Contents lists available at GrowingScience

International Journal of Data and Network Science

homepage: www.GrowingScience.com/ijds

Android malicious attacks detection models using machine learning techniques based on permissions

Mousa AL-Akhras[a], Abdulrhman ALMohawes[b], Hani Omar[c], Samer Atawneh[b] and Samah Alhazmi[b*]

[a]*King Abdullah II School for Information Technology, The University of Jordan, Amman 11942, Jordan*
[b]*College of Computing and Informatics, Saudi Electronic University, Riyadh 11673, Saudi Arabia*
[c]*Faculty of Information Technology, Applied Science Private University, Amman, Jordan. MEU Research Unit, Middle East University, Amman, Jordan*

C H R O N I C L E

Article history:
Received: April 5, 2023
Received in revised format: July 26, 2023
Accepted: August 21, 2023
Available online: August 21, 2023

Keywords:
Android, permission
Attack
Machine Learning
Noise

A B S T R A C T

The Android operating system is the most used mobile operating system in the world, and it is one of the most popular operating systems for different kinds of devices from smartwatches, IoT, and TVs to mobiles and cockpits in cars. Security is the main challenge to any operating system. Android malware attacks and vulnerabilities are known as emerging risks for mobile devices. The development of Android malware has been observed to be at an accelerated speed. Most Android security breaches permitted by permission misuse are amongst the most critical and prevalent issues threatening Android OS security. This research performs several studies on malware and non-malware applications to provide a recently updated dataset. The goal of proposed models is to find a combination of noise-cleaning algorithms, features selection techniques, and classification algorithms that are noise-tolerant and can achieve high accuracy results in detecting new Android malware. The results from the empirical experiments show that the proposed models are able to detect Android malware with an accuracy that reaches 87%, despite the noise in the dataset. We also find that the best classification results are achieved using the RF algorithm. This work can be extended in many ways by applying higher noise ratios and running more classifiers and optimizers.

1. Introduction

Android is a mobile operating system that relies on the altered Linux kernel and other software open source. Android has been developed by Open Handset Alliance (OHA) to develop open standards in mobile devices. OHA is a group of top telecom companies. Including HTC, Sony, Motorola, Google, Samsung, and LG (OHA, 2007). Smartphones have become a significant element of our everyday lives. Statistics show that the Android mobile operating system (OS) is the leading OS used in the globe, with a market share of 88 percent (O'Dea, 2020). According to Android director Stephanie Cuthbertson, there are over two and a half billion active Android devices worldwide. More than 180 hardware manufacturers manufactured these devices, and not all of them are smartphones (Kerns, 2019).

Android malware is increasingly becoming a constant risk to users. The number of Android malware is rising exponentially; they become dramatically sophisticated and cause possible financial and data losses for users. Hence, there is a need for optimized and practical techniques to detect Android malware applications. Due to the large number of users, Android has become the critical target for mobile malware. Since 2011, Android has become the most widespread and most attacked smartphone platform. The attackers' essential rewards are stealing the user's data or money and getting hold of the system and SMS trojans. Other features found in mobile malware include intrusive ads, the use of paid programs, and ransomware (Skovoroda et al., 2015). The use of unified smartphone apps stores made it easier for attackers to distribute mobile malware.

* Corresponding author.
E-mail address: s.alhazmi@seu.edu.sa (S. Alhazmi)

ISSN 2561-8156 (Online) - ISSN 2561-8148 (Print)

doi: 10.5267/j.ijdns.2023.8.019

Any marketplaces, for example, Apple Application Store and Windows Phone Store, search the applications submitted manually, while the Google Play Store, automatically scans the latest applications and developers' accounts using Google Bouncer. A proper analysis dramatically decreases the amount of malware released, although it is still existing (Skovoroda et al., 2015). According to a study by McAfee laboratories, in the first quarter of 2020, 1,5 million new malwares were detected, and most of those malwares are targeting Android devices (McAfee, 2020).

Malware or malicious software is any software with a misbehaving intention. It can be written to interrupt normal functioning, circumvent access controls, gather confidential information, show unauthorized ads, or get control of the system without the user's awareness (Doğru & KİRAZ, 2018). Moreover, malware and accidentally harmful applications were collectively termed badware. The acceleration of Android has been growing fast. From the first-ever virus, malware for mobile devices has grown tremendously (Qamaret al., 2019). According to a study on mobile malware in the duration from the year 2000 to 2018, mobile malware was a virus, worms, trojans, ransomware, rootkits, botnet, backdoors, and keyloggers.

Android malware detection has been an active research field recently. The mobile malware identification and approaches to detect malicious intent and other security threats can be classified into three primary groups: (Qamar et al., 2019):

1. Structural or static analysis: an application inspection technique that tests the software code without being executed.
2. Behavioral or dynamic analysis: Dynamic analysis is the process by which a program is analyzed in runtime.
3. Hybrid analysis (static and dynamic): The hybrid analysis's underlying theme is to incorporate the characteristics of static and dynamic analysis related to an application's code and behavior.

1.1. Android permissions

The Android operating system uses a permission-based security mechanism to identify the assets and resources that can be used or accessed and which transactions can be performed by an application. In order to use permission in the application, the developer must add the permission in the AndroidManifest.xml file. Before Android version 6.0, the user had to accept all the permissions at the installation time since the Android OS was using all or nothing (Krajci et al., 2013). This policy has changed from version 7.0, where there were install time permissions and run time permissions. Developers identify the requested permissions in the manifest file under the <permission> tag in a unique format as an example: android.permission.INTERNET. Notwithstanding a high number of permissions within Android, various researchers have investigated and revealed that the permission-based security mechanism cannot grant enough security against malicious software (Arslan et al., 2019; Doğru & Önder, 2020).

Android malware attacks and vulnerabilities are known as emerging risks to mobile devices. It is a rising security issue expected to continue as users take crucial activities on their smartphones (Xiao et al., 2020). Most Android users have no idea how perpetrators and intruders deeply infiltrate their mobile devices without their privileges and knowledge. The worst case is that users feel protected from threats like these and have inadequate awareness of the rapid growth in mobile malware and attacks. In many attacks, user vulnerabilities remain critical as they become critical vulnerabilities to mobile attackers' entry. Most users download and install third-party applications (games, image editors, chatting apps). Because of this, it is also possible to install ransomware and adware applications without their knowledge. That is because malware applications are usually appearing in legitimate-looking applications. The security breaches permitted by permission misuse are amongst the most critical and prevalent issues threatening Android OS security.

Malware identification in the Android platform has been made in a variety of ways. The signature-based approach is the most common. Malicious sample signatures are saved in a database, which is later used to detect malware. Only known malware can be detected using this method. In order to identify zero-day malware, machine learning algorithms have been implemented. Because of the increase in malware activity in the Android environment, researchers have been focusing their attention on detecting malicious Android apps.

We noticed that most of the studies were conducted on old Android malware and benign dataset. Most of the datasets were collected for applications before 2015. In the last five years, Android application development changed many times due to the Android framework updates and new permissions added in these frameworks. In addition, we did not find any work that studies the noise insertion effect in our search. The feature selection and classification algorithm is noise-tolerant and can perform well with noisy datasets. In this work, we will perform an ML study on an Android applications dataset collected recently. In addition to studying the effect of noise in the dataset and which ML feature selection and classification algorithms will perform well despite the noise (Xin et al., 2018).

This study uses multiple datasets to perform Android malware detection approaches using new Machine learning algorithms (ML). The first dataset is called dataset for Android malware detection (Xiong, 2020). It was collected from three different sources: the Drebin dataset, AMD dataset, and Google play store. The second dataset was collected from many sources, including ApkMetaReport, AssetReport, ByteCodeReport, and VirusTotalReport (Thon, 2018). A JSON file contains a lot of data about the applications generated for the previously mentioned resources. Then we noticed that only ApkMetaReport contains permission, so we write code to extract the permissions and app name and hash and save it in csv in order to deal with the dataset in an easier way. After that, we will use new ML algorithms to detect Android malware by using the application permissions from the previously mentioned dataset.

The following sections are organized as follows. Section 2 shows a literature review of Android and malware. The methodology is presented in Section 3. In Section 4, we show the implementation and the results. Conclusion in section 5. Finally, future work is presented in section 6.

2. Literature review

Recently, more attempts have been spent on analyzing permission in the Android framework. Researchers have used machine learning and data mining techniques to detect Android malware based on permission use. In this section, we will look at how serious the issue of abusing permissions is. In addition, we will discuss many permission-based methods that have been proposed to identify malware in Android applications.

2.1. Abusing Android permissions

The studies conducted by Alenezi and Almomani (2017, 2018) pay more attention to Android permissions. Also, how attacks can exploit them. They study 71 apps and check the requested and used permissions. They find out that: 80.3% of the apps request permissions more than they need.11.2% of apps have requested less than what they have used 8.5% of the apps have used exactly what they requested. Many permissions, with 63.77 percent, are normal permissions. Dangerous permissions come next with 27.17 percent. Both Signature and SignatureOrSystem represent 9.05 percent of the total permissions.

2.2. Using permissions for malware detection

DroidPermissionMiner is a static analysis mechanism suggested by (Aswini et al., 2014) to detect malware by evaluating applications' permissions. The research included reviewing 436 applications APK files and mined unique features related to malicious activities. The proposed model categorized the apps based on machine learning classifiers.

Utku et al. (2017) employed machine-learning methods such as Naive Bayes and KNN algorithms in Android malware detection focused on app permissions. They tested their method on more than 5000 malicious applications of the Drebin dataset and about 1300 benign applications retrieved from the Android play store. They obtained a good malware detection ratio of 97.29 percent with the Naive Bayes classifier and 97.74 percent with the K-Nearest Neighbors algorithm.

The study of Alsoghyer et al. (2019) offers a comprehensive investigation. One thousand samples were screened from Android to collect all permissions required or used by the app whether it is benign or ransomware. The experiments show that the framework gets an accuracy of 96.9% by using only the permissions.

Almomani et al. (2020) present a thorough review of the Android permissions framework from 2008 until 2020. Case research was performed for the last five years' versions of the top Android apps, and all the API levels (1 to 30), have been examined. For each release update of permission, the previous permissions were not deleted to allow existing applications to use them. The authors notice that the permissions were expanded in all third-party applications over the last five years. Besides the continuous enhancements of Android permission implement numerous security issues and disadvantages. Consequently, the application can exploit this system feature to improve its privileges. In the work of (Alqatawna et al., 2021), Android botnets are examined with static analysis to extract features from source code after reverse-engineering the applications. The features are then used to create efficient machine-learning models for such malicious applications. They extract features from 2224 Android benign applications and 1928 botnet applications ISCX dataset. They download and scan the applications using the tools developed in this work. The features extracted are trained and tested with four common ML classifiers. For the training of classifiers in this work, 406 features were used. The best results can be achieved using the RF and MLP classifier, respectively. RF classifier achieved 98%. Yerima et al. 2021 presents an analysis of the Android botnet using deep learning techniques. They use 342 static features extracted from the application to unclassify applications into clean or botnets. Deep learning has many models (Wang, 2018). The results of these models are evaluated on 6802 apps consisting of 1929 ISCX dataset applications. Convolutional Neural Networks (CNN), Dense Neural Networks (DNN), Gated Recurrent Units (GRU), Long Short-Term Memory (LSTM), CNN-LSTM, and CNN-GRU were among the models investigated. The best results were obtained by the DNN and CNN-GRU models. Both models obtained the highest accuracy of 99.1 percent, which correlates to the highest F1-score of 0.984. (Abdullah et al., 2020) proposed a technique for detecting Android ransomware using dynamic analysis. There were two datasets used: a ransomware dataset and a benign dataset. The system calls obtained from dynamic analysis were used as features in the proposed method. The proposed features were used to classify the instances using the classification algorithms Random Forest, J48, and Naive Bayes.

3. Methods and the materials

We will introduce the multiple datasets we identify to perform malware detection approaches using new Machine learning algorithms (ML). After that, we will select the appropriate datasets. We will perform the pre-processing of these datasets to make them ready for the experiments. In addition, we will show the overall methodology for this work.

