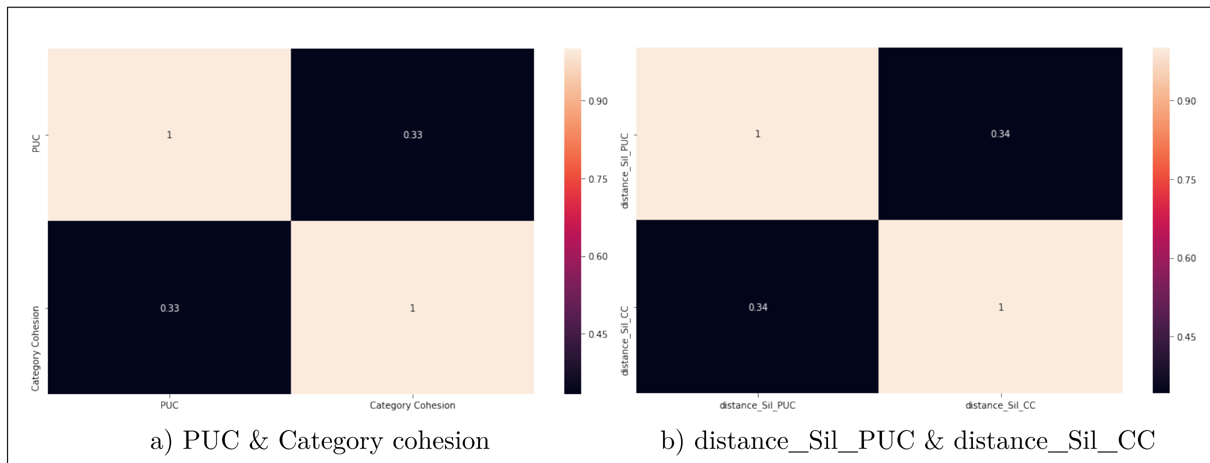
The PUC outcomes also provide evidence that SOM+K-means exhibits consistent cohesion with regard to the identified usage patterns. We found that at least 50% of the applications are used together with high PUC. A noteworthy number of the apps have 100% PUC. For

example, an average PUC would be 60% and a standard deviation value (0.2). As well, the category cohesion matrix carried out the qualitative aspect of the obtained results. From it, we observe that an average 40% of the clusters contain apps that belong to the same categories. Indeed, it is worth mentioning that we observed a trade-off between usage cohesion of detected patterns and their distributed categories of apps.
Next, to acquire a better understanding of the correlation of the findings between cohesion matrices with respect to the silhouette matrix, we calculated the distance between the silhouette matrix and each PUC and CC matrix. This resulted in two new matrices: (distance_Sil_PUC) and (distance_Sil_CC).

Figure 5.4 shows the correlation between the cohesion matrices on each axis. As can be seen, the correlation ranges from -1 to $+1$. Values closer to zero mean that the two cohesion matrices show no linear trend. The closer to 1 the correlation is, the stronger their correlation; in other words, as one increases, so does the other. Thus, the closer to 1, the stronger the relationship is. A correlation closer to -1 indicates similarity, However, rather than both rising, one variable will drop as the other increases. The diagonals are all 1 (light), since the squares relate each variable to themselves. Our motivation here is to study the correlation between the cohesion matrices in order to see the relation between them and possibly to discard some of them. Based on this motivation, we investigated the correlation between the PUC and CC matrices, and that between (distance_Sil_PUC) and (distance_Sil_CC). Figure 5.4a provides the correlation between the PUC and CC matrices, while Figure 5.4b shows the correlation between (distance_Sil_PUC) and (distance_Sil_CC). It worth noting that both correlations yield very close results. As well, Figure 5.5 shows the correlation between the clusters and cohesion matrices. We observe that the correlation result is not sufficiently close to be useless and not far enough away to be independent. Hence, it is important to consider all cohesion matrices. The presence of correlation implies the absence of a linear relationship that demonstrates the quality. From this, we can assume that cohesion matrices assess inferred patterns from various perspectives.

Table 5.6 SOM-Kmeans average cohesiveness and summary of inferred usage patterns

	SIL	PUC	CC
Avg	0.981	0.6	0.4
StdDev	0.1	0.2	0.2
Nb Pattern	250		



a) PUC & Category cohesion b) distance_Sil_PUC & distance_Sil_CC

Figure 5.4 The correlation between the quality matrices

5.6.2 Produced inferred pattern

The purpose of this study is to determine the reliability of the permission usage patterns detected using SOM+K-means. We seek to answer the following research question:

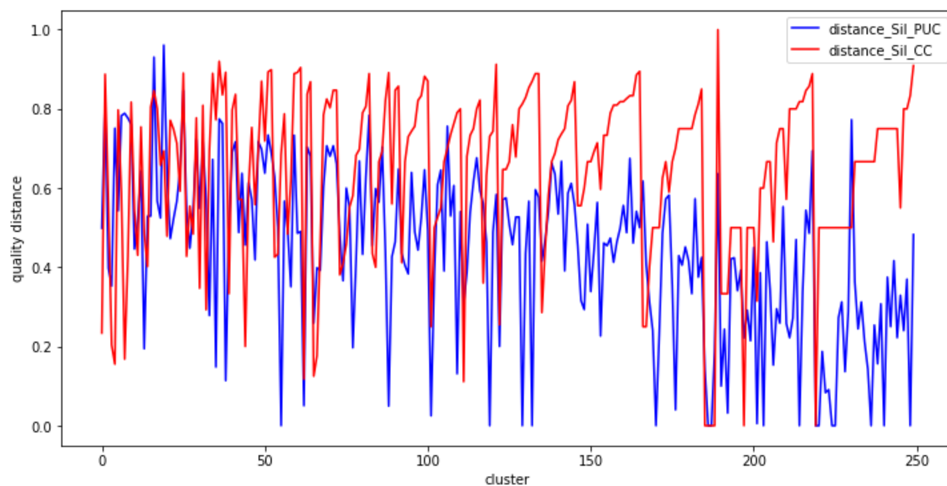


Figure 5.5 Overview of the quality matrices cohesion

RQ2. How far does the concept of cohesive matrices go in obtaining representative permission usage patterns?

5.6.2.1 Analytical technique

To address our second research question (RQ2), we study whether the patterns are representative of the permission usage by applying two selective thresholds to maintain a high level of usage cohesion.

First Selective Threshold: Based on PUC matrix, we calculate the median for the inferred patterns in our case ($Median = 0.42$). Our motivation for applying this threshold concept is as follows: We believe that the median for the PUC matrix is far enough away to include a valuable pattern. In other words, the median covers the patterns that have sufficient quality and generate a sufficient number of patterns. Thus, the median is considered the threshold, and the second cut follows this criterion. This step resulted in representative permission usage patterns, such that the representative permission usage pattern $\geq Median$.

Second Selective Threshold: To perform this step for each cluster, we only consider apps that belong to the same category and have the highest value. Thus, based on the app categories ($Apps_{Cat_i}$) and Category Cohesion (CC) matrix, we calculate the number of ($Apps_{Cat_i}$) in each cluster (C_i).. Then we calculate the average for the ($Apps_{Cat_i}$) matrix. Our motivation is to apply the average as the threshold. In so doing, we observe that the average is not particularly high when compared with total apps per cluster. This observation leads us to remove some clusters, even though this may badly impact our study.
As well, the average is not sufficiently small to be not representative enough; so, based on this motivation, the average was chosen as the first threshold. According to this criterion, the first cut was applied, leaving 58 clusters remaining. After this, each cluster was assigned to the more representative app category. In other words, we selected the

category with the highest percentage of apps to represent the cluster's pattern, as follows. Category pattern = $Max\ (Apps_{Cat_i}) \in (C_i)$. In this step, we are logically motivated.

