

Gwanyong Lim

Feature Evaluation in Malicious Android App Detection

Master Thesis

at the Research Group for IT Security
(University of Münster)

Principal Supervisor: Prof. Dr. Thomas Hupperich
Associate Supervisor: Henry Hosseini, M.Sc.

Presented by: Gwanyong Lim [465480]
Heekweg 12
48161 Münster
+49 176 86963409
g_lim001@uni-muenster.de

Submission: 31st March 2021

Any sufficiently advanced technology is indistinguishable from magic.

ARTHUR C. CLARKE

Contents

Figures	V
Tables	VI
Listings	VII
Abbreviations	VIII
1 Introduction	1
1.1 Motivation	1
1.2 Contribution	3
1.3 Organization of this thesis	4
2 Background on Android	5
2.1 Android System Architecture	5
2.2 Android Application Fundamentals	8
2.3 Android Security Model	10
2.4 Application Permission Model	12
2.5 Android Malwares	13
2.6 Android Malware Detection Approaches	13
2.7 Feature selection methods	14
3 Android Malware Analysis	16
3.1 Static analysis	16
3.2 Dynamic analysis	22
3.3 Hybrid analysis	23
3.4 Discussion of literature review	24
4 Framework of Android malware detection	27
4.1 Feature Extraction	27
4.2 Feature Vectorization	29
4.3 Data Pre-processing	30
4.4 Feature selection	31
4.4.1 Information Gain	31
4.4.2 Chi-square	32
4.4.3 Genetic Algorithm	33
4.4.4 Binary Particle Swarm Optimization	37
4.4.5 Ensemble Feature selection	40
4.4.6 Hybrid Feature selection	41
4.5 Classification Model	42
4.5.1 SVM	42
4.5.2 XGBoost	43
4.5.3 DNN	45
4.6 Evaluation	46

5	Implementation	48
5.1	Data Collection	49
5.2	Data preparation	49
5.3	Experiment procedure	51
6	Experiment results and discussion	53
6.1	Data Pre-processing	53
6.2	Results of filter methods	55
6.2.1	Information Gain	55
6.2.2	Chi-square	57
6.2.3	Ensemble feature selection	60
6.3	Discussion of filter methods	62
6.4	Results of wrapper methods	63
6.4.1	Genetic Algorithm	63
6.4.2	Binary Particle Swarm Optimization	69
6.5	Discussion of wrapper methods	70
6.6	Results of hybrid method	72
6.7	Discussion of all methods	73
7	Deployment and Continuous Learning	76
8	Conclusion	78
	Appendix	81
	References	82

Figures

Figure 1	Cumulative Android malware samples from 2014 to 2020 [AV-] ..	2
Figure 2	The Android software stack [Devc]	6
Figure 3	Flowchart of reverse engineering for <i>DEX</i> file [Sto]	14
Figure 4	Flowchart of dynamic feature extraction.....	14
Figure 5	The approach of filter feature selection	15
Figure 6	The approach of wrapper feature selection	15
Figure 7	Frequency of feature categories.....	25
Figure 8	Frequency of filter feature selection algorithms	25
Figure 9	Frequency of wrapper feature selection algorithms	26
Figure 10	Flowchart of research methodology	27
Figure 11	Example of feature vectorization	30
Figure 12	1-point crossover	34
Figure 13	Uniform crossover	34
Figure 14	Bit mutation	35
Figure 15	Flowchart of GA	35
Figure 16	Flowchart of BPSO.....	39
Figure 17	A scheme of heterogeneous ensemble approach	41
Figure 18	A scheme of hybrid method	42
Figure 19	Support Vector Machines	42
Figure 20	Overview of DNN structure	45
Figure 21	Structure of Project Directory	48
Figure 22	Structure of source Directory	49
Figure 23	Example of extracted features in JSON format	50
Figure 24	Evolving Graph of GA(survivor=" $(\mu+\lambda)$ ", recombination="1-point")	67
Figure 25	Evolving Graph of GA(survivor=" $(\mu+\lambda)$ ", recombination="uniform")	67
Figure 26	Evolving Graph of GA(survivor=" (μ,λ) ", recombination="1-point")	68
Figure 27	Evolving Graph of GA(survivor=" (μ,λ) ", recombination="uniform")	68
Figure 28	Iterations of BPSO	70
Figure 29	Evolving Graph of GA in hybrid method	73
Figure 30	The process of the finally selected feature selection method	75
Figure 31	Flowchart of REST API	76
Figure 32	Example of web application	77
Figure 33	Example of API request.....	77

Tables

Table 1	Description of components in Java API Framework	8
Table 2	Top10 features of Information Gain	32
Table 3	Top10 features of Chi-square.....	33
Table 4	Parameter GA	36
Table 5	Parameter of BPSO	40
Table 6	Top10 features of Ensemble Feature selection	41
Table 7	Parameter of XGBoost	44
Table 8	Parameter of DNN	46
Table 9	Confusion Matrix for malware detection	46
Table 10	Example of Table for features (after vectorization)	50
Table 11	Example of Table for binary vector (permission).....	51
Table 12	Number extracted features	53
Table 13	Performance of all features	54
Table 14	Number features after pre-processing.....	54
Table 15	Performance of pre-processed features	55
Table 16	Number of features selected by Information Gain	56
Table 17	Performance of SVM with Information Gain	56
Table 18	Performance of XGBoost with Information Gain.....	57
Table 19	Performance of DNN with Information Gain	57
Table 20	Number of features selected by Chi-square	58
Table 21	Performance of SVM with Chi-square	58
Table 22	Performance of XGBoost with Chi-square	59
Table 23	Performance of DNN with Chi-square	59
Table 24	Number of features selected by ensemble feature selection.....	60
Table 25	Performance of SVM with ensemble feature selection	61
Table 26	Performance of XGBoost with ensemble feature selection	61
Table 27	Performance of DNN with ensemble feature selection	62
Table 28	Comparison of different filter methods (Threshold = 75%)	63
Table 29	Performance of GA(survivor=" $(\mu+\lambda)$ ", recombination="1-point") ...	64
Table 30	Performance of GA(survivor=" $(\mu+\lambda)$ ", recombination="uniform")...	65
Table 31	Performance of GA(survivor=" μ,λ ", recombination="1-point")	66
Table 32	Performance of GA(survivor=" (μ,λ) ", recombination="uniform")...	66
Table 33	Performance of BPSO	69
Table 34	Number of features selected by GA	71
Table 35	Performance of hybrid method	72
Table 36	Elapsed time of the first generation	74
Table 37	Number of features selected by Hybrid method	75

Listings

1	Example of Permissions	28
2	Example of Activities	28
3	Example of Services	28
4	Example of Broadcast Receivers	29

Abbreviations

ANNs	Artificial Neural Networks
ART	Android Runtime
BPSO	Binary Particle Swarm Optimization
CHI	Chi-square
DEX	Dalvik Executable format
DNN	Deep Neural Network
EA	Evolutionary Algorithm
FN	False Negative
FP	False Positive
GA	Genetic Algorithm
GBM	Gradient Boosting Machine
HAL	Hardware Abstraction Layer
IDC	International Data Corporation
IDF	Inverse Document Frequency
IG	Information Gain
IPC	Interprocess communication
KNN	K-Nearest Neighbors
LDA	Linear Discriminant Analysis
MLP	Multilayer Perceptron
NMF	Non-Negative Matrix Factorization
p.p.	percentage point
PCA	Principal Components Analysis
PSO	Particle Swarm Optimization
RBF	Radial Basis Function
ReLu	Rectified Linear Unit
SGD	Stochastic Gradient Descent
SVM	Support Vector Machine
TF	Term Frequency
TF-IDF	Term Frequency–Inverse Document Frequency
TN	True Negative
TP	True Positive
UID	user ID

Abstract

As Android platform is widely used around the world, the number of malware applications targeting the Android operating system has been rapidly increasing. Therefore, the threat to Android platform security is also increasing. Machine learning techniques can be used to address this security issue. Previous studies have proved that machine learning or deep learning techniques successfully helps detect Android malicious applications with high accuracy, and feature selection methods that reduce the dimensionality of data can improve the performance of the classification model. We propose a framework for Android malware detection utilizing feature selection algorithms and learning algorithms. We collected 1500 benign android applications and 1500 malicious android applications. A total of 119,427 features were extracted from applications using a reverse engineering tool. In our experiment, we compared various feature selection algorithms and machine learning and deep learning algorithms to find the most suitable feature selection algorithm and learning algorithm. Through our experiment, we chose Deep Neural Network (DNN) for the learning algorithm and the hybrid method for the feature selection. To be specific, the hybrid method is a combination of filter method and wrapper method, and we made the hybrid method by combining the ensemble feature selection (i.e., the combination of Chi-square (CHI) and Information Gain (IG)) and Genetic Algorithm (GA). In our experiment, we finally selected 3,902 features using the hybrid method, and the classification model using DNN achieved an accuracy of 0.943. Finally, we present a framework for deploying the Android malware detection model through REST API and the continuous updating of the classification model.



Unless explicitly specified otherwise, this work is licensed under the license Attribution-ShareAlike 4.0 International.

1 Introduction

In this chapter, we give an overview for Android malware detection in Section 1.1, followed by Section 1.2 that addresses our contribution to this field of study. Finally, the structure of this paper is briefly described in Section 1.3.

1.1 Motivation

Smartphones have been an indispensable stuff in the life of modern people. In fact, smartphones are used by 3.6 billion users around the world in 2020 [Sta] and for various purposes, such as web browsing, online banking, online shopping, and social networking, and so on [Ras+20; Fer+16]. So, smartphones can keep people connected to the Internet at all times. Also, smartphones are utilized by important institutions such as corporations, government agencies, and the military.

Among several platforms for smartphones, the Android operating system has become the most popular and commonly used platform. "Android is an open-source operating system for mobile devices and a corresponding open source project led by Google" [Proa]. According to International Data Corporation (IDC), Android has 84.8% of market share of smartphone, while iOS (Apple) has 15.2% in 2020 [IDC]. As a result of the open environment of Android, many mobile device manufacturers and companies adopt Android because they can customize the Android system for their devices. Therefore, the open environment is a significant reason why Android has become the most popular platform.

Along with the high growth in smartphone usage, smartphones have been increasingly targeted by attackers and infected with malicious software. In particular, attacks on Android are more frequent. Due to its' open environment, attackers can easily abuse the platform features to develop malware applications or include a piece of malicious code in a benign application [Prob]. Besides, attackers can find vulnerabilities in the platform, hardware, or other installed applications and exploit them to execute malicious actions [Ala+16]. According to F-Secure, a cybersecurity company, it was reported in the F-Secure State of Cyber Security 2017 [F-S] that 99% of all malicious applications targeting mobile devices are designed for Android. According to Nokia Threat Intelligence Report - 2020 [Nok20], Android is the most targeted platform by malware among smartphone platforms. Android devices accounted for 26.64% of the total infection cases, Windows and personal computers for 38.92%, followed by IoT devices for 32.72%. iPhones, which use another smartphone platform, iOS, accounted for only 1.7%, which is relatively low compared

to Android. The number of Android malware samples reached about 27 million in January 2020, an increase of 17% from previous year, 2019 [Nok20]. Moreover, according to the statistics [AV-], a huge number of Android malware applications are created each year, and the total number of Android malware has been rising rapidly year by year, as depicted in Fig. 1.

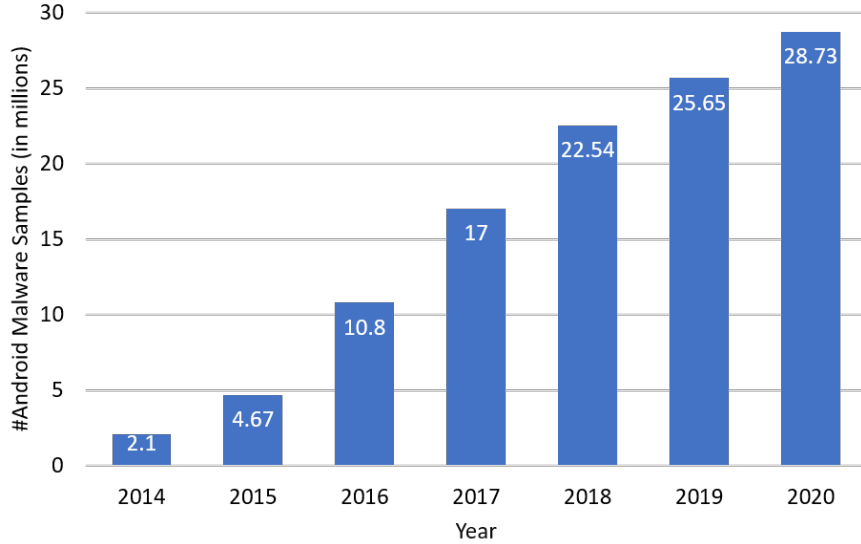


Figure 1 Cumulative Android malware samples from 2014 to 2020 [AV-]

As Android devices are used by users for various purposes, they contain various important and sensitive data of users, such as emails, banking, pictures, locations, and so on. As we discussed, there are currently many malicious applications targeting Android, and the amount will continue to increase in the future. Therefore, it is very likely that Android devices are exposed to security threats by malware applications, such as transmitting the user’s sensitive data or passing control of the device to the attacker. Thus, detecting malicious applications is significantly important to protect users’ devices. Through Android malware detection approaches, numerous features can be extracted from applications and analyzed to detect malware applications. Detection of known Android malware has been commonly performed using traditional approaches such as a signature-based method. In the signature based method, malware signature definitions are stored in a database, and malware is detected by comparing each application’s signature against the database of known malware signatures. However, it cannot efficiently detect new unknown Android malware [Odu+18; Fei+15]. It is because that new variants of malicious applications are constantly emerging, and small modifications in malicious applications also can lead to new variants. These new variants are able to pass the signature-

based detection as unknown attacks [Fei+15; Gar+09]. Therefore, machine learning algorithms that overcome the limitations of traditional methods can be utilized.

In order to implement machine learning techniques, data consisting of various features are required. A large number of features can be extracted from benign applications and malicious applications through Android malware detection approaches. A classifier model can be generated by training the extracted features. Consequently, an unknown application can be classified into either benign or malware. The performance of machine learning algorithms depends on the quality of features. Thus, improving the quality of features can play an important role in detecting android malware. If a method of selecting more important, informative features is used, a high-quality subset of features will be fed into a machine learning algorithm, which in turn will generate a more accurate classification model than simply using a large number of features.

1.2 Contribution

Our contribution can be summarized as follows:

- (1) We looked at the recent research trends on Android malware detection through a literature review. In addition, Android malware detection approach, feature selection algorithms, machine learning and deep learning algorithms, and categories of features to be considered in this paper were determined.
- (2) We presented a framework of android malware detection. The series of processes from extracting features to creating a classification model is implemented by using *Python*, *Docker*, and *MongoDB*.
- (3) We conducted extensive experiments on real-world data and compared not only four feature selection algorithms, but also two methods of combining the considered feature selection algorithm (i.e., ensemble and hybrid) to find the best subset of features for the framework. We discuss the effectiveness of the feature selection algorithm based on the performance results of training the features selected by the feature selection algorithm on three machine learning and deep learning algorithms and the number of reduced features. Also, the comparison of three learning algorithms based on their performance is part of this thesis.
- (4) To the best of our knowledge, no researchers have worked on the practical deployment of the classification model for Android Malware Detection

and the updating the model. We designed a framework for the deployment, continuous training, and updating of the classification model. A REST API is created using the *Flask* framework, and the classification model is deployed through it. A web application that shows an example of using a user interface to communicate with the API is implemented utilizing the *Django* framework.

1.3 Organization of this thesis

The rest of this thesis is organized as follows. Chapter 2 briefly provides background about Android OS, which includes architecture, security features, application components, and permissions model. It also talks about the type of malware in Android applications. This chapter also provides theoretical background on approaches for Android malware detection and feature selection. In Chapter 3, we review related research in Android malware detection. Chapter 4 proposes the methodology used for this thesis, which covers the framework, feature selection algorithms, and machine learning and deep learning algorithms. Chapter 5 provides an overview of the project package for this paper and how the experiment works. In Chapter 6, we present the experimental results, discuss them, and present the most suitable feature selection algorithm and learning algorithm. Chapter 7 proposes a framework for continuous classification model improvement. Finally, in Chapter 8, this thesis presents the conclusion and possible future work.

2 Background on Android

Before discussing the details of the framework for Android malware detection, we should briefly explain how Android and Android applications work. We also should explain the background of what is needed to help understand what will be discussed in the future. Therefore, this chapter defines the basic structure of the Android system and its detailed elements in Section 2.1. Afterward, Section 2.2 covers fundamental concepts for the Android application with respect to the file structure of Android applications and the essential components of Android applications. Next, Section 2.3 addresses the security aspects of the Android operating system. This will be followed by the discussion of the Android application permission in Section 2.4. Afterward, Section 2.5 explains the types of malicious applications that threaten Android. Section 2.6 is dedicated to the presentation of approaches for Android malware detection. Next, Section 2.7 explains how to select optimal features to be used for detecting malicious Android applications.

2.1 Android System Architecture

Android is a Linux-based open-source software stack designed for use on a variety of devices and form factors. Fig. 2 illustrates the major components of the Android software stack. The Android stack consists of five layers. The description of each layer is as follow [Smy20; Devc].

- (1) **Linux Kernel:** This is located at the bottom of the Android software stack and represents the core of the Android system. The Linux kernel includes security settings, memory management, process management, and power management. It also handles device drivers for hardware such as a camera, a device’s display, and audio. For instance, the Android Runtime (ART) depends on some underlying functionalities that the Linux kernel possesses, such as threading and low-level memory management. The device driver is to provide a software interface to a hardware device so that it can allow access to hardware features without knowing the operating system and other details.
- (2) **Hardware Abstraction Layer:** It is positioned between the hardware and other software stacks in the upper layers. It defines standard interfaces that allow hardware functions to be used in the higher-level Java API framework. Multiple library modules make up the Hardware Abstraction Layer (HAL). Interfaces of hardware components, such as Blue-

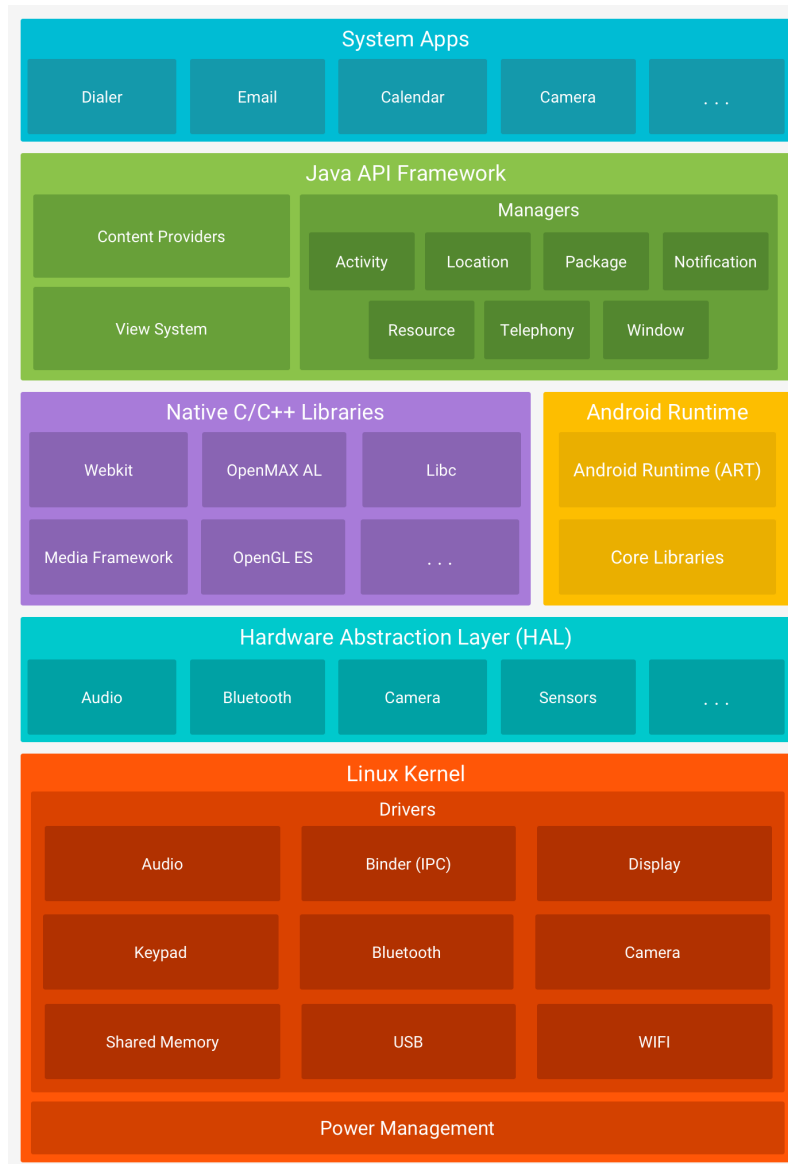


Figure 2 The Android software stack [DevC]

tooth and audio, are implemented through modules. When the API in the Java framework executes a call command to access device hardware, the Android system loads the library module for the hardware component. HAL allows functionalities to be implemented without affecting or modifying higher-level systems.

- (3) **Android Runtime (ART):** Android applications are written in Java, Kotlin, and C++ languages. In Android, it is compiled into an intermediate bytecode format (i.e., Dalvik Executable format (DEX) format). The bytecode format is designed especially for Android, and it is optimized for minimal memory footprint. "Each app runs in its own process and with its own instance of the ART. ART is written to run multiple

virtual machines on low-memory devices by executing *DEX* files." [DevC]. For earlier versions of Android, Dalvik virtual machine was the Android runtime. So, apps that can run on ART can be run on Dalvik.

- (4) **Native Libraries:** These are libraries based on C/C++ native code that can be used on Android. Many of the core Android system components and services, such as ART and HAL, are built on native code. Therefore, native libraries written in C/C++ are required to use the components and services. At this time, native C/C++ libraries help to use components and services written in native code in Java. The functionality of some native libraries is exposed to apps through Java framework APIs. OpenGL ES (C++ library), which is supporting for drawing 2D and 3D graphics on the device display, can be accessed through the Android framework's Java OpenGL API.
- (5) **Java API Framework:** Java API Framework provides various APIs necessary to develop applications. So, all functions of Android OS can be accessed through the Java API framework. These APIs simplify the reuse of core modular system components and services, forming the building blocks needed to create Android applications. The key components included in this layer are described in Tab. 1.
- (6) **System Apps:** It is located at the top of the Android software stack, where the device's functions are provided to the end-user. Android provides a set of core applications, such as email, SMS messaging, calendars, internet browsing, contacts, and so on. Native apps included in the platform and third-party apps installed by users can be used without any special distinction. Thus, third-party apps can replace the default apps, such as the default web browser, SMS messenger, or even the default keyboard.

Table 1 Description of components in Java API Framework

Categories	Description
Activity Manager	controls the lifecycle of applications and activity stack
Content Providers	enables applications to publish and share their data with other applications
Resource Manager	gives access to non-code resources such as graphics, strings, and user interface layouts
Notifications Manager	enables applications to display alerts and notifications in the status bar.
View System	is used to create user interfaces of applications, such as grids, lists, text boxes, and buttons
Package Manager	contains information about the application packages currently installed on the device.
Telephony Manager	provides information about the telephony service available on the device, such as status and subscriber information
Location Manager	provides access to the location services and allows applications obtain updates about location changes

2.2 Android Application Fundamentals

Android applications are generally ZIP files in APK file format. If an APK file is unzipped, the contents are as follows:

- **META-INF:** This directory includes the certificate of the application.
- **AndroidManifest.xml:** This file contains application metadata such as package name, permissions, version, referenced library files, and app components.
- **classes.dex:** This file contains all source codes of the application compiled into the *DEX* file format translatable by ART (or Dalvik virtual machine)
- **assets:** This directory holds any other files the application may need.

- **lib:** This directory basically holds native libraries for the application. There are CPU-specific directories under the lib/ directory. For instance, armeabi, mips.
- **resources.arsc:** This file holds precompiled resources
- **res:** This directory contains resources not compiled into resources.arsc

AndroidManifest.xml describes essential information about the application. Activities, broadcast receivers, services, and content providers, which are essential app components of an Android application, are declared in this file. These are the key building blocks of an Android application. Each component is an entry point for a system or user to enter the application [Devb]. The description of the app components is as follows [Devb].

Activities: Activities account for all visible actions regarding user interfaces. For instance, an email app can have multiple activities such as showing a list of new emails, composing an email, and reading emails. Several activities work together within an email app, but each activity is independent of the others.

Services: Services run in the background to perform long-running operations without a user interface. For example, music can be played in the background while the user is doing other tasks. Service also is used to implement a rich communications API that is called by other applications.