3.1. Methodology

We will identify the Android datasets that contain benign and malware applications. We will study each one of them, and we will select the appropriate datasets in terms of the number of applications and whether it is updated recently or not. Then we will divide the dataset into two different datasets: training dataset and testing dataset. After that will try the training dataset with free noise insertion, then will insert noise in different percentages. The main goal of the noise insertion is to see which ML classifier will be noise tolerant and provide better results. Next, we will use noise-cleaning algorithms to increase the efficiency of the ML algorithms. For the two datasets, training, and testing, we might perform optimization and feature selection. As a result, the unused features will be eliminated, increasing the performance, and reducing the time. The optimized dataset will be tested using many ML algorithms like decision tree, random forest, and support vector machine. Finally, we measure the performance using accuracy, recall, and precision equations. Fig. 1 summarizes the overall methodology, while Fig. 2 presents the algorithm of the steps that will be conducted during the research.

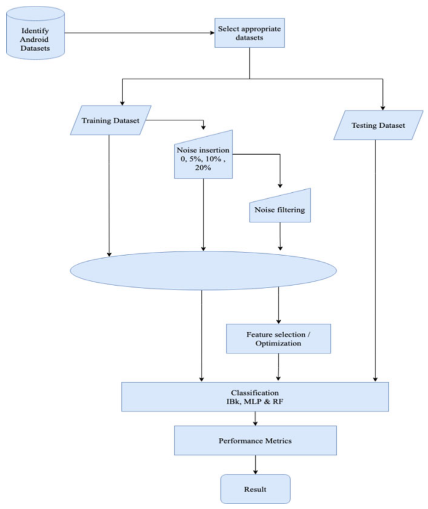

Identify Android Datasets
Select appropriate datasets
Training Dataset
Testing Dataset
Noise insertion 0, 5%, 10% , 20%
Noise filtering
Feature selection / Optimization
Classification IBk, MLP & RF
Performance Metrics
Result

Fig. 1. **Overall Methodology**

```
Procedure Detect_Malicious (datasets $D$, noise_level = {0, 5, 10, 20})
Return trained models

//Select an appropriate dataset $D_a$ and split it into train set $D_{TN}$ and test set $D_{TT}$
$D_{TN}$, $D_{TT}$ = train_test_split ($D_a$)

//Call Noise_Insertion function with noise level
For each noise_level:
            $D_{TN\_noise\_level}$ = Noise_Insertion ($D_{TN}$, noise_level)

// Noise Filtering
For each dataset with inserted noise:
            $D_{TN\_noise\_level\_f}$ = Noise_Filtering ($D_{TN\_level}$)

// Feature Selection for all datasets (TN, TN_level, TN_level_f)
$D_{TN\_s}$ = Feature_Selection ($D_{TN}$)
For each dataset after noise insertion:
      $D_{TN\_noise\_level\_s}$ = Feature_Selection ($D_{TN\_level}$)
For each dataset with noise insertion after filtering:
      $D_{TN\_noise\_level\_f\_s}$ = Feature_Selection ($D_{TN\_noise\_level\_f}$)

// Train classification (IBK, MLP & RF) models for each datasets
For each dataset (TN, TN_noise_level, TN_noise_level_f, TN_s, TN_noise_level_s,
            TN_noise_level_f_s):
      For each classification model:
Train_model (classification_model, dataset)
Evaluate_model ($D_{TT}$)
end
```

Fig. 2. Research algorithm

3.2. Dataset

In machine learning, data plays a vital role. The data are used to train and evaluate any machine learning model, most often as static data. Such datasets' properties can affect a model's behavior: if its deployment environment does not fit its training or assessment datasets, or if those datasets represent undesirable distortions, a model will probably not function well. Android datasets usually contain both benign applications and malicious applications. Most researchers obtain these from app stores such as Google Play Store for benign applications. On the other hand, the malware application depends on malicious actions in the study. For instance, VirusTotal was a significant source for many researchers of malware Android applications. A study done by (Almomani et al., 2019) disclosed the most popular datasets used in research work in 2018 for Androzoo. It was used in 42% of Android malware research as shown in Fig. 3. After that, the Derbin dataset was used in 25% of the research. At the same time, the third most used dataset with 23% is Genome.

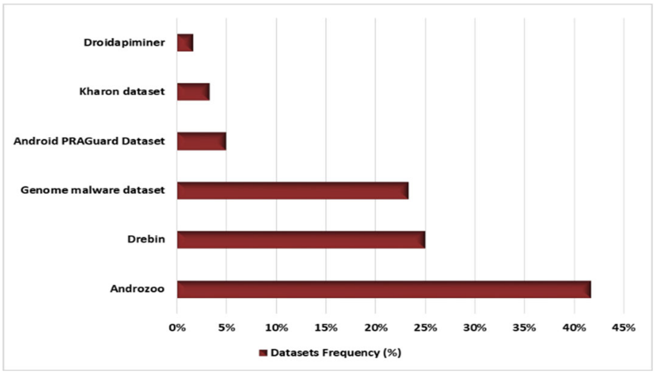


Fig. 2. Android malware Datasets usage in 2018 research

3.2.1. Available Dataset

Datasets are primarily categorized into two types. In the first category, the datasets of the Android apps. This covers both benign applications and malware apps. Most researchers are gathering them from app stores like Google play store for the benign apps. While for the malicious software, it depends on the malicious activity under analysis. VirusTotal was one of the critical sources for many researchers of malware Android apps. In the second category, the datasets are created after extracting the application features. The researchers can either use the existing dataset considering the features under analysis or create new ones by removing the framework's features (Almomani et al., 2019). In the following subsection, we will explore the most used malware datasets in the literature.

3.2.2. AMD Project

Android malware dataset (AMD) has 24,650 samples from 71 malware families, classified into 135 varieties, between 2010 and 2016. The dataset offers a current image of the Android malware landscape and is available to the public. Android malware apps are obtained from many sources, the samples are analyzed, and detailed behavior is reported (Wei et al., 2017).

3.2.3. Kharon Malware Dataset

The Kharon dataset is a completely reversed and recorded malware collection. This dataset was designed to assist the researchers in assessing their research experiments. Its building required an incredible amount of work to comprehend and create documentation for malicious applications. Kharon dataset focused on seven types of malware (Kiss et al., 2016).

3.2.4. AndroZoo

AndroZoo is a growing list of Android apps gathered from many different sources like the Google Play store. Currently, 14.456.513 multiple APKs are available, each with numerous antivirus products analyzed to identify malware applications. They distribute this dataset for ongoing research and to promote new research topics on Android apps (AndroZoo, 2020). According to a study done by Almomani and Khayer, this is the most used dataset in the 2018 research publications (Almomani et al., 2019).

3.2.5. RansDS-Permissions

This dataset was conducted to overcome the shortage of ransomware malware detection for Android applications. The creators study the old datasets HelDroid and RansomProber and remove the duplications. They also add new applications from VirusTotal and Koodous by searching for Android ransomware applications. There are 500 malicious applications. The creators also add 500 benign applications to study the difference in permissions between the benign and ransomware apps (Alsoghyer & Almomani, 2019, 2020).

3.2.6. Drebin

The dataset comprises 5,560 applications of 179 separate families of malware. During the period from August 2010 to October 2012, samples were collected. Drebin carries out a static code analysis on Android applications that extract as many features as possible from the application. These features are arranged in string sets, for example, permissions, API calls, and addresses of the network (Arp et al., 2014).

3.2.7. Dataset for Android Malware Detection

This dataset was published in the IEEE data port by (Xiong, 2020) for Android malware detection. It was obtained from various sources: Drebin dataset, AMD dataset, and Google play store. It consists of 7,681 benign applications from the Android default store, 8000 samples of benign and malicious from this dataset, and 12000 benign and malicious samples from the AMD dataset. The dataset was published as text files, and it misses some features.

3.2.8. Static Analysis of Android Benign and Malware Applications

Thon published this dataset by performing static analysis to process permissions and intents of Android apps by reading the applications' manifest files (Thon, 2018). It uses these files to perform many collected data operations on many static analyzing tools, including ApkMetaReport, AssetReport, ByteCodeReport, and VirusTotalReport. A JSON file contains a lot of data about the applications generated for the previously mentioned tools. The dataset consists of more than 8000 benign and malware Android applications and is published as two different datasets, one for benign apps and the other for malicious apps.

3.2.9. Preprocessing for the dataset

The first dataset we work on is the dataset for Android malware detection. It was collected from different sources. The first source is the Derbin dataset. The creator takes from it 4000 malicious apps and another 4000 benign apps. The second source was the AMD project dataset, where the creators take from it 6000 malicious apps and another 6000 benign apps. The dataset was published in two different folders; each folder represents one of the previously mentioned datasets, and inside each folder, there are five different text files. Each file has one specific purpose; the first file contains malware families (Derbin has 179 families and AMD 71 families). Another file presents the indexing of malware families. The third file is called label and consists of each application's labels, either benign or malware. The authors show a sample for the apps in each dataset in the fourth file where the fifth file was containing the features. We contribute to this dataset by merging the files into single dataset files (.csv) and classifying them. The challenge we faced was that the feature's title was not mentioned in the dataset.

The second dataset we work on is a static analysis of Android benign and malware applications. It was published as two different datasets, one for benign applications and the other for malicious applications. The creator uses many static code analyzers, including ApkMetaReport, AssetReport, ByteCodeReport, and VirusTotalReport. Each dataset contains four folders, and each folder represents the used tool. Inside every folder, multiple JSON files. In each file, the file name is the hash of the application, and inside the file is the result of the analyzing tool. To work with the dataset, we write a Python code that reads the JSON files and converts them into a single dataset file to make it easier to read and work with. After that, we notice that the result of the ApkMetaReport is the only one consisting of permissions, so we write another Python code to read the JSON files and convert them into a dataset that contains the name of every application along with the permissions used in the application.

3.2.10. Dataset statistics

A key design of the Android security architecture is that no app can perform operations that could adversely affect the user. Therefore, the Android permissions framework regulates the applications' requests for components, confidential data, and particular system functionality. Table 1 shows 10 Android system permissions, where there are 172 permissions (AndroidDeveloper, 2020). The aim of showing all Android System Permissions is to help us to understand the dataset statistics in the following sections.

Table 1
Some Android system permissions (AndroidDeveloper, 2020)

#	Constants	Added in API level	Deprecated
1	ACCEPT_HANDOVER	28	NO
2	ACCESS_BACKGROUND_LOCATION	29	NO
3	ACCESS_CHECKIN_PROPERTIES	1	NO
4	ACCESS_COARSE_LOCATION	1	NO
5	ACCESS_FINE_LOCATION	1	NO
6	ACCESS_LOCATION_EXTRA_COMMANDS	1	NO
7	ACCESS_MEDIA_LOCATION	29	NO
8	ACCESS_NETWORK_STATE	1	NO
9	ACCESS_NOTIFICATION_POLICY	23	NO
10	ACCESS_WIFI_STATE	1	NO

The first dataset mentioned previously is called Android malware detection. It contains 20000 applications, collected from two datasets AMD and Derbain. AMD dataset contains 12000 applications, half of them benign, and the other half is malicious application. The dataset after the processing contains 434 features. While Derbain has 8000 applications, half of them benign, and the other half is malicious application. Derbain contains 491 features after processing.

The second dataset, which was collected from Kaggle, contains 8315 applications—divided into two classes, benign and malware. The number of benign applications is 4304, while the number of malicious applications is 4011. The number of features in this dataset is 175; the first three present the information about the applications like the application name, the hash of the application, and the class (either benign or malicious), and the rest of the features (172) present the names of the Android permissions which are same as the permissions in Table 1. The following 3D recommendation chart presents the usage of these permissions based on their classes in Figure 4. We can notice that the most used permission in the benign applications is:

- android.permission.VIBRATE.
- android.permission.WAKE_LOCK.
- android.permission.WRITE_EXTERNAL_STORAGE.
- android.permission.ACCESS_NETWORK_STATE.
- android.permission.INTERNET.

While the most used permissions in the malware applications are:

- android.permission.ACCESS_WIFI_STATE.
- android.permission.READ_PHONE_STATE.
- android.permission.WRITE_EXTERNAL_STORAGE.
- android.permission.ACCESS_NETWORK_STATE.
- android.permission.INTERNET.