5.6.2.2 Results for RQ2

The obtained results are as follows. The analysis study provides 30 representative permission usage patterns, including 12 different categories. Some of the categories have more than one pattern. This step resulted in a dataset of inferred patterns. Figure 5.6 shows the statistical distribution for the cohesion matrices and provides additional information about selective criteria.

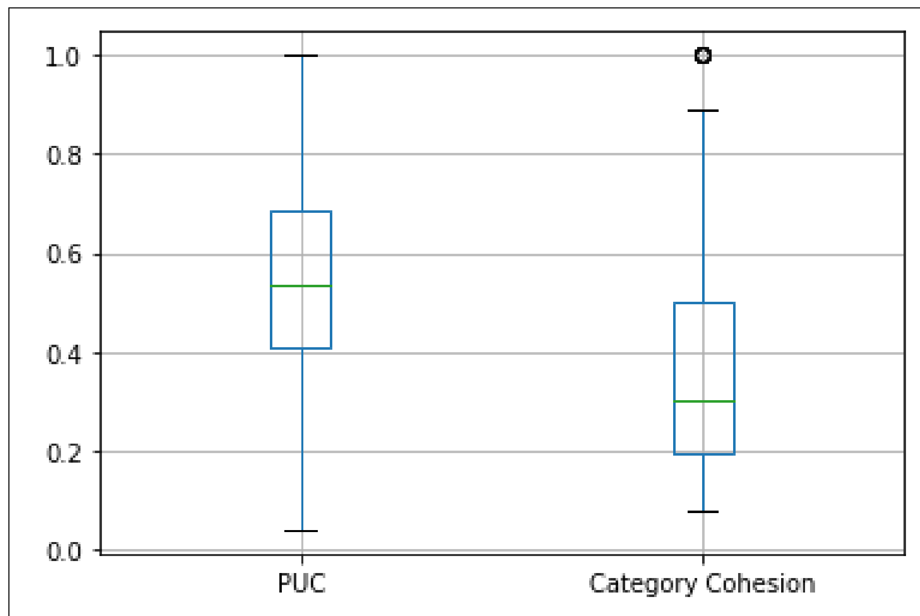


Figure 5.6 PUC & Category cohesion results of the identified permissions usage patterns.

5.6.3 Pattern generalization evaluation

In this study, our objective is to evaluate whether the representative permission usage patterns identified with SOM+K-means can be generalizable in terms of being able to

identify malicious and benign apps, which would then validate our work. Our goal is to address the following research question.

RQ3. To which extent are the discovered permission usage patterns consistent enough to increase the ability to distinguish between malware and benign apps?

5.6.3.1 Analytical technique

To answer RQ3, we look at whether the discovered patterns will be sufficiently consistent to aid in the differentiation of malicious and benign applications, and thus evaluate their generalizability. We address RQ3 through the following experiment: The inferred patterns are used as references to calculate the distance between each pattern's category P_{cat_i} in the inferred pattern dataset with patterns for the same category in the main dataset $P_{main_{cat_i}}$. We called this new set potential malware (PM). Hence, $PM_i = min_i|P_{cat_i} - P_{main_{cat_i}}|$. Our motivation here is to validate representative permission usage patterns and provide evidence of their quality.

Algorithm 5.3 explains the procedure to calculate potential malware (PM). As input 1, $Patterns_i$ is the inferred pattern category, and Apps refer to all apps in our dataset. After the variables are initialized in Line 3, we filter the apps based on their categories. Then the app permissions were compared with inferred patterns, as shown in Line 9. We count the differences and store it in pm_i. If the cat_i has many inferred patterns, we select min pm_i., which means that the pattern has more similarities than the others. The chosen result is then stored in PM, as shown in Line 18. Next, Line 19 is reinitialized to the variables, after which we repeat all the procedures for all the apps. In the end, each app will be mapped with integer values in PM, as follows:

If the value in PM equals zero, the app's pattern is equal to one of the inferred patterns with respect to its category. Otherwise, we count the differences between the app's pattern

and category's inferred patterns. The value with the smallest difference is then assigned to the category.

$$PM_{app} \rightarrow \begin{cases} 0 & ifIdenticalPattern \\ min & otherwise \end{cases}$$

Algorithm 5.3 A potential Malware (PM)

```
   Input  : Patterns_i , i = 1,2,··· ,n, where set of Patterns.
            Apps = [app, app_2, ..., app_m]
1  p_i is a permission.
   Output : PM list.
2  PM = [ ]
3  count = 0, min = ∞ , pm_i = 0
4  for app ∈ Apps do
5      cat_i = get_category(app) ▷getting all apps for same category.
6      pattern_cat_i = get_category(cat_i) ▷getting the pattern for each category.
7      for p ∈ pattern_cat_i do
8          for perm ∈ app.permissions do
9              if perm ∉ p.permissions then
10                 count = count + 1
11             end
12         end
13         if min < count then
14             min = count
15             pm_i = min
16         end
17     end
18     PM.append(pm_i)
19     count = 0, min = ∞ , pm_i = 0 reinitialize the variables
20 end
```

Study 1: We labelled our dataset based on vt_detection features as benign and malware by applying the Derbin (Arp *et al.*, 2014) standard, which considers apps with 0& 1 flag as benign and apps with ≥ 2 as malware, as shown below.

$$App \rightarrow \begin{cases} Benign & if vt_detection < 2 \\ Malware & if vt_detection \geq 2 \end{cases}$$

Next, the machine learning classifier Support Vector Machine (SVM) was selected, as it has been successfully used in many research-related works. Therefore, in this study, the SVM method will be used to classify and distinguish between benign and malware apps. Also, we aim to validate our inferred patterns in this study. The SVM model is applied as follows:

1. SVM model were fed with the permissions as features.
2. The cross validation is applied 80% in the training phase and 20% in the testing phase.
3. The hyper parameters C& Gamma are tuned in the training phase to fit our data.
4. The model is tested using 20% cross-validation.

Study 2: In this study, we add MP to the dataset as a feature and deploy the same hyper parameters C & Gamma from **Study 1**. Thus, we apply the same model with respect to C& Gamma hyper parameters in order to observe the results under the same conditions.

To assess our model, we used three performance parameters: Accuracy, F1, and AUC. These parameters are frequently used in machine learning to evaluate performance models.

To assess our model, we used three performance parameters: Accuracy, F1, and AUC. These parameters are well-known in machine learning to evaluate the performance models.

1. This denotes the percentage of correctly classified apps: $(TP+TN)/(TP+TN+FP+FN)$(Chicco & Jurman, 2020).

2. F1-Measure: This indicates a performance indicator that takes into account both the precision and recall of the obtained classification: $2 * (Recall * Precision)/(Recall + Precision)$(Chicco & Jurman, 2020).

3. Area under ROC Curve (AUC): This is a measure of the predictive power of the classifier that basically informs us how much the model is capable of distinguishing between classes (benign apps vs malware).

5.6.3.2 Results for RQ3

In Study 2, the training phase results were significantly higher than those in the testing phase, causing overfitting in the model. Thus, we solve the overfitting using the random oversampling technique. Random oversampling is the simplest strategy for balancing a dataset's imbalanced nature. It balances out the data by duplicating minority class samples. The overfitting was solved and the performance in the training phase was almost the same as that in the testing phase.
The obtained results are as follows.

Table 5.7 summarizes the results of the SVM classifier both without using the MP as a feature and including the MP as a feature. We observe that there are improvements in terms of distinguishing between malware and benign apps when we added the potential malware feature. Hence, adding the detected patterns is more informative and creates a notable change in the performance of the model. More specifically, the results from the experiment confirm the above-mentioned findings. We believe that our approach can be achieved and will succeed at improving Android security for developers and users. The adaptation of our variant SOM+K-means method is one of the most important contributions of this work for mining permission usage patterns.