Broadcast receivers: Broadcast receivers receive and respond to broadcast announcements from the system or other applications. For example, the broadcast message is delivered to the broadcast receiver when the alarm time comes.

Content providers: Content providers are used to store the data of the application in the file system. The application can access the data of other applications provided through the content providers. In addition, an application can define its own content providers to share its own data as well.

Three of the app components, namely activities, broadcast receivers, and services, are activated by an asynchronous message called an **intent**. For activities and services, intent includes an action to be carried out. For broadcast receivers, the

intent simply defines the current announcement that is being broadcast. Also, it allows components to communicate and share information with each other.

2.3 Android Security Model

Android has built-in robust and flexible security models so as to secure the open-source system. Moreover, the security of Android depends on the user's comprehension of applications and systems. So, users can visibly see permissions requested by each app and control those permissions. Android's security mainly aims to protect user data and system resources and provide application isolation from the system, other apps, and from the user. To achieve these goals, the main foundations of Android's security rely on Linux kernel, application sandboxing, secure Interprocess communication (IPC), application signing, and application-defined and user-granted permissions [Prob].

Linux kernel: The Linux kernel is the foundation of the Android platform as discussed in Section 2.1. At the operating system level, the security of the Linux kernel is provided by the Android platform. The Linux kernel has been commonly used in a variety of sensitive-security environments. The Linux kernel has been constantly enhanced by security specialists, developers who fix and patch security bugs, and attackers who find vulnerabilities to be exploited. So, Linux has become a stable and secure kernel. The Linux kernel is the base of a mobile computing environment and offers several key security features to Android: a user-based permissions model, process isolation, an extensible mechanism for secure IPC, and the ability to remove the unnecessary and insecure parts from the kernel. As a multi-user operating system, the basic security goal of the Linux kernel is to separate user resources from each other. Therefore, the Linux kernel prevents other users from invading the user's resources. Such as, the Linux kernel prevents user A from reading user B's files and from exhausting user B's memory, CPU resources, devices (e.g., GPS, Bluetooth) [Prob].

Application sandboxing: The application sandboxing isolates applications from each other and protects the system and applications from malicious applications. To be specific, Android utilizes Linux user-based protection for identification and isolation of applications using a unique user ID (UID). Android assigns each Android application a unique UID to run in its own process. The applications cannot interact with each other and have limited access to the operating system. Therefore, it is

theoretically impossible for application A to try something malicious to application B , such as reading data stored under the UID of application B [Prob].

Application signing: The application signing requires all applications to be signed by the developers with a digital certificate. If an application is not digitally signed, either Google Play store or package blocks its installation attempt. Application signing is utilized to identify the author of the application and to update the application without complicated procedures and further permissions. On Android, the application signing is the beginning of placing an application in its application sandbox. A signed application certificate defines the association between UID and application. So, different applications will run under different UIDs. The application signing makes the application inaccessible to any other application without going through a well-defined IPC. Applications can share UID, if they are signed with the same certificate. The Android system grants the security permissions at the signature protection level to the requesting application if it is signed with the same certificate of other apps that declared the permissions [Prob].

Interprocess communication (IPC): The IPC refers to the communication between processes using the traditional UNIX-type mechanisms, such as local sockets and file systems. Mechanisms of Android IPC are described as follows [Prob]:

- **Binder:** It is a remote procedure call mechanism to handle in-process and cross-process calls with high performance.
- **Services:** Services (discussed in Section 2.2) can provide interfaces that can be accessed directly using a binder.
- **Intents:** An intent is a communication mechanism that represents an intention to do some action (explained in Section 2.2). For example, if a website is to be opened, an intent instance about the 'intent' of viewing the URL is created and sent to the system.
- **Content providers:** Content provider (explained in Section 2.2) is a data store that provides access to data on a device.

Android Permissions model: The Android Permissions model (i.e., Application-defined and user-granted permissions) manage the access privileges to system resources and functions. Therefore, developers need to have specific permissions to be

allowed to access or exploit the system resources and functions. Section 2.4 explains the Android permission model in more detail.

2.4 Application Permission Model

Permission is the main security concept that Android security relies on. Permissions refer to the permissions granted during the installation or while the app is running on devices. Permissions are requested by Android apps to access sensitive user data (e.g., contacts and SMS) and certain system features (e.g., connecting to a paired device and recording audio). Therefore, malicious applications try to circumvent security mechanisms by requiring sensitive permissions and exploiting them. For instance, *"SEND_SMS"* requests permission to send SMS. *"ACCESS_COARSE_LOCATION"* is the permission to allow to access approximate location [Deva].

The provided permissions are classified into 4 types according to the level of protection: normal, dangerous, signature, and signature or system.

- **Normal:** A low-risk permission that provides applications requesting permission access to isolated application-level features, with minimal threat to other applications, the user, or the system. This type of permission is automatically granted by the system without notifying the user.
- **Dangerous:** A high-risk privilege that grants an application requesting permission to access private user data or to control a device that could negatively affect users. Because this type of permission poses a potential risk, the system does not automatically grant permissions to the application requesting permission. Therefore, dangerous permissions are shown to the user. Subsequently, the user can decide whether to grant the permissions to the application.
- **Signature:** A permission granted by the system only if the application requesting a permission and the application that declared the permission are signed with the same certificate. If both applications have the same certificate, the system automatically grants permission without notifying the user or requesting explicit approval from the user.
- **Signature or System:** A permission granted by the system only to the applications that are installed in a dedicated folder on the Android system image or signed with the same certificate of the application that declared the permission.

2.5 Android Malwares

Malicious software is referred to as Malware. In this section, referring to [Ars+16] and [Deve], we discuss some types of malware families found in Android mobile and their characteristics.

- **Trojan:** It disguises itself as a normal application. It can commit malicious activities, such as stealing sensitive information in the background without the user noticing it. Examples: FakeNetflix, Zsone, Zitmo.
- **Backdoor:** It enables any remote-controlled operation on a device by taking root privileges. It has the ability to hide itself from antiviruses. Examples: Exploid, Rageagainstthecage and Zimperlich.
- **Worm:** It copies and spreads itself over a network. It can spread over Bluetooth networks, even to devices paired with Bluetooth. Example: Android.Obad.OS.
- **Spyware:** It transmits user's data such as contacts, messages, location, and other confidential data to a remote server without knowledge of the user. Examples: Nickyspy, GPSSpy.
- **Botnet:** It is a network consisting of comprised Android devices. It enables an attacker to remotely control the device from a server called Botmaster using a command and control network. Example: Geinimi.
- **Ransomware:** It locks the device to prevent users from accessing their data until some ransom is paid. Example: FakeDefender.

2.6 Android Malware Detection Approaches

Approaches with respect to Android malware detection can be classified into static, dynamic, and hybrid according to the nature of the feature. The hybrid analysis is the consolidation of static analysis and dynamic analysis.

The **Static analysis** utilizes the executable files without the execution of application. It decomposes an application to source code by applying a reverse engineering technique. *AndroidManifest.xml* file and *DEX* file are generally obtained through reverse engineering. *AndroidManifest.xml* includes various kinds of features such as permissions, app components, intent, and so on. *DEX* file is not human-readable. Thus, *DEX*'s bytecode needs to be transformed into a readable file through reverse

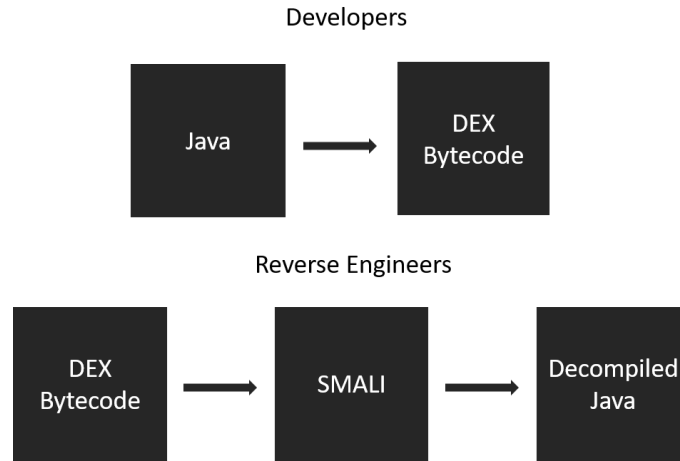


Figure 3 Flowchart of reverse engineering for *DEX* file [Sto]

engineering, as described in Fig. 3. Smali is the most common human-readable version of Dalvik bytecode [Sto]. Classes, methods, or strings can be extracted from the *DEX* file.

The **Dynamic analysis** is the method used to capture behaviors of an application. While an application is executed on either a virtual machine or a physical device, behaviors of the application are recorded and monitored for further processes. Features obtained using the dynamic analysis can be network connections, function calls, system calls, resource usage, and so on. Fig. 4 depicts the process of dynamic feature extraction.

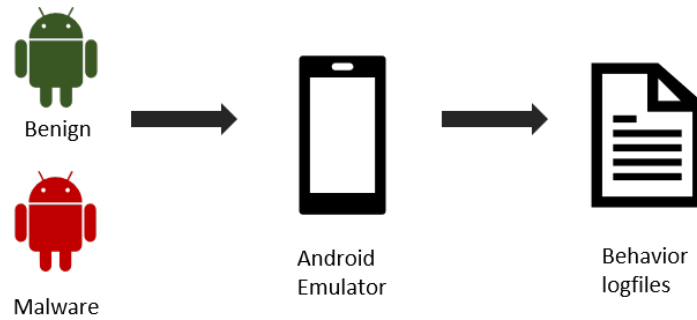


Figure 4 Flowchart of dynamic feature extraction

2.7 Feature selection methods

A number of features can be extracted using the Android malware detection approaches discussed above. In this respect, feature selection methods are necessary. Feature selection is a widely used technique to offset the curse of dimensionality.

A huge dataset probably contains redundant and irrelevant features, which can be noisy for the classification. The aim of feature selection is to find out the most informative features and filter out features lacking in decisiveness. It can give some advantages. Data with reduced dimensionality leads to improving the efficiency of a classification algorithm (i.e., less training elapsed time) and performance [Cen+15; Liu+20]. Feature selection methods are categorized into filter, wrapper, and embedded methods.

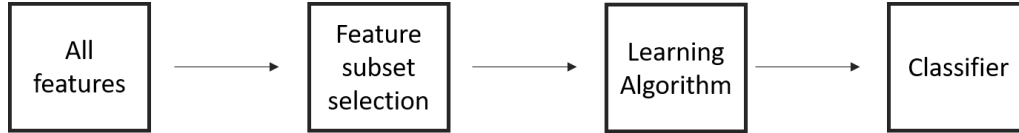


Figure 5 The approach of filter feature selection

Filter methods are independent of any classification algorithms so that it is applied prior to a learning algorithm. Thus, filter methods are generally considered less costly than other methods [QSL16; SSK20]. Filter methods usually rank features based on the statistical criteria, and the features are kept or filtered out based on their score [TAL14]. Fig. 5 illustrates the approach of filter feature selection.

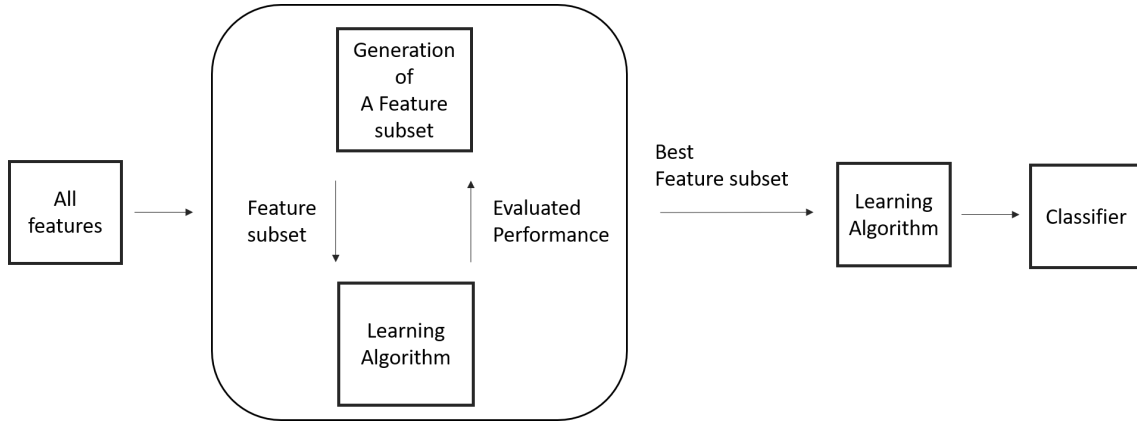


Figure 6 The approach of wrapper feature selection

Wrapper methods generate subsets of features and evaluate them by training in a learning algorithm, whereas filter methods evaluate features individually, such as scoring a feature using a statistical method. According to the performance of the trained model, the subset of features is iteratively revised by adding or removing features [SSK20]. The framework of wrapper feature selection is described in Fig. 6. Embedded methods perform feature selection and model training simultaneously [QSL16]. Random Forest algorithm is one example of the embedded method.

3 Android Malware Analysis

A variety of research has been conducted to detect Android malware using machine learning techniques. In this section, we review the relevant literature work. For the collection of relevant articles, one academic database, "SCOPUS", was utilized. We include papers which cover Android malware detection using machine learning as well as deep learning. We applied inclusion and exclusion criteria, which are in the English language, and with a custom time range from 2016 to 2020. Moreover, the articles that do not contain a specific process of feature extraction, feature selection (or reduction) or model training were removed. After removing duplication between papers, 34 papers were finally selected.

The following subsections explain how features are extracted, what methods are used for feature selection, and what learning algorithms are used from each method. Section 3.1 reviews papers employing the static analysis. Papers using the dynamic analysis are discussed in Section 3.2. Afterward, Section 3.3 discusses papers exploiting the hybrid analysis. The final Section 3.4 comprehensively analyzes and discusses the papers discussed in the previous sections.

3.1 Static analysis

Alatwi et al. [Ala+16] suggested category-based malware detection. They extracted features (permissions, broadcast receivers, and APIs) from *AndroidManifest.xml* and Java classes. They generated a binary vector matrix for the Music & Audio application category, the personalization category, and all categories. A subset of features for each dataset is selected using the Random Forest algorithm. The authors trained a model using a Support Vector Machine (SVM) classifier for each dataset. So, two category-based models showed better performance than the non-category based model.

Cen et al. [Cen+15] explored and compared diverse ways for each step of the generation of an optimized detection model. First, they compared the different level of feature granularity. The Java function level was the most informative feature level (i.e., the highest F1-score) compared to the Java packaged level and class level. When it comes to feature representation, the usage of binary showed a better F1-score than the usage of Term Frequency (TF) and $\log(TF + 1)$. The researchers took Information Gain (IG), Chi-square (CHI), and Regularization algorithms for feature selection. As a result, they created a probabilistic discriminative learning model exploiting logistic regression with regularization. This model achieved 0.95 in terms of F1-score.

Zhao et al. [Zha+16] devised a feature extraction tool called *AppExtractor*, which extracted more than 3,200 features of four different feature categories, such as permission features, APIs from smali code, action features, and IP & URL features. They introduced a novel feature selection algorithm, "*FrequenSel*", which overcomes the shortcomings of the existing well-known algorithms, CHI and IG. *FrequenSel* rank features by calculating the differences of the feature occurrence frequency between malware and benign applications. So, the authors eventually selected features that are frequent and have data coverage. The authors made a model using the K-Nearest Neighbors (KNN) and SVM algorithms using the features extracted by three different feature selection algorithms and compared their performance results. The *FrequenSel* algorithm showed considerably better performance over the other two algorithms. As a result, the authors made a classification model using SVM that achieved accuracy and recall of about 98%. KNN showed slightly weaker performance than SVM, but because the training time is considerably faster than SVM, KNN was also considered as an optimal classification algorithm.

Fereidooni et al. [Fer+16] extracted different feature sets (i.e., intents, used permissions, system commands, suspicious API calls, and malicious activities). They employed a randomized Decision Tree algorithm so as to select features by evaluating the importance of features. They trained the selected features with several classifier algorithms to find the one with the best performance. It appeared to be that XGBoost performed more accurately than the other eight algorithms. Additionally, it achieved a 97% accuracy score.

Chang and Wang [CW17] obtained permission features from *AndroidManifest.xml* and API features from *DEX* file through the reverse engineering procedure. The researchers used frequency-based analysis to select features. They calculated the difference in the usage rate $(x-y)\%$ between the usage rate among malware applications $(x)\%$ and the usage rate among benign applications $(y)\%$ for each feature. Therefore, the features can be ranked and sorted by the score. SVM and KNN were utilized to train detection models. SVM performed better than KNN in terms of accuracy, precision, recall, and false-positive rate scores.

Li et al. [Li+17a] obtained features from decompiled source code and *AndroidManifest.xml*. It exploited Principal Components Analysis (PCA) based on singular value decomposition to reduce dimensionality. Six different machine learning algorithms (i.e., Naive Bayes, Decision Tree, Logistic Regression, SVM, and Random Forest) were taken into account to figure out the most suitable algorithms. This paper proposed a novel classification idea named *Fine-grained* classification. The *Fine-grained* classification considers multi-class (various malware families), as opposed to binary-class (i.e., whether an application benign or not), in order to show a more detailed result. When experimenting in two situations, binary-class and multi-class,

two models out of Logic Regression and Random Forest showed the best performance scores. It also proved that training time decreased due to the reduction of dimensions.

Wang et al. [WLZ18] got 285 permission features and 1,526 API call features. It analyzed the weight of the features utilizing a wrapper method based on Random forest-based selection algorithm. Therefore, features can be ranked by their accuracy score resulting from the Random Forest. As a threshold was set to 0.5, the number of features decreased to about 300. The authors made three models using SVM, XGBoost, and XGBoost with only selected features. The XGBoost showed better accuracy and training time efficiency than the SVM model. Moreover, XGBoost with the selected features from the combination of two types (i.e., permissions and API calls) showed better performance than the XGBoost on any single type of feature. Thus, the case of considering both types using function selection algorithm showed better results than simply considering only one type.

Li et al. [LWX18] extracted features including hardware components, requested permissions, application components, filtered intents, restricted API calls, used permissions, suspicious API calls, and network addresses. PCA was used for feature reduction. They subsequently trained the dataset using Deep Neural Network (DNN) algorithm. DNN algorithm showed better performance in precision, recall, and F1-score than other traditional machine learning algorithms (i.e., Decision Tree, Random Forest, Logistic Regression, and SVM).

Zhang et al. [Zha+18] extracted API calls from decompiled smali code. However, they did not consider the smallest granular API calls. It determined the level of package to consider using n-grams. After that, they compared the performance of package n-grams to opcode n-grams. The authors trained models with Random Forest, KNN, Naive Bayes, and Decision Tree algorithms. Using package n-grams in all models achieved the best performance and showed better performance than using opcode n-grams

Fatima et al. [Fat+19] used Genetic Algorithm (GA), which returns a set of features having the best fitness score. Two machine learning algorithms, SVM and Neural Network, were used to apply the GA. After getting two optimized sets of features, these features are trained with two classifiers respectively. Both models showed better training time complexity than models without the feature selection process and accuracy higher than 94%.

Nivaashini et al. [Niv+18] extracted permission information by dissecting APK files. They chose the CHI algorithm after comparing it to other algorithms (Gain ratio, Relief attribute evaluator, and Correlation-based feature subset evaluator) for feature selection. This paper took five machine learning algorithms into account, namely, SVM, Random Forest, KNN, Naive Bayes and Multilayer Perceptron (MLP) based

Neural Network. It appeared that the SVM showed better performance than the other machine learning algorithms with the selected features by CHI.

Han et al. [Han+20] obtained API call features and found that 5% of malicious instances were noisy data, so that those were removed. In addition, the APIs had an extremely low accessed frequency, or zero were removed as well. It created a detection model based on SVM with Radial Basis Function (RBF) kernel, which achieved above 98% in overall performance scores.

Eom et al. [Eom+18] experimented with three different processes of feature selection with APIs. First, approach 1 just used light pre-feature selection. Regardless of malicious or benign, the top 200 frequent API features were selected. Second, approach 2 used pre-feature selection, which selects the top 150 frequent features each from malicious and benign. After removing duplicates, the top 200 features remained. Third, after the pre-feature selection step, approach 3 added post-feature selection that considered the importance of each feature by using Random Forest. When it trained the three approaches through Random Forest, the performance was ranked from best to worst in the order of Approach 3, 2, 1.

Le et al. [Le+20] utilized Document Frequency to decrease the number of features (e.g., APIs-based behaviors and permissions). Document Frequency was used to rank features by a difference in the rate of each feature used in all malicious features and the rate of each feature used in all benign features. This paper executed some experiments with different combinations of datasets. The authors found out that using the set of API-based behaviors and the set of permissions individually did not show good performance results compared to using both datasets or also with features of the APK files. For this experiment, it used Naive Bayes, Decision Tree, Random Forest, Gradient Boosting, and AdaBoost as machine learning algorithms.

Bhattacharya et al. [BGM19] extracted permissions from the *AndroidManifest.xml* file. They subsequently generated a binary feature vector. The researchers came up with a new feature selection algorithm called Particle Swarm Optimization (PSO) and rough set-based feature selection (PSORS-FS) which is a combination of PSO with rough set theory. They conducted an experiment with several machine learning algorithms (e.g., Random Forest, Naive Bayes, MLP, and so on.) to compare PSORS-FS to other existing feature selection algorithms (e.g., IG, Gain ratio, Pearson’s correlation coefficient, CHI, and so on.). As a result, they proved that the suggested feature selection algorithm (i.e., PSORS-FS) performed better than the other algorithms.

Salah et al. [SSK20] extracted seven different categories of static features. During URL feature merging step, the URL feature was turned into a URL_score feature. The authors left out low frequent features using a Term Frequency–Inverse Document Frequency (TF-IDF) based feature selection method. A linear SVM model out-

performed other classification models using Logistic Regression, AdaBoost, Stochastic Gradient Descent (SGD) and Linear Discriminant Analysis (LDA). Compared to the original linear SVM's accuracy score, a linear SVM with selected features by CHI showed 4p.p. higher accuracy score. Moreover, the linear SVM utilized remarkably less memory size and showed efficient training time compared to a nonlinear SVM model. It also maintained that the proposed feature selection method effectively saved the feature space.

Roy et al. [Roy+20] exploited Non-Negative Matrix Factorization (NMF) to reduce the dimensionality of the API feature set. The authors exploited Logistic Regression, SVM, Random Forest, KNN as classification algorithms. Before reducing the dimensionality of the feature set, Random Forest outperformed other classifiers. Through NMF, this paper decreased the number of features from 209 to 50-dimensional-representation. Although many features were excluded, classification models did not degrade significantly, and KNN showed the best performance in this case.

Razak et al. [Raz+18] extracted permissions and compared PSO, IG and evolutionary computation algorithms to optimize feature set. They used various machine learning algorithms, including Random Forest, J48, KNN, MLP, and AdaBoost, for its experiment. Results of this experiment showed that the overall evaluation indicators were good when the classifier models were created with the optimized feature set using PSO.

Bhattacharya and Goswami [BG17] abstracted permissions and carried out a comparative experiment of six different feature set, selected by IG, Pearson coefficient, Gain ratio, CHI, One R, and Relief. And it evaluated their performances using various machine learning algorithms.

Bhattacharya and Goswami [BG18] used permissions as features and proposed community based feature selection methods. This methodology is based on Cosine similarity and Levenshtein Distance to measure the similarity between permission vectors and utilized community detection mechanisms of Infomap [RB08], Louvain Algorithm [Blo+08], and VOS clustering algorithm [VW07], which is not commonly applied in the context of feature selection of permission-based Android malware detection. The proposed feature selection method enhanced performances of classification models over the feature selection algorithms utilized in [BG17].

Badhani and Muttou [BM18] utilized API classes and compared two different classification approaches, pre-classification ensemble and post-classification ensemble. For two approaches, CHI, OneR, and Relief were used as feature selection methods and Extreme Learning Machine was used as a classification method. In terms of pre-classification ensemble, ensemble learning is applied at the feature selection level. In other words, three feature sets were generated by each feature selection algorithms and those were united into one feature set. After that, the Extreme

Learning Machine classifier was applied. In contrast, ensemble learning is applied at the prediction level with respect to post-classification. Hence, three classifiers were generated using the individual feature set generated by each feature selection algorithms. So, the final decision is made by a majority vote of three individual classifiers. For low values of the threshold of feature subset, the pre-classification ensemble performed better, whereas the post-classification performed better for high values.

Firdaus et al. [Fir+18] obtained features including five different categories (i.e., permission, code-based, directory path, and system command). It adopted the Genetic search algorithm, which is based on GA, to find the best set of features. The authors used Naive Bayes, Functional Trees, J48, Random Forest, and MLP learning algorithms so as to compare the difference in performance before using Genetic search and after. It appeared that using Genetic search improved the prediction accuracy and prediction time when evaluated using k-fold. Among the machine learning algorithms, functional trees showed the highest scores with the accuracy of 95%.