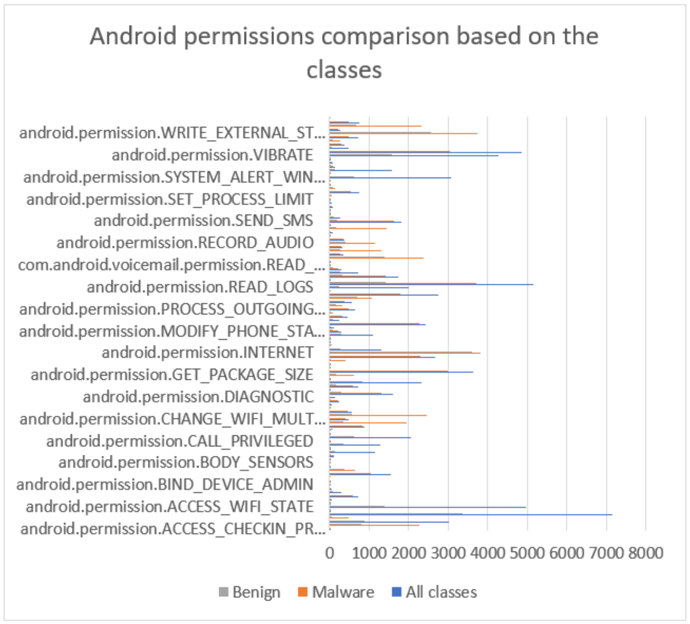


Fig. 3. Android permissions comparison based on classes

3.3. *Feature selection techniques*

Choosing an appropriate set of features is called feature selection. When we make feature selection, we restrict ourselves to a subset of the terms as features; some machine learning models do better. We should exclude terms that are very uncommon as well as terms that are used by both classes and categories. The goal of feature selection is to improve the predictors' prediction accuracy, provide quicker and more cost-effective predictors, and understand the data's underlying mechanism (Russell et al., 2020; Choudhary & Kishore, 2018). In this work, we will use different types of feature selection: evolutionary and statistical feature selection techniques.

3.3.1. Statistical technique

Various statistical feature selection techniques have been proposed. Based on how the feature selection module functions in the overall classification process, feature selection approaches can be divided into three categories: filter approach, wrapper approach, and embedded approach. Filter approaches are usually quicker, while wrapper approaches are more efficient yet computationally costly. Feature selection strategies often aim for output that is comparable to wrapper approaches that use a variety of computational approaches. Feature selection strategies are usually based on how feature–feature and feature–class correlation is computed. The correlation measure that is used heavily influences these two computations (Borah et al., 2014). In this work, we will use one of the filter methods called: Correlation-based Feature Selection (CFS). The feature subset is ranked according to the correlation with the class label and other features by CFS. A higher value is given to subcategories that display strong correlations with the class label and less correlations with other features. This multivariant approach does not take into account irrelevant and redundant dataset features (Kumar et al., 2017).

3.3.2. Evolutionary technique

Feature selection becomes important as the size and complexity of datasets grow. The selection process can bring advantages for better model performance, computation efficiency and simpler models. Evolutionary feature selection includes many naturally inspired techniques such as genetic algorithms, genetic programming, ant colony optimization, or particle swarm optimization algorithms. Such strategies are ideal for selecting features since the rendering of a feature subset is simple and the assessment can also be performed easily by wrapping or filter algorithms. In addition, the ability of such heuristic algorithms to search vast areas efficiently benefits the selection of features (Iglesia, 2013). Many algorithms can be used for evolutionary feature selection including:

- Gray Wolf Optimizer (GWO): algorithm imitates the hierarchy of leadership and the system for hunting gray wolves. To simulate the hierarchy of leadership, four groups of gray wolves were employed, such as alpha, beta, delta, and omega. The three key stages of hunting are also introduced, search for prey, surrounding prey, and attacking prey. The results show that GWO is very competitive in relation to popular heuristics algorithms like PSO, Gravitational Search Algorithm (GSA), Differential Evolution (DE), Evolutionary Programming (EP), and Evolution Strategy (ES) (Mirjalili et al., 2014).
- Firefly algorithm (FFA): evolutionary feature selection algorithm focused on the pattern of fireflies and their behavior. Essentially, FFA uses three idealized rules (Yang et al., 2013):

1. Fireflies are unisexual so they attract another firefly regardless of sex.
2. The attraction is directly proportional to the brightness, and both reduce with increasing space. Therefore, the darker one would move to the lighter one with any two flashing fireflies. It will move randomly if there is not one lighter than another.
3. The brightness of a firefly depends on the landscape of the objective function.

A pseudocode that summarizes how the FFA algorithm works is shown in Figure 5. Many studies confirmed that the FFA algorithm is very efficient in solving nonlinear constrained optimization tasks. FFA was also used in solving optimization problems. Optimization results show that the Firefly algorithm is potentially better than other proposed algorithms, for example, the particle swarm optimization (Yang et al., 2010).

```
Objective function $f(\mathbf{X})$, $\mathbf{X}=\{x_1, \ldots, x_d\}^T$
Generate an initial population of fireflies $\mathbf{X}_i = (i = 1, 2, \ldots, n)$
Light intensity $\mathbf{I}_i$ at $x_i$ is determine by $f(\mathbf{X}_i)$
Define absorption coefficient $\gamma$
  while ($t <$ MaxGeneration)
    for $i = \mathbf{1}$: n all n fireflies
      for $j = \mathbf{1}$ : i all n fireflies
        if ($\mathbf{I}_j > \mathbf{I}_i$), Move firefly i toward j in two dimension; end if
          Attractiveness with distance r via exp[$-\gamma r$];
          Evaluate new solutions and update light intensity;
      end for j
    end for i
    Rank fireflies and find the current best;
  end while
end
```

Fig. 4. Firefly Algorithm (FFA) pseudo code (Yang et al., 2010)

3.4. Noise insertion and cleaning

The existence of noise can have a negative impact on the performance of machine learning algorithms. Mislabeled data or incidents that are wrongly identified are examples of noise. In this work, we will obtain many noise insertion and cleaning experiments to find out the ML algorithms that provide the best results with a noisy dataset.

3.4.1. Noise insertion to training dataset

The accuracy of ML algorithms may be affected by the presence of noise. Noise may take various forms, such as wrongly labeled and classified data or instances. Training a neural network with a small dataset will cause the network to record all training samples. The model will learn the unique input examples and their related outputs rather than the general mapping between inputs and outputs. This can lead to a well-performing model in the training dataset and bad in new data, resulting in a straggler dataset being overfitted and in poor results. Random noise is an approach to improve the general error and improve the mapping structure (Brownlee, 2018). In our work, we will add noise as label change and data change. The noise might be a mistake or intentional. The noise will be inserted at different levels: no noise, 5 percent, 10 percent, and 20 percent. We will measure the performance result for each one of these noises.

3.4.2. Noise cleaning

Algorithms for noise-cleaning are widely used in ML. ML algorithms store the training data during training. For any new instance, noise cleaning looks for a memory of the most similar instances via a distance function and uses the class for the new instance with the majority of votes. Several algorithms can be used to conduct noise filtering, which include (Amro et al., 2020):

- Instance-Based Learning (IBL): has made heavy use of instance reduction algorithms (noise cleaning). IBL saves the training data during training, and when a new instance needs to be categorized, they use a distance feature to scan their memory for the most similar instance(s) and use the class with the most votes as the expected class for the new instance (Amro et al., 2020). To save memory and preserve classification accuracy, instance reduction algorithms attempt to classify and retain only the most important instances. IBL measures the distance to determine how close the new instance is to others to predict the output class (Wilson and Martinez 1997). To measure the distance between two examples, a variety of distance functions can be used. For continuous and nominal attributes, respectively, the Euclidean distance function and Value Difference Metric (VDM) may be used, but not for both types of attributes. In our experiments, we used the Heterogeneous Value Difference Metric (HVDM), which is suitable for heterogeneous applications of both types of attributes. It can also calculate distance when missing values are present. The distance between two input vectors *x* and *y* is calculated by HVDM as shown in Eq. (1) (Amro et al., 2020):

$$\mathrm{HVDM}(\mathrm{x},\mathrm{y}) = \sqrt{\sum_{a=1}^{m} d_a 2(x_a, y_a)} \tag{1}$$

The number of attributes is denoted by the letter m. For attribute a, the function d (x_a, y_a) returns the distance between x_a and y_a of vectors x and y. If the attribute is an integer, it uses the Euclidean distance function (Eq. (2)), and if the attribute is nominal, it uses the VDM distance metric (Eq. (3)).

$$d\,(x_a, y_a) = (x_a - y_a)/(\,x_{max} - y_{max}) \tag{2}$$

$$d\,(x_a{'}\,y_a) \;=\; vdm(x_a{'}\,y_a) \;=\; \sqrt{\sum_{c=1}^{c} (P(c|x_a) \;-\; P(c|y_a))^2} \tag{3}$$

where:

- *x* and *y* are two vectors (documents)
- The values of attributes for the vectors are x_a and y_a, respectively. And *m* is the total attributes.
- *c* is the number of classes (document categories).

- The Edited Nearest Neighbor Algorithm (ENN): the algorithm will check if the instance has a class different from that of the majority of its k nearest neighbors. It will delete an instance. The reason for this is possibly a noisy instance or close-border instance, and it may lead to a wrong classification by classifying new instances.
- Encoding Length (Explore): First, the ELGrow algorithm is performed, and then 1000 mutations are made to improve the classifier's performance. In each mutation, delete, insert, or replace an instance is changed to the reduced set. The move would be retained only if the expense of the classification is not increased.
- DROP5: This algorithm considers first eliminating and proceeding outward instances that are close to its nearest opponent. This will lead to the deletion of most internal instances. After that, instances of elimination are then tested, starting at the closest opponent's most distant instance. An instance is omitted if, at least, as many instances, it is correctly identified without its nearest k neighbors. This is done many times until there are no further improvements can be made.

- DROP3: To delete noisy instances first, the algorithm employs a noise-filtering pass, such as ENN. The instances are sorted by the distance between them and the nearest opponent (nearest neighbor with a different class). The instances that are the furthest away from their closest opponent are then eliminated first. As a result, this algorithm tends to strip center points while retaining border points; this is why it is critical to remove the noisy instances with ENN.

3.5. Machine Learning Algorithms

Here, we will show different machine learning techniques, which are used in our experiment.

3.5.1. Neural Network

The human brain was the inspiration for neural networks (NNs). In artificial intelligence, the NNs are implemented using various artificial neurons scattered through several layers. The input layer, the hidden layer, and the output layer are the three layers that shape the NNs. The dataset is fed into the input layer, and the classification results are shown in the output layer (AlGethami et al., 2021). Deep Learning relies heavily on artificial neural networks (NNs). They are adaptable, efficient, and scalable, making them suitable for large-scale, high-complexity machine learning tasks like classifying billions of photos, powering speech recognition systems, and suggesting the best videos to watch to millions of users every day (Géron, 2019).

3.5.2. Decision Trees

A decision tree (DT) represents a function that maps a vector with the values of an attribute to a single decision output value. To draw a conclusion, a decision tree does a series of tests beginning at the root and then doing tests as appropriate before it reaches the leaf (Russell et al., 2020). DT utilizes a system of rules to classify information depending on the meanings of the attributes. Classification is shown in a tree structure format, where branches reflect the value collection of the input characteristics, leading to those classifications and the leaves are class labels. The DTs distinguish high classification accuracy and ease of implementation (AlGethami et al., 2021).

3.5.3. Random Forest

The random forest (RF) model variates decision trees, which take additional steps to diversify the arboreal complex and reduce change. Random forests can be used for classification or regression. The concept is arbitrarily to vary the choices for attributes rather than to prepare them. We choose a random sample of attributes in each split point to calculate which data is most beneficial (Russell et al., 2020). By a plurality of weighted voting, the results of the RF can be controlled. RF can be viewed as an ensemble method (Li et al., 2022) of learning, as the RF combines the results of various decision trees (AlGethami et al., 2021)

3.5.4. Support Vector Machine

A support vector machine (SVM) is a very flexible and robust learning machine, which can perform classification, regression, and even outer detection. SVMs are ideal for classifying complex but small or medium datasets (Géron, 2019). SVM has been trained to classify various data categories from various disciplines as a supervised learning technique. SVM creates a hyperplane or multiple hyperplanes, and the best hyperplane in it divides data into different classes, with the most significant class division (Ahmad et al., 2018).

3.5.5. Instance Based Learning (IBL)

It uses a very easy method to store and label the unknown data by determining the distance of the most equivalent example already stored. The algorithm has various implementations such as IBL and IBK (K-nearest neighbour algorithm). For which IBK algorithm decides on the classification of a new example using a voting system, with the number of votes denoted by the "k" value. To prevent tie situations, the value of k must be odd (Kanwal & Bostanci, 2016).

3.5.6. Multilayer perceptron (MLP)

A feed-forward artificial neural network underpins the multilayer perceptron classifier. It is made up of several layers. Each neural layer in the network is fully linked to the next neural layer, and the input layer's nodes represent the input data. All other nodes use a linear combination of the inputs with the nodes' weights and bias, as well as an activation or relation mechanism, to map inputs to outputs (Dua et al., 2017).