Table 5.7 Comparison between two models (with and without MP)

Gamma = 1		C = 100	
Performance Matrix	Accuracy	F1	AUC
Without PM	93.5	92.9	92.5
With PM	94.1	93.2	94.1

5.7 RELATED WORK

5.7.1 Research related to dataset generation

Numerous repositories have been proposed over the years for the study of mobile apps. Recently, the AndroZoo[15] dataset was released, which includes over 13 million Android apps from Google Play, other stores, and app repositories. The aim of AndroZoo is to build robust app collections for software engineering research. F-Droid onlineF is a repository of free open-source Android apps that have been used in an impressive number of studies. Even more recently, Geiger et al. (Geiger *et al.*, 2018) made available a graph-based database with information (e.g., metadata and commit/code history) on 8,431 open-source Android apps located on GitHub and the Google Play Store. Also notable, although slightly older, is Krutz et al.'s study (Krutz *et al.*, 2015)], with a public dataset centered on the lifecycle of 1,179 Android apps from F-Droid. Arp D et al. (Arp *et al.*, 2014) established the well-known DREBIN dataset, which is comprised of 131,611 applications of benign and malicious software. Samples were obtained in the August 2010 to October 2012 time-frame. To find out whether an application is malicious or benign, each sample was sent to the VirusTotal service to examine the output of ten common antivirus scanners (AntiVir, AVG, BitDefender, ClamAV, ESET, FSecure, Kaspersky, McAfee, Panda, and Sophos). Any application that was scanned by at least two scanners was detected as malicious.

[15] https://androzoo.uni.lu/

Li et al. (Li *et al.*, 2017) built a dataset of 1,497 apps pairs, where one application piggybacks another that may contain malicious payloads. Their work was based on AndroZoo. Using VirusTotal's results, they flagged the relevant malware apps. F. Wei et al. (Wei *et al.*, 2017) prepared a dataset containing 24,650 samples dating from 2010 to 2016 that labeled Android malware. The samples were collected from several sources, including Google Play, VirusShare, and security companies of third-parties. VirusTotal was used to flag their apps. X. Jiang et al. (Zhou & Jiang, 2012) managed to collect around 1,200 malware samples in August 2010 and manually analyzed the malware samples. Wang H et al.built a dataset of 9,133 malware samples and set a threshold of 20 to indicate suspicious applications based on the number of VirusTotal[16] engines recorded.

Size and coverage: Apart from AMD dataset (Wei *et al.*, 2017), the great majority of datasets currently available are limited and obsolete. For example, MalGenome (Zhou & Jiang, 2012) and Drebin (Arp *et al.*, 2014) are two of the most popular datasets. Their production was done five years ago, and only a limited number of samples are included. The literature also reports that the Drebin dataset has a replication issue (Irolla & Dey, 2018). The AMD dataset, which contains a large number of malware samples, was developed in 2016. It includes several samples that overlap with the MalGenome and Drebin projects, since it gathered samples from a broad range of sources, including previously collected malware datasets.

Methods used to flag the ground truth: The rest of the three datasets heavily depend on VirusTotal for accuracy in labelling the ground truth. It is worth noting that various thresholds are utilized on VirusTotal to label malware samples. For example, Drebin was developed based on the findings of ten well-known engines on VirusTotal. At least two of the ten engines found one type of malicious activity in the original sample and flagged it as a malware. As a threshold, one engine was employed in the Piggybacking dataset, while AMD made use of 28 different engines (which, at that time, represented

[16] https://www.virustotal.com/gui/

over 50% of the engines). Furthermore, despite the fact that VirusTotal is commonly used in academia and industry, it contains very little exclusivity.

App Metadata: After looking into the issue, we assert that, to the best of our knowledge, no other studies have focused on metadata (e.g., app description, app ratings, etc.) relevant to malware in their samples. Furthermore, because previous works (Gorla *et al.*, 2014; Ma *et al.*, 2015) have suggested incorporating app metadata for malicious/anomaly detection, we believe it is critical to build a malware dataset containing all of the app metadata to enable malware detection evaluation.

5.7.2 Permissions based study

The permission system has attracted considerable research interest. Several studies have been conducted recently to investigate how permissions are used in Android apps and whether or not they can help identify malware apps. In (Felt *et al.*, 2011), Felt et al.conducted a survey of 100 paid apps and 856 free apps from the Android Market. They identified the most requested permissions and observed that both free and paid apps make requests for at least one dangerous permission. Additionally, they created a tool that is able to detect whether an app requests more permissions than necessary, noting that one-third of the examined applications were over-privileged. In (Barrera *et al.*, 2010), Barrera et al. conducted a survey of the 1,100 most popular applications downloaded in 2009. They discovered that only a small portion of the specified permissions are actively used by developers. In (Wei *et al.*, 2012), Wei et al. investigated the evolution of permissions in the Android ecosystem, finding that dangerous permissions often outnumber other permission types in all Android. Meanwhile, in (Krutz *et al.*, 2017), Krutz et al. also carried out a study on app permissions. They discovered that more experienced developers are more likely to make permission-based modifications, and that permissions are usually introduced earlier in an app's lifetime.

In (Frank *et al.*, 2012), the authors selected 188,389 applications from the official Android market and studied the different requested permission combinations made by them. The

authors identified more than 30 common patterns of permission requests and found that low-reputation applications often diverge from the permission request pattern observed in high-reputation applications.

Other research has focused on defining risk signal as a way to identify malware applications. In (Sarma *et al.*, 2012), Sarma et al proposed a set of risk signals by analyzing the permission patterns in apps taken from the Android Market within a dataset of 121 malicious apps. in (Zhou *et al.*, 2012), Zhou et al. developed a system for detecting malicious applications in official and alternative Android markets.

In (Scoccia *et al.*, 2019), the authors performed an empirical research of 574 open-source Android app GitHub repositories. They examined the incidence of four distinct sorts of permission-related concerns throughout the duration of the apps' lifetimes. Their findings indicate that permission-related difficulties are a common occurrence in Android applications. In (Almomani & Al Khayer, 2020), authors have conducted for the last five years' versions of the top Android apps to examine the Android platform's permissions mechanism. Additionally, the paper addresses Android's user-permissions model, which defines how applications manage sensitive data and resources. In(Xiao *et al.*, 2020), the authors introduced MPDroid, It is a new technique that combines static analysis and collaborative filtering to determine the minimum permissions required for an Android application based on its description and API usage. MPDroid begins by utilising collaborative filtering to determine the app's basic minimal permissions. Then, using static analysis, the final minimal permissions required by an app are determined. Finally, it assesses the danger of over privilege by analysing the app's excess privileges, i.e., the rights sought by the programme that are not essential. Experiments are run on 16,343 popular Google Play applications. In(Wu *et al.*, 2021), the authors manually annotated 2,254 app descriptions from the Google Play Store to include 26 permissions classified into ten categories. They used two natural language processing approaches to enhance our annotated dataset in order to acquire additional permission semantics. In(Arif *et al.*, 2021), the authors proposed a multi-criteria decision-making-based (MCDM) mobile malware detection system that evaluated Android mobile applications using a

risk-based fuzzy analytical hierarchy process (AHP) method. The study focuses on static analysis, which employs permission-based features to evaluate the approach used by mobile malware detection systems. Risk analysis is used to raise the mobile user's awareness when accepting any permission request that carries a high risk level. 10,000 samples were collected from Drebin and AndroZoo for the assessment. The findings indicate a high rate of accuracy of 90.54%. In(Jiang *et al.*, 2020), the authors devised a method for identifying Android harmful applications called fine-grained dangerous permission (FDP), which collects characteristics that more accurately describe the difference between malicious and benign applications. Among these features, for the first time, a fine-grained feature for harmful permissions issued to components is offered. We examine 1700 benign and 1600 malicious apps and show that FDP has a 94.5% TP rate.

