Decision Science Letters 13 (2024) 29–44

Contents lists available at GrowingScience

Decision Science Letters

homepage: www.GrowingScience.com/dsl

A machine learning technique for Android malicious attacks detection based on API calls

Mousa AL-Akhras[a], Saud Alghamdi[b], Hani Omar[c] and Hazzaa Alshareef[b*]

[a]*King Abdullah II School of Information Technology, The University of Jordan, Amman 11942, Jordan*
[b]*College of Computing & Informatics, Saudi Electronic University, Riyadh 11673, Saudi Arabia*
[c]*Faculty of Information Technology, Zarqa University, Zarqa 13110, Jordan*

C H R O N I C L E

Article history:
Received: August 12, 2023
Received in revised format: November 1 , 2023
Accepted: December 7, 2023
Available online: December 7, 2023

Keywords:
Attack Detection
API Calls
Machine Learning
Malware
Android

A B S T R A C T

Android malware is widespread and it is considered as one of the most threatening attacks recently. The threat is targeting to damage access data or information or leaking them; in general, malicious software consists of viruses, worms, and other malware. Current malware attempts to prevent being detected by any software or anti-virus. This paper describes recent Android malware detection static and interactive approaches as well as several open-source malware datasets. The paper also examines the most current state-of-the-art Android malware identification techniques including identifying by comparative evaluation the gaps between these techniques. As a result, an API-based dynamic malware detection framework is proposed for Android to provide a dynamic paradigm for malware detection. The proposed framework was closely inspected and checked for reliability where meaningful API packages and methods were discovered.

1. Introduction

According to the latest study, 85% of smartphones in the world use Android OS, which has been developed by Google, and the Play Store is rising 3 times as fast as the Apple App Store. This store is not the only source of Android applications; many other unofficial third-party application developers exist (Abuthawabeh & Mahmoud., 2020). Thus, it increases the possibility of being targeted by malicious software. This growth also leads to the risk of privacy issues, such as accessing contact details, GPS locations, and personal data. Recently, smartphone industry exploitation has uncovered malware and repackaged software where malicious components were embedded (T. Sharma & Rattan, 2021).

1.1 Importance of Android Systems

The open-source Android OS operates on the Linux kernel. Applications written for Android were built using the Java programming language. Google offers an SDK, which allows these Java codes to power cell phones, laptops, and other computers. The Android operating system became popular with developers because of its configurability. Building an application in one platform and distributing it across several platforms concurrently does not necessitate thinking about platform updates. Like any other OS, Android is also susceptible to malware attacks (Alam & Demir., 2023). Android consumers have the power to authorize or reject applications that have been installed (Sarkar et al., 2019). There are several ways to obfuscate code in an Android world, and many off-the-shelf code obfuscators are available for that reason (Nellaivadivelu et al., 2020).

1.2 Risk of Android malware attacks

Since the functionality of smartphones has expanded, malware risks will rise. use cache policies to increase the launch time (Alagarsamy et al., 2022). Malicious Android malware systems have a range of security methods at clearance. The latest

* Corresponding author.
E-mail address: h.alshareef@seu.edu.sa (H. Alshareef)

doi: 10.5267/dsl.2023.12.004

Android malware detection strategies are contrasted in this paper to propose more robust and efficient detection methods. Malware software can be defined as any malicious software program code aims to obtain access to sensitive information stored on the device (Agrawal & Trivedi, 2019). Approximately, 350,000 new malware and potentially unwanted applications are identified per day by the AV-TEST Institute.

1.3 Android API Calls

An Application Programming Interface (API) represents a software programming interface that enables connectivity between various software components such as between the user program and the operating system services. APIs allow access to the data and features that reside inside Android devices. Looking at component calls inside the executable file allows app functionality to be explored. In certain instances, though, adversaries obfuscate the API calls with encryption, reflection, or complex code-loading techniques (Almomani & Alenezi, 2019).

1.4 Contribution

This paper aims to systematically explore the various variables that may affect the efficacy of malware API calls using the ML engine with multiple newly collected datasets for Android malware. This will pave the way towards focusing on processes that will positively affect and mitigate possible negative impacts precisely. This paper also investigates the effect and efficacy of noise filtering techniques on ML techniques' accuracy. The paper also examined classification accuracy based on Android API calls using ensembles of classifiers.

The rest of this paper is organized as follows: Section 2 introduces preliminary concepts. The literature is reviewed in Section 3. Methods and Materials are explained in Section 4. Experimental results are discussed in Section 5. Conclusions and avenues for future work are explored in Section 6.

2. Preliminaries

2.1 Android API's

An application programming interface (API) allows software components to interact with each other in various ways. In many situations, security attackers' APIs are encapsulated with encryption, reflection, or dynamic code. Since this is the case, the analysis of the app has been more complex (Alsoghyer & Almomani, 2019). As seen in Figure 1, API calls in Android are performed in three key steps. The first step is for a program in the library to access the API. Then, the library uses a private interface to operate. Lastly, the private interface issues a Remote Procedure Call (RPC) to the device. The code sets up an API to be run in the process (Almomani & Khayer, 2020).