3.5.7. Performance Metrics

The appropriate performance metrics to assess ML algorithms using Android permission datasets must be defined. The performance metrics should therefore be appropriate for the topic under review. We will use notable methods of machine learning in this research: SVM, random forest, and extreme machine learning (ELM) since these methods are known for their classification capabilities (AlGethami et al., 2021). These performance metrics are accuracy, precision, and recall. A simpler way to measure a classifier's performance is to use the confusion matrix. The classification of each testing instance falls within four cases described as follows:

a. True positive (TP): malicious cases that were correctly classified by the algorithm as malicious.
b. True negative (TN): benign cases that were correctly classified by the algorithm as benign.
c. False positive (FP): benign cases that were incorrectly classified by the algorithm as malicious.
d. False negative (FN): malicious cases that were incorrectly classified by the algorithm as benign.

To determine the performance metrics, the four above values are used (AlGethami et al., 2021). Accuracy is the proportion of correctly categorized instances in the test dataset with the total number of instances as shown by Eq. (4):

$$accuracy = \frac{\mathrm{TP} + \mathrm{TN}}{\mathrm{TP} + \mathrm{TN} + \mathrm{FP} + \mathrm{FN}} \tag{4}$$

The accuracy of the positive predictions; is called the precision and can be calculated by the following equation:

$$precision = \frac{\mathrm{TP}}{\mathrm{TP} + \mathrm{FP}} \tag{5}$$

Precision, or a true positive rate, is widely used with another metric called recall (Géron, 2019). The recall is the ratio of positive instances correctly identified by the classifying classifier, which the following equation can calculate:

$$recall = \frac{TP}{TP + FN} \tag{6}$$

Accuracy is the most important metric for this study since it shows the efficient ML algorithm percentage of the testing instances being classified (AlGethami et al., 2021).

4. Implementation

This section shows all the information related to the methods of implementing the various sets of experiments in this project. This section will illustrate the steps needed to preprocess the Android permission datasets. Furthermore, we will demonstrate the most common testing option for several tools such as DROP for noise insertion and cleaning, EvoloPy for feature selection, and Weka for statistical features selection and classification. In addition to the device configurations for the system used to conduct the experiments.

4.1. Dataset balancing

In real-world datasets, there is an imbalanced distribution of samples from various groups. The classifier techniques are incapable of dealing with this issue. In the literature, typical data balancing approaches rely on data sampling, which involves either under sampling or oversampling the majority or the minority, respectively. The learning performance should be improved by a clever combination of under-sampling the majority class and oversampling the minority class (Susan et al., 2019). Table 2 shows the details of the dataset understudy, it is obvious that the dataset is imbalanced since the benign label is more than the malware labels. Imbalance data need preprocesses to get balanced data from the majority or minority. One of the solutions suggested was Synthetic Minority Oversampling (SMOTE). In SMOTE extra minority samples are produced along the line segment between two available minority samples, with no regard for the samples in the adversarial majority class. SMOTE has been studied on various datasets that have different degrees of imbalance and differing quantities of data in training sets. The result shows that it provides an approach to oversampling that can improve the accuracy of classifiers for a minority class. SMOTE algorithm is shown in Fig. 9 (Chawla et al,. 2002).

Table 1
dataset under study details

Name	Format	Source	Instances	Features	Labels	Label /Instance
Static Analysis of Android benign and malware apps	json	Kaggle	8315	175	2	Benign: 4304 Malware:4011

To oversample our dataset, we used open-source ML software called the workbench for machine learning (WEKA). WEKA offers learning algorithm implementations that you can easily implement on your dataset. It also contains several methods for dataset transformation, such as discretionary and sampling algorithms. WEKA has been developed in New Zealand at the University of Waikato (Witten et al., 2016). Fig. 6 shows our dataset before balancing, and as we can notice the benign is bigger than malware by 293 instances. SMOTE is one of the algorithms supported in WEKA, hence we used it using the configuration shown in Fig. 7. The percentage number that appears in the config window represents the amount of over sampling and we used here 7.305 since 1.07305 * 4011 (number of malware class instances) = 4034, which equals the number of benign instances. Fig. 10 shows the number of class instances after oversampling using the SMOTE algorithm presented in Figure 9. Our Android permission dataset is binarized which only has usable values: 0 permission is not used, 1: used permission. When we observe the dataset file after sampling, we notice that some of the new values entered by SMOTE are not 0 or 1 as shown in Fig. 8. In order to overcome this issue, we rounded the values to the nearest number (below 0.5 will be 0, and 0.5 or above will be considered 1).

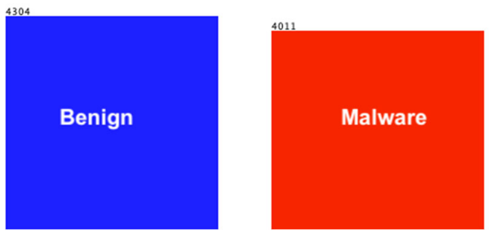
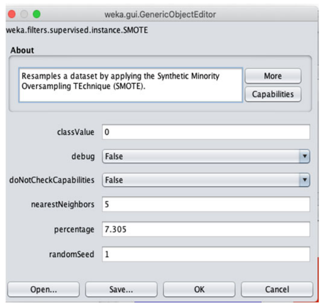


Fig. 5. dataset labels count before balancing



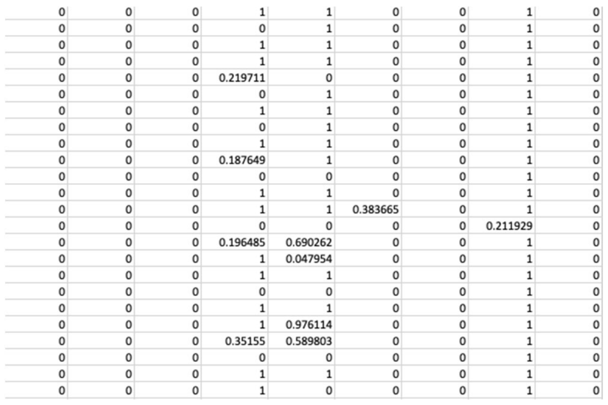
weka.filters.supervised.instance.SMOTE

About

Resamples a dataset by applying the Synthetic Minority Oversampling TEchnique (SMOTE).

classValue	0
debug	False
doNotCheckCapabilities	False
nearestNeighbors	5
percentage	7.305
randomSeed	1

Fig. 6. SMOTE Configuration in WEKA



0	0	0	1	1	0	0	1	0
0	0	0	0	1	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	0.219711	0	0	0	1	0
0	0	0	0	1	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	0	1	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	0.187649	1	0	0	1	0
0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	1	1	0.383665	0	1	0
0	0	0	0	0	0	0	0.211929	0
0	0	0	0.196485	0.690262	0	0	1	0
0	0	0	1	0.047954	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	1	0.976114	0	0	1	0
0	0	0	0.35155	0.589803	0	0	1	0
0	0	0	0	0	0	0	1	0
0	0	0	1	1	0	0	1	0
0	0	0	1	0	0	0	1	0

Fig. 7. dataset after balancing

```
Algorithm SMOTE(T, N, k)
Input: Number of minority class samples T; Amount of SMOTE N%; Number of nearest
neighbors k
Output: (N/100) * T synthetic minority class samples
1. (∗ If N is less than 100%, randomize the minority class samples as only a random
percent of them will be SMOTEd. ∗)
2. if N < 100
3.          then Randomize the T minority class samples
4.               T = (N/100) ∗ T
5.               N = 100
6. endif
7. N = (int) (N/100) (∗ The amount of SMOTE is assumed to be in integral multiples of
100. ∗)
8. k = Number of nearest neighbors
9. numattrs = Number of attributes
10. Sample[ ][ ]: array for original minority class samples
11. newindex: keeps a count of number of synthetic samples generated, initialized to 0
12. Synthetic[ ][ ]: array for synthetic samples
(∗ Compute k nearest neighbors for each minority class sample only. ∗)
13. for i ← 1 to T
14.         Compute k nearest neighbors for i, and save the indices in the nnarray
15.         Populate (N, i, nnarray)
16. endfor
   Populate (N, i, nnarray) (∗ Function to generate the synthetic samples. ∗)
17. while N ≠ 0
18.         Choose a random number between 1 and k, call it nn. This step chooses one of
the k nearest neighbors of i.
19.         for attr ← 1 to numattrs
20.              Compute: dif = Sample[nnarray[nn]][attr] − Sample[i][attr]
21.              Compute: gap = random number between 0 and 1
22.              Synthetic[newindex][attr] = Sample[i][attr] + gap ∗ dif
23.         endfor
24.         newindex++
25.         N = N − 1
26. endwhile
27. return (∗ End of Populate. ∗)
End of Pseudo-Code.
```

Fig. 8. SMOTE algorithm (Chawla et al., 2002)

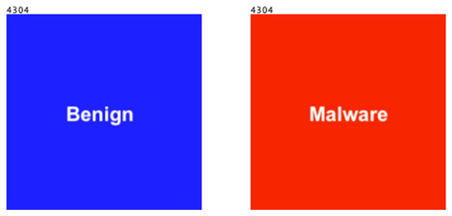


Fig. 9. dataset labels count after balancing

4.2. *Noise insertion and filtering*

In this section, we present an overview of the use of noise insertion and cleaning algorithms and the tools we used to perform the noise cleaning and insertion. After that, we will use these algorithms in our Android permission datasets and present the details and results of the experiment.

4.2.1. *Using Noise insertion and cleaning in ML algorithms*

The accuracy of ML algorithms may be affected by the presence of noise. Noise may take various forms, such as wrongly labeled and classified data or instances. A noise presence may affect the efficiency and accuracy of ML-based dataset, particularly in real-life situations where noisy information is more likely to occur. The effect of noise on the accuracy of the ML-based Android permission in this project is empirically investigated through a significant number of experiments. We will examine the impact on the efficacy of ML-based Android permissions of mislabeled instances. Various noise level is injected, and the accuracy of classification is investigated. The aim of using an ML algorithm to analyze noise is to measure the robustness of various ML algorithms in noisy situations. In other words, our objective is to find the most noise tolerant ML algorithm. This section shows an explanation of the proposed set of experiments on our Android permission datasets, specifically, noise injection, noise filtering, and the tool that is used to conduct the experiments.

Our Android permission dataset originally was in json format, we converted it into csv in order to handle the data easily. The tool we used for the noise insertion and cleaning algorithms is developed in C language and requires the dataset to be in NUM format (Wilson et al., 2000). In order to make our dataset in num file format we write a program in Python to convert our dataset from csv to num and also to return the files from num to csv format. Before any instance reduction algorithm was implemented, noise was also injected into full datasets. Adding noise is primarily accomplished by altering the original class of certain instances in a dataset. The noise can be applied manually or automatically; in our experiments, the noise was automatically inserted. The noise ratio determines the number of instances affected: (i.e., 5 percent, 10 percent, or 20 percent of instances). The instances that are affected by this process are randomly selected. The experiments were run over 10-fold cross-validation and then averaged for each combination of a dataset and an instance reduction algorithm. Each fold created nearly the same number of instances. The noise insertion and cleaning algorithms used in this experiment were kNN, ENN, Explore, DROP3, and DROP5. We ran the algorithms with different ratios at the same time (i.e., ENN with 0%, 5%, 10%, and 20%).

4.3. *Feature selection*

Choosing a subset of appropriate features to be used in the model. Features that are relatively uncommon, as well as features that are used by all classes, should be excluded. The aim of feature selection is to boost predictor accuracy, provide faster and more cost-effective predictors, and learn more about the data's underlying mechanism. In this work we will use evolutionary and statistical feature selection techniques, we will perform feature selection techniques on datasets of different noise ratios to observe whether the noise will have an impact on feature selection or not.

4.3.1. *Statistical feature selection*

Various statistical feature selection techniques have been proposed. We will use correlation-based feature selection (CFS). In order to perform the statistical feature selection on our dataset we will use WEKA. The input for the WEKA is the result of the best noise cleaning algorithm with different ratios from the previous section. Our objective is to observe the effect of the noise on the feature selection on different noise ratios.