Peynirci et al. [PEK20] extracted permissions, API calls, and string features and suggested a new feature selection algorithm which is called Delta_IDF. Delta_IDF is the difference in Inverse Document Frequency (IDF) value in benign applications and a IDF value in malicious applications for each feature (i.e. $IDF_{\text{benign}} - IDF_{\text{malware}}$). Results of this paper showed that performance of classifiers using the proposed feature selection were better than using the lowest IDF method, statistical ranking method, and no feature selection.

Chen et al. [Che+19] extracted permissions, intent actions, and API calls. The authors exploited Random Forest as a feature selection method. Random Forest method enables the calculation of the impurity reduction for each feature. Thus, this paper selected features above the mean of decrease impurity. The authors used Deep Belief Network based algorithm for malware classification. As a result, the deep learning model of this paper achieved superior results in comparison to other detection tools.

Wang et al. [Wan+17] extracted four types of features from *AndroidManifest.xml*, which are namely application components, intents, requested permissions, and hardware. Also, they extracted four types from *DEX* code, which are namely API calls, protected strings, commands, and networks. They collected 65,804 features in total. Two hundred eighty-seven features which are named as the *APIC* feature set were selected from four types of features, API calls, permissions, intents, and components using *FrequenSel*. In the meantime, 99 features are called the *CHPN* feature set were selected from four types of features, protected strings, commands, hardware, and networks, using the IG algorithm. KNN, J48, and Random Forest were utilized for classification algorithms. The three algorithms train each of the two datasets to

return a probability output. Six probability outputs were joined and input to the information fusion method with probability analysis and DempsterShafer theory to identify Android malware samples.

Kabakus [Kab19] used the number of disassembled strings in place of codes from an XML or a *DEX* file themselves. Therefore, extracted features from *Android-Manifest.xml* are the number of activities, services, receivers, features, dangerous permissions, custom permissions, and other permissions. The author also counted the number of lines of code in the decompiled Java source file. It utilized the IG method to remove uninformative features. The author utilized a variety of machine learning algorithms to compare their performance. As a result, using Random Forest achieved the highest precision score of 0.987.

3.2 Dynamic analysis

Alzaylaee et al. [AYS17] compared two different environments for the execution of applications which are emulator-based analysis and phone-based analysis. They utilized the *DynaLog* dynamic analysis framework to monitor and extract logged features with respect to API calls and intents during runtime. IG was used to rank and select the top 100 features. During the analysis and feature extraction, the phone-based analysis was able to analyze 23.8% more applications compared to the emulator-based analysis. Therefore, the phone-based analysis could extract a larger number of features more effectively. The authors exploited various machine learning algorithms for its experiment, which are namely SVM, Naive Bayes, Logistic Regression, MLP, Partial Decision Trees [BMD06], Random Forest, and J48 Decision Tree. It appeared that the top 100 features from the phone-based analysis outperformed overall over the top 100 features from the emulator-based. When it comes to the phone-based analysis, all classifiers obtained a TPR score of 90% or more except for the Naive Bayes classifier. In contrast, four classifiers (i.e., Naive Bayes, Logistic Regression, Partial Decision Trees, and J48 Decision Tree) with the emulator-based features obtained a TPR score of 90% or less.

Singh and Hofmann [SH18] executed 216 malicious applications and 278 benign applications installing in an emulator. They gained 337 system calls as a feature, and the frequency of each system call was filled into the feature vector. Subsequently, features with zero variance were removed. This paper took three feature selection methods (i.e., IG, CHI, and Correlation) into account. The authors utilized Decision Tree, Random Forest, Gradient Boosting trees, KNN, SVM, Neural Network, and Deep learning as machine learning algorithms. As a result, the classifier SVM with selected features by correlation reached an accuracy of 97.16% and recall of 99.54%. Feng et al. [Fen+18] used *DroidBox* [Dro], which is an open-source dynamic anal-

ysis tool. During the dynamic analysis, nine types of features were extracted, such as Cryptographic Operation, information leaks, service start, receiver action, sent SMS, system call, and so on. The authors used CHI for feature selection and Cryptographic Operation, information leaks, service start, receiver action, sent SMS, and system call for training. They compared stacking learning algorithm (i.e., probability-wise) to majority voting learning algorithm. The results of this paper’s experiment demonstrated that using the stacking method performed better than using the Majority Voting method.

Thangaveloo et al. [Tha+20] put collected APK files into an emulator to monitor and extracted system call, CPU usage, memory usage, and network packets as features. They utilized a method of Gain ratio for a feature selection process. With Random Forest classifier, the authors conducted several experiments with a different combination of types of features and feature selection processes. The best result was an accuracy of 91.7%.

3.3 Hybrid analysis

Li et al. [Li+18] extracted ten categories of features through static analysis and dynamic analysis, which were permissions, hardware, software, intent action, intent categories, sensitive APIs, IP addresses, advertisement modules, system services, and system security settings. Attribute subset selection and PCA were exploited to reduce the dimensionality of the data. They compared some machine learning algorithms: Naive Bayes, Bayesian network Decision Tree, KNN, and Random Forest, to find the best classifier with the selected features in terms of the performance of 10-fold cross-validation. The Random Forest classifier achieved the highest performance score as a result.

Malik et al. [MGM19] extracted permission and intents as static analysis features. On the other side, network traffic features were captured by running applications on smartphones with a specific application, tPacket Capture, to help analyze the traffic of the Android smartphone. IG was utilized to rank all features and select informative features. They conducted experiments using Naive Bayes and Decision Tree machine learning algorithms. As a result, the integration of three categories of features (i.e., permission, intents, and network traffic) outperformed compared to using each of them or using them separately as static and dynamic.

Wen and Yu [WY17] obtained permission, intent, uses-feature, API, application features through static analysis and the CPU consumption, the battery consumption, the number of running processes, and the number of short message through dynamic analysis with *DroidBox* [Dro]. They devised a feature selection method

by combining PCA and Relief, which is capable of finding the most discriminating subset of features. The combination of PCA and Relief performed better with SVM than using either of them with SVM as a result.

Alzaylaee et al. [AYS20] proposed an automated dynamic analysis framework with added static analysis. For the dynamic analysis, they compared stateless (*Monkey* [Devd]) and stateful (*DroidBot* [Li+17b]) input generation. *Monkey* is a random-based testing tool, which generates random inputs of user events such as clicks, touches, or gestures. Whereas *DroidBot* is a user interface-guided input generation tool based on a state transition model. Prior to the dynamic analysis, they extracted permissions using static analysis. Four feature sets were generated based on IG. In other words, the top-ranked features of when only stateless and stateful input generation were considered and when the static analysis was also considered. After training the four sets with deep learning algorithms, it appeared that using stateful performed better than using stateless input generation regardless of the features from the static analysis. However, adding static analysis improved performance in stateless or stateful input generation.

3.4 Discussion of literature review

The proposed papers above demonstrate the effectiveness of the reduction of feature dimensionality and the usage of machine learning or deep learning algorithms for Android malware detection. The majority 76% of papers used the static analysis to collect features. It is because there is no need for static analysis to run applications. In contrast, the dynamic analysis requires high computational cost, high resource consumption, and high time consumption. So, we adopt the static analysis to extract features in this thesis.

Fig. 7 describes that the frequency of feature categories used for static analysis in the proposed papers. In terms of app components or intent, even if only one of their components (e.g., content provider, service, intent action) is used, it is added to the frequency. The most used category is permission, followed by API call. So, we use these two categories for this thesis. In addition, we use the third and fourth categories, which are namely intent and app components.

Fig. 8 shows the frequency of the filter feature selection algorithms showed the best results or used alone in the papers above. IG is the most used algorithm. We chose another feature ranking algorithm, CHI, as well as IG. When it comes to the wrapper method (Fig. 9), PSO is the most used algorithm. But, Genetic search is

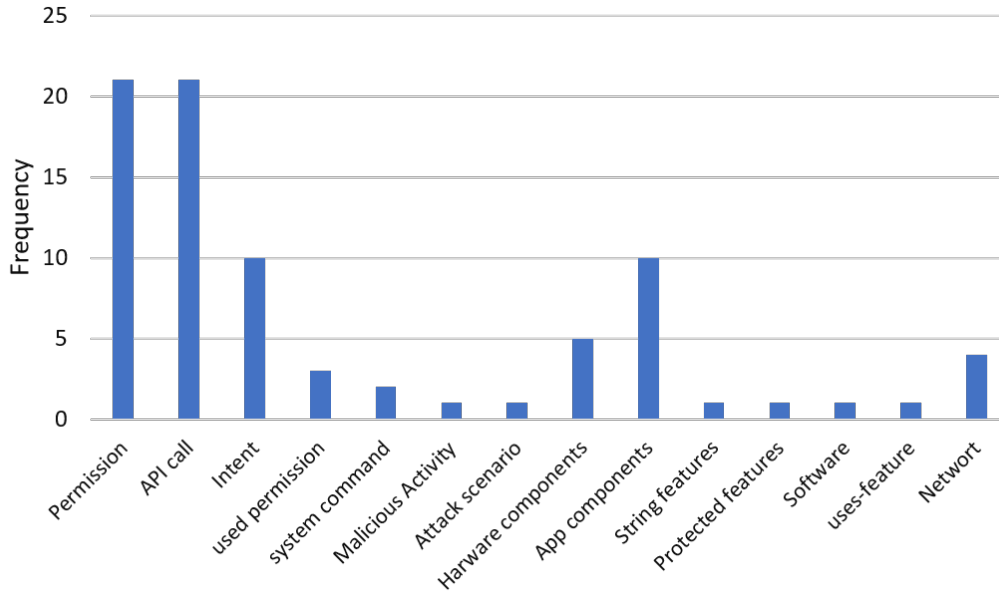


Figure 7 Frequency of feature categories

based on GA. So, GA is also considered the most frequent algorithm. In this thesis, PSO and GA are also used for feature selection. According to training algorithms, we utilize SVM most commonly used in the papers and growing trendy algorithms, deep learning, and XGBoost. A detailed description of the selected algorithms and methods is covered in Chapter 4.

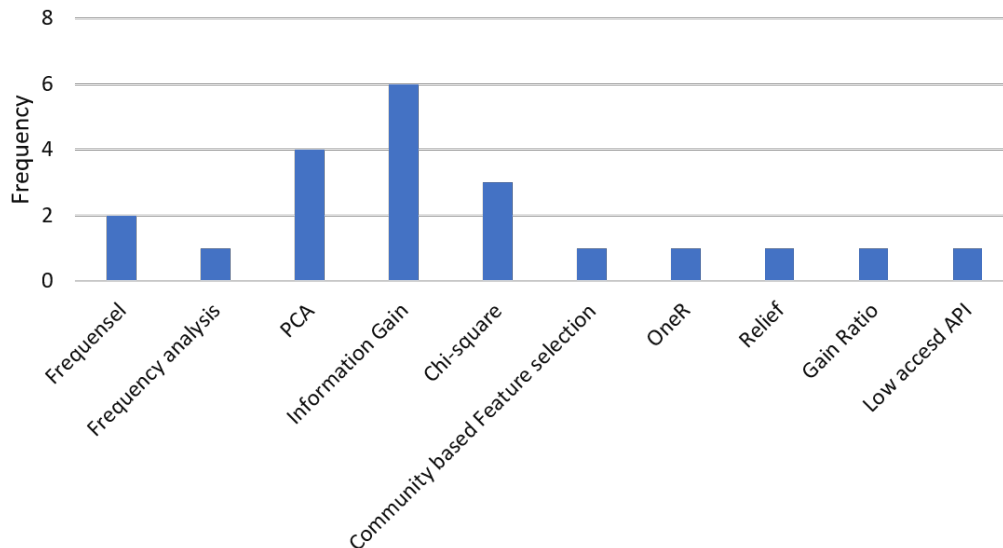


Figure 8 Frequency of filter feature selection algorithms

The approaches proposed by the papers above are relatively simple. For feature selection, only simple comparisons or applications are made. In this paper, we present

various usages and advanced applications of feature selection methods. Moreover, some papers merely evaluated the performance of the classification models, and it did not deal with how to use the model. In this paper, we cover the more practical framework with respect to a classification model application and continuous model training.

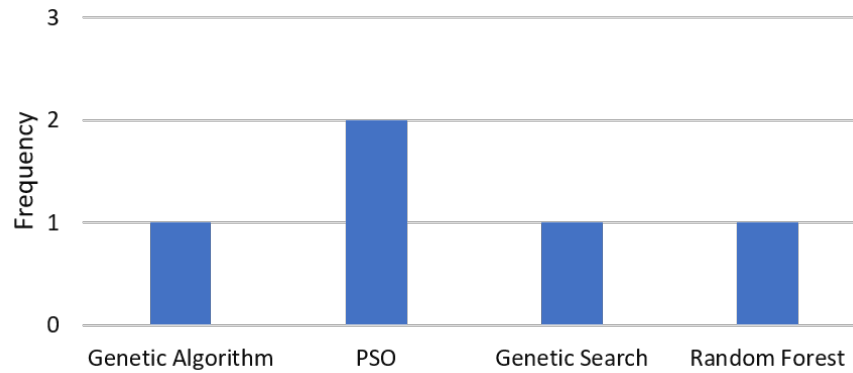


Figure 9 Frequency of wrapper feature selection algorithms

4 Framework of Android malware detection

This chapter explains this research's approach and the theological background of algorithms used for the experiment. Fig. 10 describes the process of the framework for Android malware detection and includes how we conduct the experiment.

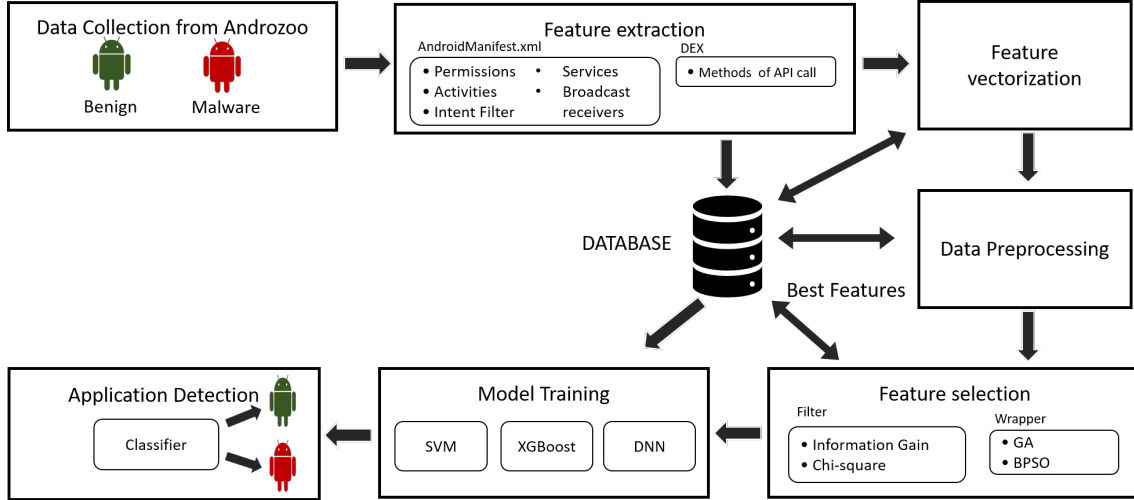


Figure 10 Flowchart of research methodology

First, Android applications in .apk file format are collected. Second, Section 4.1 discusses the way to extract features from the collected applications. Next, features are transformed into vectorized matrices, which is discussed in Section 4.2. After going through the data pre-processing (Section 4.3), the data is now ready for the next experimentation steps. In the experiment, we compare various combinations of feature selection algorithms and learning algorithms. Section 4.4 theoretically explains the feature selection algorithms used in this thesis. Section 4.5 contains a description of the machine learning and deep learning algorithms for classification used in this paper. Afterward, Section 4.6 addresses the methods to evaluate feature selection algorithms and learning algorithms. Finally, the best combination of a feature selection algorithm and a learning algorithm is selected, and a classifier is created based on the selected feature selection algorithm and learning algorithm. An unknown app can be therefore predicted to be either a malware application or a benign application.

4.1 Feature Extraction

In this thesis, static analysis is employed. We utilize the *Python* library, *Androguard* [Des], as a tool to extract features from the APK files. From *AndroidManifest.xml*, permission, activity, service, broadcast receiver, and intent filter feature categories

are extracted. In this paper, we only consider components including intent among app components (i.e., the content provider is not extracted). API methods are extracted from the *DEX* file. In terms of features from *AndroidManifest.xml*, *Androguard* extracts the string value corresponding to the *android:name* attribute from the syntax. We only use the last part of the string that seems to make the most meaningful when splitting a string. For instance, only *"ACCESS_NETWORK_STATE"* is used out of *"android.permissions.ACCESS_NETWORK_STATE"*. Each category of features is parsed as follows:

Permission: Permissions are represented by *<permission>* elements in an XML file. Listing 1 is an example of permission syntax in an XML file.

```

1 <permission android:name=".permissions.CD2_MESSAGE" android:protectionLevel="signature"/>
2 <uses-permission android:name="android.permission.INTERNET"/>
3 <uses-permission android:name="android.permission.VIBRATE"/>
4 <uses-permission android:name="android.permission.ACCESS_NETWORK_STATE"/>
5 <uses-permission android:name="android.permission.ACCESS_FINE_LOCATION"/>

```

Listing 1 Example of Permissions

Activity: Activities are defined by *<activity>* elements as described in Listing 2. The name extracted from *android:name* is the name of the class that implements the activity (i.e., an activity subclass) [Deva].

```

1 <activity android:name="com.philips.simpleset.gui.MainActivity">
2   <intent-filter>
3     <action android:name="android.intent.action.MAIN"/>
4     <category android:name="android.intent.category.LAUNCHER"/>
5   </intent-filter>
6   <intent-filter>
7     <action android:name="android.nfc.action.TECH_DISCOVERED"/>
8     <category android:name="android.intent.category.DEFAULT"/>
9   </intent-filter>
10 </activity>

```

Listing 2 Example of Activities

Service: Services are represented by *<service>* elements in an XML file (Listing 3). Also, the value of *android:name* refers to the subclass of service [Deva].

```

1 <service android:name="com.gcm.MyInstanceIdListenerService" android:exported="false">
2   <intent-filter>
3     <action android:name="com.google.android.gms.iid.InstanceID"/>
4   </intent-filter>
5 </service>
6 <service android:name="com.gcm.RegistrationIntentService" android:exported="false"/>

```

Listing 3 Example of Services

Broadcast Receiver: Broadcast Receivers are represented by `<receiver>` elements in an XML file (Listing 4). Receivers declares subclass of broadcast receiver. [Deva].

```

1 <receiver android:name="com.google.android.gms.GCMBroadcastReceiver" android:permission="com.google.
  android.c2dm.permission.SEND">
2   <intent-filter>
3     <action android:name="com.google.android.c2dm.intent.RECEIVE"/>
4     <action android:name="com.google.android.c2dm.intent.REGISTRATION"/>
5   </intent-filter>
6 </receiver>
7 <receiver android:name="com.google.android.gms.measurement.AppMeasurementReceiver" android:enabled="
  true" android:exported="false"/>

```

Listing 4 Example of Broadcast Receivers

Intent filter: Intent filter is declared by `<intent-filter>` element, which is located inside the activity, service, and broadcast receiver as demonstrated in Listings 2, 3, 4. Intent filter must contain at least one `<action>` element. Action defines an action or a specific state to be performed when a parent component is invoked. In addition, intent filter can contain `<category>` elements. The category contains information on the type of component that will handle intents. We extracted the *android:name* attribute of action and category belonging to the intent filter [Deva].

API method: *Androguard* analyzes a *DEX* file and extracts methods that are not defined in the application itself as external methods. Therefore, it can be said that the external methods are potential API call methods to communicate with external systems. Malicious behaviors can gain access to sensitive data through API calls. For example, *sendTextMessage()* can be exploited for SMS fraud.

4.2 Feature Vectorization


We transform the data of extracted features into binary-vectors. All extracted features are denoted as $F_T = [f_1, f_2, \dots, f_n]$. An application A has a feature set as F_a . Each feature of the application can be mapped into a vector $V_a = [v_1, v_2, \dots, v_n]$ according to Equation 4.1.

$$v_i = \begin{cases} 1, & \text{if } f_i \in F_a \\ 0, & \text{Otherwise,} \end{cases} \quad (4.1)$$

Thus, v_i is 1, if a feature f_i is represented in the application. In contrast, v_i is 0, a feature f_i is not contained in the application. Therefore, the feature vector of

the application can be translated into binary values as $V_a = [0, 1, 1, 0, 1, \dots]$. Fig. 11 describes how data is transformed and what structure it has after the binary vectorization process.

App name	label	Features
APK1	benign	F1:["WRITE_CALENDAR", "INTERNET", "ACCESS_WIFI_STATE", "WRITE_EXTERNAL_STORAGE", ...]
APK2	benign	F2:["INTERNET", "ACCESS_NETWORK_STATE", "WRITE_EXTERNAL_STORAGE", ...]
APK3	malware	F3:["ACCESS_LOCATION_EXTRA_COMMANDS", "ACCESS_COARSE_LOCATION", "INTERNET", ...]
APK4	malware	F4:["ACCESS_LOCATION_EXTRA_COMMANDS", "ACCESS_COARSE_LOCATION", "GET_ACCOUNTS", ...]
APK5	malware	F5:["READ_LOGS", "INTERNET", "GET_TASKS", "READ_EXTERNAL_STORAGE", ...]



App name	label	"WRITE_CALENDAR"	"INTERNET"	"ACCESS_COARSE_LOCATION"	"GET_TASKS"	"GET_ACCOUNTS"	...
APK1	benign	1	1	0	0	1	...
APK2	benign	0	1	0	0	0	...
APK3	malware	0	1	1	0	0	...
APK4	malware	0	1	1	1	1	...
APK5	malware	0	1	0	1	0	...

Figure 11 Example of feature vectorization

4.3 Data Pre-processing

During this step, features with very low variance are removed. There are many features with random names that make it difficult to assume their function, such as arbitrarily named features and features written as special characters. Also, there are features that are only found in very few applications or most applications have. Then, these kinds of features have no influence on the performance of the classification models. Thus, these features don't need to be taken into account in this experiment. We only select features with variance greater than 0.0099, $var(x) = 0.99(1 - 0.99)$. 0.0099 indicates that a feature is comprised of the 99% same value, either 0 or 1. For example, there is a feature named "zlzs". It can be considered as a random name that cannot be inferred any meanings and might be only invoked by one application. Therefore, it can be reasonable to remove these before the experiment. Through this process, data is cleansed. The need for data pre-processing will be briefly explained in Chapter 6.

4.4 Feature selection

This section discusses the theoretical background of the feature selection algorithm used in the paper. In this paper, IG and CHI are used as filter methods. These are the most used and adopted methods in the literature review. In terms of wrapper methods, a bio-inspired heuristic search technique, namely, Binary Particle Swarm Optimization (BPSO) and Genetic Algorithm (GA) are utilized. In addition, we discuss how to apply the selected filter and wrapper algorithms as ensemble feature selection and hybrid methods.

4.4.1 Information Gain

Information Gain (IG) measures the mutual dependence between a feature and a label [YJ97; Zha+18]. IG is the value obtained by subtracting the posterior uncertainty from the prior uncertainty. The prior uncertainty means the entropy of variable Y (i.e., label), which are benign or malware, is defined in Equation 4.2.

$$H(Y) = - \sum_{i=1}^n P(y_i) \log_2(P(y_i)) \quad (4.2)$$

The posterior uncertainty means the conditional entropy of variable Y when X (i.e., a feature) is given. The conditional entropy is calculated as Equation 4.3.

$$H(Y|X) = - \sum_{j=1}^m P(x_j) \sum_{i=1}^n P(y_i|x_j) \log_2(P(y_i|x_j)) \quad (4.3)$$

Hence, the IG shows how much uncertainty decreases for each feature, and it is defined in Equation 4.4. Therefore, a high value of IG indicates that a feature considerably decreases uncertainty and is informative. For instance, features are scored and ranked by IG as shown in Tab. 2 (based on features used in our experiment).

$$IG(X) = H(Y) - H(Y|X) \quad (4.4)$$

Table 2 Top10 features of Information Gain

Category	Feature name	value
API method	setVisibility	0.1349
API method	requestFocus	0.1318
API method	startActivity	0.1309
API method	setText	0.1294
API method	getExtras	0.1280
API method	scheme	0.1279
API method	getText	0.1276
API method	setContentView	0.1259
API method	getAttributeValue	0.1264
Permission	MOUNT_UNMOUNT_FILESYSTEMS	0.1260

4.4.2 Chi-square

Chi-square (CHI) measures the independence between a label and a feature [YJ97]. A high score of CHI indicates the significant dependence between a feature and a class, and the feature is therefore relevant. CHI is defined in Equation 4.5. c is denoted as a label of application (i.e., benign or malware), and t is denoted as the feature to be evaluated.

$$\chi^2(t, c) = \frac{N \times (AD - CB)^2}{(A + C) \times (B + D) \times (A + B) \times (C + D)} \quad (4.5)$$

where N is the total number of applications (i.e., $N = A + B + C + D$), A and B are the number of times that malware and benign apps co-occur with t , C and D are the number of times that malware and benign apps occur without t [Zha+16]. In this thesis, CHI is implemented by the *Python* library provided by *Scikit-learn* [Ped+11]. For instance, Tab. 3 demonstrates how features are scored and ranked by CHI.