Our approach is similar to (Scoccia *et al.*, 2019; Almomani & Al Khayer, 2020; Xiao *et al.*, 2020; Wu *et al.*, 2021) in terms of permission-related concerns, we dissimilar in terms of the dataset (including the size, features, and the number of permissions), using machine learning, and considering the categories'apps in their studies. In our present work, we expand on the existing research. We also investigate similar properties and propose new ones, which we define as application sustainability and malware risk.

5.7.3 Category based study

Apps in Android app stores are classified into various categories, such as Health&Fitness, News&Magazine, Books&References, Music&Audio, etc. Each category has its own set of functionalities, which means that applications in the same category have similar functionalities. Permissions are one of these features. Several state-of-the-art studies make a link between the apps' requested permissions and the features that are standard in its category. Some researchers proposed using category-based machine learning classifiers to improve the efficiency of classification models in identifying malicious applications within a certain category.

Table 5.8 Comparison between various state-of-art solutions

Reference	Generated Dataset	Category based	Identify permissions usage pattern	Machine Learning	Malware detection	Dataset size
(Scoccia *et al.*, 2019)	✓	✗	✗	✗	✗	574
(Almomani & Al Khayer, 2020)	✓	✗	✗	✗	✗	20 apps with their versions (2016 to 2020)
(Xiao *et al.*, 2020)	✗	✗	✗	✗	✗	16,343
(Wu *et al.*, 2021)	✗	10 categories	✗	✓	✗	2,254
(Yuan *et al.*, 2016a)	✗	18 categories	✗	✓	✓	13,005
(Arif *et al.*, 2021)	✗	✗	✗	✓	✓	10,000
(Jiang *et al.*, 2020)	✓	✗	✗	✓	✓	3100
Our work	✓	46 categories	✓	✓	✓	16,000

In (Grampurohit *et al.*, 2014), as a feature, the authors used the category of applications named by Google Play. Their results reveal that by using machine learning technology to detect malicious malware, they used the applications' permissions at app-level. Further, they found that adding the application category feature improves detection efficiency and accuracy. In (Lin *et al.*, 2017), the target consists of both static and dynamic analyses. The static analysis is focused on source code, user permissions and signatures, while the dynamic analysis is based on the behavior of applications in running time. A machine learning algorithm known as OKNN is then used to determine which category an application belongs to. The size of the dataset in that study is 3,600 apps. In (Yuan *et al.*, 2016a) Yuan et al. presented an automated method for categorising Android apps. They conducted experiments with 13,005 applications composed of 18 categories with Naive Bayes. More specifically, in their approach, the malware application publisher can choose an application category at random in order to avoid detection by the application market. As a consequence, a method that can automatically categorize multiple types of apps can be useful for organizing the Android Market as well as identifying malicious applications. Studies show that the addition of an application category will greatly increase the efficiency and accuracy of the detection when using machine leaning technology to detect malicious

apps (Grampurohit *et al.*, 2014). Thus, application category is important for Android malware detection. Several works involved category-based investigations, but for different purposes. The one most related to our work was conducted by Sarma et al. (Sarma *et al.*, 2012). Thus, application category is important for Android malware detection.

Several works involved category-based investigations, but for different purposes. The one most related to our work was conducted by Sarma et al. (Sarma *et al.*, 2012). Their approach is most similar to ours, in that it is also focused on permission use through categories. However, it has a different purpose, with Sarma et al.(Sarma *et al.*, 2012) focusing on the similarities between app permission usage and their categories to distinguish between malware and benign entities. We, on the other hand, are more concerned with the overall app permission usage and in finding requested permission patterns among different categories. Moreover, our work takes into account a different level of granularity than previous works whose approaches infer malware app usage permissions at the category level.

Nonetheless, to improve Android security, Sarma et al.(Sarma *et al.*, 2012) investigated the feasibility of using the permissions that an app requires, the category of the app, and the permissions that other apps of the same category require. They created their 158,062-app dataset in February 2011. The malware dataset consists of 121 apps obtained from the Contagio Malware Dump. Some related work used category as a feature (Grampurohit *et al.*, 2014) in their training model to improve performance, whereas in our case, we are more interested in exploring possible use permissions patterns across whole categories of applications. Previous approaches assumed that the necessary permissions were selected by the developer in advance and that he/she chooses an application category at random in order to avoid detection by the application market (Yuan *et al.*, 2016a). Without using this assumption, our study will meaningfully supplement other research. Indeed, our approach may be used as a preliminary step to infer sets of permissions that are consistently used together, such that existing approaches could be used to learn how to improve the ability to distinguish between benign and malware within the patterns'

permissions and category apps. Our novel findings focus on producing usage patterns of permissions for various categories and on providing in-depth analysis of pattern cohesion and the impact of patterns on malware detection. Table 5.8 shows the comparison between various state-of-art solutions that study the Android permissions system in different purposes.

5.8 Conclusion

With the exponential growth in the number of smartphones being used in services such as banks, hospitals, and m-commerce, smartphone security has become a major concern. The use of unofficial sources to upload applications is likewise concerning. Malicious apps can be used to steal passwords, leak information, and build windows into phones. Existing anti-virus software relies on static signatures that must be modified on a regular basis and are incapable of detecting zero-day malware. The Android permission scheme is the core Android security framework that governs application task execution. Despite recent advancements in research that have provided a variety of approaches and detection methods for locating malware applications, the available literature lacks a comprehensive examination of the topic. We addressed this deficiency in this work by investigating all the larger issues, resulting in two main achievements. 1) We created a huge dataset of malware and benign apps in a systematic and automated manner and made it accessible to the community. 2) We conducted a preliminary analytical analysis of various forms of Android permissions and their potential associations with malicious intents, as well as users' impressions of the nature of the applications that use them.

Our research examined 118 separate features, 103 of which are permissions, on approximately 16K apps. Further, we proposed tentative findings on the ties between the use of Android permissions tagged as unsafe by the permission scheme. Additionally, we introduced a model that combines a self-organizing map (SOM) and K-means clustering. Based on a clustering validity test, we built the resultant SOM+K-means using permis-

sions as features. Our overall achieved purpose was to describe pictures or patterns of how applications in a particular category behave by optimizing our model.

CONCLUSION AND RECOMMENDATIONS

6.1 General conclusion

Third party apps are key drivers of the popularity of Android-run devices but the openness of the Android ecosystem carry risks to its users. These users store sensitive information, conduct confidential activities and complete financial transactions on their smartphones on a daily basis. That state of affairs provides strong incentives for cyber criminals and malware creators in search of systems to hack and compromise. Malware for smartphones are getting more dangerous and difficult to detect. According to Legal-jobs (Vuleta, 2021), malware infections have increased by 87% in the last decade, and Android applications and users accounted for 98% of all malware targets on mobile devices. With the exponential growth and sophistication of malware targeting mobile networks, it is critical to create effective strategies to secure these systems.

An analysis of the literature on the topic revealed that the two major strategies used to detect malware were signature-based techniques and anomaly-based techniques.

In this thesis, we focused on anomaly-based identification strategies. Anomaly-based methods are more resistant to evasion and obfuscation threats, and they can detect unknown malware and variants of known malware. However, these strategies do need additional research and development to enhance prediction performance.

This thesis aims to provide innovative and effective malware identification strategies for Android applications based on their vulnerabilities. In this respect, the major contribution of our thesis are as below:

1. To help detect malware embedded within mobile applications, we proposed AndroVul, a repository for security vulnerabilities. It consists of a sample of apps from the well

known Android app repository AndroZoo, complete with vulnerability information extracted through reverse engineering and diverse static analyses.