4.3.2. *Evolutionary feature selection*

Feature selection becomes important as the size and complexity of datasets grow. Evolutionary feature selection includes such as genetic algorithms, genetic programming, ant colony optimization, or particle swarm optimization algorithms. Such strategies are ideal for selecting features since the rendering of a feature subset is simple and the assessment can also be performed easily by wrapping or filter algorithms. In this work, we will conduct our experiments on the same datasets we used on the statistical feature selection. The experiment will be on two evolutionary feature selection algorithms: Grey Wolf Optimizer (GWO) and Firefly Algorithm (FFA). The tool we used is a Python open-source optimization framework called EvoloPy-FS. EvoloPy-FS consists of a variety of well-considered swarm intelligence (SI) algorithms. It is aimed at optimizing feature selection issues. It is a framework that is simple to use, reusable, and adaptable. Eight metaheuristic optimizers were introduced in the framework, including particle swarm optimization (PSO), firefly algorithm (FFA), gray wolf optimizer (GWO), whale optimization algorithm (WOA), multi-verse optimizer (MVO), moth-flame optimization (MFO) algorithm, bat algorithm (BAT), and cuckoo search (CS). EvoloPy-FS allows you to load your dataset as a CSV file either as a physical path or URL (Abu Khurma et al., 2020). In this experiment we will use the result of the best noise cleaning algorithm, the input will be four datasets based on different noise ratios. While the result will be 8 datasets since we are going to use two different optimizers GWO and FFA.

4.4. Classification

The purpose of this work is to look at the factors that may affect the classification accuracy of ML-based Android malware detection systems. To do this, we will use WEKA. There are many categories of classifiers in WEKA including Bayesian classifiers, trees, rules, functions, lazy classifiers, and a final miscellaneous category. In this work, we will use different classification algorithms from different categories in order to evaluate their performance. The algorithms we will use are K-Nearest-Neighbor (IBk), Multilayer Perceptron (MLP), and Random Forest (RF). In order to get the accuracy of the classification algorithms, the datasets class values should be nominal rather than numeric. Since ML algorithms can only be compared to datasets with nominal class values, we convert the class values in our datasets from numeric values to nominal for the class attribute only.

4.5. Device Configuration

In this work, we used a MacBook Pro laptop (Retina, 13-inch, Early 2015), with a CPU processor 2.9 GHz Intel Core i5, 8 GB 1867 MHz DDR3. The operating system is MacOS Catalina version 10.15.7.

5. Research results and discussions

We will discuss the result of the experiments we perform in this work. The experiments are grouped based on their types: noise insertion and cleaning, feature selection, and finally, classification. The goal of the research is to find the most noise tolerance classifier for detecting Android malware applications.

5.1. Noise insertion and cleaning

The existence of noise can influence the accuracy of machine learning algorithms. Data or instances that have been incorrectly labeled and categorized are examples of noise. In this experiment, we will perform noise insertion and cleaning. Since many datasets have noise by default the instance cleaning will result in a more capable learning model. Our objective is to find the most accurate noise cleaning algorithm. We will use the noise cleaning algorithms introduced in subsection 4.4.2.

Before any noise cleaning algorithm is implemented, the noise was injected into datasets. In our experiments, the noise was automatically inserted. The noise ratio determines the number of instances affected: (i.e., 5 percent, 10 percent, or 20 percent of instances). The instances that are affected by this process are randomly selected. The noise insertion and cleaning algorithms used in this experiment were: kNN, ENN, Explore, DROP3, and DROP5. We ran the algorithms with different ratios at the same time (i.e., ENN with 0%, 5%, 10%, and 20%). The execution time varies based on the algorithm used. For example, the ENN takes about two to three hours for execution and DROP3 takes 40 hours on average for different noise ratios. The experiment of executing DROP5 with 0% noise took more than 96 hours of running time. Other DROP5 noise insertion took a significant amount of time also as shown in Table 3.

Table 2
Noise cleaning algorithms execution duration

Noise cleaning algorithm	0%	5%	10%	20%
kNN (none)	4 Hours	6 Hours	7 Hours	8 Hours
DROP5	96 Hours	88 Hours	80 Hours	70 Hours
DROP3	46 hours	40 hours	35 Hours	27 Hours
ENN	8 Hours	12 Hours	14 Hours	18 Hours
Explore	6 Hours	8 Hours	9 Hours	10 Hours

The detailed result of our experiment with all the trails information for all algorithms is shown in Table 4. Another summarized table for the result is shown in Table 5, and a comparison of the accuracy between the algorithms is shown in Fig. 11.

Table 3
detailed result of using noise insertion and cleaning algorithm

Noise Ratio	Trail	None (kNN)		ENN		DROP5		DROP3		Explore	
		Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
0%	1	89.00	100.00	87.83	89.22	89.33	8.20	88.00	6.30	86.67	0.04
	2	89.00	100.00	86.83	88.98	89.33	7.85	87.67	6.30	86.83	0.04
	3	88.83	100.00	86.83	89.13	87.67	7.74	86.17	6.46	85.17	0.04
	4	90.00	100.00	88.50	89.19	88.83	8.56	88.83	6.54	86.17	0.04
	5	88.17	100.00	87.17	89.13	86.50	7.93	86.33	7.17	86.17	0.04
	6	90.33	100.00	89.67	88.81	89.17	8.04	88.67	6.85	86.17	0.04
	7	86.67	100.00	86.33	88.91	87.17	7.70	86.67	6.46	82.67	0.04
	8	88.00	100.00	88.33	89.26	87.17	8.35	87.83	6.98	84.00	0.04
	9	87.83	100.00	87.67	89.13	87.50	8.59	88.00	7.02	85.00	0.04
	10	89.50	100.00	87.67	89.04	88.83	8.13	90.50	6.89	85.50	0.04
	Avg.	**88.73**	**100.00**	**89.83**	**89.08**	**88.15**	**8.11**	**87.87**	**6.70**	**85.43**	**0.04**
5%	1	88.33	100.00	87.17	84.85	88.50	9.11	88.33	5.76	86.17	0.04
	2	87.83	100.00	86.33	84.46	87.50	7.94	86.67	6.59	85.33	0.07
	3	88.33	100.00	86.17	84.78	87.67	8.44	86.67	6.00	86.17	0.17
	4	88.33	100.00	88.5	84.89	88.00	9.70	89.00	6.48	87.33	0.11
	5	86.17	100.00	85.83	84.96	85.83	8.04	85.50	6.63	85.83	0.15
	6	90.17	100.00	89.17	84.57	90.17	9.09	89.67	6.67	88.00	0.19
	7	85.17	100.00	84.33	84.94	85.00	10.06	85.67	5.91	84.83	0.15
	8	86.33	100.00	88.00	84.89	85.83	8.78	87.67	6.72	86.33	0.13
	9	86.67	100.00	86.5	84.91	86.17	9.33	86.17	7.11	87.00	0.15
	10	89.50	100.00	89.17	84.56	90.17	8.61	89.00	6.22	87.00	0.11
	Avg.	**87.68**	**100.00**	**87.12**	**84.78**	**87.48**	**8.91**	**87.43**	**6.41**	**86.40**	**0.13**
10%	1	86.83	100.00	86.33	80.24	88.33	9.35	87.17	6.83	86.17	0.04
	2	87.50	100.00	86.17	79.67	87.50	9.09	87.17	6.93	85.67	0.13
	3	85.83	100.00	86.33	80.17	86.83	9.87	86.50	6.61	86.00	0.11
	4	87.50	100.00	87.83	80.22	87.00	10.20	88.33	6.57	84.17	0.11
	5	84.83	100.00	85.33	80.15	84.33	10.39	85.83	7.02	85.33	0.15
	6	88.00	100.00	89.17	79.83	87.50	10.07	90.33	7.43	83.67	0.13
	7	83.83	100.00	83.83	79.96	84.67	10.07	84.00	6.39	80.67	0.11
	8	84.67	100.00	87.33	80.24	85.00	9.63	87.17	6.76	81.83	0.09
	9	85.17	100.00	86.33	80.52	85.83	10.37	85.17	7.43	82.50	0.07
	10	87.83	100.00	88.00	80.07	87.83	10.19	88.67	7.37	84.33	0.09
	Avg.	**86.20**	**100.00**	**86.67**	**80.11**	**86.48**	**9.92**	**87.03**	**6.93**	**84.03**	**0.10**
20%	1	83.50	100.00	83.33	71.61	83.50	11.94	83.50	7.43	84.67	0.04
	2	82.00	100.00	82.17	71.35	81.83	12.85	83.17	8.94	82.67	0.13
	3	81.67	100.00	85.00	71.46	83.83	11.65	84.33	7.81	87.17	0.09
	4	83.17	100.00	87.00	71.41	86.83	12.91	87.33	8.43	87.50	0.09
	5	80.17	100.00	83.50	71.59	80.83	11.93	81.67	8.80	85.83	0.11
	6	81.17	100.00	85.17	71.70	84.33	12.02	87.00	9.09	88.50	0.13
	7	80.17	100.00	81.17	71.52	81.83	12.81	81.50	8.57	85.00	0.13
	8	81.33	100.00	85.33	71.65	83.83	11.59	85.83	8.46	86.50	0.11
	9	81.33	100.00	83.83	72.43	82.50	13.09	85.33	8.80	85.83	0.13
	10	82.50	100.00	85.33	71.93	85.00	11.87	85.00	8.63	85.67	0.15
	Avg.	**81.70**	**100.00**	**84.18**	**71.66**	**83.43**	**12.27**	**84.47**	**8.50**	**85.93**	**0.11**

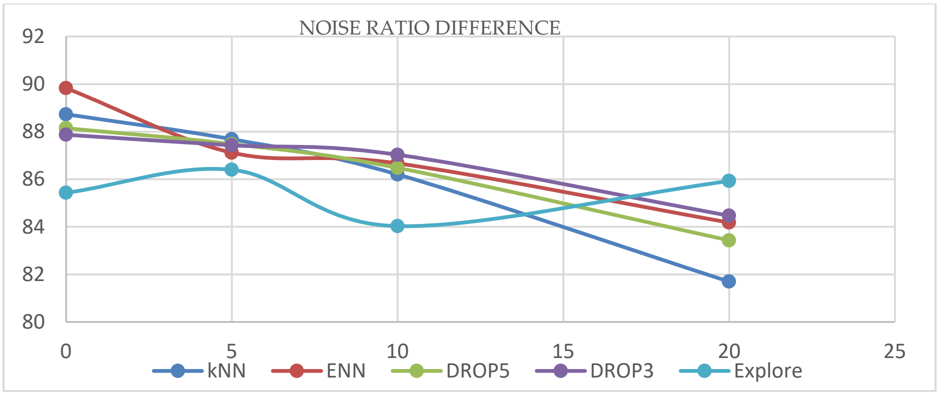


Fig. 10. Comparison of the accuracy between the algorithms

Table 4
Summary of using noise insertion and cleaning algorithm

Noise Ratio	None (kNN)		ENN		DROP5		DROP3		Explore	
	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
0%	88.73	100.00	89.83	89.08	88.15	8.11	87.87	6.70	85.43	0.04
5%	87.68	100.00	87.12	84.78	87.48	8.91	87.43	6.41	86.40	0.13
10%	86.20	100.00	86.67	80.11	86.48	9.92	87.03	6.93	84.03	0.10
20%	81.70	100.00	84.18	71.66	83.43	12.27	84.47	8.50	85.93	0.11

We can notice from the previous results that DROP3 and ENN provide the best accuracy at all the noise insertion levels. We also observe that their result might be close to each other but DROP 3 was better in almost all the noise insertion levels. The following experiments will work on the Datasets of DROP 3 only since it is the best noise-cleaning algorithm.

5.2. Feature selection

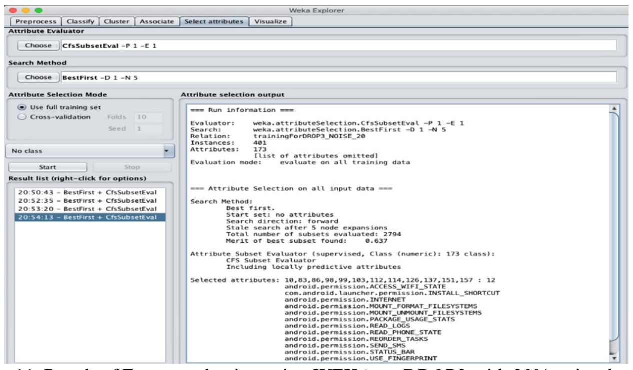
Features that are not used by both classes, as well as features that are relatively rare, should be omitted. The objective of the feature selection is to increase the accuracy and the performance. In this work, we will use evolutionary and statistical feature selection techniques. The experiment will be performed on datasets of different noise ratios to observe whether the noise will have an impact on feature selection or not, and finally, we will compare the provided results.