Table 3 Top10 features of Chi-square

Category	Feature name	value
Permission	MOUNT_UNMOUNT_FILESYSTEMS	373.0437
Permission	SYSTEM_ALERT_WINDOW	297.7732
Activity	WXEntryActivity	294.912
Permission	CHANGE_WIFI_STATE	294.5684
Permission	GET_TASKS	291.2124
Intent filter	USER_PRESENT	289.8119
Permission	WRITE_SETTINGS	256.6192
Service	DownloadService	243.0556
Permission	READ_LOGS	240.3105
Permission	CHANGE_NETWORK_STATE	236.4371

4.4.3 Genetic Algorithm

Genetic Algorithm (GA) utilized for this experiment is organized based on [ES15]. GA is one of the variants of Evolutionary Algorithm (EA) and frequently adopted for the data type of bistring (e.g., 0/1) highly relevant to the data purpose of this experiment [ES15]. EA is inspired by the concept of natural evolution [ES15; Fat+19]. To be specific, features are fed into GA and each individual (i.e., a subset of features) in the GA is subsequently fed to a fitness function including a learning algorithm. Throughout each generation (i.e., iteration), the subsets of features are selected based on a fitness function and evolve to the next generation. Consequently, it ends up generating the best subset of features for the learning algorithm [ES15; Fat+19]. GA consists of following components [ES15]:

- **Fitness function:** The fitness function is to measure each individual and determine which individuals meet requirements and would be passed on next generation [ES15]. In this thesis, a subset of features is fed into a learning algorithm and the result of the fitness function is accuracy score.
- **Population:** Population means groups of individuals. In other words, how many individuals are in a generation. The size of population is denoted as μ [ES15]. In this experiment, an individual is represented as an array of binary values. For instance, if there are features $[f1, f2, f3, f4, f5]$,

one randomly generated subset (i.e., individual) can be $[1, 0, 1, 0, 1]$. When this subset is passed to the fitness function, the data with features corresponding to 1 (i.e., $[f_1, f_3, f_5]$) is given to a classifier.

- **Parent Selection:** The purpose of parent selection is to select the better individuals to become parents of the next generation [ES15]. Tournament selection, which is a simple and fast way, is utilized for this experiment. It picks k individuals randomly with replacement. The individuals (i.e., subsets) are ranked based on accuracy defined by the fitness function, and an individual with the highest accuracy is selected as a parent. If there is a tie accuracy score, an individual with a lower number of features will become the winner. These processes are repeated until we get two parents.
- **Recombination:** Recombination is also called crossover. The genomes of two parents are combined to produce the specific size of offspring which is denoted as λ . Recombination occurs with crossover probability (p_c). [ES15]. In this experiment, we implemented two crossover strategies for binary representation. First, '1-point' crossover chooses a random index r in the range of the array of features. From that point, the genomes of the parents who were split in two are mixed in half and half in order to create two children [ES15]. Fig. 12 illustrates how '1-point' crossover works. Second, 'uniform' crossover chooses the parent randomly to originate from for each gene in isolation [ES15]. The second offspring is the inverse of the first one, which is illustrated in Fig. 13

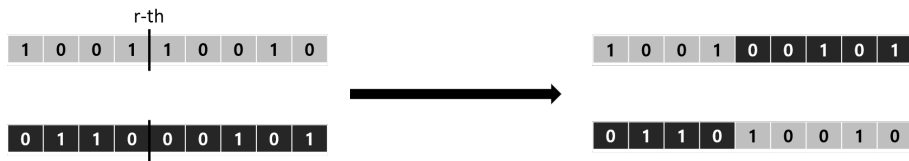


Figure 12 1-point crossover



Figure 13 Uniform crossover

- **Mutation:** The role of mutation is used to modify an offspring. For binary data type, bit mutation can be applied [ES15]. As illustrated in

Fig. 14, k genes are randomly chosen, and their values are flipped. The mutation occurs with mutation probability (p_m).



Figure 14 Bit mutation

- **Survivor Selection:** Survivor selection is similar to parent selection. In contrast, it is applied after the creation of the offspring are done and just before the next generation begins or the end of the iteration. In this experiment, $(\mu+\lambda)$ selection and (μ,λ) selection are used. $(\mu+\lambda)$ selection selects the best μ individuals out of $P_\mu \cup P_\lambda$. (μ,λ) selection selects the best μ individuals out of P_λ . Thus, the whole current population is not taken into account for the next generation.

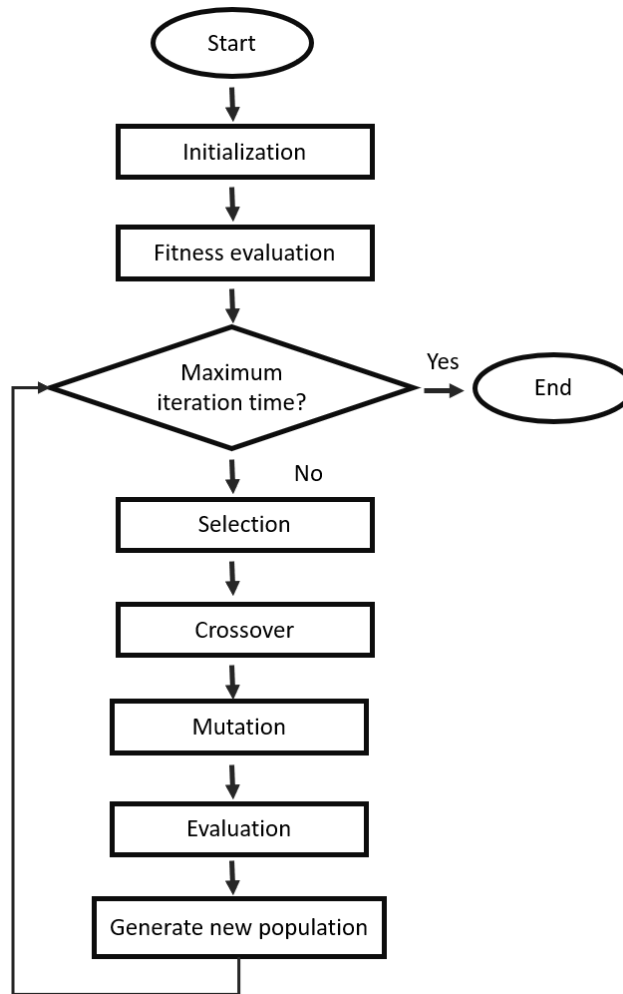


Figure 15 Flowchart of GA

Fig. 15 illustrates the process of GA for feature selection. In the first step, it initializes GA parameters and randomly generates an initial population. In step 2, each individual's fitness score is computed. One individual represents a subset of features. Data with the subset of features is divided into training data and test data at a 75% to 25% ratio. Subsequently, training data is trained using a learning algorithm. The trained model is evaluated by the test data. The accuracy score, which is the evaluation result, is the individual's fitness score. A detailed description of model evaluation is covered in Section 4.6. In step 3, two parents are selected with a good fitness score based on a parent selection strategy. In step 4, crossover using the selected parents is performed on the basis of the given crossover probability. In step 5, it performs mutation to the generated offspring on the basis of the given mutation probability. The newly created individual is also fed into a learning algorithm, and the fitness score of the individual is subsequently evaluated in step 6. Then, it repeats steps 3, 4, 5 and 6 until the number of offspring is fulfilled. In step 7, it decides which individuals to save based on a survivor selection strategy and generates a new population. Steps 3 to 7 are repeated until the given number of iterations is reached. Finally, an optimally selected subset of features comes out as a result.

Table 4 Parameter GA

Parameter	value
Recombination	1-point, uniform
Mutation	bit-flip
Parent selection	tournament (k=4)
p_m	0.2
p_c	0.8
μ	20
λ	20
Survivor Selection	$(\mu+\lambda)$, (μ, λ)
Number of generations	100

We refer to *pyeasyga* Python library [Rem] to reinvent and implement GA suitable for this framework. Tab. 4 summarizes the value of the chosen parameters for GA. We set the values of parameters p_m , p_c , and the number of generations by referring to the default values of *pyeasyga* library. In addition, we also implement multi-

parallel processing to reduce computational time. Algorithm 1 describes the process in pseudocode.

Algorithm 1 Pseudocode of Genetic Algorithm

```

iteration = 0;
initialize parameters;
initialize individuals for  $P_\mu$ ;
Evaluate individuals;
while iteration <= maxIteration do
    current_member = 1;
     $P_\lambda = []$ ;
    while current_memeber <=  $\lambda$  do
        parent1 = selectParent();
        parent2 = selectParent();
        if crossover probability is fulfilled then
            Apply recombination to generate an individual i;
            if mutation probability is fulfilled then
                Mutate i
            Evaluate i;
            Add i to  $P_\lambda$ ;
        current_member = current_member + 1;
    Apply survivor selection to Select  $\mu$  individuals for the next iteration;
    iteration = iteration + 1;

```

4.4.4 Binary Particle Swarm Optimization

Particle Swarm Optimization (PSO) is first devised by J. Kennedy and R. Eberhart [KE95]. The concept of PSO is inspired by the social behaviour of some species, such as bird flocking or fish schooling. A set of candidate solutions is represented as a particle. A particle is similar to an individual in EA. The i -th particle is denoted as $X_i = (x_{i1}, x_{i2}, \dots, x_{iD})$. Each particle is dispersed as a point in the D-dimensionality of a search space [SE98]. Particles traverse their search space to find the optimized position. The position of each particle's best performance (e.g., fitness score) in all iterations is recorded and denoted as $P_i = (p_{i1}, p_{i2}, \dots, p_{iD})$ [SE98]. Each particle has velocity which means the rate of change for a particle's position is represented as $V_i = (v_{i1}, v_{i2}, \dots, v_{iD})$ [SE98]. While each particle is moving, its velocity and position are updated based on its neighbors. In the same manner as an individual in GA, a particle is comprised of binary (i.e., $x_{id} \in \{0, 1\}$). Hence, one of PSO variants for a binary space, Binary Particle Swarm Optimization (BPSO) proposed in [KE97], is utilized for this thesis. The update rule of velocity is defined as Equation 4.6. The role of inertia weight w is to balance the global search and local search by tuning the impact of previous velocity. The $c1$ (cognitive parameter) and $c2$ (social parameter)

show positive constants. r_1 and r_2 have a random value distributed between 0 and 1. The g of p_{gd} indicates that the index of the particle having the best performance among neighbors in all iterations (i.e, Global best).

$$v_{id}^{t+1} = \underbrace{w * v_{id}^t}_{part1} + \underbrace{c_1 \times r_1 \times (p_{id}^t - x_{id}^t)}_{part2} + \underbrace{c_2 \times r_2 \times (p_{gd}^t - x_{id}^t)}_{part3} \quad (4.6)$$

The second and third parts of Equation 4.6 affect the movement of particles with different intention. The second part (i.e., $c_1 \times r_1 \times (p_{id}^t - x_{id}^t)$) implies the cognitive think to search the best position individually and locally. The third part (i.e., $c_2 \times r_2 \times (p_{gd}^t - x_{id}^t)$) represents the social and collaborative action of a group or neighbors [SE98; KE97]. Each particle moves to the new position according to Equation 4.7 and 4.8. $rand()$ is a random number from a uniform distribution in $[0.0, 1.0]$ and $S(x)$ is a sigmoid limiting transformation [KE97].

$$x_{id}^{t+1} = \begin{cases} 0, & \text{if } rand() \geq S(v_{id}^{t+1}) \\ 1, & \text{if } rand() < S(v_{id}^{t+1}) \end{cases} \quad (4.7)$$

$$S(x) = \frac{1}{e^{-x}} \quad (4.8)$$

In order to evaluate a fitness score for each particle, Equation 4.9 used in Susana et al. [Vie+13] is applied into an objective function. A subset of features with the maximized model performance and the minimized number of used features can be considered as the optimized subset, and it meets our thesis's research goals. P is a performance score of a classifier. Accuracy score is used in this thesis. N_f is the number of the subset's features, and N_t is the total number of features. α defines the trade-off between the performance and the subset size.

$$f(X) = \alpha(1 - P) + (1 - \alpha)(1 - \frac{N_f}{N_t}) \quad (4.9)$$

The steps involved in feature selection using BPSO is shown in Fig. 16. First, BPSO parameters are initialized and particles with random position and velocity are generated. One particle represents a subset of features. In step 2, it computes a fitness score for each particle. Data with the subset of features is divided into training data and test data at an 75% to 25% ratio. Subsequently, training data is trained using a learning algorithm. The trained model is evaluated by the test data. The accuracy score, which is the evaluation result becomes P and the number of

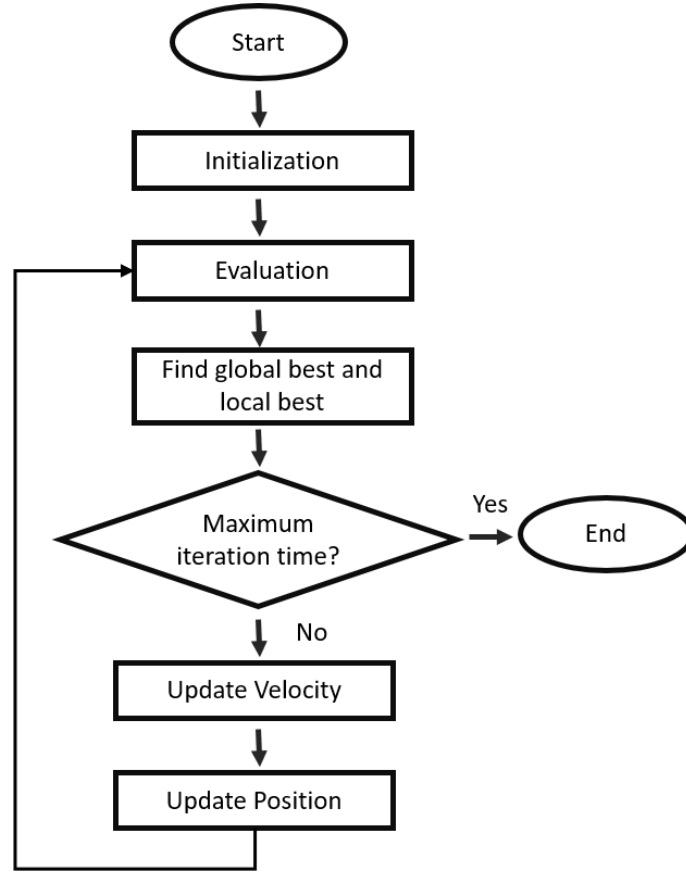


Figure 16 Flowchart of BPSO

particle's features becomes N_f in Equation 4.9. In step 3, it determine the local best position (p_{id} in Equation 4.6) and the global best position (p_{gd} in Equation 4.6) of each particle. In step 4, the velocity of each particle is updated based on Equation 4.6. In step 5, the position of each particle is updated based on Equation 4.7. Steps 2 to 5 are repeated until the given number of iteration is achieved. Finally, an optimally selected subset of features comes out as a result. The result corresponds to the position of global best.

In this thesis, we utilize *PySwarms* [VMib] *Python* library. Tab. 5 summarizes the value of the chosen parameters for BPSO. We determine the values of the parameters by referring to the example of the *PySwarms* library [VMia]. The number of particles and parameter k are set to 20 (i.e., population size of GA used in this thesis) different from the example. Parameter k indicates the number of neighbors to be considered, and parameter p is the Minkowski p -norm to use. $p = 1$ is the sum-of-absolute values (or $L1$ distance), whereas 2 is the Euclidean (or $L2$ distance). Setting the number of neighbors equal to the number of particles will find a global-best solution. The number of iterations is set to 100, which is the same value as the number of generations in GA used in this thesis. In this thesis, the value of α in Equation

Table 5 Parameter of BPSO

Parameter	value
Number of particles	20
c1	0.5
c2	0.5
w	0.9
k	20
p	2
Number of iterations	100

4.9 is set to 0.88, the value given in the example of the *PySwarms* library [VMia]. Algorithm 2 describes the process in pseudocode.

Algorithm 2 Pseudocode of the Binary Particle Swarm Optimization

```

iteration = 0;
initialize parameters;
 $x_{id}, v_{id}$  for all  $i = 1, 2, \dots, N$  and  $D = 1, 2, \dots, n$ ;
Evaluate the fitness value of each particle based on the objective function;
while iteration  $\leq$  maxIteration do
  for  $i = 1$  to  $N$  do
    Update velocity  $V_i$ ;
    Update the position of particle  $X_i$ ;
    Calculate the fitness score of the new  $X_i$ 
    if the fitness score of  $X_i$  is better than the fitness score of  $P_i$  then
       $P_i = X_i$ ;
    if the fitness score of  $X_i$  is better than the fitness score of  $P_g$  then
       $P_g = X_i$ ;
  iteration = iteration + 1;

```

4.4.5 Ensemble Feature selection

Using a heterogeneous ensemble approach for feature selection, we can combine the ranking results of the filter methods, CHI and IG, to create new indicator for feature selection. As described in Fig. 17, the heterogeneous approach is that different feature selection methods are applied to the same data and the results are aggregated using a combination method [BA19]. In order to aggregate the ranking results of CHI and IG, we applied arithmetic mean of two ranking results. For instance, features are ranked by the ensemble approach as shown in Tab. 6.

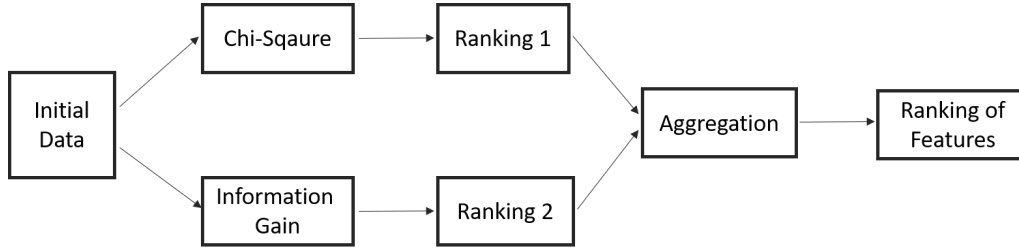


Figure 17 A scheme of heterogeneous ensemble approach

Table 6 Top10 features of Ensemble Feature selection

Category	Feature name	CHI rank	IG rank	mean rank
Permission	MOUNT_UNMOUNT_FILESYSTEMS	1	10	5.5
Permission	GET_TASKS	5	131	68
API method	isVisible	48	92	70
Permission	SYSTEM_ALERT_WINDOW	2	141	71.5
API method	scheme	139	6	72.5
API method	getComponentName	115	46	80.5
Permission	CHANGE_WIFI_STATE	4	159	81.5
API method	dispatchKeyEvent	153	12	82.5
API method	invalidateSelf	107	72	89.5
API method	getFragmentManager	141	62	101.5

4.4.6 Hybrid Feature selection

Hybrid methods are the combination of different feature selection methods to use benefits of different methods together [Pou18]. In this thesis, we combine the filter method (i.e., CHI, IG, and ensemble feature selection) and the wrapper method (i.e., GA and BPSO). As described in Fig. 18, the role of the filter methods is pre-feature selection, and selected features are passed on to the wrapper method to find the best subset of features. In this sense, the hybrid method takes advantage of both methods. Irrelevant and redundant features are removed by the filter method by selecting the top N features. This can reduce the time and computational complexity of the wrapper methods, since the wrapper methods just need to consider less features and more informative features in the beginning [Naq11].



Figure 18 A scheme of hybrid method

4.5 Classification Model

The objective of malware detection deals with the problem of classification whether an app is malicious or benign. Sample data is fed to a machine learning or deep learning algorithm in order to train and test a classification model. The classification model can predict whether an unknown app is malicious or benign. In this thesis, three classification algorithms are applied, namely SVM, XGBoost, DNN. The following contents briefly introduce these algorithms.

4.5.1 SVM

Support Vector Machine (SVM) is one of the machine learning techniques used for classification, regression, and other learning tasks. SVM identifies two classes with an optimal separating hyper-plane with a maximum margin between two classes (Fig. 19) [HTF09].

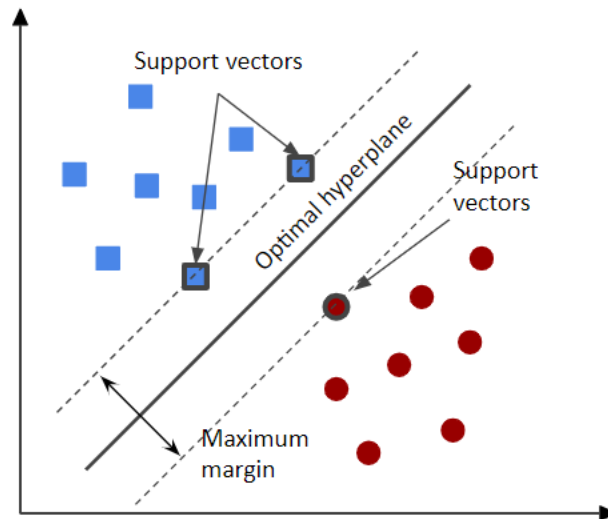


Figure 19 Support Vector Machines

SVM solves the Lagrange dual problem as Equation 4.10 [Han+20; CL11; HTF09].

$$\begin{aligned}
& \max_{\alpha} \quad \sum_{i=1}^n \alpha_i - \frac{1}{2} \sum_{i=1}^n \sum_{j=1}^n \alpha_i \cdot \alpha_j \cdot y_i \cdot y_j \cdot (\phi(x_i) \cdot \phi(x_j)) \\
& \text{subject to} \quad \sum_{i=1}^n y_i \alpha_i = 0 \\
& \quad \quad \quad 0 \leq \alpha_i \leq C
\end{aligned} \tag{4.10}$$

where x_i is given data vectors and y_i indicates two classes for x_i . α_i is the Lagrange multipliers. C is the regularization parameter. The parameter C tunes a trade-off between the size of the margin and the error about misclassified points. A smaller C has a tendency to generate a model with a larger margin. In other words, the model is highly regularized so that the influence of the points around the decision boundary is weak. Therefore, a hyper-plane close to linear is formed. So, SVM maximizes the dual problem subject to $\sum_{i=1}^n y_i \alpha_i = 0$ and $0 \leq \alpha_i \leq C$. $\phi(x_i) \cdot \phi(x_j)$ is the inner product two observations. $\phi(x_i)$ transforms x_i and maps x_i into a transformed (i.e., higher-dimensional) space. For this, the knowledge of the kernel function (Equation 4.11) that yields the inner product in higher dimensions is applied.

$$K(x_i, x_j) = \phi(x_i) \cdot \phi(x_j) \tag{4.11}$$

For this thesis, we employed an Radial Basis Function (RBF) defined as Equation 4.12 as a kernel function.

$$K(x_i, x_j) = \exp(-\gamma \|x_i - x_j\|^2) \tag{4.12}$$

where γ determines the distance over which a single training sample exerts influence, with low values meaning ‘far’ and high values meaning ‘close’. To implement SVM, the *Python* library provided by *Scikit-learn* [Ped+11] is utilized in this thesis. When it comes to SVM using RBF kernel, we configure two tuning parameters as $C = 100$ and gamma (γ) = 0.0001.

4.5.2 XGBoost

XGBoost [CG16] is one of the machine learning algorithms and stands for Extreme Gradient Boosting. As the name suggests, this is based on the Gradient Boosting. The idea of boosting is an ensemble of weak learners (e.g., typically decision trees)

in series, which results in a strong learner. The boosting algorithm enhances performance by increasing the weights of misclassified observations in the previous model during sequential learning of multiple weak learners [HTF09]. Gradient boosting strengthens an ensemble of weak learners by minimizing the loss function using gradient descent rather than manipulating weights of observations [Fri01; HTF09].

XGBoost is an improvement over the Gradient tree boosting, which is also known as Gradient Boosting Machine (GBM) and offers advantages. It is highly scalable due to an efficient approximation for finding splits within each tree based on the exact greedy algorithm. XGBoost can exploit sparse data in a unified way. It also can prevent overfitting using regularization and shrinkage [Fri02]. In the case of standard GBM, there is no overfitting regularization, but XGBoost has strong durability with its own overfitting regularization. The regularization restricts the complexity of tree structure. Shrinkage is similar to a learning rate in stochastic optimization. Shrinkage decreases the impact of each individual tree and leaves space for future trees to improve the model. Another advantage of XGBoost is that it provides a flexible learning system to manipulate various parameters.