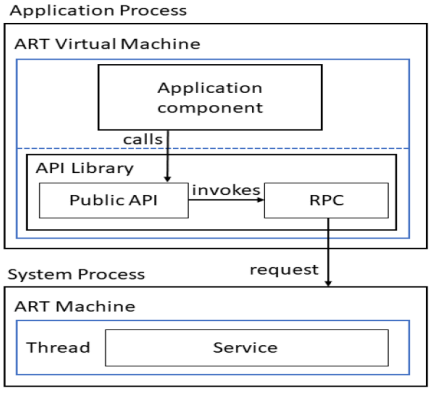

Fig. 1. Handling API call by android (Almomani & Khayer, 2020)

2.2 Android API's Level

A collection of APIs that programs can use to interface with the device. When it comes to the Android platform, API Level is an integer used to identify the corresponding application revision. It has classes, properties, methods, and events, among other things. The system API is defined as set of the following:

- Packages and classes;
- XML elements and attributes for declaring a manifest file;
- XML elements and attributes for declaring and accessing resources;
- Intents;
- Permissions that applications can request, as well as permission enforcements included in the system.

Changes to the API are intended to keep it consistent with previous releases. Changes in the API are, for the most part, additive and result in the introduction of new or replacement features. For each API enhancement, current APIs are updated

but not replaced to allow old ones to continue to be used (Zhang & Cai, 2019). Most updates to the API are designed to improve robustness and device security. The Android framework introduction started with API Level 1 and later supported API Levels up to API Level 7. In Table 1 every version of the API level is specified by the supported Android platform.

Table 1
Android API's Versions (Android, 2020)

API Level	Version Code	Platform Version
1	BASE	Android 1.0
2	BASE_1_1	Android 1.1
3	CUPCAKE	Android 1.5
4	DONUT	Android 1.6
5	ÉCLAIR	Android 2.0
6	ECLAIR_0_1	Android 2.0.1
7	ECLAIR_MR1	Android 2.1.x
8	FROYO	Android 2.2.x
9	GINGERBREAD	Android 2.3, 2.3.1, 2.3.2
10	GINGERBREAD_MR1	Android 2.3.3, 2.3.4
11	HONEYCOMB	Android 3.0.x
12	HONEYCOMB_MR1	Android 3.1.x
13	HONEYCOMB_MR2	Android 3.2
14	ICE_CREAM_SANDWICH	Android 4.0, 4.0.1, 4.0.2
15	ICE_CREAM_SANDWICH_MR1	Android 4.0.3, 4.0.4
16	JELLY_BEAN	Android 4.1, 4.1.1
17	JELLY_BEAN_MR1	Android 4.2, 4.2.2
18	JELLY_BEAN_MR2	Android 4.3
19	KITKAT	Android 4.4
20	KITKAT_WATCH	Android 4.4W
21	LOLLIPOP	Android 5.0
22	LOLLIPOP_MR1	Android 5.1
23	M	Android 6.0
24	N	Android 7.0
25	N_MR1	Android 7.1, 7.1.1
26	O	Android 8.0
27	O_MR1	Android 8.1
28	P	Android 9
29	Q	Android 10
30	R	Android 11

2.3 Machine Learning

Machine Learning (ML) is a subclass of Artificial Intelligence (AI). ML techniques perform some learning tasks where they gain experience in executing a specific task (Ray, 2019). ML is a subclass of Artificial Intelligence (AI).

2.4 Supervised Learning

In supervised learning, the input dataset is paired with the target output. It involves algorithms that accept a dataset and feedback so that the program can identify the class of each piece of data. Many techniques are used for classification, such as regressions, classification trees, support vector machines, random forests, and artificial neural networks (ANNs). The study of ANNs is a significant branch of AI (Louridas & Ebert, 2016). Supervised learning can be categorized into two categories:

- Regression, forecasting a continuous result.
- Classification, forecasting discrete values.

2.5 Unsupervised Learning

In unsupervised learning the solution must discover natural grouping. It can be categorized into two categories (Louridas & Ebert, 2016):

- Dimension reduction, which is ideal for reducing the details as much as possible while retaining the facts.
- Clustering, which involves grouping data into groups that meet those requirements.

2.6 Semi-supervised Learning

It is a computational approach, which helps in the automatic labeling of huge unlabeled data based on smaller and similar labeled data. "Semi-supervised learning" is taking advantage of labeled and unlabeled data. Labeled data are hard to be

found, so semi-supervised learning is promising for grasping unlabeled human category learning (van Engelen & Hoos, 2020).

2.7 Reinforcement Learning (RL)

Sequential decision-making research in computer learning is known as “reinforcement learning”. The principal characteristic of RL is that it relies on iterative discovery and only information learning occurs in an environment (François-Lavet et al., 2018).

3. Literature review

Smartphones have become an essential part of the lives of an individual in present-day society. Smartphones are the most used devices for accessing Internet services in the present day (Faris et al., 2020). With the increase in the popularity and frequency of the use of smartphones, they become targets for various attacks, malware, and malicious activities over the Internet (Dubey et al., 2023). The attacks on smartphones have excessively increased over the past few years. The number of ransomware attacks on smartphones is rapidly increasing as compared to other types of malware attacks (Sharma et al., 2021). Ransomware attacks on smartphones aim to control smartphone system, data, and demand payment before they release it.