2. Building on our generated dataset, we also investigated well-known classifiers and feature selection methods used to detect malware. Experiments with the metrics available in our dataset demonstrated that they can be very successful at detecting malwares and can outperform comparable work, although with notable variations depending on the classifiers selected. Thus, our work offers malware researchers insights into the selection and tuning of classifiers to use while identifying malware.

3. We proposed and tuned two advanced classification techniques, Support Vector Machine and Deep learning and found that they were able to achieve excellent results, outperforming the state-of-art.

4. Finally, we carefully assess permissions requested by apps with respect to these apps' categories and proposed a model integrating self-organizing maps (SOM) and K-means clustering to get insignts into permission usage patterns depending on app categories.

6.2 Articles in peer-reviewed journals and conferences

1. Namrud Z, Kpodjedo S, Talhi C. AndroVul: a repository for Android security vulnerabilities. InProceedings of the 29th Annual International Conference on Computer Science and Software Engineering 2019 Nov 4 (pp. 64-71). Published.

2. Namrud Z, Kpodjedo S, Talhi C, Boaye Belle A. Probing AndroVul dataset for studies on Android malware classification. Journal of King Saud University-Computer and Information Sciences. 2021 Sep 22.(H-indexed and impact Factor is 13.473). Published.

3. Namrud Z, Kpodjedo S, Talhi C, Bali A, Boaye Belle A. Deep Learning Based Android Anomaly Detection Using a Combination of Vulnerabilities Dataset. Applied Sciences. 2021 Aug;11(16):7538. (H-indexed and impact Factor is 2.736). Published.

4. Namrud Z, Kpodjedo S, Bali A, Talhi C. Deep-layer clustering to identify permission usage patterns of Android app categories. IEEE Access. 2022 Mar 2. (H-indexed and impact Factor is 3.367). Published.

BIBLIOGRAPHY

Aafer, Y., Du, W. & Yin, H. (2013). Droidapiminer: Mining api-level features for robust malware detection in android. *International conference on security and privacy in communication systems*, pp. 86–103.

Abaei, G., Rezaei, Z. & Selamat, A. (2013). Fault prediction by utilizing self-organizing map and threshold. *2013 ieee international conference on control system, computing and engineering*, pp. 465–470.

Abraham, A. (2020, March, 12). Mobile-security [Format]. Retrieved from https://github.com/MobSF/Mobile-Security-Framework-MobSF.

Ahmadi, M., Sotgiu, A. & Giacinto, G. (2017). Intelliav: Toward the feasibility of building intelligent anti-malware on android devices. *International cross-domain conference for machine learning and knowledge extraction*, pp. 137–154.

Aittokallio, A. (2021, January, 29). Report: mobile malware infection rate accelerating [Format]. Retrieved from https://telecoms.com/281581/report-mobile-malware-infection-rate-accelerating/.

Al Ali, M., Svetinovic, D., Aung, Z. & Lukman, S. (2017). Malware detection in android mobile platform using machine learning algorithms. *2017 international conference on infocom technologies and unmanned systems (trends and future directions)(ictus)*, pp. 763–768.

Allix, K., Bissyandé, T. F., Klein, J. & Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. *2016 ieee/acm 13th working conference on mining software repositories (msr)*, pp. 468–471.

Almomani, I. M. & Al Khayer, A. (2020). A comprehensive analysis of the android permissions system. *Ieee access*, 8, 216671–216688.

Alqatawna, J., Ala'M, A.-Z., Hassonah, M. A., Faris, H. et al. (2021). Android botnet detection using machine learning models based on a comprehensive static analysis approach. *Journal of information security and applications*, 58, 102735.

Amos, B., Turner, H. & White, J. (2013). Applying machine learning classifiers to dynamic android malware detection at scale. *2013 9th international wireless communications and mobile computing conference (iwcmc)*, pp. 1666–1671.

Arif, J. M., Ab Razak, M. F., Mat, S. R. T., Awang, S., Ismail, N. S. N. & Firdaus, A. (2021). Android mobile malware detection using fuzzy ahp. *Journal of information security and applications*, 61, 102929.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K. & Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. *Ndss*, 14, 23–26.

Barrera, D., Kayacik, H. G., Van Oorschot, P. C. & Somayaji, A. (2010). A methodology for empirical analysis of permission-based security models and its application to android. *Proceedings of the 17th acm conference on computer and communications security*, pp. 73–84.

Baskaran, B. & Ralescu, A. (2016). A study of android malware detection techniques and machine learning.

Bengio, Y., Buhmann, J., Abu-Mostafa, Y., Embrechts, M. & Zurada, J. (2000). Special issue on 'neural networks for data mining and knowledge discovery'. *Ieee transactions on neural networks*, 11(3), 545–822.

Bhatia, T. & Kaushal, R. (2017). Malware detection in android based on dynamic analysis. *2017 international conference on cyber security and protection of digital services (cyber security)*, pp. 1–6.

Bhattacharya, A. & Goswami, R. T. (2017). Dmdam: data mining based detection of android malware. *Proceedings of the first international conference on intelligent computing and communication*, pp. 187–194.

Bhattacharya, A. & Goswami, R. T. (2018). Community based feature selection method for detection of android malware. *Journal of global information management (jgim)*, 26(3), 54–77.

Bose, A., Hu, X., Shin, K. G. & Park, T. (2008). Behavioral detection of malware on mobile handsets. *Proceedings of the 6th international conference on mobile systems, applications, and services*, pp. 225–238.

Calciati, P. & Gorla, A. (2017). How do apps evolve in their permission requests? a preliminary study. *2017 ieee/acm 14th international conference on mining software repositories (msr)*, pp. 37–41.

Canfora, G., Medvet, E., Mercaldo, F. & Visaggio, C. A. (2015). Detecting android malware using sequences of system calls. *Proceedings of the 3rd international workshop on software development lifecycle for mobile*, pp. 13–20.

Canfora, G., Medvet, E., Mercaldo, F. & Visaggio, C. A. (2016). Acquiring and analyzing app metrics for effective mobile malware detection. *Proceedings of the 2016 acm on international workshop on security and privacy analytics*, pp. 50–57.

Catak, F. O., Yazı, A. F., Elezaj, O. & Ahmed, J. (2020). Deep learning based sequential model for malware analysis using windows exe api calls. *Peerj computer science*, 6, e285.

Catak, F. O., Ahmed, J., Sahinbas, K. & Khand, Z. H. (2021). Data augmentation based malware detection using convolutional neural networks. *Peerj computer science*, 7, e346.

Chakradeo, S., Reaves, B., Traynor, P. & Enck, W. (2013). Mast: Triage for market-scale mobile malware analysis. *Proceedings of the sixth acm conference on security and privacy in wireless and mobile networks*, pp. 13–24.

CHEBYSHEV, V. (2021). Mobile malware evolution 2020. Retrieved from https://securelist.com/mobile-malware-evolution-2020/101029/.

Chebyshev, V. & Unuchek, R. (2014). Mobile malware evolution: 2013. *Kaspersky lab zao's securelist*, 24.

Chen, T. M. & Peikari, C. (2008). Malicious software in mobile devices. In *Handbook of Research on Wireless Security* (pp. 1–10). IGI Global.

Chen, W., Aspinall, D., Gordon, A. D., Sutton, C. & Muttik, I. (2016). More semantics more robust: Improving android malware classifiers. *Proceedings of the 9th acm conference on security & privacy in wireless and mobile networks*, pp. 147–158.

Chicco, D. & Jurman, G. (2020). The advantages of the matthews correlation coefficient (mcc) over f1 score and accuracy in binary classification evaluation. *Bmc genomics*, 21(1), 1–13.

Chin, E., Felt, A. P., Greenwood, K. & Wagner, D. (2011). Analyzing inter-application communication in android. *Proceedings of the 9th international conference on mobile systems, applications, and services*, pp. 239–252.