Statistical feature selection

Statistical feature selection methods involve evaluating the relationship between the features and the class using statistical methods. In this work, we will use a statistical feature selection algorithm called Correlation-based Feature Selection (CFS). In order to perform the statistical feature selection on our dataset we will use WEKA. The input for the WEKA is the result of the DROP3 noise cleaning with different ratios. Our objective is to observe the effect of noise on the feature selection on different datasets. That means we are doing the feature selection on four different datasets with noise ratios of 0%, 5%, 10%, and 20%. After executing the statistical features selection, the total number of features we got in the four experiments was 66. Of these features, only 35 are unique as shown in Table 6. Five features from those 35 appear in all four experiments, which are:

1. com.android.launcher.permission.INSTALL_SHORTCUT.
2. android.permission.MOUNT_UNMOUNT_FILESYSTEMS.
3. android.permission.READ_PHONE_STATE.
4. android.permission.REORDER_TASKS.
5. android.permission.SEND_SMS.

These features are shown in bold font in Table 6. The execution result for the feature selection on our dataset with a 20% noise percentage is shown in Fig. 12.



Attribute Evaluator: CfsSubsetEval -P 1 -E 1

Search Method: BestFirst -D 1 -N 5

Attribute Selection Mode: (●) Use full training set; ( ) Cross-validation, Folds 10, Seed 1

Result list (right-click for options)

```
20:50:43 - BestFirst + CfsSubsetEval
20:52:35 - BestFirst + CfsSubsetEval
20:53:20 - BestFirst + CfsSubsetEval
20:54:13 - BestFirst + CfsSubsetEval
```

Attribute selection output

```
=== Run information ===

Evaluator:    weka.attributeSelection.CfsSubsetEval -P 1 -E 1
Search:       weka.attributeSelection.BestFirst -D 1 -N 5
Relation:     trainingForDROP3_NOISE_20
Instances:    401
Attributes:   173
              [list of attributes omitted]
Evaluation mode:    evaluate on all training data



=== Attribute Selection on all input data ===

Search Method:
        Best first.
        Start set: no attributes
        Search direction: forward
        Stale search after 5 node expansions
        Total number of subsets evaluated: 2794
        Merit of best subset found:    0.637

Attribute Subset Evaluator (supervised, Class (numeric): 173 class):
        CFS Subset Evaluator
        Including locally predictive attributes

Selected attributes: 10,83,86,98,99,103,112,114,126,137,151,157 : 12
                     android.permission.ACCESS_WIFI_STATE
                     com.android.launcher.permission.INSTALL_SHORTCUT
                     android.permission.INTERNET
                     android.permission.MOUNT_FORMAT_FILESYSTEMS
                     android.permission.MOUNT_UNMOUNT_FILESYSTEMS
                     android.permission.PACKAGE_USAGE_STATS
                     android.permission.READ_LOGS
                     android.permission.READ_PHONE_STATE
                     android.permission.REORDER_TASKS
                     android.permission.SEND_SMS
                     android.permission.STATUS_BAR
                     android.permission.USE_FINGERPRINT
```

Fig. 11. Result of Feature selection using WEKA on DROP3 with 20% noise dataset

Table 5
WEKA Statistical unique values with duplicate in all noise ratio in bold.

SN	Permission	SN	Permission
1	android.permission.ACCOUNT_MANAGER	19	android.permission.WRITE_SECURE_SETTINGS
2	android.permission.BIND_DEVICE_ADMIN	20	android.permission.ACCESS_LOCATION_EXTRA_COMMANDS
3	android.permission.BLUETOOTH	21	android.permission.CALL_PRIVILEGED
4	android.permission.BODY_SENSORS	22	android.permission.DIAGNOSTIC
5	android.permission.DELETE_CACHE_FILES	23	android.permission.INSTALL_PACKAGES
6	android.permission.DELETE_PACKAGES	24	android.permission.MOUNT_FORMAT_FILESYSTEMS
7	**com.android.launcher.permission.INSTALL_SHORTCUT**	25	android.permission.RECEIVE_MMS
8	android.permission.INTERNET	26	android.permission.REQUEST_INSTALL_PACKAGES
9	android.permission.MANAGE_DOCUMENTS	27	com.android.alarm.permission.SET_ALARM
10	android.permission.MEDIA_CONTENT_CONTROL	28	android.permission.STATUS_BAR
11	**android.permission.MOUNT_UNMOUNT_FILESYSTEMS**	29	android.permission.USE_SIP
12	android.permission.PROCESS_OUTGOING_CALLS	30	android.permission.ACCESS_WIFI_STATE
13	android.permission.READ_LOGS	31	android.permission.BIND_APPWIDGET
14	**android.permission.READ_PHONE_STATE**	32	android.permission.REBOOT
15	**android.permission.REORDER_TASKS**	33	android.permission.REQUEST_IGNORE_BATTERY_OPTIMIZATIONS
16	**android.permission.SEND_SMS**	34	android.permission.PACKAGE_USAGE_STATS
17	android.permission.SET_PREFERRED_APPLICATIONS	35	android.permission.USE_FINGERPRINT
18	android.permission.WRITE_EXTERNAL_STORAGE		

5.2.1 *Evolutionary feature selection*

Evolutionary feature selection includes many naturally inspired techniques. Such techniques are ideal for selecting features since the rendering of a feature subset is simple and the assessment can also be performed easily. The ability of such heuristic algorithms to search vast areas efficiently benefits the selection of features. We will conduct our experiments on the same datasets we used on the statistical feature selection. The experiment will be on two evolutionary feature selection algorithms: Grey Wolf Optimizer (GWO) and Firefly Algorithm (FFA). The first experiment we conducted on our dataset using the EvoloPy tool was using the gray wolf optimizer (GWO). In the experiment, we use four datasets with different noise ratios. The total number of features in the dataset was 172. The unique values selected in all the datasets (every dataset with different noise ratio: 0, 5, 10, and 20%) was 159 features. Only eleven features were selected in the four datasets and are listed below:

1. android.permission.ACCEPT_HANDOVER.
2. android.permission.ACCESS_COARSE_LOCATION.
3. android.permission.ACCESS_NOTIFICATION_POLICY.
4. android.permission.ACCESS_WIFI_STATE.
5. android.permission.BLUETOOTH.
6. android.permission.BODY_SENSORS.
7. android.permission.CHANGE_WIFI_STATE.
8. android.permission.CLEAR_APP_CACHE.
9. android.permission.INTERNET.
10. android.permission.MANAGE_OWN_CALLS.
11. android.permission.SET_TIME.

Our second experiment in evolutionary feature selection is done using the Firefly algorithm (FFA). We used four datasets of varying noise levels in the experiment of using FFA for feature selection. In the dataset, there were 172 features. The unique features selected from the 4 datasets, while twelve features selected in every dataset are listed below:

1. android.permission.ACCESS_MEDIA_LOCATION.
2. android.permission.ACCESS_WIFI_STATE.
3. android.permission.BIND_CARRIER_MESSAGING_SERVICE.
4. android.permission.DELETE_PACKAGES.
5. android.permission.INTERNET.
6. android.permission.KILL_BACKGROUND_PROCESSES.
7. android.permission.MOUNT_UNMOUNT_FILESYSTEMS.
8. android.permission.READ_PHONE_STATE.
9. android.permission.READ_SYNC_STATS.
10. android.permission.REORDER_TASKS.
11. android.permission.UPDATE_DEVICE_STATS.
12. android.permission.WRITE_CALL_LOG.

5.2.2 *Comparison between statistical and evolutionary feature selection results*

In this section, we will compare the statistical and evolutionary feature selection in terms of the number of features selected by different algorithms and the most selected features by all algorithms, in addition to the relation between the top ten benign and malware applications and the feature selection. In order to start our comparison, we should know that the statistical feature selection algorithms work based on the correlation between a feature to feature or feature to class. On the other hand, the evolutionary feature selection algorithms work based on finding global optimization with randomization in feature selections and natural selection of parameters. The way each type of feature selection works explains the difference in numbers between the features selected. The number of features in the statistical algorithms is below 20 (out of 172 features) due to its work since it focuses on the correlation between the features and classes. While for the evolutionary, the algorithms always try to optimize their selection. This can be shown in the number of selected features in these algorithms, which vary between 79 and 93 features, as shown in Table 7.

Table 7
Number of features selected by each algorithm

Noise ratio/ Algorithm	GWO	FFA	Statistical
0%	79	88	19
5%	87	90	18
10%	93	87	16
20%	84	88	12

In this experiment, we found that some of the features are selected by the optimizer or FS algorithm despite the noise ratio injected into the dataset. As shown in Table 8.

We can observe from the table permissions like READ_PHONE_STATE, MOUNT_UNMOUNT_FILESYSTEMS, INTERNET, and ACCESS_WIFI_STATE is selected in two FS algorithm and at every noise ratio level. This is because these permissions have an effect on the accuracy of the classification, and it has an impact on whether the classifier will classify this application as benign or malware. Moreover, the following two permissions were never selected by the FS algorithm:

- android.permission.BIND_TELECOM_CONNECTION_SERVICE
- android.permission.GLOBAL_SEARCH

That is because the first permission was never used by any application, while the second permission appears in seven applications out of more than 8000 applications.

FS algorithms often select the features or permission that have an impact on the classification. To clarify that, we will show the top ten benign and malware permissions and how many times the FS algorithms selected them in Table 9 and Table 10. We can notice that the FS algorithms detect the importance of these features with the exitance of noise. From Table 9, we can see the popular permissions used by benign applications. We notice from the table that the permissions WAKE_LOCK, VIBRATE, and READ_PHONE_STATE are used widely among benign applications. Moreover, when we compare Table 9 and Table 10, we can observe that, in general, benign applications tend to request fewer permissions than malware applications. On the other hand, we can observe from Table 10: WRITE_EXTERNAL_STORAGE, SYSTEM_ALERT_WINDOW, and CHANGE_WIFI_WINDOW are used widely among malware apps since attackers used these permissions to achieve their malicious goals.

Table 8
Permission that appears in every evolutionary and statistical optimizers and all noise ratios

SN	Permission	Appear in
1	com.android.launcher.permission.INSTALL_SHORTCUT	Statistical
2	android.permission.MOUNT_UNMOUNT_FILESYSTEMS	Statistical & FFA
3	android.permission.READ_PHONE_STATE	Statistical & FFA
4	android.permission.REORDER_TASKS	Statistical & FFA
5	android.permission.SEND_SMS	Statistical
6	android.permission.ACCEPT_HANDOVER	GWO
7	android.permission.ACCESS_COARSE_LOCATION	GWO
8	android.permission.ACCESS_NOTIFICATION_POLICY	GWO
9	android.permission.ACCESS_WIFI_STATE	GWO & FFA
10	android.permission.BLUETOOTH	GWO
11	android.permission.BODY_SENSORS	GWO
12	android.permission.CHANGE_WIFI_STATE	GWO
13	android.permission.CLEAR_APP_CACHE	GWO
14	android.permission.INTERNET	GWO & FFA
15	android.permission.MANAGE_OWN_CALLS	GWO
16	android.permission.SET_TIME	GWO
17	android.permission.ACCESS_MEDIA_LOCATION	FFA
18	android.permission.BIND_CARRIER_MESSAGING_SERVICE	FFA
19	android.permission.DELETE_PACKAGES	FFA
20	android.permission.KILL_BACKGROUND_PROCESSES	FFA
21	android.permission.READ_SYNC_STATS	FFA
22	android.permission.UPDATE_DEVICE_STATS	FFA
23	android.permission.WRITE_CALL_LOG	FFA

Table 9
Top 10 permissions selected by benign apps and number selections by the FS algorithms

SN	Permission	Number of used apps	Number of times selected by optimizer (max = 12)
1	android.permission.INTERNET	3609	12
2	android.permission.ACCESS_NETWORK_STATE	3365	4
3	android.permission.WRITE_EXTERNAL_STORAGE	2553	5
4	android.permission.WAKE_LOCK	1825	4
5	android.permission.VIBRATE	1588	5
6	android.permission.READ_PHONE_STATE	1424	12
7	android.permission.RECEIVE_BOOT_COMPLETED	1391	4
8	android.permission.ACCESS_WIFI_STATE	1384	12
9	android.permission.READ_EXTERNAL_STORAGE	963	5
10	android.permission.ACCESS_FINE_LOCATION	874	4

Table 6
Top 10 permissions selected by benign Apps and number selections by the FS algorithms

SN	Permission	Number of used apps	Number of times selected by optimizer (max = 12)
1	android.permission.INTERNET	3827	12
2	android.permission.ACCESS_NETWORK_STATE	3797	4
3	android.permission.WRITE_EXTERNAL_STORAGE	3733	5
4	android.permission.READ_PHONE_STATE	3719	12
5	android.permission.ACCESS_WIFI_STATE	3582	12
6	android.permission.WAKE_LOCK	3033	4
7	android.permission.GET_TASKS	2998	3
8	android.permission.VIBRATE	2692	5
9	android.permission.SYSTEM_ALERT_WINDOW	2460	2
10	android.permission.CHANGE_WIFI_STATE	2451	12

In Fig. 13, we combine the top ten benign and malware features and how many times they were selected. We can notice that all FS algorithms selected permissions like permissions: INTERNET, READ_PHONE_STATE, ACCESS_WIFI_STATE, and CHANGE_WIFI_STATE. We can also observe that most of the features were selected by at least two algorithms. This clarifies that FS algorithms focus on the important features which usually have an impact on the classifications.