API has become a crucial part of system development in the present-day technological world. They function as rules that facilitate communications between programs or hardware devices. Current systems have changed this approach because of the demand for data regardless of geographical position. In these cases, there is a need for an interface for sending and receiving data to or from servers (Alsoghyer & Almomani, 2019).

Alsoghyer and Almomani (2019) reported that API-based ransomware detection systems (API-RDS) achieved 97% in detecting ransomware attacks when compared to other security systems that were currently being used. The current security systems used in Android devices attempt to detect the ransomware once they have entered the systems but often fails because ransomware is difficult to detect and can capture the device instantly. The API-based ransomware detection system, on the other hand, can detect the malware at the period when the user requests information from a given data.

API-based ransomware detection systems play a crucial role in enabling a developer to detect the source code vulnerability. (Alenezi & Almomani, 2018) noted that developers used the analysis approach in detecting malware and preventing possible attacks on systems. The research also examined static software metrics and the ability of these metrics to predict security vulnerabilities. They found that the static analysis approach is very effective in detecting vulnerabilities in a source code. For this reason, software developers can consider API as a tool for providing security against ransomware because it provides access to the source codes.

The application of static analysis framework as an approach that could be used in API-based ransomware protection systems applies to Android devices. Although there is limited research on the application of the static analysis framework, the available literature provides critical information that demonstrates its usefulness. (Khayer et al.,2020) presented an Android Static Analysis Framework (ASAF) which is a static analysis framework designed for an Android system that provides a possible solution to the problem of ransomware attacks. For this reason, they suggested the application of ASAF in Android devices to perform various tasks including protection against malware. Furthermore, (Khayer et al., 2020) recommended the application of Android Static Parse (ASParse) in the future as an approach to dealing with Android malware, particularly ransomware. ASParse tool could also prove to be a crucial element in API because it enables the parsing of files when they are requested by the users. This is because parsed files are very easy to analyze and attempt to establish the presence of malware. Another important aspect of Android API in the development of ransomware protection systems is that they consist of crucial information about the behavior of the malware. Data on the ransomware API call and permissions can be extracted and its patterns analyzed to determine the behavior of the ransomware. Research by Faris et al. (2020) explored the effectiveness of the Salp Swarm Algorithm (SSA) and Kernel Extreme Learning Machine (KELM) in monitoring an information system. These two systems are proposed as an information security system that is designed through the static analysis framework. The application of API in the development of a security system to protect Android devices from ransomware attacks can also involve the use of Machine learning to collect and statistically analyze data by a computer system to create patterns and make decisions without the interference of human beings. (Jung et al., 2018) explore the effectiveness of an Android ML API and establish that it can achieve a high accuracy of 99.98%. However, the researchers did not apply any feature selections on the datasets and noise insertion and filtering.

Another research by Jung et al. (2019), explored the need for developing an accurate malware detection system for Android devices. This research builds on the effectiveness of Android API in providing security for Android devices against ransomware. The authors concluded that the AI-based cybersecurity systems are always considered to be the future of information security given their accuracy and efficiency they have in monitoring information systems.

In conclusion, an API-based malware protection system through the use of a static analysis framework can prove effective in the prevention of malware attacks in Android systems. API plays a fundamental role as an interface between the end-user and software of a hardware device. Android API, collects information about the actions of various programs making them a key data source for understanding the behavior of ransomware attacks.

4. Methods and materials

4.1 Overall Methodology

This paper intends to explore methods for malware detection for malicious API calls based on ML and the robustness of ML algorithms in the presence and absence of injected noise. The main aim is to detect Android malware using ML algorithms. Fig. 2 illustrates the general approach used in this paper through the following four experiments, whereas steps presented in the algorithm 1, will be conducted in this research.

- The first experiment involves retrieving a dataset from the Kaggle repository as JSON files and converting the dataset to a CSV file after preprocessing. Subsequently, injecting choosing percentage noise from 0%, 5%, 10%, 15%, and 20% then optimizing data by using feature selection running ML algorithms for classification to determine the effectiveness of noise filtering on detecting malware API calls.
- The second experiment includes the previous steps without optimization.
- The third experiment involves collecting the dataset from the IEEE Dataport as separated text files and converting the dataset to a CSV file after preprocessing. Subsequently, injecting noise by choosing noise percentages from 0%, 5%, 10%, 15% and 20% then running ML algorithms for classification to determine the effectiveness of noise filtering on detecting malware API calls.
- The fourth experiment involves collecting a dataset from IEEE Dataport as separated text files then starting pre-processing the data by reassembling the data and converting the dataset to a CSV file. Subsequently, injecting noise by choosing noise percentage from 0%, 5%, 10%, 15%, and 20% then optimizing data by using feature selection and applying one of three methods to clean noise filtering and running ML algorithms to determine the effectiveness of noise cleaning on detecting malware API calls.