Table 7 Parameter of XGBoost

Parameter	value
booster	gbtree
objective	binary:logistic
n_estimator	50
eta	0.3
gamma	0
max_depth	8
colsample_bylevel	0.9
colsample_bytree	0.8

In this thesis, we utilize XGBoost [CG] library. Tab. 7 summarizes the set of parameters used for this thesis. The booster parameter refers to the type of learner. *gbtree* uses tree-based models as weak learners. The objective determines the learning task, thus the type of the target variable. This thesis deals with a binary problem (benign or malware), so we set it to *binary:logistic*. The parameter *n_estimator* is the number of weak learners to create. The rest of the parameters are for the tree booster. The *eta* parameter corresponds to the learning rate. It helps to prevent overfitting

in the boosting process by giving a weight for each boosting step. As the value of the gamma parameter increases, the tree depth decreases, resulting in a conservative model. The parameter *colsample_bytree* is the subsample ratio of features used for each tree. The parameter *colsample_bylevel* is the subsample ratio of features for each level [CG].

4.5.3 DNN

Deep Neural Network (DNN) is one of the common deep learning algorithms used for classification, which is comprised of more than two hidden layers of Artificial Neural Networks (ANNs). ANNs are inspired by the characteristics of biological neural networks, a structure in which several neurons are connected. While DNN trains large-scale observational data, weights and biases of neurons and nodes are adjusted by gradient descent using back-propagation to find where the value of loss function is minimized [GBC16; Nie15]. In this thesis, we utilize *Keras* [Cho+15] which is a high-level API for *Tensorflow* [Mar+15] to build a DNN model.

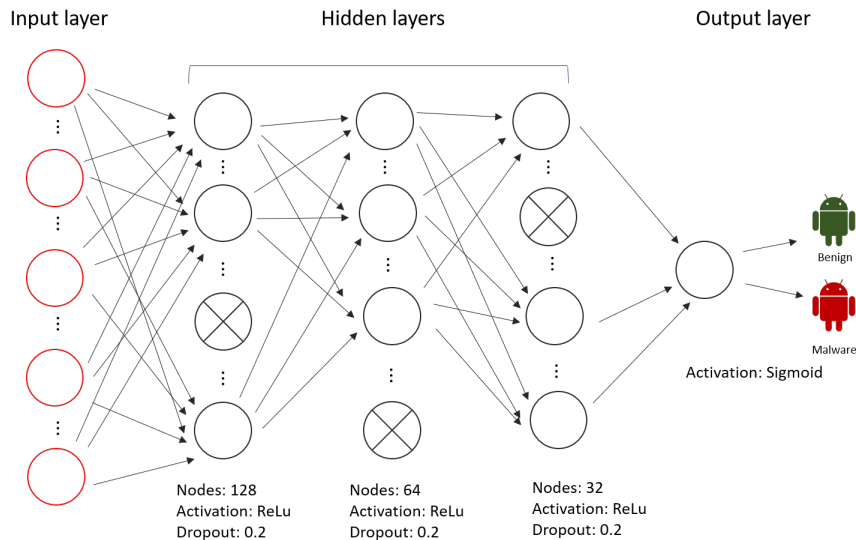


Figure 20 Overview of DNN structure

Fig. 20 illustrates the overview structure of the DNN used for this thesis. It consists of an input layer and three hidden layers, a output layer. The input layers contains nodes for each of the features. The nodes of the input layer are connected to a series of hidden layers. The three hidden layers consist of 128, 64, and 32 nodes, respectively, and Rectified Linear Unit (ReLu) activation function is applied to all. All layers use drop out (rate = 0.2). Drop out literally drops out randomly selected nodes with a given drop-out rate. It is helpful for preventing overfitting. Lastly, the output layer uses sigmoid as the activation function. During the training, the

parameters are set as described in Tab. 8. The loss function, *binary_crossentropy*, is suitable for two classes and works well in combination with the sigmoid activation function as output.

Table 8 Parameter of DNN

Parameter	value
loss function	binary_crossentropy
optimizer	Stochastic Gradient Descent (SGD), learning rate = 0.1
epochs	100

4.6 Evaluation

This section covers a variety of metrics to evaluate the performance of the classification model. This thesis uses some numeric metrics based on a confusion matrix (Tab. 9), which is a common performance assessment method for binary classification. True Positive (TP) indicates the number of the malicious applications that are accurately classified as malicious. False Negative (FN) indicates the number of the malicious applications that are wrongly classified as benign. True Negative (TN) denotes the number of benign applications that are exactly classified as benign. False Positive (FP) represents the number of benign applications that are wrongly classified as malicious.

Table 9 Confusion Matrix for malware detection

		Predicted	
		Malware	Benign
Actual	Malware	TP	FN
	Benign	FP	TN

Using the above, we can calculate the following metrics:

Accuracy: the ratio of benign and malicious applications ($TP + TN$) that are correctly classified to the total number of applications.

$$Accuracy = \frac{TP + TN}{TP + TN + FN + FP} \quad (4.13)$$

Precision: the ratio of malicious applications (TP) that are accurately classified to all applications (TP + FN) that are predicted as malware.

$$Precision = \frac{TP}{TP + FP} \quad (4.14)$$

Recall: the ratio of malicious applications (TP) that are correctly classified to all applications that are actually malicious applications (TP + FN).

$$Recall = \frac{TP}{TP + FN} \quad (4.15)$$

F1-score: is the harmonic mean of precision and recall.

$$F1 = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall} = \frac{2 \cdot TP}{2 \cdot TP + FP + FN} \quad (4.16)$$

According to all metrics, higher values means that the model has better performance. On top of that, K-fold cross-validation is conducted to evaluate. As we set K to 10, data is randomly split into ten blocks. Thus, each block, 10% of data, acts as test data, and the remaining 90% of data is utilized as training data. As 10 folds are repeated, *Accuracy*, *Precision*, *Recall*, and *F1-score* are measured for each fold. As a result, the averages of these values become an indicator for evaluating the model's performance.

5 Implementation

In this section, we discuss this thesis’s project package for the Android malware detection framework and how we structure and conduct this thesis’s experiment. For the project, we utilize *Python 3.6*, *MongoDB*, and *Docker* [Mer14]. Due to the nature of this project, a relational data management system is not necessary. Therefore, using *MongoDB*, one of NoSQL database, can handle large-scale data more flexibly. *Docker* is a container-based open source virtualization platform. *Docker Compose* is written in yaml format and allows developers to manage the execution of multiple containers at once. In this thesis, four containers for, running *Python*, *MongoDB*, a REST API, and a web application, are built. The API and web application will be covered in Chapter 7. The structure of the project directory is shown in Fig. 21. This project can be run using *Docker*.

```

android_feature_analysis
├── feature_selection/
├── source/
├── myweb/
├── classifier/
├── csv_results/
├── Dockerfile
├── db.py
├── docker-compose.yml
├── feature_extraction.py
├── feature_vectorization.py
├── data_preprocessing.py
├── select_best_features.py
├── generate_detection_model.py
├── api.py
└── requirement.txt

```

Figure 21 Structure of Project Directory

The *feature_selection* directory contains *Python* files regarding algorithms for feature selection, such as IG, GA, and for executing feature selection algorithms, such as ranking features based on CHI or running BPSO with SVM. The directory, *source*, is where APK files are placed. Project outputs can be saved in .csv format in the *csv_results* directory. The *requirements.txt* file contains *Python* packages that are required to run the project. *db.py* contains configurations for connecting *Python* and *MongoDB*. We select the feature selection algorithm that best suits the purpose of the project in this paper, and the *select_best_features.py* file contains the algorithm. The *generate_detection_model.py* file contains the learning algorithm that showed the best performance in the experiment, and the classification model for

malware detection is created using the learning algorithm, and the model is stored in the *classifier* directory. We discuss further on implementing this project in the following sections.

5.1 Data Collection

We exploited *AndroZoo* [All+16] as data source. *AndroZoo* offers a CSV file containing lists of APKs to download through an API. This CSV file contains several fields such as SHA-256, dex_size, dex_date, pkg_name, vt_detection. To download, we specify vt_detection field. vt_detection indicates the number of anti-virus tools that detected this APK as malware from *VirusTotal* [Vir]. First, we download 1500 APKs where the value of vt_detection is 0, and these APKs will be treated and labeled as 'benign' in the future steps. Second, 1500 APKs where the value of vt_detection is larger than 0 are downloaded, and these APKs will be treated and labeled as 'malware' in the future steps.

$$\begin{aligned} \text{if } vt_detection = 0 : \text{APK is Benign} \\ \text{if } vt_detection > 0 : \text{APK is Malware} \end{aligned} \quad (5.1)$$

Consequently, the benign APKs and malware APKs are stored in benign and malware directories, respectively, under the *source* directory (Fig. 22).

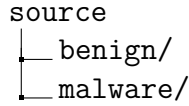


Figure 22 Structure of source Directory

5.2 Data preparation

After APKs are collected, we can extract features by running *feature_extraction.py*. During the extraction process, APKs in the *source/benign* directory are labeled as "benign", APKs in the *source/malware* directory are labeled as "malware". The extracted features for each APK are stored in the database in JSON format, and the category names (i.e., permission, activity, service, receiver, intent, and API method) are used as keys (Fig. 23). For each APK, the package name, the label of the APK, and the extracted features in the form of JSON are stored in the database.

Next, the extracted features can be transformed into the binary-vector matrix by running *feature_vectorization.py*. The output of this step is stored in the database

```

{
  "permissions":["WRITE_CALENDAR", "ACCESS_FINE_LOCATION", ... ],
  "activities":["SplashActivity", "SettingsActivity", ... ],
  "services":["EngageGeoFenceService", ... ],
  "receivers":["GeofenceRegistrationReceiver", "GCMCustomBroadcastReceiver", ... ],
  "intent-actions":["ACTION_QUICK_CHECKIN", "VIEW", ... ],
  "intent-categories":["BROWSABLE", ... ],
  "API_methods":["setUserAgent", "getSortKey", ... ]
}

```

Figure 23 Example of extracted features in JSON format

differently from the vectorized table depicted in Fig. 11 for the efficiency of data management. Column names (i.e., extracted features) and binary vector values are separated and stored in the database as different tables. Then, the column names are saved in JSON format for each category, as illustrated in Tab. 10. Intent are stored separately as action and category, but when actually used, they are used as one category.

Table 10 Example of Table for features (after vectorization)

Categories	Feature name
permissions	{"feature":["ACCEPT_HANDOVER", "ACCESSORY_FRAMEWORK", ...]}
activities	{"feature":["AccountInfoSafeActivity", "AutoBoostActivity", ...]}
services	{"feature":["DeviceIdService", "CallService", ...]}
receivers	{"feature":["CustomBroadcastReceiver", "SystemReceiver", ...]}
intent-actions	{"feature":["ACTION_VIEW_DOWNLOADS", "ACTION_VIDEO_CALL", ...]}
intent-categories	{"feature":["LAUNCHER", "EVENT_GUIDE_CONTROLLER", ...]}
API methods	{"feature":["GetVisible", "GetLocation", ...]}

As illustrated in Tab. 11, arrays of binary vector values are stored in JSON format in a table for each category. Even after data pre-processing (*data_preprocessing.py*), the output is stored in the database in this way.

Table 11 Example of Table for binary vector (permission)

App name	Label	Binary vectors
com.example1	benign	{"binary": [0,1,1,0,0...]}
com.example2	malware	{"binary": [1,0,1,0,0...]}

5.3 Experiment procedure

We carried out our experiments on a computer with hardware specifications described in Appendix A. In the experiment, we first discuss the necessity of pre-processing by comparing the performance of the features before pre-processing with the features remaining after pre-processing. The features resulting from data pre-processing are used in the next experiments.

Next, the thresholds for selecting the best features of the filter methods (i.e., CHI, IG, and the ensemble feature selection) are set to the top 1%, 5%, 25%, 50%, and 75% score and 'mean' (i.e., features with above the mean of the score). Each threshold is trained and evaluated on three machine learning and deep learning algorithms (i.e., SVM, XGBoost, DNN). Subsequently, we choose the threshold that shows the best performance for each method by comparing the performance of each threshold. Finally, considering only the threshold that showed the best performance for each method, the best method is selected among CHI, IG, and the ensemble feature selection.

The next step is the wrapper methods. First, we find the survival selection strategy and the recombination strategy that best suits GA. '1-point' and 'uniform' are considered and compared to find the suitable value for the recombination parameter. (μ, λ) and $(\mu + \lambda)$ are considered and compared to find the best value for the survival selection parameter. One of the three learning algorithms is applied to the fitness function of GA to train and evaluate an individual during iteration. In this experiment, a total of 12 subsets are obtained from GA. For example, one optimal subset of features is returned by a GA with '1-point' recombination, $(\mu + \lambda)$ survivor selection, and SVM for fitness function. Then, the subset of features is trained by SVM and evaluated. In this way, 12 subsets are generated, and their performance and evolutionary trends are considered to find the best combination of strategies. Likewise, for BPSO, a total of three optimal subsets can be created by applying each of the three learning algorithms to the fitness function of BPSO, such as a BPSO

with SVM, a BPSO with XGBoost, a BPSO with DNN. Finally, we compare and discuss the performance results of GA and BPSO.

Hybrid methods were created by combining the methods that showed good performance in the filter method experiment and the methods that showed good performance in the wrapper method experiment. Finally, we compare and discuss the hybrid methods, the filter methods, and the wrapper methods and select the most appropriate feature selection method and learning algorithm for this framework.

6 Experiment results and discussion

In this chapter, we show the results of the experiment and discuss them. Section 6.1 discusses the necessity of data pre-processing by comparing results before and after data pre-processing. Section 6.2 describes the experimental results of filter methods, and the results are discussed in Section 6.3. Experimental results of wrapper methods are covered in Section 6.4, followed by Section 6.5 that discusses the experimental results. Experimental results of a hybrid method are covered in Section 6.6. Finally, we discuss all methods and choose the best feature selection and learning algorithm in Section 6.7.

6.1 Data Pre-processing

We obtained 119,427 features in total. Tab. 12 summarizes the number of features per category as well as the total number of features.

Table 12 Number extracted features

Categories	Number
Permission	660
Activity	48,954
Service	3,045
Broadcast Receiver	2,229
Intent filter	4,027
API method	60,512
Total	119,427

Tab. 13 represents the performance results when training a learning algorithm with all features that are not pre-processed. The classifier with XGBoost shows slightly better performance than the classifier with SVM in terms of accuracy, precision, F1-score. Although the SVM classifier performs better than the XGBoost classifier in recall, the difference can be seen as very slight and almost the same. When it comes to all evaluation metrics, the DNN classifier outperforms the other two classifiers. In the table, the column, 'Avg. Time', represents the average of the 10 training times while running 10 folds. According to average time, the XGBoost takes an average of 250s to train 119,427 features. On the other hand, SVM and DNN take 748s and

718s, respectively. In this respect, SVM has the longest training time and the worst performance when it comes to training all features.

Table 13 Performance of all features

Models	Accuracy	Precision	Recall	F1	Avg. Time
SVM	0.8433	0.8518	0.8331	0.8417	748s
XGBoost	0.8553	0.8731	0.8324	0.852	250s
DNN	0.9197	0.9292	0.9096	0.9186	718s

Tab. 14 summarizes the number of features after data pre-processing (explained in Section 4.3). Tab. 15 represents the performance results when training a learning algorithm with features that are pre-processed. In terms of all evaluation metrics, the classifier with XGBoost shows slightly better performance than the classifier with SVM. The classifier with DNN has the best performance in all evaluation metrics. Likewise, XGBoost shows the fastest average training time (23s) compared to the other two (58s and 59s).

Table 14 Number features after pre-processing

Categories	Number
Permission	106
Activity	258
Service	70
Broadcast Receiver	57
Intent filter	116
API method	9,614
Total	10,221

As illustrated in Tab. 12 and Tab. 14, the number of features decreased from 119,427 to 10,221. Although the dimensions of the data decreased, the performance of classification models trained on each learning algorithm did not deteriorate. The accuracy of the SVM trained with the pre-processed features is 0.8413, and the accuracy of the SVM using all the features is 0.8433, showing similar performance without significant difference between the two. According to XGBoost, the accuracy has increased slightly from 0.8553 to 0.86, but it is still believed to have a similar performance.

Table 15 Performance of pre-processed features

Models	Accuracy	Precision	Recall	F1	Avg. Time
SVM	0.8413	0.8506	0.8305	0.8394	58s
XGBoost	0.86	0.8761	0.8383	0.8565	23s
DNN	0.9223	0.9405	0.9018	0.9205	59s

As with DNN, there is a slight increase in accuracy, but it is insignificant. However, considering the average time, the difference between the two is striking. The average time for SVM is reduced from 748s to 58s, XGBoost from 250s to 23s, and DNN from 718s to 59s. Thus, we can save computational time in experiments by filtering out data that is unnecessary for experiments. For instance, GA trains a learning algorithm 20 times for each iteration. In simple mathematics, using all features would take roughly 14,960s per iteration when using SVM. On the other hand, if the data is pre-processed, it takes around 1,160s. Therefore, from an overall point of view, this process is necessary for future experiments. As a result, initial data in future experiments are the pre-processed features.

6.2 Results of filter methods

6.2.1 Information Gain

Tab. 16 summarizes the number of features selected by IG per threshold. In terms of SVM (Tab. 17), the performance scores have a tendency to improve as the threshold increases. The top 1% features according to the IG score shows the worst performance, such as its accuracy is 0.733, which is around 10p.p. lower than that of using the entire feature set. The top 75% features according to the IG score show the best performance among other thresholds. It achieves a 0.8477 accuracy score, which is a bit higher than using the whole initial features. It also shows better performance in precision and F1-score, and in particular, it shows the greatest improvement in precision, around 1p.p.. However, in terms of recall, the score falls slightly, but it is about the same as the recall rate of all features. Threshold 'mean' achieves the second-best performance, but performance scores in terms of all evaluation metrics are lower than when using the threshold 75%. For XGBoost (Tab. 18), it shows better performance as the threshold increases, and when the threshold is 1%, an accuracy of 0.7447 is about 12p.p. lower than when using all features. The threshold

75% shows the best performance compared to other thresholds, which has around 11p.p. higher accuracy than that of the threshold 1%. The performance scores of threshold 75% are a little lower than using whole initial features, but it shows similar scores to some extent in terms of all evaluation metrics.

Table 16 Number of features selected by Information Gain

Categories	Thresh. 1%	Thresh. 5%	Thresh. 25%	Thresh. 50%	Thresh. 75%	Thresh. mean
Permission	1	6	27	64	87	37
Activity	0	1	15	115	186	21
Service	0	0	5	44	57	9
Broadcast Receiver	0	0	3	32	47	5
Intent filter	0	1	21	73	96	31
API method	101	503	2,484	4,782	7,193	2,910
Total	102	511	2,555	5,110	7,666	3,013

Table 17 Performance of SVM with Information Gain

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.733	0.8512	0.5648	0.6783	0.6s
5%	0.7617	0.8503	0.6358	0.7270	3s
25%	0.8413	0.8697	0.8041	0.8350	13s
50%	0.8406	0.8580	0.8173	0.8366	27s
75%	0.8477	0.8630	0.8285	0.8445	43s
mean	0.8437	0.8660	0.8139	0.8387	15s
100%	0.8413	0.8506	0.8305	0.8394	58s

When it comes to applying features to DNN (Tab. 19), likewise, the increase in the threshold and the improvement in performance show a direct proportional trend. The thresholds 1% and 5% show a score of less than 0.8 in accuracy, but with a threshold of 25% or more, the accuracy scores are higher than 0.9. According to Tab. 16, the service and receiver feature categories are excluded in the thresholds 1% and 5%, whereas all categories are used in the other thresholds. Therefore, it

Table 18 Performance of XGBoost with Information Gain

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.7447	0.8705	0.5748	0.6919	0.3s
5%	0.787	0.8590	0.6886	0.7635	1s
25%	0.855	0.8675	0.8377	0.8521	6s
50%	0.8507	0.8615	0.8358	0.8482	11s
75%	0.8567	0.8671	0.8422	0.8543	17s
mean	0.8563	0.8727	0.8345	0.8529	7s
100%	0.86	0.8761	0.8383	0.8565	23s

can be seen that using all categories leads to an improvement in performance. The threshold of 75% gives the best performance results. In all three learning algorithms, using the top 75% of the features shows similar or better performance than using the 100% feature. On top of that, as the number of features decreases, the average training time also decreases.

Table 19 Performance of DNN with Information Gain

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.7613	0.9009	0.5874	0.7106	6s
5%	0.7999	0.9041	0.674	0.7712	7s
25%	0.9017	0.9394	0.8598	0.8973	17s
50%	0.9193	0.9225	0.9177	0.9196	33s
75%	0.9253	0.9369	0.9158	0.9254	48s
mean	0.9080	0.9329	0.8799	0.9048	19s
100%	0.9223	0.9405	0.9018	0.9205	59s

6.2.2 Chi-square

Tab. 20 shows the number of features selected by CHI per threshold. Tab. 21 summarizes the results for each threshold based on CHI score trained by SVM. It shows a trend of gradually improving performance as the threshold value increases. Only

1% of the threshold has an accuracy less than 0.8, which is 0.7637, and this value is around 8p.p. less than the accuracy score when using the entire feature. The threshold 75% and the threshold 'mean' show the same accuracy score, 0.8483. However, in terms of precision, the two thresholds show similar performance, but the threshold 75% outperforms the mean threshold for the recall and F1-score. Therefore, the CHI shows the best performance when training with SVM at the threshold 75%. Also, it shows better performance than using all the features.

Table 20 Number of features selected by Chi-square

Categories	Thresh. 1%	Thresh. 5%	Thresh. 25%	Thresh. 50%	Thresh. 75%	Thresh. mean
Permission	14	19	35	63	86	50
Activity	6	8	22	115	187	61
Service	2	4	11	44	58	27
Broadcast Receiver	1	1	5	33	47	20
Intent filter	14	16	31	75	97	47
API method	65	463	2,451	4,780	7,191	3,615
Total	102	511	2,555	5,110	7,666	3,820

Table 21 Performance of SVM with Chi-square

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.7637	0.7605	0.7697	0.7647	0.5s
5%	0.8297	0.8602	0.7866	0.8215	2s
25%	0.8403	0.8667	0.8050	0.8341	13s
50%	0.8467	0.8646	0.8233	0.8430	27s
75%	0.8483	0.8632	0.83	0.8455	43s
mean	0.8483	0.8678	0.8226	0.8441	20s
100%	0.8413	0.8506	0.8305	0.8394	58s

Tab. 22 summarizes the results of each threshold trained by XGBoost. It shows good performance in the order of threshold 50%, 'mean', and 75%. The threshold 50% reaches an accuracy of 0.8627, which is slightly higher than the accuracy score

Table 22 Performance of XGBoost with Chi-square

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.8217	0.8397	0.7960	0.8168	0.4s
5%	0.8413	0.8621	0.8127	0.8363	1s
25%	0.852	0.8664	0.8327	0.8489	6s
50%	0.8627	0.8755	0.8455	0.8599	11s
75%	0.8533	0.868	0.8385	0.8527	17s
mean	0.859	0.876	0.8364	0.8555	8s
100%	0.86	0.8761	0.8383	0.8565	23s

of using the whole features. The performance scores have a tendency to improve as the threshold increases. So, the difference between the lowest (threshold 1%) and highest accuracy (threshold 50%) is approximately 8p.p..

Table 23 Performance of DNN with Chi-square

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.8957	0.9237	0.8637	0.8921	6s
5%	0.8923	0.9295	0.8511	0.8876	7s
25%	0.9100	0.9309	0.8869	0.9080	17s
50%	0.9217	0.9465	0.8947	0.9196	31s
75%	0.926	0.9311	0.9207	0.9257	47s
mean	0.9183	0.9472	0.8867	0.9156	24s
100%	0.9223	0.9405	0.9018	0.9205	59s

For training with DNN (Tab. 23), using the threshold 75% shows the best performance among thresholds. It achieves an accuracy of 0.926. This value is higher than an accuracy value of 0.9223 obtained by using all the features. In addition, the threshold 75% shows better performance in terms of recall and F1-score than using all the features. Likewise, for DNN, there is a difference in performance depending on the threshold. To sum up, the threshold 75% shows the best performance with SVM and DNN. In XGBoost a threshold of 75% shows the third-best performance

and shows slightly worse performance scores than using the entire feature. Therefore, the threshold 75% is considered the best since it shows the best performance in SVM and DNN and not a bad performance in XGBoost. Likewise, the threshold 75% reduces the dimensions while also reducing the average training time.