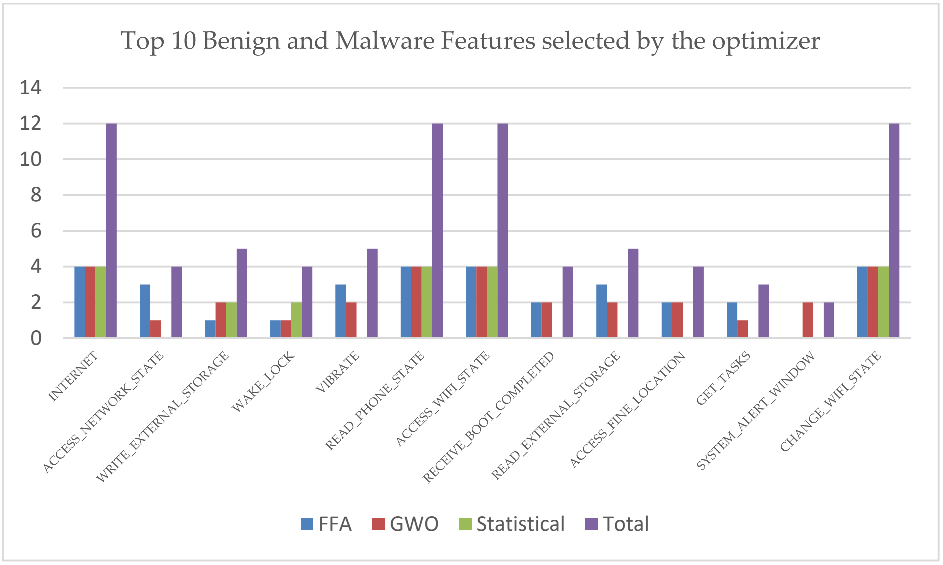


Fig. 12. Top 10 Benign and Malware Features selected by the optimizer

5.3 *Classification*

This section will present the result of our experiment on the classification accuracy of the ML-based Android malware detection dataset. The algorithms we will use are k-nearest-neighbor (IBk), multilayer perceptron (MLP), and random forest (RF). The experiments were done based on the result of the feature selection in the previous section; we ran the experiment on the not optimized dataset (all features included). The result of these experiments is shown in Table 11. The experiments using no optimization achieve the best result in all the experiments. Nevertheless, on the other side, it takes a long time. As we can see in Table 11, at noise level 0, the time to build a model using the MLP classifier takes about 35 seconds, but the same classifier takes less than nine seconds in GWO and FFA. ML algorithms accuracy using no optimizer varies between 84.16% - 88.88%, as we can see in Fig. 14. The best ML algorithm accuracy result was by the RF classifier.

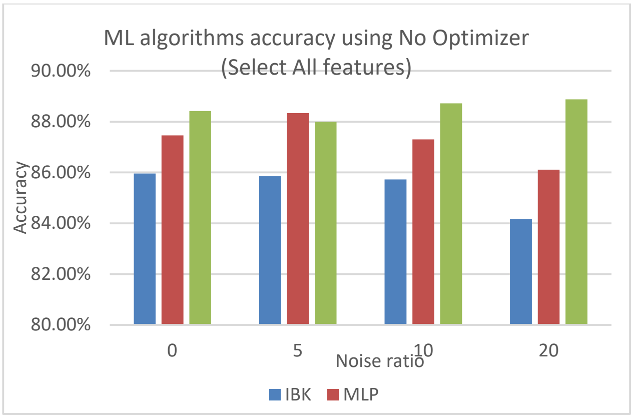


Fig. 13. ML algorithms accuracy using No Optimizer

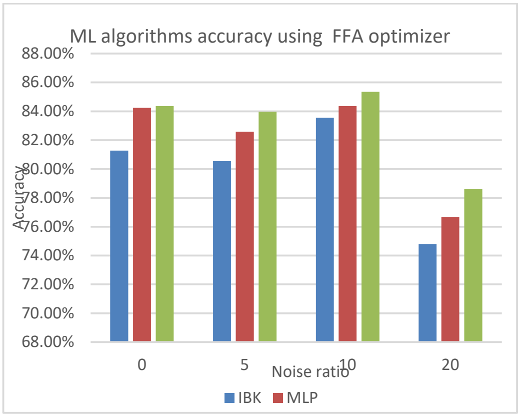


Fig. 14. ML algorithms accuracy using FFA optimizer

The worst result among all FS algorithms occurs using the FFA optimizer, where the accuracy varies between 74.80% - 85.35%, as shown in Figure 15. The best ML algorithm accuracy result was by the RF classifier. In addition, the ML algorithms' accuracy using statistical feature selection varies between 80.82% - 84.24%, as shown in Figure 16. There was no best ML algorithm in this experiment since the best ML algorithm changes with different noise levels. The ML algorithms accuracy using GWO optimizer achieve the best results in our experiments. The accuracy varies between 81.01% - 87.34%, as shown in Figure 17. The best ML algorithm accuracy result was by the RF classifier.

Table 71
The Results of Running ML Algorithms on Android malware dataset

Noise ratio	Optimizer	ML Algorithm	Accuracy	Recall	Precision	TP	TN	FP	FN	Time/s
0%	**No Optimizer**	IBK	85.96%	0.860	0.860	1148	156	1094	210	0
		MLP	87.46%	0.875	0.875	1168	136	1113	191	35.95
		RF	88.42%	0.884	0.884	1191	113	1115	189	0.54
	Firefly Algorithm (FFA)	IBK	81.28%	0.813	0.816	1122	182	998	306	0
		MLP	84.24%	0.842	0.843	1129	175	1068	236	8.54
		RF	84.35%	0.844	0.846	1152	152	1048	256	0.29
	Grey Wolf Optimizer (GWO)	IBK	81.71%	0.817	0.819	1119	185	1012	292	0
		MLP	83.01%	0.830	0.830	1081	223	1084	220	7.49
		RF	**85.16%**	0.852	0.852	1128	176	1093	211	0.22
	Statistical	IBK	82.66%	0.827	0.835	1181	123	975	329	0
		MLP	81.55%	0.816	0.825	1173	131	954	350	0.57
		RF	82.63%	0.826	0.836	1189	115	966	338	0.1
5%	**No Optimizer**	IBK	85.85%	0.859	0.859	1144	160	1095	209	0
		MLP	88.34%	0.883	0.884	1172	132	1132	172	31.53
		RF	87.99%	0.880	0.882	1196	108	1099	205	0.4
	Firefly Algorithm (FFA)	IBK	80.55%	0.806	0.812	1143	161	958	346	0
		MLP	82.59%	0.826	0.828	1132	172	1022	282	8.21
		RF	83.97%	0.840	0.843	1157	147	1033	271	0.41
	Grey Wolf Optimizer (GWO)	IBK	82.66%	0.827	0.827	1109	195	1047	257	0
		MLP	81.44%	0.814	0.814	1058	246	1066	238	7.41
		RF	**85.19%**	0.852	0.852	1105	199	1117	187	0.28
	Statistical	IBK	84.12%	0.841	0.851	1208	96	986	318	0
		MLP	84.24%	0.842	0.852	1207	97	990	314	0.51
		RF	83.39%	0.834	0.845	1206	98	969	335	0.08
10%	**No Optimizer**	IBK	85.73%	0.857	0.859	1158	146	1078	226	0
		MLP	87.30%	0.873	0.873	1140	164	1137	167	37.88
		RF	88.72%	0.887	0.888	1184	120	1130	174	0.47
	Firefly Algorithm (FFA)	IBK	83.55%	0.836	0.838	1143	161	1036	268	0
		MLP	84.35%	0.844	0.846	1154	150	1046	258	9.02
		RF	85.35%	0.854	0.854	1145	159	1081	223	0.4
	Grey Wolf Optimizer (GWO)	IBK	85.12%	0.851	0.851	1122	182	1098	206	0
		MLP	84.89%	0.849	0.849	1131	173	1083	221	10.75
		RF	**87.34%**	0.873	0.873	1136	168	1142	162	0.29
	Statistical	IBK	81.51%	0.815	0.815	1082	222	1044	260	0
		MLP	80.82%	0.808	0.809	1081	223	1027	277	0.51
		RF	81.59%	0.816	0.816	1084	220	1044	260	0.1
20%	**No Optimizer**	IBK	84.16%	0.842	0.846	1173	131	1022	282	0
		MLP	86.11%	0.861	0.862	1158	146	1088	216	43.12
		RF	88.88%	0.889	0.889	1179	125	1139	165	0.45
	Firefly Algorithm (FFA)	IBK	74.80%	0.748	0.748	961	343	990	314	0
		MLP	76.68%	0.767	0.767	970	334	1030	274	10.63
		RF	78.60%	0.786	0.788	971	333	1079	225	0.3
	Grey Wolf Optimizer (GWO)	IBK	81.01%	0.810	0.820	1173	131	940	364	0
		MLP	82.86%	0.829	0.830	1125	179	1036	268	9.87
		RF	**85.92%**	0.859	0.860	1139	165	1102	202	0.37
	Statistical	IBK	83.81%	0.838	0.848	1202	102	984	320	0
		MLP	82.09%	0.821	0.831	1183	121	958	346	0.36
		RF	82.82%	0.828	0.836	1180	124	980	324	0.07

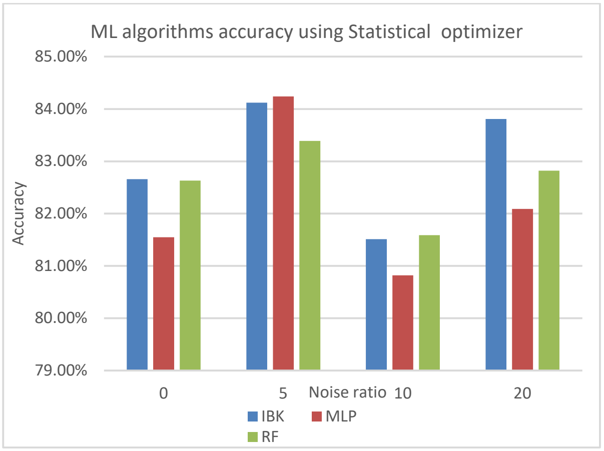


Fig. 15. ML algorithms accuracy using Statistical feature selection

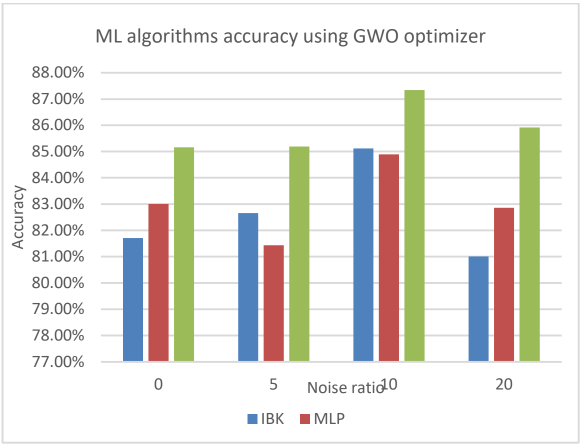


Fig. 16. ML algorithms accuracy using GWO optimizer

Since the previous experiment shows that the best accuracy results were achieved by using GWO, we want to compare the classifier accuracy results in GWO and without GWO. Our aim from the comparison is to find the effect of the feature selection on the classification accuracy. Fig. 18 shows the result of the comparison. We can notice that the difference between using GWO and no optimization accuracy varies between 1.38% to 3.26% for the advantage of no optimization. In terms of time, Table 13 shows the amount of time taken to build the model. We can notice that the time difference is up to 40% for the advantage of GWO. The previous results show that using feature selection will have a small decrease in accuracy, but there will be a significant increase in performance.