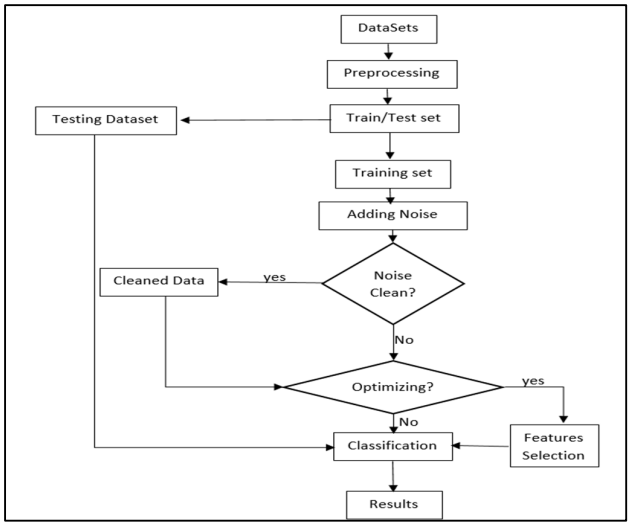

Fig. 2. The Overall methodology used in this paper

Algorithm 1. Algorithm of the steps that will be conducted during the research

Procedure Detect_Malicious (datasets *D,* noise_level = {0, 5, 10, 20})
Return trained models
//Select an appropriate dataset D_a and preprocessing and split it into train set D_{TN} and test set D_{TT}
Preprocessing (D_a)
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 (D_{TN}, $D_{TN\_noise\_level}$, $D_{TN\_noise\_level\_f}$)
D_{TN_s} = Feature_Selection (D_{TN})
For each dataset after noise insertion:
    $D_{TN\_noise\_level\_s}$ = Feature_Selection ($D_{TN\_noise\_level}$)
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 dataset
For each dataset (D_{TN}, $D_{TN\_noise\_level}$, $D_{TN\_noise\_level\_f}$, D_{TN_s}, $D_{TN\_noise\_level\_s}$, $D_{TN\_noise\_level\_f\_s}$):
    For each classification model (RF, MLP, IBK):
        Train_model (classification_model, dataset)
        Evaluate_model (D_{TT})
end

4.2 Dataset

Using datasets is essential to training and validating machine learning algorithms. The focus was on Android API datasets as we found many datasets with different aspects like ransomware, malware, adware, and other aspects. Computer- or program-generated solutions are needed to handle the increasing number of mobile malware samples (Lindorfer et al., 2014). The chosen datasets from Kaggle and IEEE Dataport were significant for malware attacks using Android API.

Pre-Processing for the Dataset

The data must be processed to clarify the features of the data. The phases of pre-processing of the datasets include:

- Data Collection Phase: The collected data was divided and duplications were identified which needs to be cleaned. The collected dataset from IEEE Dataport was divided into two datasets AMD and DREBIN where every dataset has five files containing data, labels, family, sample, and family index. Similarly, the Kaggle dataset was divided into two datasets: the first for benign and the second for malware.
- IEEE Dataset Pre-processing Phase: In this phase, the file has been grouped and is ready to add family and family index then add samples. Finally, adding data and labels together into one CSV file.
- Kaggle Dataset Pre-processing Phase: In this phase, the created CSV dataset with full features and classify apps into two CSV files were created: one for benign apps and the other for malware apps where the merge between them was done manually.

Dataset Statistics

The dataset has 35 API calls from 170 API calls in API level 23. In the pre-processed dataset, 4304 benign and 4011 malware Android applications were analyzed, which contained 8315 instances and 35 features.

Feature Selection

Feature selction is dealing with high-dimensional to find a limited subset of features that describe the target definition while elemenation irrelevant features (Priyadarsini et al., 2011). Feature selection attempts to identify an optimal subset of variables from the data while reducing the impact of noise or unnecessary variables (Chandrashekar & Sahin, 2014).

Statistical Feature Selection

Hall (1999) suggested Correlation-based Feature Selection (CFS) which is a straightforward filter algorithm that ranks feature subsets according to a heuristic evaluation function based on correlation. Irrelevant characteristics should be overlooked due to their poor association with the class. Delete redundant features that are heavily associated with one or more of the remaining features. For convenience, the CFS element subset evaluation function is presented in Eq. (1),

$$M_s = \frac{k\overline{r_{cf}}}{\sqrt{k + k(k-1)\overline{r_{ff}}}} \quad (1)$$

Evolutionary Feature Selection

According to Abd-Alsabour (2014), evolutionary algorithms are population-based heuristic search techniques (i.e., groups of individuals representing different candidate solutions to the optimization problem being solved). Each individual represents a subset of features in feature selection problems. Particle swarm optimization (PSO) is used as an evolutionary feature selection method. Particle swarm optimization is an incredibly straightforward algorithm that seems to be efficient at optimizing various functions.

Noise Insertion

Data in the real world has an element in Data Mining algorithms and must be dealt with realistically that way. We insert noise intentionally percentage from 5% and then will grow to 20%. There are two types, label noise and attribute noise as well.