6.2.3 Ensemble feature selection

The number of features selected by the ensemble feature selection per threshold is summarized in Tab. 24. The results for each threshold based on ensemble feature selection trained by SVM are summarized in Tab. 25. According to all evaluation metrics, it shows better scores as the threshold value increases. The threshold 75% outperforms the other thresholds in terms of all metrics. It reaches an accuracy of 0.8487. Its accuracy, recall, and F1-score are higher than the scores of using all features. Moreover, the accuracy of the threshold of 75% is around 8p.p. higher than that of the threshold 1%, which shows the lowest accuracy of 0.7647. Tab. 26 shows the performance results of using XGBoost. In this case, it tends to show better performance as the threshold value increases. The threshold 'mean' shows the best performance, and the threshold 75% shows the second-performance. Considering the performance scores, there is no significant difference in performance between the thresholds 75% and 'mean'. In addition, when considering the reduction in the number of features, there is not much reduction in performance compared to the use of all features.

Table 24 Number of features selected by ensemble feature selection

Categories	Thresh. 1%	Thresh. 5%	Thresh. 25%	Thresh. 50%	Thresh. 75%	Thresh. mean
Permission	6	10	30	63	86	63
Activity	1	5	17	115	187	115
Service	1	2	7	44	57	44
Broadcast Receiver	0	1	3	32	47	32
Intent filter	1	5	23	73	96	73
API method	93	488	2,475	4,783	7,193	4,779
Total	102	511	2,555	5,110	7,666	5,106

Tab 27 summarizes the results of using DNN and ensemble feature selection. Likewise, the higher the threshold value, the better the performance. In terms of accu-

Table 25 Performance of SVM with ensemble feature selection

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.7647	0.8538	0.6391	0.7305	0.6s
5%	0.7953	0.8392	0.7302	0.7803	2s
25%	0.838	0.8668	0.7999	0.8314	13s
50%	0.842	0.8598	0.8186	0.8382	28s
75%	0.8487	0.8645	0.8284	0.8453	43s
mean	0.8423	0.8599	0.8193	0.8386	28s
100%	0.8413	0.8506	0.8305	0.8394	58s

racy score, the threshold 75% shows the best result compared to the other thresholds. In addition, this value is higher than the accuracy score (0.9223) of using all features. The threshold 75% has a better score than using all features in recall and F1-score, but not in precision. It is more important not to misclassify actual malware as benign, so a higher recall value is more important for Android malware detection. The threshold 75%, therefore, is better than using the whole features. To sum up, using the threshold 75% shows comprehensively good performance for all three learning algorithms compared to using other thresholds and all features. It likewise reduces the dimensionality of the data and shortens the average training time.

Table 26 Performance of XGBoost with ensemble feature selection

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.791	0.8532	0.7044	0.7712	0.4s
5%	0.8257	0.8503	0.79	0.8187	1s
25%	0.8543	0.8716	0.8313	0.8506	6s
50%	0.8513	0.8642	0.8336	0.8483	11s
75%	0.857	0.8686	0.8409	0.8545	17s
mean	0.858	0.8718	0.8395	0.8551	11s
100%	0.86	0.8761	0.8383	0.8565	23s

Table 27 Performance of DNN with ensemble feature selection

Threshold	Accuracy	Precision	Recall	F1	Avg. Time
1%	0.8213	0.8852	0.7407	0.8055	7s
5%	0.8513	0.9056	0.7866	0.8407	7s
25%	0.9057	0.9263	0.8830	0.9037	18s
50%	0.9157	0.932	0.8975	0.9139	33s
75%	0.9257	0.9273	0.9245	0.9254	43s
mean	0.917	0.9275	0.9103	0.9177	31s
100%	0.9223	0.9405	0.9018	0.9205	59s

6.3 Discussion of filter methods

As a result of the above, performance tends to decrease when the threshold value is low in all three methods. Nevertheless, at low thresholds (i.e., 1% and 5%), CHI is less degraded than IG. In Tab. 17, 18, and 19, feature selection based on IG shows an accuracy of less than 0.8, when the threshold is 1% and 5%. In contrast, for CHI as shown in Tab. 21, 22, and 23, all learning algorithms show higher accuracy than 0.8 when the threshold is 1% and 5%, except when the threshold is 1% with SVM. To find out the reason for the difference in performance, we can look at the number of features per category. In IG, according to Tab. 16, the number of features in some categories is 0 when the threshold is 1% or 5%. In contrast, in CHI, there is no category with zero number of features at all thresholds as shown in Tab. 20. In addition, at the thresholds of 1% and 5%, a greater number of features in the permission category are selected by CHI compared to IG, and the number of features selected by CHI is better distributed throughout the entire category. The ensemble feature selection is affected by CHI and IG, so that performance at a low threshold is between the two.

The threshold 75% can be the best for all filter methods. Tab. 28 is a collection of performance scores for each method when the threshold is 75%. In SVM, all three methods show similar performance. However, the ensemble method shows the highest accuracy of 0.8487 with a very small difference from the other two methods. Moreover, CHI outperforms IG in terms of all evaluation metrics. When it comes to XGBoost, IG has higher scores of accuracy, recall, and F1-score than CHI. The ensemble feature selection overall outperforms the other two methods. In DNN, the

Table 28 Comparison of different filter methods (Threshold = 75%)

Models	Filter method	Accuracy	Precision	Recall	F1
SVM	IG	0.8477	0.8630	0.8285	0.8445
	CHI	0.8483	0.8632	0.83	0.8455
	Ensemble	0.8487	0.8645	0.8284	0.8453
XGBoost	IG	0.8567	0.8671	0.8422	0.8543
	CHI	0.8533	0.868	0.8385	0.8527
	Ensemble	0.857	0.8686	0.8409	0.8545
DNN	IG	0.9253	0.9369	0.9158	0.9254
	CHI	0.926	0.9311	0.9207	0.9257
	Ensemble	0.9257	0.9273	0.9245	0.9254

CHI gives the highest accuracy score, but there is no significant difference from the other selection methods.

In conclusion, the three methods show similar performance in the three algorithms without significant difference. However, in SVM and XGBoost, the ensemble feature selection shows slightly better performance than other methods. Also, in DNN, the ensemble feature selection shows the second-highest accuracy score and the highest recall of 0.9245. In addition, the ensemble feature selection shows better performance than using the entire features and shows no significant difference from the other two methods. Therefore, the ensemble feature selection is regarded as the most suitable filter method and applied to the hybrid method. Finally, the ensemble feature selection with the threshold 75% selects the 7,666 informative features as illustrated in Tab. 24 among the 10,221 features.

6.4 Results of wrapper methods

6.4.1 Genetic Algorithm

Each learning algorithm is applied to the GA to obtain the best subset of features. The best subset is evaluated using the learning algorithm applied to the GA. Tab. 29 summarizes the experimental results using a GA with a parameter set (survivor selection=" $(\mu+\lambda)$ ", recombination="1-point"). In case that the subset is chosen and trained using SVM, the subset selected by GA shows better results than using the initial feature set in all performance metrics. For instance, the accuracy score

increased by more than 1p.p. compared to using all features, reaching 0.8533. The number of features decreased from 10,221 to 5,123 by about 5,000, and the average training time is also reduced. In terms of XGBoost, the number of features is reduced by around 60%, and training time is shortened. The features selected by GA using XGBoost show slightly less performance overall than when using all features, such as the accuracy is reduced by about 0.5p.p. and the precision is reduced by around 1p.p.. In the case of using DNN, selected features show better performance scores than all features in terms of all evaluation metrics. Besides, the smallest number of features is obtained by applying DNN to GA. The performance scores are increased to a greater extent than using SVM. For example, the accuracy score increases by about 2p.p. from 0.9223 to 0.942, and also 2p.p. for other metrics. Therefore, in this case, the dimension reduction and performance improvement are successfully achieved in SVM and DNN. In XGBoost, there is a slight performance deterioration.

Table 29 Performance of GA(survivor=" $(\mu+\lambda)$ ", recombination="1-point")

Models	No. features	Accuracy	Precision	Recall	F1	Avg. Time
SVM	5,123	0.8533	0.8636	0.8402	0.8512	28s
XGBoost	3,822	0.854	0.8661	0.8377	0.8513	9s
DNN	884	0.942	0.9523	0.9311	0.9412	10s

The experimental results using a GA with a parameter set (survivor selection=" $(\mu+\lambda)$ ", recombination="uniform") is shown in Tab. 30. When it comes to using SVM, the subset selected by GA outperforms the initial feature set in terms of all performance metrics. For example, it achieves a 0.848 accuracy score and a 0.8615 precision score, which is around respectively 0.7p.p. and 1p.p. higher than those of the entire features. Moreover, the number of features decreased by about 5,000, and the average training time is also reduced. In the case that the subset is selected by GA with XGBoost, it shows a better recall score than that of using all features, whereas it shows slightly lower performance for the other three evaluation metrics. However, the difference is insignificant. The dimensions of about 3,000 are reduced. In the case of using DNN, selected features show better performance scores than all features in terms of all evaluation metrics. For instance, the accuracy score increases by about 1p.p. from 0.9223 to 0.9323. Also, the recall increases by around 2p.p. to 0.9234. As a result, in SVM and DNN, the amount of features is reduced, and performance is improved at the same time. Furthermore, in XGBoost, the amount of features is reduced, and the performance remains almost the same.

Table 30 Performance of GA(survivor=" $(\mu+\lambda)$ ", recombination="uniform")

Models	No. features	Accuracy	Precision	Recall	F1	Avg. Time
SVM	5,202	0.848	0.8615	0.8307	0.8451	29s
XGBoost	7,270	0.8583	0.8706	0.8418	0.8556	16s
DNN	5,073	0.9323	0.9416	0.9234	0.9319	32s

Tab. 31 illustrates the experimental results using a GA with a parameter set (survivor selection=" (μ,λ) ", recombination="1-point"). In the case where SVM is used for the fitness function of GA, and the selected features are trained by SVM, the number of features is drastically reduced to 550. Nevertheless, the overall performance scores are reduced compared to using the initial features. For instance, the accuracy score dropped from 0.8413 to 0.8377, and the recall score even decreased to 0.7986, which is less than 0.8. When it comes to exploiting XGBoost, the performance scores decreased in all evaluation metrics. In addition, compared to the case of using all features, the decrease in performance is a little larger than in SVM. For example, there is a drop of about 2.5p.p. in accuracy score, and the rest of the indicators show a similar drop. The number of selected features is 434. In terms of using DNN, the final selected features show better performance than using all features in the overall evaluation metrics, such as the accuracy reaches 0.9470, and it is about 2.5p.p. higher than the accuracy of the overall features. Also, it shows the largest increase in recall score over other indicators. The recall score increases by 4p.p. from 0.9018 to 0.9443. The number of features is declined to 382, which is relatively less than the number of features selected by using SVM and XGBoost. To sum up, in this case, the dimensions of the data are significantly reduced in the three learning algorithms. However, in cases where XGBoost and SVM are used, the performance is degraded. Only when DNN is used, the performance increases remarkably.

Tab. 32 summarizes the experimental results using a GA with a parameter set (survivor selection=" (μ,λ) ", recombination="uniform"). In terms of using SVM, it shows better performance than using the entire feature set in F1-score, accuracy, and precision. According to precision, 0.8610 is achieved, and this score is approximately 1p.p. higher than the precision score of all features. However, using the selected features yields a slightly lower score than using the entire feature set in terms of recall. The number of features is reduced to 3,803, and the training time is cut in half as well. In terms of using XGBoost, the number of features is declined to 5,358,

Table 31 Performance of GA(survivor=" μ, λ ", recombination="1-point")

Models	No. features	Accuracy	Precision	Recall	F1	Avg. Time
SVM	550	0.8377	0.8657	0.7986	0.8304	2s
XGBoost	434	0.8337	0.8595	0.7985	0.8271	1s
DNN	382	0.9470	0.9502	0.9443	0.9471	7s

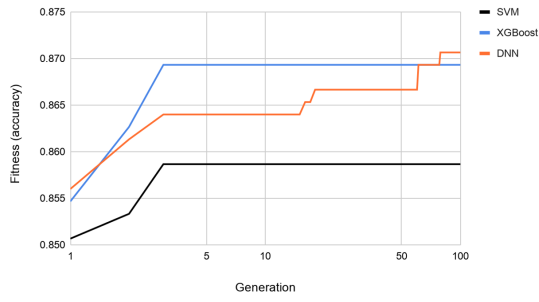
and training time is shortened. The features selected by GA using XGBoost show slightly less performance overall than when using all features, such as the recall score is reduced by about 0.7p.p., and the accuracy score is reduced by around 0.4p.p.. When it comes to using DNN, selected features show better performance scores than all features in terms of all evaluation metrics. For example, the accuracy and the F1-score increase by about 2p.p. from 0.9223 to 0.9403 and from 0.9205 to 0.9404, respectively, and the recall increases by 3p.p.. As a result, in SVM and XGBoost, there is no major performance deterioration. Nevertheless, in DNN, performance improvement is achieved while reducing the number of dimensions.

Table 32 Performance of GA(survivor=" (μ, λ) ", recombination="uniform")

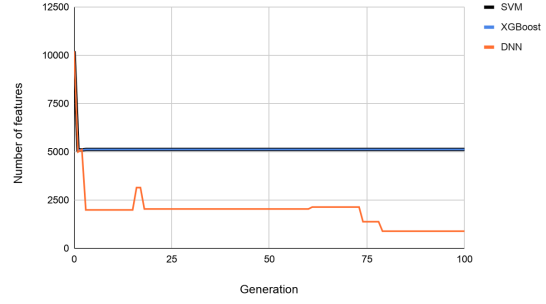
Models	No. features	Accuracy	Precision	Recall	F1	Avg. Time
SVM	3,803	0.847	0.8610	0.8287	0.8440	20s
XGBoost	5,358	0.8563	0.8753	0.8311	0.8523	12s
DNN	4,321	0.9403	0.9497	0.9316	0.9404	29s

In this experiment, we consider not only the performance results for each parameter set discussed above but also the evolutionary trend of each parameter set to figure out the most suitable parameter values of the survival selection for GA. The most optimal individual (i.e., high fitness score and a low number of features) of each generation is recorded to understand the evolutionary trend.

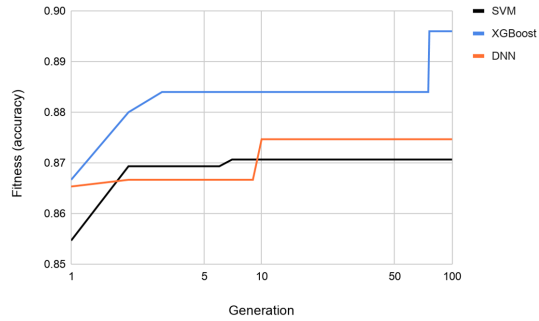
For the survival selection, we can compare Fig. 24 and Fig. 25 to Fig. 26 and 27. Fig. 24 and Fig. 25 depict the evolutionary trend over 100 generations of GA using $(\mu + \lambda)$ as survival selection. And, Fig. 26 and Fig. 27 depicts the evolutionary trend over 100 generations of GA using (μ, λ) as survival selection. In the case of using (μ, λ) , it is difficult to judge the trend of change in the number of features and change in fitness value as generation passes. Graphs of the number of features and fitness



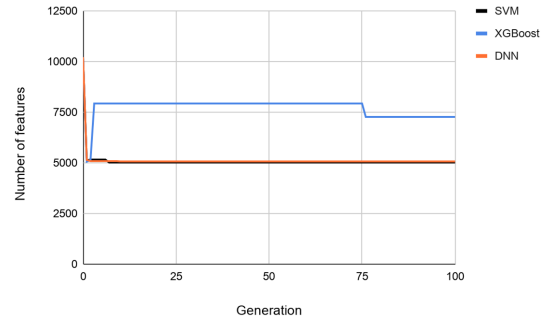
(a) Fitness (accuracy)



(b) Number of features

Figure 24 Evolving Graph of GA(survivor=" $(\mu+\lambda)$ ", recombination="1-point")

(a) Fitness (accuracy)



(b) Number of features

Figure 25 Evolving Graph of GA(survivor=" $(\mu+\lambda)$ ", recombination="uniform")

values show considerably fluctuating trends. It is because (μ, λ) as survival selection can lead to losing the best individuals of the current population. In other words, it presents that there is no guarantee that the selected subset using (μ, λ) is close to a true optimal subset. In contrast, the case of using $(\mu+\lambda)$ shows a more stable evolutionary process than the case of (μ, λ) . In this case, the fitness graphs show a growing trend, and the amount of features graphs show a decreasing trend. But, in certain sections, the fitness value and the amount of features are almost unchanged, and tend to remain almost constant. This is because an individual that surpasses the most optimal individual of the previous generation fails to be created in the offspring generation, and the individual is continuously transmitted over several generations. It appears that the evolutionary trend is determined by survivor selection. So, rather than (μ, λ) in which the evolutionary trend cannot be grasped, $(\mu+\lambda)$, which shows a stable and weakly evolving trend through generations, is adopted as a survival selection.

For the recombination, we can consider the number of features. First, Tab. 29 and Tab. 30 are the results of adopting the same survival selection, $(\mu+\lambda)$. However,

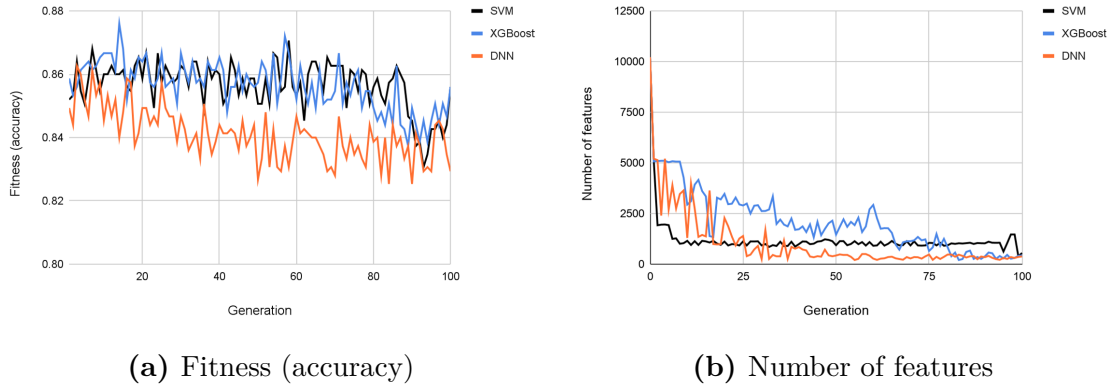


Figure 26 Evolving Graph of GA(survivor=" (μ, λ) ", recombination="1-point")

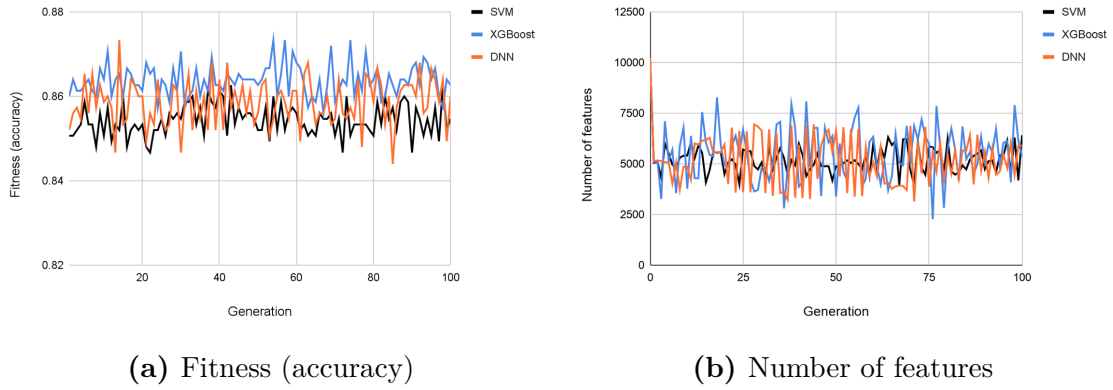


Figure 27 Evolving Graph of GA(survivor=" (μ, λ) ", recombination="uniform")

in the case of selecting '1-point', fewer features are finally selected in all learning algorithms than in the case of selecting uniform. For instance, in DNN, when $(\mu + \lambda)$ and 'uniform' are used, 5,073 features are finally selected, and when $(\mu + \lambda)$ and '1-point' are applied, a much smaller number of 884 features are selected. Also, the results summarized in Tab. 29 show better scores than the results summarized in Tab. 30. Comparing Tab. 31 and Tab. 32, it appears that the case of applying the points reduces the dimensions more effectively than the case of applying the uniform. Also, '1-point' can be more suitable for this project than 'uniform' considering the evolutionary trend. In Fig. 25b, the number of features does not fall below 5,000 in any case, whereas in Fig. 24b, when DNN is used, the number of features falls gradually to 884. When (μ, λ) is used as a survival strategy, the number of features fluctuates around 5,000 when 'uniform' is applied (Fig. 27b), whereas the number of features ends up fluctuating below 2,000 when points are applied (Fig. 26b). In conclusion, we choose '1-point' for recombination, which is capable of reducing the number of features more effectively.

As a result, $(\mu+\lambda)$ for survivor selection and '1-point' for recombination are chosen as the best parameter set for GA. This is considered a set of parameters that is only able to achieve both stable and continuous evolution and dimensionality reduction. In this experiment, only this parameter set achieved an accuracy of over 0.85 in terms of using SVM (Tab. 29). Also, in XGBoost and DNN, the result of this parameter set outperforms or shows similar results to other parameter sets.

6.4.2 Binary Particle Swarm Optimization

Each learning algorithm is used for the BPSO to search for the best subset of features. The best subset obtained by applying a learning algorithm to BPSO is trained and evaluated on the same learning algorithm.

Table 33 Performance of BPSO

Models	No. features	Accuracy	Precision	Recall	F1	Avg. Time
SVM	6,128	0.839	0.8499	0.8250	0.8367	33s
XGBoost	5,452	0.8503	0.8686	0.8258	0.8463	12s
DNN	5,580	0.924	0.9504	0.8946	0.9205	37s

Tab. 33 summarizes the experimental results using BPSO. In case that SVM is applied to the fitness function of BPSO, the number of features decreased by about 40% from 10,221 to 6,128. However, the performance of the final selected subset is slightly less than that of the entire features. For example, the accuracy of the selected subset is 0.839, which is slightly lower than the accuracy of the entire feature set, 0.84. In addition, the recall decreased by about 0.5p.p. from 0.8305 to 0.8250, which shows the biggest performance drop compared to other evaluation metrics. Considering the advantage of reducing the features, BPSO with SVM provides an advantage to this data. When it comes to utilizing XGBoost, despite the decrease in the number of features to 5,452, the overall performance scores of the selected features are lower than those of all features. For example, the accuracy of the selected features is 0.8503, but it is about 1p.p. lower than the accuracy of the entire features, 0.86. In addition, other evaluation indicators show similar differences in performance decline. In terms of applying DNN to BPSO, the experimental results are not significantly different from the experimental results using all features. The selected features achieved an accuracy of 0.924, which is slightly higher than that of using the entire feature set. Also, its precision increased by about 1p.p. to

achieve 0.9504. However, in terms of recall, 0.8946 is achieved, about 0.5p.p. drop from 0.9096. While the scores of the evaluation indicators remain similar, BPSO to which DNN is applied provides the advantage of reducing the dimension. As a result, in SVM, there is no considerable performance deterioration. In XGBoost, there is a slightly larger performance drop than in case of SVM. In DNN, the performance is kept similar. In all three ways, a dimension reduction is achieved.