Clement, J. (2021, January, 29). Mobile app downloads worldwide from 2018 to 2024, by store [Format]. Retrieved from https://www.statista.com/statistics/1010716/apple-app-store-google-play-app-downloads-forecast/.

Dai, S., Tongaonkar, A., Wang, X., Nucci, A. & Song, D. (2013). Networkprofiler: Towards automatic fingerprinting of android apps. *2013 proceedings ieee infocom*, pp. 809–817.

Developer.android.com. (2021a, March, 2). Permissions on android [Format]. Retrieved from https://developer.android.com/.

Developer.android.com. (2021b, February, 1). Manifest.permission [Format]. Retrieved from https://developer.android.com/reference/android/Manifest.permission.

Drake, J. (2015). Stagefright: Scary code in the heart of android. *Blackhat usa*, 8.

Dunham, K. (2008). *Mobile malware attacks and defense*. Syngress.

Enck, W., Ongtang, M. & McDaniel, P. (2009). On lightweight mobile phone application certification. *Proceedings of the 16th acm conference on computer and communications security*, pp. 235–245.

Etud.iro.umontreal. (2021, March, 2). Replication package [Format]. Retrieved from http://www-etud.iro.umontreal.ca/~saiedmoh/MobileSoftRP/index.html.

F-droid. (2018, February, 11). android apks [Format]. Retrieved from https://www.f-droid.org/.

Fahl, S., Harbach, M., Muders, T., Baumgärtner, L., Freisleben, B. & Smith, M. (2012). Why eve and mallory love android: An analysis of android ssl (in) security. *Proceedings of the 2012 acm conference on computer and communications security*, pp. 50–61.

Fan, W., Sang, Y., Zhang, D., Sun, R. & Liu, Y. (2017). Droidinjector: A process injection-based dynamic tracking system for runtime behaviors of android applications. *Computers & security*, 70, 224–237.

Felt, A. P., Greenwood, K. & Wagner, D. (2011). The effectiveness of application permissions. *Proceedings of the 2nd usenix conference on web application development*, pp. 7–7.

Ferdous, R. et al. (2009). An efficient k-means algorithm integrated with jaccard distance measure for document clustering. *2009 first asian himalayas international conference on internet*, pp. 1–6.

Forensics, C. (2021, June, 29). Report: mobile malware infection rate accelerating [Format]. Retrieved from https://virusshare.com/.

Fowler, M. (2018). *Refactoring: improving the design of existing code*. Addison-Wesley Professional.

Frank, M., Dong, B., Felt, A. P. & Song, D. (2012). Mining permission request patterns from android and facebook applications. *2012 ieee 12th international conference on data mining*, pp. 870–875.

Gadient, P., Nierstrasz, O. & Ghafari, M. (2017). Security in android applications. *Phd diss., master s thesis. university of bern.*

Gadient, P., Ghafari, M., Frischknecht, P. & Nierstrasz, O. (2019). Security code smells in android icc. *Empirical software engineering*, 24(5), 3046–3076.

Gartner.com. (2021, March, 2). Mobile users will provide personalized data streams [Format]. Retrieved from http://www.gartner.com/newsroom/id/2654115.

Geiger, F.-X., Malavolta, I., Pascarella, L., Palomba, F., Di Nucci, D. & Bacchelli, A. (2018). A graph-based dataset of commit history of real-world android apps. *Proceedings of the 15th international conference on mining software repositories*, pp. 30–33.

Ghafari, M., Gadient, P. & Nierstrasz, O. (2017). Security smells in android. *2017 ieee 17th international working conference on source code analysis and manipulation (scam)*, pp. 121–130.

Gkortzis, A., Mitropoulos, D. & Spinellis, D. (2018). Vulinoss: a dataset of security vulnerabilities in open-source systems. *Proceedings of the 15th international conference on mining software repositories*, pp. 18–21.

GmbH, A.-T. (2020, May, 14). Malware [Format]. Retrieved from www.av-test.org.

Gorla, A., Tavecchia, I., Gross, F. & Zeller, A. (2014). Checking app behavior against app descriptions. *Proceedings of the 36th international conference on software engineering*, pp. 1025–1035.

Gottschalk, M., Josefiok, M., Jelschen, J. & Winter, A. (2012). Removing energy code smells with reengineering services. *Informatik 2012*.

Grampurohit, V., Kumar, V., Rawat, S. & Rawat, S. (2014). Category based malware detection for android. *International symposium on security in computing and communication*, pp. 239–249.

Grano, G., Di Sorbo, A., Mercaldo, F., Visaggio, C. A., Canfora, G. & Panichella, S. (2017). Android apps and user feedback: a dataset for software evolution and quality improvement. *Proceedings of the 2nd acm sigsoft international workshop on app market analytics*, pp. 8–11.

Habchi, S., Moha, N. & Rouvoy, R. (2019). The rise of android code smells: who is to blame? *2019 ieee/acm 16th international conference on mining software repositories (msr)*, pp. 445–456.

Hecht, G., Rouvoy, R., Moha, N. & Duchien, L. (2015). Detecting antipatterns in android apps. *2015 2nd acm international conference on mobile software engineering and systems*, pp. 148–149.

Hecht, G., Moha, N. & Rouvoy, R. (2016). An empirical study of the performance impacts of android code smells. *Proceedings of the international conference on mobile software engineering and systems*, pp. 59–69.

Herong. (2021, January, 30). Android application project build process [Format]. Retrieved from http://www.herongyang.com/Android/Project-Android-Application-Project-Build-Process.html.

Huang, C.-Y., Tsai, Y.-T. & Hsu, C.-H. (2013). Performance evaluation on permission-based detection for android malware. In *Advances in intelligent systems and applications-volume 2* (pp. 111–120). Springer.

Huang, G., Liu, Z., Van Der Maaten, L. & Weinberger, K. Q. (2017). Densely connected convolutional networks. *Proceedings of the ieee conference on computer vision and pattern recognition*, pp. 4700–4708.

IDC. (2021, January, 27). Smartphone market share [Format]. Retrieved from https://www.idc.com/promo/smartphone-market-share/os.

Ikram, M., Vallina-Rodriguez, N., Seneviratne, S., Kaafar, M. A. & Paxson, V. (2016). An analysis of the privacy and security risks of android vpn permission-enabled apps. *Proceedings of the 2016 internet measurement conference*, pp. 349–364.

Iqbal, S., Yasin, A. & Naqash, T. (2018). Android (nougats) security issues and solutions. *2018 ieee international conference on applied system invention (icasi)*, pp. 1152–1155.

Irolla, P. & Dey, A. (2018). The duplication issue within the drebin dataset. *Journal of computer virology and hacking techniques*, 14(3), 245–249.

Jeong, E. S., Kim, I. S. & Lee, D. H. (2017). Safeguard: a behavior based real-time malware detection scheme for mobile multimedia applications in android platform. *Multimedia tools and applications*, 76(17), 18153–18173.

Jiang, X., Mao, B., Guan, J. & Huang, X. (2020). Android malware detection using fine-grained features. *Scientific programming*, 2020.

Jiang, X. (2021, February, 03). Security alert: New nickibot spyware found in alternative android markets [Format]. Retrieved from https://www.csc2.ncsu.edu/faculty/xjiang4/NickiBot/.

Johnson, J. (2021, February, 03). Distribution of leading android malware types in 2019 [Format]. Retrieved from https://www.statista.com/statistics/681006/share-of-android-types-of-malware/.

Karbab, E. B., Debbabi, M., Derhab, A. & Mouheb, D. (2020). Scalable and robust unsupervised android malware fingerprinting using community-based network partitioning. *computers & security*, 97, 101965.

Kirchner, K., Delibašić, B. & Vukićević, M. (2010). Designing clustering process with reusable components. *Info m*, 9(34), 23–29.