Noise Filtering

Inaccurate and noisy data must be filtered out to obtain more reliable results. Several noise-filtering algorithms may be employed, including (Amro et al., 2021):

- The Repeated Edited Nearest Neighbor Algorithm (RENN): As long as there are no further deleted instances, the algorithm loops through ENN many times. By expanding on the above argument, all individual instances are also bound to their k closest neighbors (have the same class).

- Encoding Length (Explore): first applies the ELGrow algorithm to the data and then experiments with 1,000 possible classifier improvements. For each mutation, one instance is removed, one is added, or one is substituted. This change is kept only if it does not increase the classification costs.
- DROP5: this algorithm looks at the instances next to their closest first opponent and works its way outwards. This excludes most internal events. After this move, the instances are then reviewed for elimination, starting with the closest opponent. An instance is eliminated if no more than k of the instances, that have it among their neighbors, are correctly classified without it. This is done several times until there is no further progress to be achieved.

4.3 Machine Learning Algorithms

Neural Network

Artificial Neural networks are important in the implementation of Deep Learning. They are efficient and scalable, making them useful for tremendous and complex ML tasks. ANNs consist of three layers, input, output, and hidden nodes (Louridas & Ebert, 2016).

Decision Trees (DT-J48)

There are several tree analogies in the real world; they are the root of a large portion of machine learning, which deals with classification and regression. A decision tree can represent decisions and decision-making by visually and clearly showing decision nodes and their choices (Yang, 2019).

Random Forest (RF)

Creates several DTs, each with randomly assigned attributes. Much of the real results are in the distribution of files and folders may be managed by voting, where people have the option to vote for or to weigh vote (Buczak & Guven, 2016). As a learning tool, it is essential to understand that RF can play a big role in the use of other tools such as trees, predictive models, and clustering (Rathore et al., 2016).

Support Vector Machine (SVM)

Support vector machines (SVM) are supervised learning algorithms for classification and regression analysis. SVM constructs models that will automatically allocate examples to one or the other (Sabar et al., 2018).

Performance Metrics

The model is tested by performance-metric calculations. In research (Seliya et al., 2009), some performance metrics were calculated to determine how successful the ML algorithm is in detecting malware. To calculate performance, it needs to determine the following:

- **True Positive (TP)** the prediction was right and real results indicated that it was a positive
- **False Positive (FP)** predictions are inaccurate, Where negative values are predicted as positive
- **True Negative (TN)** the prediction was right and real results indicated that it was a negative
- **False Negative (FN)** predictions are inaccurate, Where positive values are predicted as negatives.

Precision: That is equal to the ratio of right positive examples / overall positive examples (Eq. (2)).

$$\frac{TP}{(TP + FP)} \tag{2}$$

Recall quantifies the number of positive predictions out of all the possibilities (Eq. (3)).

$$\frac{TP}{(TP + FN)} \tag{3}$$

Accuracy calculates the closeness of a measured or estimated value to its real value (Eq. (4)).

$$\frac{(TP + TN)}{(TP + TN + FP + FN)} \times 100 \tag{4}$$

5. Experimental results and discussions

Datasets balancing

SMOTE

Chawla et al. (2002) proposed an approach to balance data. The minority party is oversampled by the use of "synthetic" cases by working in "feature space". The used dataset was imbalanced and SMOTE was applied to balance the dataset to get 4304 benign instances and 4304 malware instances. Also, we round up decimal portion for better model training

Noise insertion and filtering

Firstly, we divided the datasets into two training datasets (6456 instances) and testing dataset (2152 instances). Both datasets are divided equally between benign and malware instances. Inserting noise is done automatically. The noise ratio calculates the percentage of affected instances: 0%, 5%, 10%, and 20% of selected instances randomly. The tests were performed using ten equal fold cross-validation and then averaged with each combination of dataset and instance reduction algorithm. For noise filtering, kNN, DROP5, RENN, and Explore have been applied as part of our experiment. Table 2 and Table 3 report the accuracy and size results for every iteration and noise ratio.

Table 2
Kaggle dataset noise insertion and cleaning iterations (10 Trials)

Noise Ration (average)	None (kNN)		DROP5		RENN		Explore	
	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
0%	82.54	100.00	80.61	15.40	80.79	81.11	69.10	0.03
5%	81.35	100.00	79.14	16.60	80.61	77.25	72.00	0.11
10%	79.96	100.00	78.58	17.26	80.65	73.19	72.00	0.15
20%	82.54	100.00	80.61	19.66	80.79	64.20	69.10	0.10

Table 3
Binarized Kaggle dataset noise insertion and cleaning iterations (10 Trials)

Noise Ration (average)	None (kNN)		DROP5		RENN		Explore	
	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
0%	82.81	100.00	81.68	10.21	81.99	82.99	75.54	0.03
5%	81.92	100.00	80.92	10.36	81.55	79.20	75.28	0.03
10%	81.21	100.00	80.75	10.61	81.57	76.06	75.54	0.03
20%	77.85	100.00	77.83	13.75	79.37	68.87	75.06	0.03

Based on the accuracy average for every run in Table 4 and Table 5 and Fig. 3 and Fig. 4, it can be noticed that DROP5 and RENN have similar accuracy. However, Explore and kNN did not give good results. We compared size and accuracy as shown in Fig. 5 and Fig. 6. RENN has been chosen because it is the best among all used filtering algorithms.