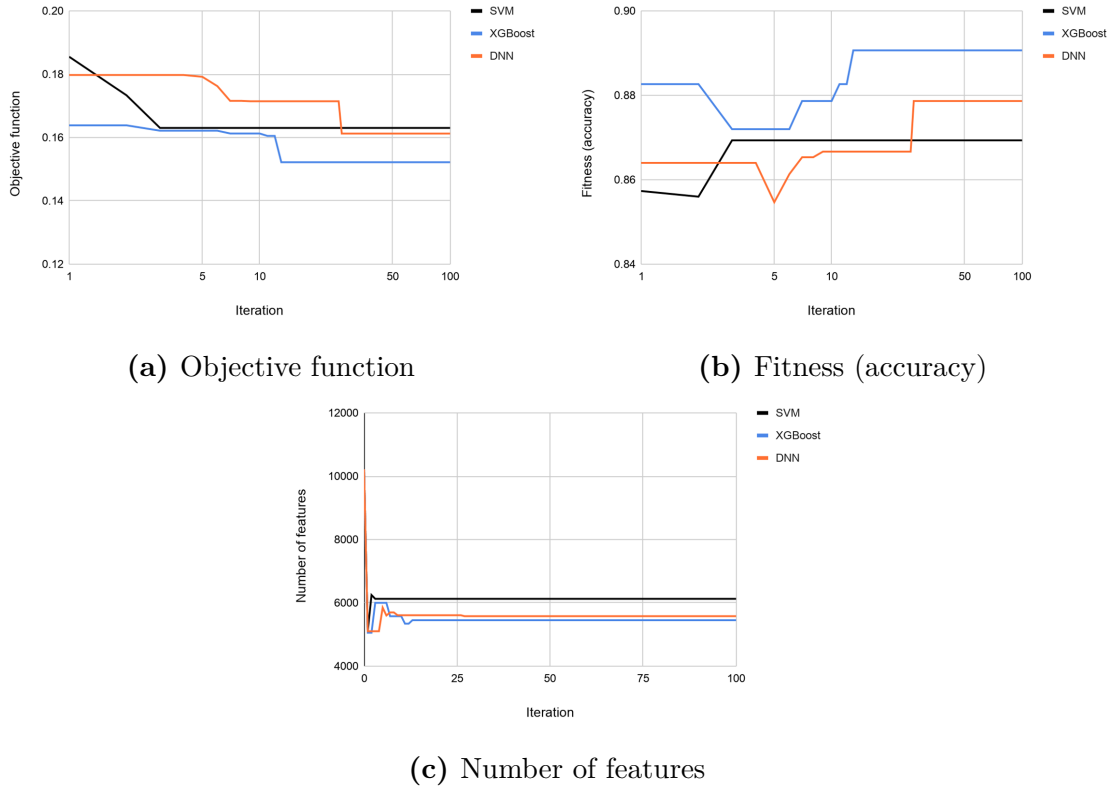


Figure 28 Iterations of BPSO

Fig. 28 contains graphs showing the history of the global best subset by iteration. The particle with the lowest objective function result is the global best, and the history over 100 iterations is depicted in Fig. 28a. Just as particles go to an optimal subset, the outcome of the objective function shows a trend of decreasing with iteration. At the same time, the fitness score increase has a tendency to incline as shown in Fig. 28b, and the amount of features in the subset tends to decrease as depicted in Fig. 28c.

6.5 Discussion of wrapper methods

As we discussed above, the parameter set, survivor selection=" $(\mu+\lambda)$ " and recombination="1-point", is the most suitable for GA. Thus, in this section, we compare and discuss GA with the best parameters and BPSO.

Both methods show that performance improves with iteration as shown in Fig. 24 and Fig. 28. Both methods have advantages over simply using the entire feature set. Considering the reduction in the number of features, GA works more effectively than BPSO. According to Fig. 28 and Tab. 33, in all three learning algorithms, the number of features does not fall below 5,000 during iteration, and the number of features in the selected subsets are all over 5,000 and even the case of SVM results in the largest number of 6,128 features. In contrast, only in the case of applying SVM, 5,123 features in which the number of features exceeds 5,000 are selected according to Tab. 29, but this number is less than the number of features in any subset of features selected by BPSO. In addition, in DNN, the number of features is greatly reduced up to 884.

Comparing the performance of the subsets selected by BPSO and GA, respectively, GA outperforms BPSO as shown in Tab. 29 and Tab. 33. In SVM, GA shows about 1.5p.p. higher performance than BPSO in all the indicators. In XGBoost as well, GA leads BPSO in all performance metrics, and shows the biggest difference, especially at about 1p.p. in recall. In DNN, these two show the biggest difference in performance. In terms of accuracy and F1-score, the scores of GA are around 2p.p. higher than those of BPSO. In addition, there is a difference of nearly 4p.p. in recall.

Table 34 Number of features selected by GA

Categories	SVM	XGBoost	DNN
Permission	55	38	8
Activity	127	97	24
Service	35	33	8
Broadcast Receiver	29	22	6
Intent filter	51	41	7
API method	4,826	3,591	831
Total	5,123	3,822	884

In all three learning algorithms, GA reduces a greater number of features than BPSO, and at the same time shows better performance. In addition, GA has the potential to reduce a larger number of features to a very small number. Therefore, GA, which finally achieves the two goals of effectively reducing features and increasing performance at the same time, is selected as one of the best wrapper methods

and grafted into the hybrid method. Tab. 34 summarizes the number of features in the best subset selected by GA using each learning algorithm. The number of features decreases from 10,221 to as much as 5,123 and as little as 884. Moreover, in all three cases, features are selected from all categories without excluding any categories.

6.6 Results of hybrid method

In the previous experiment, the ensemble feature selection in the filter method and the GA in the wrapper method are chosen for a hybrid method because they can be considered as the most effective methods to reduce features and improve performance. Thus, in the hybrid method, features are first selected by the top 75% (i.e., the total of 7,666) based on the ensemble feature selection, and the pre-selected features are passed to the GA with a parameter set (survivor selection=" $(\mu+\lambda)$ ", recombination="1-point").

Table 35 Performance of hybrid method

Models	No. features	Accuracy	Precision	Recall	F1	Avg. Time
SVM	3,861	0.854	0.876	0.8266	0.8498	21s
XGBoost	3,773	0.8557	0.8688	0.838	0.8527	9s
DNN	3,902	0.943	0.9501	0.9358	0.9423	25s

As in the previous experiment, each learning algorithm is applied to the GA algorithm of the hybrid method, and the resulting subset is trained and evaluated by the applied algorithm. Tab. 35 summarizes the experimental results using the hybrid method. In case that SVM is utilized, the selected features outperform using all features in all performance indicators except recall. In the recall, there is a slight performance drop of less than 1p.p.. However, the selected features achieve an accuracy of 0.854 and a precision of 0.876, which is higher than when using the entire features by above 1p.p. and 2p.p., respectively. In addition, the number of features decreases significantly from 10,221 to 3,861, and thus the average training time is greatly reduced to 21s. When it comes to XGBoost, the features selected by the hybrid method show slightly less performance overall than when using all features, such as it achieves an accuracy of 0.8557, which is just about 0.5p.p. lower, and the precision score drops by approximately 0.8p.p.. However, the number of features successfully decreases by about 60% to 3,773. In case that the hybrid method em-

employs DNN, the selected features make significant improvements in all performance metrics. For example, it achieves an accuracy of 0.943 and an F1-score of 0.9423, an increase of about 2p.p. over the entire features. Moreover, in terms of recall, it achieves the largest performance improvement of over 3p.p.. Also, the number of features and the average training time are significantly reduced. In conclusion, in the case of using the hybrid method with SVM or XGBoost, it is possible to successfully reduce the number of features while keeping the performance similar without a significant drop. With DNN, the number of features is reduced, and the performance is improved well at the same time.

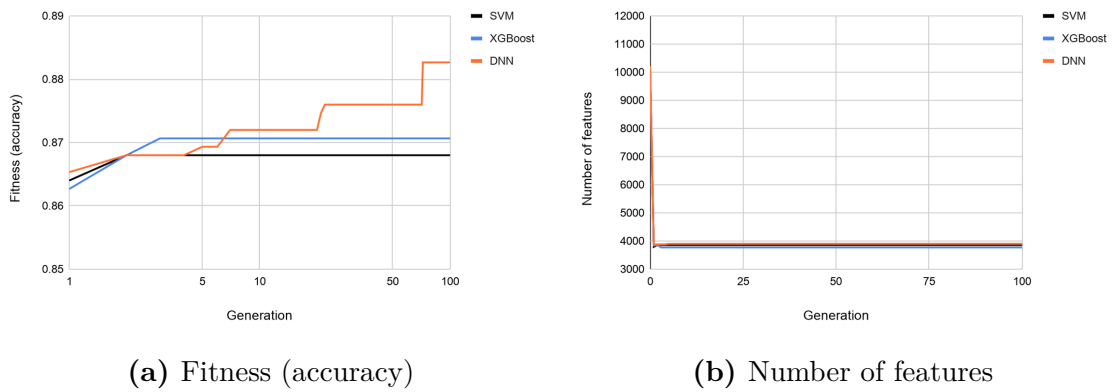


Figure 29 Evolving Graph of GA in hybrid method

Fig. 29 contains graphs showing the evolutionary trend over 100 generations according to its fitness value (i.e., accuracy) and the number of features. So, Fig. 29 illustrates the evolutionary tendency that the fitness value increases and the number of features decreases. And, this trend appears to be stronger when DNN is applied.

6.7 Discussion of all methods

In the previous sections, we looked into and discussed the experimental results of each feature selection method. It has been proven that all of the methods work well for the goals of reducing features and improving performance.

To choose the most suitable feature selection method for the framework of Android malware detection, we first compare the filter method and the wrapper method. As discussed, we chose the threshold 75% of the ensemble feature selection and GA with the best parameter set as the best feature selection algorithm for each method. So, by comparing the two algorithms, we can determine which method is better. According to Tab. 28 and Tab. 29, the GA outperforms the ensemble feature selection in SVM. The precision of the ensemble feature selection is slightly higher

Table 36 Elapsed time of the first generation

Models	Wrapper method	Hybrid method
SVM	306s	230s
XGBoost	210s	159s
DNN	719s	533s

than that of GA, but GA is higher with a larger difference in terms of accuracy, recall, F1-score. In particular, it shows a difference of more than 1p.p. in the recall. In terms of XGBoost, the ensemble feature selection shows slightly higher scores in all the metrics, but the difference is very insignificant, so we assume that the performance scores of the two algorithms are similar in XGBoost. In DNN, the features selected by the GA shows about 1.5p.p. higher scores than the features selected by the ensemble feature selection in all evaluation metrics. In terms of the number of features, 7,666 features were selected by the ensemble feature selection, and maximum 5,123 to at least 884 features were selected by GA. The selection of features by GA depends to some extent on a random contingency so that the number of features finally selected may vary. In this experiment, we can assume that even if the number of features selected by GA is large, it will be less than that of the ensemble feature selection, and GA has the potential to effectively reduce the number of features to a very small number. In other words, the GA is a way to improve performance while reducing the number of features more compared to the ensemble feature selection.

Next, we can compare the wrapper method and the hybrid method. The difference between the two methods in this experiment is the features passed to the GA. The hybrid method's GA uses pre-selected features, so it takes less time in the early generations than the GA concerning all features as shown in Tab. 36. According to Tab. 29 and Tab. 35, the two methods show almost the same performance in the three learning algorithms. However, in accuracy, the hybrid method is slightly higher than wrapper method in all learning algorithms. Considering the number of features, the number of features selected by the wrapper method ranges from 5,123 to 884, and from 3,902 to 3,861 for the hybrid method. It seems nominally that wrapper methods reduce features more, but the hybrid method adopts the same GA, so the hybrid method also have the potential to reduce the number of features to as small as 884. Since the GA is dependent on contingency to some extent, in this paper, we choose a method to stably reduce the features with a higher probability. The maximum number of features selected features by the wrapper method is 5,123,

and 3,902 for the hybrid method. Considering the use of the framework in general, we presume that we assume that since the hybrid method initially consider fewer features, fewer features will be selected than the wrapper method. Thus, in the application of the hybrid method, 5,000 features like the wrapper method will rarely be selected. To sum up, the wrapper method and the hybrid method are similar, but the hybrid method have several more advantages by considering only pre-selected features such as decreased execution time and higher probability of selecting a lower number of features.

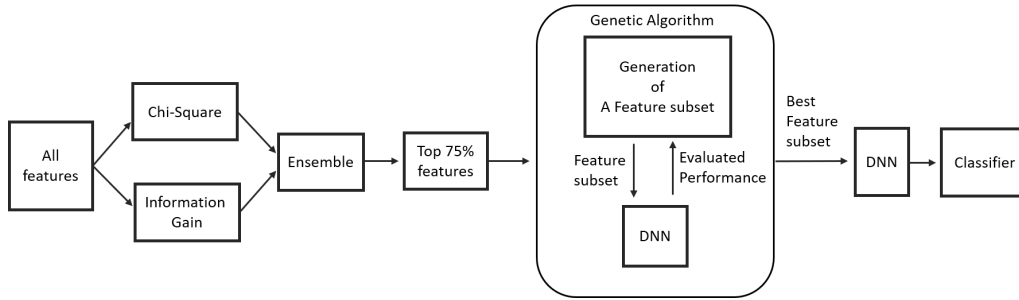


Figure 30 The process of the finally selected feature selection method

Table 37 Number of features selected by Hybrid method

Categories	Number
Permission	37
Activity	87
Service	27
Broadcast Receiver	23
Intent filter	56
API method	3,672
Total	3,902

Finally, the hybrid method is chosen for the framework. In the hybrid method, DNN, a deep learning algorithm, surpasses the classical machine learning algorithms. Also, DNN always provides the best performance in other feature selection algorithms. According to Fig. 24 and Fig. 29, while SVM and XGBoost show a tendency to stagnate in evolution, DNN continues to evolve continuously for 100 generations. Therefore, DNN is finally chosen to be applied to the GA of the hybrid method and generate the classification model. So, the process from selecting the most appropriate subset of features to creating the model for the framework is depicted in Fig. 30. Tab. 37 summarizes the number of features finally selected by the hybrid method with DNN.

7 Deployment and Continuous Learning

In this chapter, we discuss the framework for continuous improvement of the Android malware framework. Fig. 31 depicts the processes in which the classification model for Android malware detection is distributed through the API, and features of unknown applications for future usage are stored in the database.

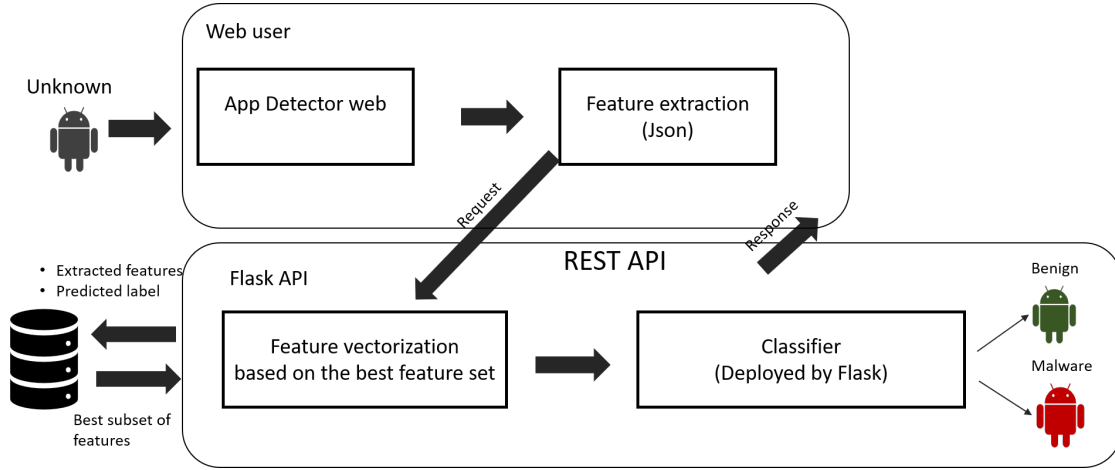


Figure 31 Flowchart of REST API

Through experimentation of this thesis, we finally obtained 3,902 features through the hybrid method and generate a classification model by training data with these features out of all 3,000 applications using DNN. The classification model is placed in the *classifier* directory (Fig. 21), and it is deployed and called by the REST API. We utilize *Flask* framework to implement the REST API procedure.

First, a user can upload an unknown application file to the web application as depicted in left side of Fig. 32. In this thesis, we create a sample web application using *Django* framework. The user can proceed with the prediction of the application uploaded by the user. In the backend of the web application, the application file is decomposed and the features of the application are extracted using *Androguard* library. Subsequently, the web application sends an API request by putting the extracted feature, the package name, and SHA-256 of the application in the body of the POST method in the form of JSON as shown in Fig. 33.

After the server dealing with the REST API receives the request, it transforms the features of the unknown application into the formation of a binary vector matrix. Subsequently, the binary vector matrix is forwarded to the classification model as input data and the unknown application is finally predicted to be either malware or benign. Afterward, the extracted features, the name of the application, SHA-256,

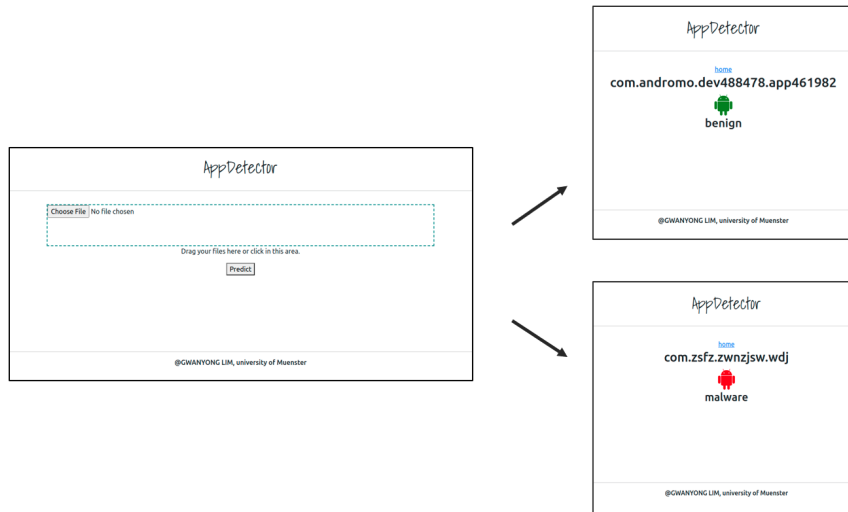


Figure 32 Example of web application

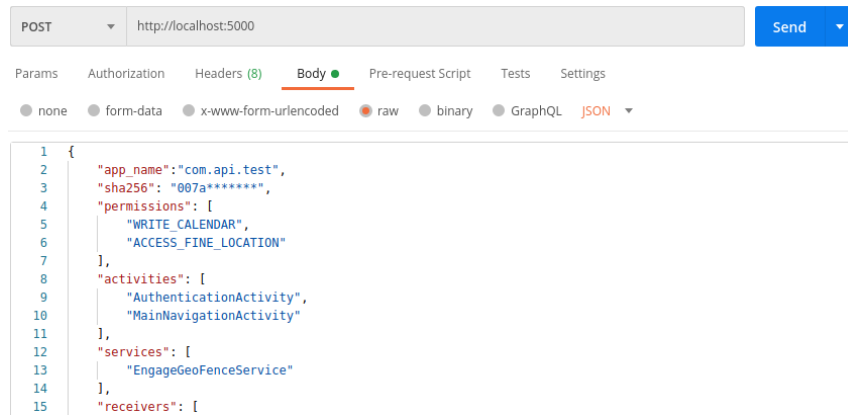


Figure 33 Example of API request

and the predicted label are stored in the database. Finally, the prediction result is delivered to the web application as a response to the API request, and the user is informed whether the uploaded application is malicious or benign.

With the process mentioned above, we can continuously accumulate new data whenever users try to predict unknown applications. It is possible to retrain and update the classification model by aggregating the new data with the existing data. Therefore, we can repeat the series of processes, data vectorization, data pre-processing, feature selection (i.e., *select_best_features.py*), and model generation (i.e., *generate_detection_model.py*), which are part of the framework of Android malware detection.

8 Conclusion

Android operating system is the most used smartphone platform, and accordingly, the number of threats to Android is increasing day by day. Therefore, Android malware detection is a significantly important research task in the fast-growing market of mobile devices.

This thesis explored the recent related research works according to the three Android malware detection approaches, namely static analysis, dynamic analysis, and hybrid analysis. It appeared that most of the previous research conducted the static analysis. Because it provides computational advantages, we also exploited the static analysis in this paper. We reviewed feature selection algorithms and machine learning and deep learning algorithms used in many studies.

In this thesis, we proposed the framework for Android malware detection. First, 1,500 benign applications and 1,500 were collected using *AndroZoo*'s API document. Next, the APK files were decomposed into *AndroidManifest.xml* file and *DEX* file using a reverse engineering technique and five categories of features, which are namely permission, activity, service, broadcast receiver, and intent filter, were extracted from *AndroidManifest.xml* file, and API methods are extracted from *DEX* file. We utilized three machine learning and deep learning algorithms, which are namely SVM, XGBoost, and DNN. We got rid of junk features through data pre-processing. Through data pre-processing, we were able to decrease the number of features from 119,427 to 10,221, and the performance has not changed much and has improved slightly in the three learning algorithms. This process helped the efficient progress of the experiment in the future.

In the experiment, we compared various feature selection algorithms. We took CHI, IG, and an ensemble of CHI and IG as filter methods into consideration. Also, GA and BPSO were considered as wrapper methods. In the filter methods, the threshold 75% of all filter methods comprehensively showed the best performance compared to the other thresholds and better performance than the initial feature set. Besides, average training time was reduced for all learning algorithms. In the comparison of filter methods, we selected the ensemble feature selection from the three filter methods. As comparing the performance of each method's threshold hold 75%, the ensemble feature selection outperformed the other two algorithms in SVM and XGBoost. In DNN, CHI achieved the highest accuracy of 0.926. However, the ensemble feature selection achieved the highest recall of 0.9245 and the second-best accuracy of 0.9257, which are almost the same as the CHI's score. Thus, the threshold of 75% of the ensemble feature selection was able to decrease the number

of features to 7,666. In addition, due to the influence of CHI, the performance drop at the low threshold of ensemble feature selection is compensated. When it comes to wrapper methods, we first defined the best parameter set for GA. It appeared that " $(\mu+\lambda)$ " for survivor selection and "1-point" for recombination are the most suitable for GA since it could guarantee that the steady and stable improvement of performance and effective feature reduction by generation. It resulted in improved performance and reduced features. Especially, GA using DNN could reduce the number of features to 884 and achieved the accuracy of 0.942 and precision of 0.9523. Another wrapper method, BPSO, kept performance similar or reduced it slightly in all learning algorithms. Considering the reduction of features and performance improvement, it appeared that the GA is a more suitable algorithm for this project than BPSO. We combined the ensemble feature selection and the GA to create a hybrid model. This hybrid method likewise improved performance while reducing the number of features. For example, in DNN, the number of features decreased to 3,903, and it reached an accuracy of 0.943.

To determine the most suitable feature selection method, we first compared the ensemble feature selection and the GA. It appeared that the use of GA guarantees a more significant number of feature reductions and greater performance improvements than the ensemble feature selection. In the contingency characteristic of the GA, it is believed that thanks to the pre-selected features, the hybrid method is more likely to reduce more features than using GA solely. At the same time, the hybrid method is able to achieve the improvement of performance slightly better. DNN performs best not only in the hybrid method but also in all other algorithms. In addition, only when DNN was applied to the GA, evolution did not stagnate and showed sustained growth. It appeared that DNN, a deep learning algorithm, surpasses the other two machine learning algorithms. Therefore, we selected the hybrid method for the feature selection process and DNN for the model generation process. A total of 3,902 features were selected using the hybrid method with DNN.

In this thesis, we also proposed the framework for deploying the classification model and continuous training. A classification model is generated by training the best subset of features with DNN. Then, the model is deployed through the REST API. Upon receiving an API request containing the extracted features of an unknown app, the model predicts whether the application is malicious or benign and sends the prediction result as a response. Simultaneously, the features of the unknown app and its label are stored in the database. Therefore, an updated classification model can be obtained if the Android malware detection framework proposed in this paper is re-executed by using the data combined with the new data and the existing data.

Our future works will consider the following aspects. First, we only applied 3,000 applications to our framework. Given good computing power, we can execute the experiment and prove the framework on a huge size of data. Second, we can utilize hybrid analysis that uses not only static analysis but also dynamic analysis. Therefore, we may enhance the classifier model by considering broader perspectives of applications. In addition to finding a subset of the optimal features, the performance of the classification model can be considerably improved by optimizing the parameters of the machine learning or deep learning algorithm. In this paper, arbitrary fixed values of parameters were applied to machine learning and deep learning algorithms. However, in future work, we can implement some parameter optimizers. For example, we can also use the idea of bio-inspired algorithms explained in this thesis, GA and PSO, as a parameter optimizer. Next, we have presented a framework for continuous model training. However, the predicted label is not 100% accurate. Therefore, it is possible to apply a method to correct the incorrect prediction by comparing the predicted label of the application stored in the database with the results of the label derived from *VirusTotal* reports. Thus, we can prevent including error data in the continuous training framework. Besides, the framework for continuous model training has the potential to be exposed to an attack that users continuously input maliciously contaminated data (i.e., Poisoning attack). This attack can cause a poorly trained classification model. Therefore, we need a method to protect our data against this attack, such as data cleansing.