Kohonen, T. (2001). Self-organizing maps.-springer series in information sciences, v. 30, springer.

Krutz, D. E., Mirakhorli, M., Malachowsky, S. A., Ruiz, A., Peterson, J., Filipski, A. & Smith, J. (2015). A dataset of open-source android applications. *2015 ieee/acm 12th working conference on mining software repositories*, pp. 522–525.

Krutz, D. E., Munaiah, N., Peruma, A. & Mkaouer, M. W. (2017). Who added that permission to my app? an analysis of developer permission changes in open source android apps. *2017 ieee/acm 4th international conference on mobile software engineering and systems (mobilesoft)*, pp. 165–169.

Kumaran, M. & Li, W. (2016). Lightweight malware detection based on machine learning algorithms and the android manifest file. *2016 ieee mit undergraduate research technology conference (urtc)*, pp. 1–3.

Lee, C., Ko, E. & Lee, K. (2020). Methods to select features for android malware detection based on the protection level analysis. *International conference on information security applications*, pp. 375–386.

Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W. & Ye, H. (2018a). Significant permission identification for machine-learning-based android malware detection. *Ieee transactions on industrial informatics*, 14(7), 3216–3225.

Li, L., Bissyandé, T. F., Le Traon, Y. & Klein, J. (2016). Accessing inaccessible android apis: An empirical study. *2016 ieee international conference on software maintenance and evolution (icsme)*, pp. 411–422.

Li, L., Li, D., Bissyandé, T. F., Klein, J., Le Traon, Y., Lo, D. & Cavallaro, L. (2017). Understanding android app piggybacking: A systematic study of malicious code grafting. *Ieee transactions on information forensics and security*, 12(6), 1269–1284.

Li, W., Wang, Z., Cai, J. & Cheng, S. (2018b). An android malware detection approach using weight-adjusted deep learning. *2018 international conference on computing, networking and communications (icnc)*, pp. 437–441.

Li, Y., Wang, G., Nie, L., Wang, Q. & Tan, W. (2018c). Distance metric optimization driven convolutional neural network for age invariant face recognition. *Pattern recognition*, 75, 51–62.

Lin, J., Zhao, X. & Li, H. (2017). Target: Category-based android malware detection revisited. *Proceedings of the australasian computer science week multiconference*, pp. 1–9.

Ma, S., Wang, S., Lo, D., Deng, R. H. & Sun, C. (2015). Active semi-supervised approach for checking app behavior against its description. *2015 ieee 39th annual computer software and applications conference*, 2, 179–184.

Ma, Z., Ge, H., Liu, Y., Zhao, M. & Ma, J. (2019). A combination method for android malware detection based on control flow graphs and machine learning algorithms. *Ieee access*, 7, 21235–21245.

Mansourov, N. & Campara, D. (2010). *System assurance: beyond detecting vulnerabilities*. Elsevier.

Martinelli, F., Marulli, F. & Mercaldo, F. (2017). Evaluating convolutional neural network for effective mobile malware detection. *Procedia computer science*, 112, 2372–2381.

Mas' ud, M. Z., Sahib, S., Abdollah, M. F., Selamat, S. R. & Yusof, R. (2014). Analysis of features selection and machine learning classifier in android malware detection. *2014 international conference on information science & applications (icisa)*, pp. 1–5.

matters. (2021, January, 27). How many apps are on google play? [Format]. Retrieved from https://42matters.com/google-play-statistics-and-trends.

Munaiah, N., Klimkowsky, C., McRae, S., Blaine, A., Malachowsky, S. A., Perez, C. & Krutz, D. E. (2016). Darwin: a static analysis dataset of malicious and benign android apps. *Proceedings of the international workshop on app market analytics*, pp. 26–29.

Murray, M. (2021, February, 03). The spectrum of mobile risk [Format]. Retrieved from https://info.lookout.com/rs/051-ESQ-475/images/lookout-spectrum-of-mobile-risk-report.pdf.

Muttoo, S. K. & Badhani, S. (2017). Android malware detection: state of the art. *International journal of information technology*, 9(1), 111–117.

Namrud, Z., Kpodjedo, S. & Talhi, C. (2019a). Androvul: A repository for android security vulnerabilities. (CASCON '19), 64–71.

Namrud, Z., Kpodjedo, S. & Talhi, C. (2019b). Androvul: a repository for android security vulnerabilities. *Proceedings of the 29th annual international conference on computer science and software engineering*, pp. 64–71.

Naway, A. & Li, Y. (2018). A review on the use of deep learning in android malware detection. *arxiv preprint arxiv:1812.10360*.

Naway, A. & Li, Y. (2019). Using deep neural network for android malware detection. *arxiv preprint arxiv:1904.00736*.

O'Dea, S. (2021, February, 3). Smartphone unit shipments worldwide by operating system from 2016 to 2023 [Format]. Retrieved from https://www.statista.com/statistics/.

Palomba, F., Di Nucci, D., Panichella, A., Zaidman, A. & De Lucia, A. (2017). Lightweight detection of android-specific code smells: The adoctor project. *2017 ieee 24th international conference on software analysis, evolution and reengineering (saner)*, pp. 487–491.

Perepletchikov, M., Ryan, C. & Frampton, K. (2007). Cohesion metrics for predicting maintainability of service-oriented software. *Seventh international conference on quality software (qsic 2007)*, pp. 328–335.

Rehman, Z.-U., Khan, S. N., Muhammad, K., Lee, J. W., Lv, Z., Baik, S. W., Shah, P. A., Awan, K. & Mehmood, I. (2018). Machine learning-assisted signature and heuristic-based detection of malwares in android devices. *Computers & electrical engineering*, 69, 828–841.

Rousseeuw, P. J. (1987). Silhouettes: a graphical aid to the interpretation and validation of cluster analysis. *Journal of computational and applied mathematics*, 20, 53–65.

Sabhadiya, S., Barad, J. & Gheewala, J. (2019). Android malware detection using deep learning. *2019 3rd international conference on trends in electronics and informatics (icoei)*, pp. 1254–1260.

Sachdeva, S., Jolivot, R. & Choensawat, W. (2018). Android malware classification based on mobile security framework. *Iaeng international journal of computer science*, 45(4), 514–522.

Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X. & Bringas, P. G. (2012). On the automatic categorisation of android applications. *2012 ieee consumer communications and networking conference (ccnc)*, pp. 149–153.

Saracino, A., Sgandurra, D., Dini, G. & Martinelli, F. (2016). Madam: Effective and efficient behavior-based android malware detection and prevention. *Ieee transactions on dependable and secure computing*, 15(1), 83–97.

Sarma, B. P., Li, N., Gates, C., Potharaju, R., Nita-Rotaru, C. & Molloy, I. (2012). Android permissions: a perspective combining risks and benefits. *Proceedings of the 17th acm symposium on access control models and technologies*, pp. 13–22.

Scoccia, G. L., Peruma, A., Pujols, V., Malavolta, I. & Krutz, D. E. (2019). Permission issues in open-source android apps: An exploratory study. *2019 19th international working conference on source code analysis and manipulation (scam)*, pp. 238–249.

Security, C. (2021, February, 03). What is the difference: Viruses, worms, trojans, and bots? [Format]. Retrieved from https://tools.cisco.com/security/center/resources/virus_differences.

Shabtai, A., Kanonov, U., Elovici, Y., Glezer, C. & Weiss, Y. (2012). "andromaly": a behavioral malware detection framework for android devices. *Journal of intelligent information systems*, 38(1), 161–190.

Sharma, A. & Sahay, S. K. (2018). An investigation of the classifiers to detect android malicious apps. In *Information and Communication Technology* (pp. 207–217). Springer.