Table 4
Average noise-cleaning accuracy for unpruned trees in the Kaggle dataset at different noise ratios

Noise Ratio	None (kNN)		DROP5		RENN		Explore	
	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
0%	82.54	100.00	80.61	15.40	80.79	81.11	69.10	0.03
5%	81.35	100.00	79.14	16.60	80.61	77.25	72.00	0.11
10%	79.96	100.00	78.58	17.26	80.65	73.19	72.00	0.15
20%	82.54	100.00	80.61	19.66	80.79	64.20	69.10	0.10

Table 5
Average noise-cleaning accuracy for unpruned trees in the binarized dataset at different noise ratios

Noise Ratio	None (kNN)		DROP5		RENN		Explore	
	Accuracy	Size	Accuracy	Size	Accuracy	Size	Accuracy	Size
0%	82.81	100.00	81.68	10.21	81.99	82.99	75.54	0.03
5%	81.92	100.00	80.92	10.36	81.55	79.20	75.28	0.03
10%	81.21	100.00	80.75	10.61	81.57	76.06	75.54	0.03
20%	77.85	100.00	77.83	13.75	79.37	68.87	75.06	0.03

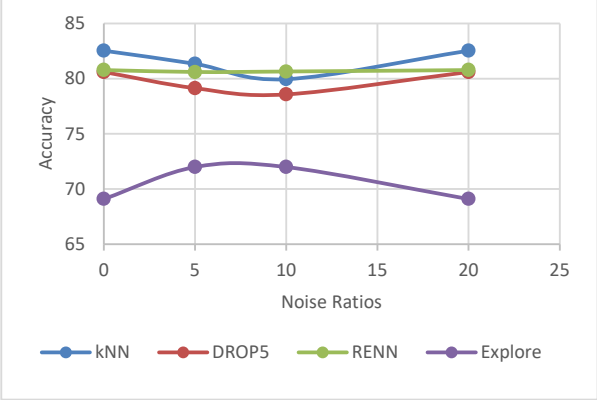

Fig. 3. Average noise-cleaning accuracy for unpruned trees in the dataset at different noise ratios

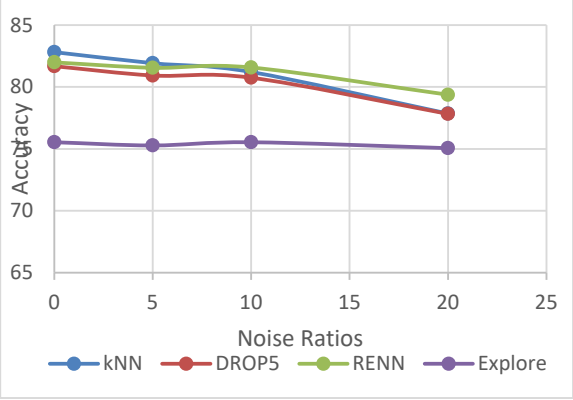

Fig. 4. Average noise-cleaning accuracy for unpruned trees in the binarized dataset at different noise ratios

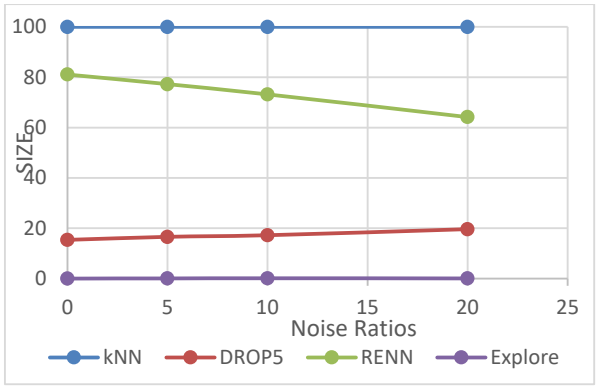

Fig. 5. Noise filtering size in non-binarized dataset at different noise ratios

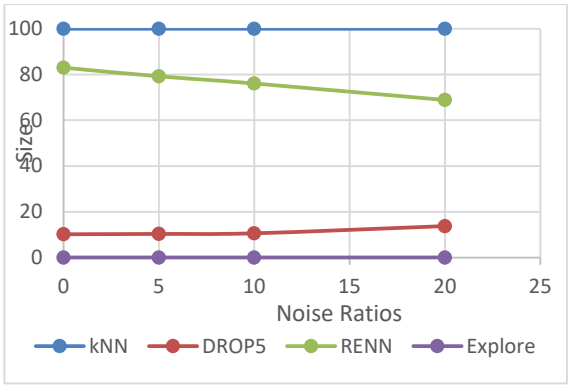

Fig. 6. Noise filtering size in the binarized dataset at different noise ratios

Feature selection

Evolutionary feature selection

We used the EvoloPy-FS tool for feature selection (Khurma et al., 2020) aims to provide a feature selection engine and continues our journey toward developing an optimized optimization environment, which began with EvoloPy for global optimization problems. The selection features were based on PSO as shown in Table 6 for binarized and non-binarized datasets with noises.