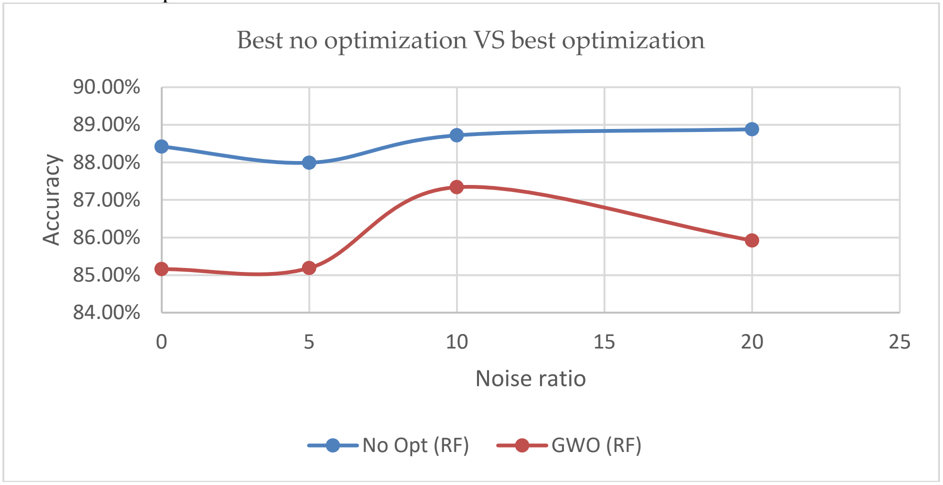


Fig. 17. Compare the best classifier accuracy results between GWO and no optimization

6. Conclusions and Future Work

The results from the empirical experiments show that the proposed model can detect Android malware with an accuracy that reaches 87%, despite the noise in the dataset. We also find that the best noise filtering algorithm is DROP3, the best feature selection algorithm is GWF, and the best classification results are achieved using the RF algorithm. This work can be extended in many ways. One of the recommended tasks to be done in the future is to apply higher noise ratios and run more classifiers and optimizers. We also believe that adding the Android intents and API calls can increase the accuracy and help in detecting more malware. Building a web-based framework that takes an Android application as an .apk file and determines whether it is benign or malware can be included in the future.

References

Abdullah, Z., Muhadi, F. W., Saudi, M. M., Hamid, I. R. A., & Foozy, C. F. M. (2020). Android ransomware detection based on dynamic obtained features. In Recent Advances on Soft Computing and Data Mining: Proceedings of the Fourth International Conference on Soft Computing and Data Mining (SCDM 2020), Melaka, Malaysia, January 22– 23, 2020 (pp. 121-129). Springer International Publishing.

Ahmad, I., Basheri, M., Iqbal, M. J., & Rahim, A. (2018). Performance comparison of support vector machine, random forest, and extreme learning machine for intrusion detection. *IEEE access, 6*, 33789-33795.

Al Khayer, A., Almomani, I., & Elkawlak, K. (2020, November). ASAF: Android static analysis framework. In 2020 First International Conference of Smart Systems and Emerging Technologies (SMARTTECH) (pp. 197-202). IEEE.

Alenezi, M., & Almomani, I. (2017, October). Abusing android permissions: A security perspective*. In 2017 IEEE Jordan Conference on Applied Electrical Engineering and Computing Technologies (AEECT)* (pp. 1-6). IEEE.

Alenezi, M., & Almomani, I. (2018). Empirical analysis of static code metrics for predicting risk scores in android applications. *In 5th International Symposium on Data Mining Applications (pp. 84-94). Springer International Publishing.*

Al-Gethami, K. M., Al-Akhras, M. T., & Alawairdhi, M. (2021). Empirical evaluation of noise influence on supervised machine learning algorithms using intrusion detection datasets. *Security and Communication Networks, 2021*, 1-28.

Almomani, I., & Alenezi, M. (2019). *Android application security scanning process. In Telecommunication Systems-Principles and Applications of Wireless-Optical Technologies.* London, UK.: IntechOpen.

Almomani, I., & Khayer, A. (2019, April). Android applications scanning: The guide. *In 2019 International Conference on Computer and Information Sciences (ICCIS) (pp. 1-5). IEEE.*

Alqatawna, J. F., Ala'M, A. Z., Hassonah, M. A., & Faris, H. (2021). Android botnet detection using machine learning models based on a comprehensive static analysis approach*. Journal of Information Security and Applications, 58*, 102735.

Alsoghyer, S., & Almomani, I. (2019). Ransomware detection system for Android applications. *Electronics, 8*(8), 868.

Alsoghyer, S., & Almomani, I. (2020, March). On the effectiveness of application permissions for Android ransomware detection. *In 2020 6th conference on data science and machine learning applications (CDMA) (pp. 94-99). IEEE.*

Amro, A., Al-Akhras, M., Hindi, K. E., Habib, M., & Shawar, B. A. (2021). Instance reduction for avoiding overfitting in decision trees. *Journal of Intelligent Systems, 30*(1), 438-459.

AndroidDeveloper. <permission>. 2020. https://developer.android.com/guide/topics/manifest/permission-element (accessed 2 20, 2021).

AndroidDeveloper. Manifest.permission. 2020. https://developer.android.com/reference/android/Manifest.permission (accessed 3 1, 2021).

AndroidDeveloper. Permissions on Android. 2020. https://developer.android.com/guide/topics/permissions/overview (accessed 2 20, 2021).

AndroidDeveloper. SDK Platform release notes. 2020. https://developer.android.com/studio/releases/platforms (accessed 2 19, 2021).

AndroZoo. Home. 2020. https://androzoo.uni.lu/ (accessed 2 21, 2021).

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., & Siemens, C. E. R. T. (2014, February). Drebin: Effective and explainable detection of android malware in your pocket. In Ndss (Vol. 14, pp. 23-26).

Arslan, R. S., Doğru, İ. A., & Barişçi, N. (2019). Permission-based malware detection system for android using machine learning techniques*. International journal of software engineering and knowledge engineering, 29*(01), 43-61.

Aswini, A. M., & Vinod, P. (2014, February). Droid permission miner: Mining prominent permissions for Android malware analysis. *In The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014)* (pp. 81-86). IEEE.

Borah, P., Ahmed, H. A., & Bhattacharyya, D. K. (2014). A statistical feature selection technique. *Network Modeling Analysis in Health Informatics and Bioinformatics, 3,* 1-13.

Brownlee, J. (2018). Better deep learning: train faster, reduce overfitting, and make better predictions. Machine Learning Mastery.

Ceci,L. (accessed 02 24, 2021). StatistaResearchDepartment. Number of apps available in leading app stores 2020. 2 4, 2021. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/

Chawla, N. V., Bowyer, K. W., Hall, L. O., & Kegelmeyer, W. P. (2002). SMOTE: synthetic minority over-sampling technique. *Journal of artificial intelligence research, 16*, 321-357.

Choudhary, M., & Kishore, B. (2018, January). Haamd: Hybrid analysis for android malware detection. *In 2018 International Conference on Computer Communication and Informatics (ICCCI) (pp. 1-4). IEEE.*

De La Iglesia, B. (2013). Evolutionary computation for feature selection in classification problems. *Wiley Interdisciplinary Reviews: Data Mining and Knowledge Discovery, 3*(6), 381-407.

Doğru, İ. A., & KİRAZ, Ö. (2018). Web-based android malicious software detection and classification system. *Applied Sciences, 8*(9), 1622.

Doğru, İ. A., & Önder, M. (2020). AppPerm analyzer: malware detection system based on android permissions and permission groups. *International Journal of Software Engineering and Knowledge Engineering, 30*(03), 427-450.

Dua, R., Ghotra, M. S., & Pentreath, N. (2017). *Machine Learning with Spark*. Packt Publishing Ltd.

Géron, A. (2022). Hands-on machine learning with Scikit-Learn, Keras, and TensorFlow. " O'Reilly Media, Inc.".

Kanwal, N., & Bostanci, E. (2016). Comparative Study of Instance Based Learning and Back Propagation for Classification Problems. arXiv preprint arXiv:1604.05429.

Kerns, T. (accessed 2 19, 2021). There are now more than 2.5 billion active Android devices. 5 7, 2019. https://www.androidpolice.com/2019/05/07/there-are-now-more-than-2-5-billion-active-android-devices/

Khurma, R. A., Aljarah, I., & Sharieh, A. (2020, July). Rank based moth flame optimisation for feature selection in the medical application*. In 2020 IEEE congress on evolutionary computation (CEC) (pp. 1-8). IEEE.*

Kiss, N., Lalande, J. F., Leslous, M., & Tong, V. V. T. (2016). Kharon dataset: Android malware under a microscope. In The LASER Workshop: Learning from Authoritative Security Experiment Results (LASER 2016) (pp. 1-12).

Klein, K. (accessed 02 19, 2021). Industry Leaders Announce Open Platform for Mobile Devices. 11 05, 2007. http://www.openhandsetalliance.com/press_110507.html.

Krajci, I., Cummings, D., Krajci, I., & Cummings, D. (2013). History and Evolution of the Android OS. Android on x86: An Introduction to Optimizing for Intel® Architecture, 1-8.

Kumar, C. A., Sooraj, M. P., & Ramakrishnan, S. (2017). A comparative performance evaluation of supervised feature selection algorithms on microarray datasets. *Procedia computer science, 115*, 209-217.

Li, Y., Xiong, Z., Zhang, T., Zhang, Q., Fan, M., & Xue, L. (2022). Ensemble Framework Combining Family Information for Android Malware Detection. The Computer Journal, bxac114.

McAfee. McAfee Labs Threats Report - November 2020. McAfee, 2020.

Mirjalili, S., Mirjalili, S. M., & Lewis, A. (2014). Grey wolf optimizer. *Advances in engineering software, 69*, 46-61.

O'Dea, S. (2020). Global market share held by the leading smartphone operating systems in sales to end users from 1st quarter 2009 to 2nd quarter 2018.

Qamar, A., Karim, A., & Chang, V. (2019). Mobile malware attacks: Review, taxonomy & future directions. *Future Generation Computer Systems, 97*, 887-909.

Russell, S. J. Norvig, P. (2020). Artificial intelligence: a modern approach. Pearson Education, Inc.

Susan, S., & Kumar, A. (2019). SSOMaj-SMOTE-SSOMin: Three-step intelligent pruning of majority and minority samples for learning from imbalanced datasets. *Applied Soft Computing, 78*, 141-149.

Taylor, P. (accessed 2 24, 2021). https://www.statista.com/statistics/266136/global-market-share-held-by-smartphone-operating-systems/

Thon, J. "Static Analysis of Android Malware of 2017 ." Static Analysis of Android Malware of 2017 . Kaggle, 7 06, 2018.

Utku, A., & Dogru, İ. B. R. A. H. İ. M. (2017). Permission based detection system for android malware. *Journal of the Faculty of Engineering and Architecture of Gazi University, 32*(4).

Wang, Z. (2018). Deep learning-based intrusion detection with adversaries. *IEEE Access, 6,* 38367-38384.

Wei, F., Li, Y., Roy, S., Ou, X., & Zhou, W. (2017). Deep ground truth analysis of current android malware. *In Detection of Intrusions and Malware, and Vulnerability Assessment: 14th International Conference, DIMVA 2017, Bonn, Germany, July 6-7, 2017, Proceedings 14 (pp. 252-276).* Springer International Publishing.

Wilson, D. R., & Martinez, T. R. (1997). Improved heterogeneous distance functions. *Journal of artificial intelligence research, 6*, 1-34.

Wilson, D. R., & Martinez, T. R. (2000). Reduction techniques for instance-based learning algorithms. *Machine learning, 38*, 257-286.

Witten, I. H., & Frank, E. (2002). Data mining: practical machine learning tools and techniques with Java implementations. *Acm Sigmod Record, 31*(1), 76-77.

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

Xin, Y., Kong, L., Liu, Z., Chen, Y., Li, Y., Zhu, H., ... & Wang, C. (2018). Machine learning and deep learning methods for cybersecurity. *Ieee access, 6*, 35365-35381.

Yang, X. S. (2010). Firefly algorithm, stochastic test functions and design optimisation. *International journal of bio-inspired computation, 2*(2), 78-84.