Shen, T., Zhongyang, Y., Xin, Z., Mao, B. & Huang, H. (2014). Detect android malware variants using component based topology graph. *2014 ieee 13th international conference on trust, security and privacy in computing and communications*, pp. 406–413.

Shezan, F. H., Afroze, S. F. & Iqbal, A. (2017). Vulnerability detection in recent android apps: an empirical study. *2017 international conference on networking, systems and security (nsyss)*, pp. 55–63.

Sirisha, P., Anuradha, T. et al. (2019). Detection of permission driven malware in android using deep learning techniques. *2019 3rd international conference on electronics, communication and aerospace technology (iceca)*, pp. 941–945.

Source.android.com. (2021, February, 1). Android permissions [Format]. Retrieved from https://source.android.com/devices/tech/config.

Statcounter. (2021, January, 27). Mobile operating system market share worldwide [Format]. Retrieved from https://gs.statcounter.com/os-market-share/mobile/worldwide.

Su, M.-Y. & Chang, W.-C. (2014). Permission-based malware detection mechanisms for smart phones. *The international conference on information networking 2014 (icoin2014)*, pp. 449–452.

Sun, J., Yan, K., Liu, X., Yang, C. & Fu, Y. (2017). Malware detection on android smartphones using keywords vector and svm. *2017 ieee/acis 16th international conference on computer and information science (icis)*, pp. 833–838.

Takahashi, T. & Ban, T. (2019). Android application analysis using machine learning techniques. In *AI in Cybersecurity* (pp. 181–205). Springer.

Tchakounté, F. & Hayata, F. (2017). Supervised learning based detection of malware on android. In *Mobile Security and Privacy* (pp. 101–154). Elsevier.

Verbraeken, J., Wolting, M., Katzy, J., Kloppenburg, J., Verbelen, T. & Rellermeyer, J. S. (2020). A survey on distributed machine learning. *Acm computing surveys (csur)*, 53(2), 1–33.

Verma, S. & Sharan, A. (2017). Enhancing the performance of svm based document classifier by selecting good class representative using fuzzy membership criteria. *2017 3rd international conference on computational intelligence & communication technology (cict)*, pp. 1–6.

Vinod, P., Zemmari, A. & Conti, M. (2019). A machine learning based approach to detect malicious android apps using discriminant system calls. *Future generation computer systems*, 94, 333–350.

virustotal. (2021, February, 03). virustotal [Format]. Retrieved from https://www.virustotal.com/gui/.

Vuleta, B. (2021, February, 26). Worrying malware statistics [Format]. Retrieved from https://legaljobs.io/blog/malware-statistics/.

Wang, H., Si, J., Li, H. & Guo, Y. (2019). Rmvdroid: towards a reliable android malware dataset with app metadata. *2019 ieee/acm 16th international conference on mining software repositories (msr)*, pp. 404–408.

Wei, F., Li, Y., Roy, S., Ou, X. & Zhou, W. (2017). Deep ground truth analysis of current android malware. *International conference on detection of intrusions and malware, and vulnerability assessment*, pp. 252–276.

Wei, X., Gomez, L., Neamtiu, I. & Faloutsos, M. (2012). Permission evolution in the android ecosystem. *Proceedings of the 28th annual computer security applications conference*, pp. 31–40.

Wu, W.-C. & Hung, S.-H. (2014). Droiddolphin: a dynamic android malware detection framework using big data and machine learning. *Proceedings of the 2014 conference on research in adaptive and convergent systems*, pp. 247–252.

Wu, Z., Chen, X., Khan, M. U. & Lee, S. U.-J. (2021). Enhancing fidelity of description in android apps with category-based common permissions. *Ieee access*, 9, 105493–105505.

Xiao, J., Chen, S., He, Q., Feng, Z. & Xue, X. (2020). An android application risk evaluation framework based on minimum permission set identification. *Journal of systems and software*, 163, 110533.

Xu, J., Rahmatizadeh, R., Bölöni, L. & Turgut, D. (2017). A sequence learning model with recurrent neural networks for taxi demand prediction. *2017 ieee 42nd conference on local computer networks (lcn)*, pp. 261–268.

Xu, L., Zhang, D., Jayasena, N. & Cavazos, J. (2016). Hadm: Hybrid analysis for detection of malware. *Proceedings of sai intelligent systems conference*, pp. 702–724.

Yalew, S. D., Maguire, G. Q., Haridi, S. & Correia, M. (2017). T2droid: A trustzone-based dynamic analyser for android applications. *2017 ieee trustcom/bigdatase/icess*, pp. 240–247.

Yamaguchi, F., Lindner, F. & Rieck, K. (2011). Vulnerability extrapolation: Assisted discovery of vulnerabilities using machine learning. *Proceedings of the 5th usenix conference on offensive technologies*, pp. 13–13.

Yerima, S. Y., Sezer, S. & Muttik, I. (2014). Android malware detection using parallel machine learning classifiers. *2014 eighth international conference on next generation mobile apps, services and technologies*, pp. 37–42.

Yousefi-Azar, M., Hamey, L. G., Varadharajan, V. & Chen, S. (2018). Malytics: a malware detection scheme. *Ieee access*, 6, 49418–49431.

Yu, S., Yang, M., Wei, L., Hu, J.-S., Tseng, H.-W. & Meen, T.-H. (2020). Combination of self-organizing map and k-means methods of clustering for online games marketing. *Sensors and materials*, 32(8), 2697–2707.

Yuan, C., Wei, S., Wang, Y., You, Y. & ZiLiang, S. G. (2016a). Android applications categorization using bayesian classification. *2016 international conference on cyber-enabled distributed computing and knowledge discovery (cyberc)*, pp. 173–176.

Yuan, Z., Lu, Y., Wang, Z. & Xue, Y. (2014). Droid-sec: deep learning in android malware detection. *Proceedings of the 2014 acm conference on sigcomm*, pp. 371–372.

Yuan, Z., Lu, Y. & Xue, Y. (2016b). Droiddetector: android malware characterization and detection using deep learning. *Tsinghua science and technology*, 21(1), 114–123.

Zarni Aung, W. Z. (2013). Permission-based android malware detection. *International journal of scientific & technology research*, 2(3), 228–234.

Zhao, M., Ge, F., Zhang, T. & Yuan, Z. (2011). Antimaldroid: An efficient svm-based malware detection framework for android. *International conference on information computing and applications*, pp. 158–166.

Zheng, M., Lee, P. P. & Lui, J. C. (2012). Adam: an automatic and extensible platform to stress test android anti-virus systems. *International conference on detection of intrusions and malware, and vulnerability assessment*, pp. 82–101.

Zheng, M., Sun, M. & Lui, J. C. (2014). Droidtrace: A ptrace based android dynamic analysis system with forward execution capability. *2014 international wireless communications and mobile computing conference (iwcmc)*, pp. 128–133.

Zhou, Y. & Jiang, X. (2012). Dissecting android malware: Characterization and evolution. *2012 ieee symposium on security and privacy*, pp. 95–109.

Zhou, Y., Wang, Z., Zhou, W. & Jiang, X. (2012). Hey, you, get off of my market: detecting malicious apps in official and alternative android markets. *Ndss*, 25(4), 50–52.

Zhu, H.-J., Jiang, T.-H., Ma, B., You, Z.-H., Shi, W.-L. & Cheng, L. (2018a). Hemd: a highly efficient random forest-based malware detection framework for android. *Neural computing and applications*, 30(11), 3353–3361.

Zhu, H.-J., You, Z.-H., Zhu, Z.-X., Shi, W.-L., Chen, X. & Cheng, L. (2018b). Droiddet: effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing*, 272, 638–646.

Zink, H. (2021, May, 3). Mobile apps: Are they stealing your information? [Format]. Retrieved from https://safeguarde.com/mobile-apps-stealing-your-information/.