Table 6
Selected and Non-selected features using EvoloPy-FS and PSO in non-binarized datasets with all noise ratio

Feature	Non-Binarized				Binarized			
	0%	5%	10%	20%	0%	5%	10%	20%
android.os.SystemProperties	No	Yes	Yes	Yes	No	Yes	Yes	Yes
android.app.SharedPreferencesImpl$EditorImpl	Yes	No	Yes	No	Yes	Yes	Yes	Yes
libcore.io.IoBridge	Yes	Yes	Yes	No	Yes	No	No	No
android.os.Debug	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
java.lang.reflect.Method	Yes	No	No	No	No	No	No	Yes
android.app.ContextImpl	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
android.content.ContentValues	No	Yes	Yes	No	Yes	Yes	Yes	Yes
android.app.ActivityThread	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
android.telephony.TelephonyManager	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
android.util.Base64	Yes	Yes	Yes	No	Yes	Yes	Yes	No
android.content.ContentResolver	Yes	Yes	No	Yes	No	No	No	Yes
android.accounts.AccountManager	No	Yes	Yes	Yes	Yes	Yes	Yes	Yes
dalvik.system.BaseDexClassLoader	Yes	Yes	Yes	No	No	No	Yes	No
java.net.URL	Yes	Yes	No	Yes	Yes	Yes	Yes	Yes
dalvik.system.DexFile	Yes	Yes	Yes	Yes	No	Yes	Yes	No
dalvik.system.PathClassLoader	No	Yes	No	No	Yes	No	Yes	Yes
android.app.Activity	Yes	No	Yes	Yes	Yes	Yes	Yes	Yes
javax.crypto.Cipher	Yes	Yes	Yes	Yes	Yes	No	Yes	Yes
javax.crypto.spec.SecretKeySpec	Yes	Yes	Yes	Yes	No	Yes	Yes	No
dalvik.system.DexClassLoader	No	Yes	Yes	Yes	Yes	Yes	No	Yes
java.lang.Runtime	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
android.net.wifi.WifiInfo	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
javax.crypto.Mac	Yes	Yes	Yes	Yes	Yes	Yes	No	Yes
android.app.NotificationManager	Yes	No	Yes	Yes	No	No	Yes	Yes
java.io.FileInputStream	Yes	Yes	No	Yes	Yes	No	No	Yes
android.app.ApplicationPackageManager	No	Yes	No	Yes	No	Yes	Yes	Yes
java.lang.ProcessBuilder	Yes	Yes	Yes	Yes	Yes	No	Yes	No
java.io.FileOutputStream	No	No	Yes	No	Yes	No	Yes	Yes
android.os.Process	Yes	Yes	Yes	No	Yes	Yes	Yes	No
android.location.Location	Yes	Yes	Yes	Yes	No	Yes	Yes	Yes
org.apache.http.impl.client.AbstractHttpClient	Yes	No	Yes	No	Yes	No	Yes	No
android.media.AudioRecord	No	Yes	Yes	Yes	Yes	Yes	No	Yes
android.app.ActivityManager	Yes	Yes	Yes	Yes	Yes	Yes	Yes	No
android.media.MediaRecorder	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
android.telephony.SmsManager	No	Yes	Yes	Yes	No	No	Yes	No

We noted that some of the features had been selected in four runs for noise ratios for non-binarized and binarized datasets, we list the most important API calls in Table 7 for non-binarized datasets which appear in noise ratios. For binarization, Table 8 shows selected features appear in noise ratios.

Table 7
Selected features in all runs for non-binarized datasets

SN	Feature
1	android.app.ContextImpl
2	android.app.ActivityThread
3	android.telephony.TelephonyManager
4	dalvik.system.DexFile
5	javax.crypto.Cipher
6	javax.crypto.spec.SecretKeySpec
7	java.lang.Runtime
8	android.net.wifi.WifiInfo
9	javax.crypto.Mac
10	java.lang.ProcessBuilder
11	android.location.Location
12	android.app.ActivityManager
13	android.media.MediaRecorder

Table 8
Selected features in all runs for binarized datasets

SN	Feature
1	android.content.ContentValues
2	android.app.ActivityThread
3	android.accounts.AccountManager
4	android.app.Activity
5	java.lang.Runtime
6	android.media.MediaRecorder

Statistical feature selection

In this paper, we used Correlation-based Feature Selection (CFS) as statistical feature selection. Table 9 shows the selected features for all noise ratios running 0%, 5%, 10% and 20%.