Appendix

A Hardware specifications

The hardware specifications of the used computer system are as follows:

- OS: Ubuntu 20.04.2 LTS
- CPU: Intel(R) Xeon(R) Gold 6242R CPU @ 3.10GHz (Core: 2)
- Memory: 32 GB

References

- [Ala+16] Huda Ali Alatwi et al. “Android malware detection using category-based machine learning classifiers”. In: *SIGITE 2016 - Proceedings of the 17th Annual Conference on Information Technology Education*. Association for Computing Machinery, Inc, Sept. 2016, pp. 54–59. DOI: 10.1145/2978192.2978218.
- [All+16] Kevin Allix et al. “AndroZoo: Collecting Millions of Android Apps for the Research Community”. In: *Proceedings of the 13th International Conference on Mining Software Repositories*. MSR ’16. Austin, Texas: ACM, 2016, pp. 468–471. DOI: 10.1145/2901739.2903508. URL: <http://doi.acm.org/10.1145/2901739.2903508>.
- [AYS20] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. “DL-Droid: Deep learning based android malware detection using real devices”. In: *Computers and Security* 89 (2020). DOI: 10.1016/j.cose.2019.101663.
- [AYS17] Mohammed K. Alzaylaee, Suleiman Y. Yerima, and Sakir Sezer. “EMULATOR vs REAL PHONE: Android Malware Detection Using Machine Learning”. In: *arXiv* (2017), pp. 65–72.
- [Ars+16] Saba Arshad et al. “Android Malware Detection Protection: A Survey”. In: *International Journal of Advanced Computer Science and Applications* 7.2 (2016). DOI: 10.14569/ijacsa.2016.070262.
- [AV-] AV-ATLAS. *AV-ATLAS - Malware PUA*. <https://portal.av-atlas.org/malware/statistics>. (Visited on 03/02/2021).
- [BM18] Shikha Badhani and Sunil K. Muttou. “Comparative analysis of pre- and post-classification ensemble methods for android malware detection”. In: *Communications in Computer and Information Science*. Vol. 906. Springer Verlag, Apr. 2018, pp. 442–453. DOI: 10.1007/978-981-13-1813-9_44. URL: https://doi.org/10.1007/978-981-13-1813-9%7B%5C_%7D44.
- [BMD06] Helmut Berger, Dieter Merkl, and Michael Dittenbach. “Exploiting partial decision trees for feature subset selection in e-mail categorization”. In: *Proceedings of the ACM Symposium on Applied Computing*. Vol. 2. Association for Computing Machinery, 2006, pp. 1105–1109. DOI: 10.1145/1141277.1141536.
- [BG18] Abhishek Bhattacharya and Radha Tamal Goswami. “Community Based Feature Selection Method for Detection of Android Malware”. In: *Journal of Global Information Management* 26.3 (2018), pp. 1–10. DOI: 10.4018/JGIM.2018070105.
- [BG17] Abhishek Bhattacharya and Radha Tamal Goswami. “Comparative analysis of different feature ranking techniques in data mining-based android malware detection”. In: *Advances in Intelligent Systems and Computing*. Vol. 515. Springer Verlag, 2017, pp. 39–49. DOI: 10.1007/978-981-10-3153-3_5.

- [BGM19] Abhishek Bhattacharya, Radha Tamal Goswami, and Kuntal Mukherjee. “A feature selection technique based on rough set and improvised PSO algorithm (PSORS-FS) for permission based detection of Android malwares”. In: *International Journal of Machine Learning and Cybernetics* 10.7 (2019), pp. 1893–1907. DOI: 10.1007/s13042-018-0838-1. URL: <http://dx.doi.org/10.1007/s13042-018-0838-1>.
- [Blo+08] Vincent D. Blondel et al. “Fast unfolding of communities in large networks”. In: *Journal of Statistical Mechanics: Theory and Experiment* 2008.10 (Oct. 2008), P10008. DOI: 10.1088/1742-5468/2008/10/P10008. URL: <https://iopscience.iop.org/article/10.1088/1742-5468/2008/10/P10008%20https://iopscience.iop.org/article/10.1088/1742-5468/2008/10/P10008/meta>.
- [BA19] Verónica Bolón-Canedo and Amparo Alonso-Betanzos. “Ensembles for feature selection: A review and future trends”. In: *Information Fusion* 52 (2019), pp. 1–12. DOI: <https://doi.org/10.1016/j.inffus.2018.11.008>. URL: <https://www.sciencedirect.com/science/article/pii/S1566253518303440>.
- [Cen+15] Lei Cen et al. “A Probabilistic Discriminative Model for Android Malware Detection with Decompiled Source Code”. In: *IEEE Transactions on Dependable and Secure Computing* 12.4 (July 2015), pp. 400–412. DOI: 10.1109/TDSC.2014.2355839.
- [CL11] Chih-Chung Chang and Chih-Jen Lin. “LIBSVM: A Library for Support Vector Machines”. In: *ACM Trans. Intell. Syst. Technol.* 2.3 (May 2011). DOI: 10.1145/1961189.1961199. URL: <https://doi.org/10.1145/1961189.1961199>.
- [CW17] Yu Chen Chang and Sheng De Wang. “The concept of attack scenarios and its applications in android malware detection”. In: *Proceedings - 18th IEEE International Conference on High Performance Computing and Communications, 14th IEEE International Conference on Smart City and 2nd IEEE International Conference on Data Science and Systems, HPCC/SmartCity/DSS 2016*. Institute of Electrical and Electronics Engineers Inc., Jan. 2017, pp. 1485–1492. DOI: 10.1109/HPCC-SmartCity-DSS.2016.0211.
- [CG] Tianqi Chen and Carlos Guestrin. *XGBoost Documentation — xgboost 1.4.0-SNAPSHOT documentation*. <https://xgboost.readthedocs.io/en/latest/index.html>. (Visited on 02/09/2021).
- [CG16] Tianqi Chen and Carlos Guestrin. “XGBoost: A Scalable Tree Boosting System”. In: *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*. KDD ’16. San Francisco, California, USA: Association for Computing Machinery, 2016, pp. 785–794. DOI: 10.1145/2939672.2939785. URL: <https://doi.org/10.1145/2939672.2939785>.
- [Che+19] Tieming Chen et al. “Droidvecdeep: Android malware detection based on word2vec and deep belief network”. In: *KSII Transactions on Internet and Information Systems* 13.4 (2019), pp. 2180–2197. DOI: 10.3837/tiis.2019.04.025.

- [Cho+15] François Chollet et al. *Keras*. <https://keras.io>. 2015.
- [Des] Anthony Desnos. *Introduction — Androguard 3.4.0 documentation*. <https://androguard.readthedocs.io/en/latest/intro/index.html>. (Visited on 02/07/2021).
- [Deva] Android Developers. *Android Developers*. <https://developer.android.com/docs>. (Visited on 02/08/2021).
- [Devb] Android Developers. *Application Fundamentals | Android Developers*. <https://developer.android.com/guide/components/fundamentals>. (Visited on 02/17/2021).
- [Devc] Android Developers. *Platform Architecture | Android Developers*. <https://developer.android.com/guide/platform>. (Visited on 02/16/2021).
- [Devd] Android Developers. *UI/Application Exerciser Monkey | Android Developers*. <https://developer.android.com/studio/test/monkey>. (Visited on 03/23/2021).
- [Deve] Google Developers. *Malware categories | Play Protect | Google Developers*. <https://developers.google.com/android/play-protect/phacategories>. (Visited on 02/17/2021).
- [Dro] Droidbox. <https://code.google.com/archive/p/droidbox/>. (Visited on 11/23/2020).
- [ES15] A. E. Eiben and James E. Smith. *Introduction to Evolutionary Computing*. 2nd. Springer Publishing Company, Incorporated, 2015.
- [Eom+18] Taehoon Eom et al. “Android malware detection using feature selections and random forest”. In: *Proceedings - 2018 4th International Conference on Software Security and Assurance, ICSSA 2018* (2018), pp. 55–61. DOI: 10.1109/ICSSA45270.2018.00023.
- [F-S] F-Secure. *2017 F-Secure State of cyber security*. Tech. rep. URL: <https://blog.f-secure.com/the-state-of-cyber-security-2017/>.
- [Fat+19] Anam Fatima et al. “Android malware detection using genetic algorithm based optimized feature selection and machine learning”. In: *2019 42nd International Conference on Telecommunications and Signal Processing, TSP 2019* (2019), pp. 220–223. DOI: 10.1109/TSP.2019.8769039.
- [Fei+15] Ali Feizollah et al. “A review on feature selection in mobile malware detection”. In: *Digital Investigation* 13 (2015), pp. 22–37. DOI: <https://doi.org/10.1016/j.diin.2015.02.001>. URL: <https://www.sciencedirect.com/science/article/pii/S1742287615000195>.
- [Fen+18] Pengbin Feng et al. “A novel dynamic android malware detection system with ensemble learning”. In: *IEEE Access* 6 (2018), pp. 30996–31011. DOI: 10.1109/ACCESS.2018.2844349.
- [Fer+16] Hossein Fereidooni et al. “ANASTASIA: ANdroid mAlware detection using STatic analySIs of applications”. In: *2016 8th IFIP International Conference on New Technologies, Mobility and Security, NTMS 2016*. Institute of Electrical and Electronics Engineers Inc., Dec. 2016. DOI: 10.1109/NTMS.2016.7792435.

- [Fir+18] Ahmad Firdaus et al. “Discovering optimal features using static analysis and a genetic search based method for Android malware detection”. In: *Frontiers of Information Technology and Electronic Engineering* 19.6 (June 2018), pp. 712–736. DOI: 10.1631/FITEE.1601491.
- [Fri02] Jerome H Friedman. “Stochastic gradient boosting”. In: *Computational Statistics Data Analysis* 38.4 (2002), pp. 367–378. DOI: [https://doi.org/10.1016/S0167-9473\(01\)00065-2](https://doi.org/10.1016/S0167-9473(01)00065-2). URL: <https://www.sciencedirect.com/science/article/pii/S0167947301000652>.
- [Fri01] Jerome H. Friedman. “Greedy Function Approximation: A Gradient Boosting Machine”. In: *The Annals of Statistics* 29.5 (2001), pp. 1189–1232. URL: <http://www.jstor.org/stable/2699986>.
- [Gar+09] P. García-Teodoro et al. “Anomaly-based network intrusion detection: Techniques, systems and challenges”. In: *Computers and Security* 28.1-2 (Feb. 2009), pp. 18–28. DOI: 10.1016/j.cose.2008.08.003.
- [GBC16] Ian Goodfellow, Yoshua Bengio, and Aaron Courville. *Deep Learning*. <http://www.deeplearningbook.org>. MIT Press, 2016.
- [Han+20] Hyoil Han et al. “Enhanced android malware detection: An SVM-based machine learning approach”. In: *Proceedings - 2020 IEEE International Conference on Big Data and Smart Computing, BigComp 2020* December 2016 (2020), pp. 75–81. DOI: 10.1109/BigComp48618.2020.00-96.
- [HTF09] Trevor Hastie, Robert Tibshirani, and Jerome Friedman. *The elements of statistical learning: data mining, inference and prediction*. 2nd ed. <http://www-stat.stanford.edu/~tibs/ElemStatLearn/>. Springer, 2009.
- [IDC] IDC. *IDC - Smartphone Market Share - OS*. <https://www.idc.com/promo/smartphone-market-share/os>. (Visited on 02/28/2021).
- [Kab19] Abdullah Talha Kabakus. “What static analysis can utmost offer for android malware detection”. In: *Information Technology and Control* 48.2 (2019), pp. 235–249. DOI: 10.5755/j01.itc.48.2.21457.
- [KE95] J. Kennedy and R. Eberhart. “Particle swarm optimization”. In: *Proceedings of ICNN’95 - International Conference on Neural Networks*. Vol. 4. 1995, 1942–1948 vol.4. DOI: 10.1109/ICNN.1995.488968.
- [KE97] J. Kennedy and R. C. Eberhart. “A discrete binary version of the particle swarm algorithm”. In: *1997 IEEE International Conference on Systems, Man, and Cybernetics. Computational Cybernetics and Simulation*. Vol. 5. 1997, 4104–4108 vol.5. DOI: 10.1109/ICSMC.1997.637339.
- [Le+20] Ngoc C. Le et al. “A Machine Learning Approach for Real Time Android Malware Detection”. In: *Proceedings - 2020 RIVF International Conference on Computing and Communication Technologies, RIVF 2020* (2020). DOI: 10.1109/RIVF48685.2020.9140771.
- [LWX18] Dongfang Li, Zhaoguo Wang, and Yibo Xue. “DeepDetector: Android Malware Detection using Deep Neural Network”. In: *Proceedings on 2018 International Conference on Advances in Computing and Communication Engineering, ICACCE 2018* June (2018), pp. 184–188. DOI: 10.1109/ICACCE.2018.8441737.

- [Li+17a] Dongfang Li et al. “FgDetector: Fine-Grained Android Malware Detection”. In: *Proceedings - 2017 IEEE 2nd International Conference on Data Science in Cyberspace, DSC 2017* (2017), pp. 311–318. DOI: 10.1109/DSC.2017.13.
- [Li+18] Jian Li et al. “An android malware detection system based on feature fusion”. In: *Chinese Journal of Electronics* 27.6 (2018), pp. 1206–1213. DOI: 10.1049/cje.2018.09.008.
- [Li+17b] Yuanchun Li et al. “DroidBot: A lightweight UI-guided test input generator for android”. In: *Proceedings - 2017 IEEE/ACM 39th International Conference on Software Engineering Companion, ICSE-C 2017*. Institute of Electrical and Electronics Engineers Inc., June 2017, pp. 23–26. DOI: 10.1109/ICSE-C.2017.8.
- [Liu+20] Kaijun Liu et al. “A Review of Android Malware Detection Approaches Based on Machine Learning”. In: *IEEE Access* 8 (2020), pp. 124579–124607. DOI: 10.1109/ACCESS.2020.3006143.
- [MGM19] Vinisha Malik, Sandip Kumar Goyal, and Naveen Malik. “A hybrid model for android malware detection”. In: *International Journal of Innovative Technology and Exploring Engineering* 8.12 (2019), pp. 2656–2662. DOI: 10.35940/ijitee.K2250.1081219.
- [Mar+15] Martin Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: <https://www.tensorflow.org/>.
- [Mer14] Dirk Merkel. “Docker: lightweight linux containers for consistent development and deployment”. In: *Linux journal* 2014.239 (2014), p. 2.
- [Naq11] Syed Naqvi. “A Hybrid Filter-Wrapper Approach for FeatureSelection”. In: 2011.
- [Nie15] Michael A. Nielsen. *Neural Networks and Deep Learning*. 2015. URL: <http://neuralnetworksanddeeplearning.com>.
- [Niv+18] M. Nivaashini et al. “Comparative analysis of feature selection methods and machine learning algorithms in permission based android malware detection”. In: *Proceedings of IEEE International Conference on Intelligent Computing and Communication for Smart World, I2C2SW 2018* (2018), pp. 72–77. DOI: 10.1109/I2C2SW45816.2018.8997527.
- [Nok20] Nokia. *Nokia: Threat Intelligence Report 2020*. <https://onestore.nokia.com/asset/210088>. 2020. DOI: 10.1016/s1361-3723(20)30115-9. (Visited on 02/28/2021).
- [Odu+18] Modupe Odusami et al. “Android Malware Detection: A Survey”. In: *Applied Informatics*. Ed. by Hector Florez, Cesar Diaz, and Jaime Chavarriaga. Cham: Springer International Publishing, 2018, pp. 255–266.
- [Ped+11] F. Pedregosa et al. “Scikit-learn: Machine Learning in Python”. In: *Journal of Machine Learning Research* 12 (2011), pp. 2825–2830.

- [PEK20] Gökçer Peynirci, Mete Eminağaoğlu, and Korhan Karabulut. “Feature Selection for Malware Detection on the Android Platform Based on Differences of IDF Values”. In: *Journal of Computer Science and Technology* 35 (2020), pp. 946–962.
- [Pou18] Shahin Pourbahrami. “Improving PSO Global Method for Feature Selection According to Iterations Global Search and Chaotic Theory”. In: *arXiv* (2018), pp. 1–17.
- [Proa] Android Open Source Project. *Android Open Source Project*. <https://source.android.com/>. (Visited on 03/01/2021).
- [Prob] Android Open Source Project. *Secure an Android Device | Android Open Source Project*. <https://source.android.com/security>. (Visited on 02/23/2021).
- [QSL16] Mengyu Qiao, Andrew H. Sung, and Qingzhong Liu. “Merging permission and api features for android malware detection”. In: *Proceedings - 2016 5th IIAI International Congress on Advanced Applied Informatics, IIAI-AAI 2016* (2016), pp. 566–571. DOI: 10.1109/IIAI-AAI.2016.237.
- [Ras+20] Ahmar Rashid et al. “Conceptualization of smartphone usage and feature preferences among various demographics”. In: *Cluster Computing* 23.3 (Sept. 2020), pp. 1855–1873. DOI: 10.1007/s10586-020-03061-x. URL: <https://doi.org/10.1007/s10586-020-03061-x>.
- [Raz+18] Mohd Faizal Ab Razak et al. “Bio-inspired for Features Optimization and Malware Detection”. In: *Arabian Journal for Science and Engineering* 43.12 (2018), pp. 6963–6979. DOI: 10.1007/s13369-017-2951-y.
- [Rem] Ayodeji Remi-Omosowon. *pyeasyga — pyeasyga 0.3.1 documentation*. URL: <https://pyeasyga.readthedocs.io/en/latest/> (visited on 03/26/2021).
- [RB08] Martin Rosvall and Carl T. Bergstrom. “Maps of random walks on complex networks reveal community structure”. In: *Proceedings of the National Academy of Sciences of the United States of America* 105.4 (Jan. 2008), pp. 1118–1123. DOI: 10.1073/pnas.0706851105. URL: www.pnas.org/cgi/content/full/.
- [Roy+20] Arindaam Roy et al. “Android Malware Detection based on Vulnerable Feature Aggregation”. In: *Procedia Computer Science* 173.2019 (2020), pp. 345–353. DOI: 10.1016/j.procs.2020.06.040. URL: <https://doi.org/10.1016/j.procs.2020.06.040>.
- [SSK20] Ahmad Salah, Eman Shalabi, and Walid Khedr. “A lightweight android malware classifier using novel feature selection methods”. In: *Symmetry* 12.5 (2020), pp. 1–16. DOI: 10.3390/SYM12050858.
- [SE98] Y. Shi and R. Eberhart. “A modified particle swarm optimizer”. In: *1998 IEEE International Conference on Evolutionary Computation Proceedings. IEEE World Congress on Computational Intelligence (Cat. No.98TH8360)*. 1998, pp. 69–73. DOI: 10.1109/ICEC.1998.699146.

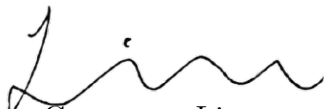
- [SH18] Latika Singh and Markus Hofmann. “Dynamic behavior analysis of android applications for malware detection”. In: *ICCT 2017 - International Conference on Intelligent Communication and Computational Techniques* 2018-January.2013 (2018), pp. 1–7. DOI: 10.1109/INTELCCT.2017.8324010.
- [Smy20] Neil Smyth. *Android Studio 4.1 Development Essentials Java Edition*. https://www.techotopia.com/pdf_previews/AndroidStudio41EssentialsPreview.pdf. 2020.
- [Sta] Statista. • *Smartphone users 2020 | Statista*. <https://www.statista.com/statistics/330695/number-of-smartphone-users-worldwide/>. (Visited on 03/23/2021).
- [Sto] Maddie Stone. *Android App Reverse Engineering 101 | Learn to reverse engineer Android applications!* <https://ragingrock.com/AndroidAppRE/index.html>. (Visited on 02/07/2021).
- [TAL14] Jiliang Tang, Salem Alelyani, and Huan Liu. “Feature Selection for Classification: A Review”. In: *Data Classification: Algorithms and Applications*. 2014.
- [Tha+20] Rajan Thangaveloo et al. “DATDroid: Dynamic analysis technique in android malware detection”. In: *International Journal on Advanced Science, Engineering and Information Technology* 10.2 (2020), pp. 536–541. DOI: 10.18517/ijaseit.10.2.10238.
- [VMia] Lester James V.Miranda. *Feature Subset Selection — PySwarms 1.0.2 documentation*. https://pyswarms.readthedocs.io/en/development/examples/feature_subset_selection.html. (Visited on 03/23/2021).
- [VMib] Lester James V.Miranda. *Welcome to PySwarms’s documentation! — PySwarms 1.3.0 documentation*. <https://pyswarms.readthedocs.io/en/latest/index.html>. (Visited on 01/25/2021).
- [VW07] Nees Jan Van Eck and Ludo Waltman. “VOS: A new method for visualizing similarities between objects”. In: *Studies in Classification, Data Analysis, and Knowledge Organization*. Kluwer Academic Publishers, 2007, pp. 299–306. DOI: 10.1007/978-3-540-70981-7_34. URL: https://link.springer.com/chapter/10.1007/978-3-540-70981-7_34.
- [Vie+13] Susana M. Vieira et al. “Modified binary PSO for feature selection using SVM applied to mortality prediction of septic patients”. In: *Applied Soft Computing Journal* 13.8 (Aug. 2013), pp. 3494–3504. DOI: 10.1016/j.asoc.2013.03.021.
- [Vir] VirusTotal. *VirusTotal*. URL: <https://www.virustotal.com/gui/> (visited on 03/25/2021).
- [WLZ18] Jiong Wang, Boquan Li, and Yuwei Zeng. “XGBoost-Based Android Malware Detection”. In: *Proceedings - 13th International Conference on Computational Intelligence and Security, CIS 2017* 2018-Janua (2018), pp. 268–272. DOI: 10.1109/CIS.2017.00065.
- [Wan+17] Xin Wang et al. “Mlifdetect: Android malware detection based on parallel machine learning and information fusion”. In: *Security and Communication Networks* 2017 (2017). DOI: 10.1155/2017/6451260.

- [WY17] Long Wen and Haiyang Yu. “An Android malware detection system based on machine learning”. In: *AIP Conference Proceedings* 1864. August 2017 (2017). DOI: 10.1063/1.4992953.
- [YJ97] Yang Yiming and Pedersen Jan O. “A Comparative Study on Feature Selection in Text Categorization”. In: *Proceeding ICML '97 Proceedings of the Fourteenth International Conference on Machine Learning* 53.9 (1997), pp. 412–420.
- [Zha+18] Peng Zhang et al. “A Novel Android Malware Detection Approach Using Operand Sequences”. In: *2018 3rd International Conference on Security of Smart Cities, Industrial Control System and Communications, SSIC 2018 - Proceedings* Ssi C (2018). DOI: 10.1109/SSIC.2018.8556755.
- [Zha+16] Kai Zhao et al. “Fest: A feature extraction and selection tool for Android malware detection”. In: *Proceedings - IEEE Symposium on Computers and Communications*. Vol. 2016-February. Institute of Electrical and Electronics Engineers Inc., Feb. 2016, pp. 714–720. DOI: 10.1109/ISCC.2015.7405598.

Declaration of Authorship

I hereby declare that, to the best of my knowledge and belief, this Master Thesis titled “Feature Evaluation in Malicious Android App Detection” is my own work. I confirm that each significant contribution to and quotation in this thesis that originates from the work or works of others is indicated by proper use of citation and references.

Münster, 31st March 2021



Gwanyong Lim

Consent Form

for the use of plagiarism detection software to check my thesis

Last name: Lim

First name: Gwanyong

Student number: 465480 **Course of study:** Information Systems

Address: Heekweg 12, 48161 Münster

Title of the thesis: “Feature Evaluation in Malicious Android App Detection”

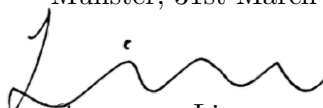
What is plagiarism? Plagiarism is defined as submitting someone else’s work or ideas as your own without a complete indication of the source. It is hereby irrelevant whether the work of others is copied word by word without acknowledgment of the source, text structures (e.g. line of argumentation or outline) are borrowed or texts are translated from a foreign language.

Use of plagiarism detection software The examination office uses plagiarism software to check each submitted bachelor and master thesis for plagiarism. For that purpose the thesis is electronically forwarded to a software service provider where the software checks for potential matches between the submitted work and work from other sources. For future comparisons with other theses, your thesis will be permanently stored in a database. Only the School of Business and Economics of the University of Münster is allowed to access your stored thesis. The student agrees that his or her thesis may be stored and reproduced only for the purpose of plagiarism assessment. The first examiner of the thesis will be advised on the outcome of the plagiarism assessment.

Sanctions Each case of plagiarism constitutes an attempt to deceive in terms of the examination regulations and will lead to the thesis being graded as “failed”. This will be communicated to the examination office where your case will be documented. In the event of a serious case of deception the examinee can be generally excluded from any further examination. This can lead to the exmatriculation of the student. Even after completion of the examination procedure and graduation from university, plagiarism can result in a withdrawal of the awarded academic degree.

I confirm that I have read and understood the information in this document. I agree to the outlined procedure for plagiarism assessment and potential sanctioning.

Münster, 31st March 2021



Gwanyong Lim