Table 9
Selected and Non-selected features using CFS algorithm in the non-binarized dataset with all noise ratios

Feature	Non-Binarized				Binarized			
	0%	5%	10%	20%	0%	5%	10%	20%
android.os.SystemProperties	Yes	No	No	No	No	No	No	No
android.app.SharedPreferencesImpl$EditorImpl	Yes	No	No	No	No	No	No	No
libcore.io.IoBridge	Yes	No	No	No	Yes	No	No	No
android.os.Debug	No	No	No	No	Yes	Yes	Yes	Yes
java.lang.reflect.Method	Yes	No	No	No	No	No	No	No
android.app.ContextImpl	Yes	No	No	No	No	No	No	No
android.content.ContentValues	No	No	No	No	No	No	No	No
android.app.ActivityThread	Yes	No	Yes	Yes	No	No	No	No
android.telephony.TelephonyManager	Yes	No	No	No	No	No	No	No
android.util.Base64	Yes	No	No	No	No	No	No	No
android.content.ContentResolver	Yes	No	No	No	No	No	No	No
android.accounts.AccountManager	No	No	No	No	No	No	No	No
dalvik.system.BaseDexClassLoader	Yes	No	No	No	No	No	No	No
java.net.URL	Yes	Yes	Yes	Yes	No	No	No	No
dalvik.system.DexFile	Yes	No	No	No	No	No	No	No
dalvik.system.PathClassLoader	No	No	No	No	No	No	No	No
android.app.Activity	Yes	Yes	Yes	Yes	No	No	No	No
javax.crypto.Cipher	Yes	No	No	No	No	No	No	No
javax.crypto.spec.SecretKeySpec	Yes	Yes	Yes	Yes	No	No	No	No
dalvik.system.DexClassLoader	Yes	Yes	Yes	Yes	No	No	No	No
java.lang.Runtime	Yes	Yes	No	No	Yes	Yes	Yes	Yes
android.net.wifi.WifiInfo	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
javax.crypto.Mac	No	No	No	Yes	No	No	No	No
android.app.NotificationManager	No	No	No	No	No	No	No	No
java.io.FileInputStream	Yes	Yes	Yes	Yes	No	No	No	No
android.app.ApplicationPackageManager	Yes	Yes	No	No	No	No	No	No
java.lang.ProcessBuilder	Yes	Yes	Yes	Yes	No	No	No	No
java.io.FileOutputStream	Yes	No	No	No	No	No	No	No
android.os.Process	Yes	Yes	Yes	Yes	No	No	No	No
android.location.Location	Yes	No	No	No	No	No	No	No
org.apache.http.impl.client.AbstractHttpClient	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes
android.media.AudioRecord	Yes	Yes	Yes	Yes	No	No	No	No
android.app.ActivityManager	No	No	No	Yes	No	No	No	No
android.media.MediaRecorder	No	Yes	Yes	Yes	No	No	No	No
android.telephony.SmsManager	Yes	Yes	Yes	Yes	No	No	No	No

CFS selects some features in all runs and we consider this very important to help in detecting malware calls such as Table 10.

Table 10
Selected features using CFS algorithm in non-binarized datasets

SN	Feature
1	java.net.URL
2	android.app.Activity
3	android.accounts.AccountManager
4	javax.crypto.spec.SecretKeySpec
5	dalvik.system.DexClassLoader
6	android.net.wifi.WifiInfo
7	java.io.FileInputStream
8	java.lang.ProcessBuilder
9	android.os.Process
10	org.apache.http.impl.client.AbstractHttpClient
11	android.media.AudioRecord
12	android.telephony.SmsManager

Classification

After applying feature selection using evolutionary and statistical optimizers, we classified the data based on noise ratio, optimizer, and ML algorithms as IBK, Decision Tree (J48), and Multilayer Perception (MLP). Table 11 reports the classification accuracy. The average accuracy, recall, Precision, and Time to build, were reported at different noise ratios and different optimizers. At a 0% ratio and no optimizer, MLP has better accuracy than IBK and finally DT(J48). At a 5% ratio and no optimizer accuracy, MLP has better performance than IBK where performance is less, and DT(J48) as well. At a 10% ratio and no optimizer, MLP has the worst accuracy in all runs, but DT(J48) performance has been improved, and IBK not doing better was the accuracy of less than 5% and 0%. At a 20% ratio and no optimizer, MLP accuracy is the best. It has a higher gap than previous runs, and then DT(J48) performance gets lower, whereas at 20% it has the worst performance. Moreover, IBK accuracy is getting better as shown in Fig. 7.

Table 11
Classification result for optimizers in the binarized dataset

Noise Ratio	Optimizer	ML Algorithm	Accuracy	Recall	Precision	Time
0%	No Optimizer	IBK	82.7138 %	0.827	0.830	0
		Decision Tree (J48)	82.6208 %	0.826	0.828	0.17
		MLP	83.0855 %	0.831	0.834	26.55
	PSO	IBK	80.4833 %	0.805	0.812	0.01
		Decision Tree (J48)	80.6227 %	0.806	0.812	0.12
		MLP	81.1338 %	0.811	0.818	13.46
	CFS	IBK	75.2788 %	0.753	0.773	0
		Decision Tree (J48)	75.2788 %	0.753	0.773	0.04
		MLP	75.2788 %	0.753	0.773	1.51
5%	No Optimizer	IBK	82.1561 %	0.822	0.823	0
		Decision Tree (J48)	82.5279 %	0.827	0.825	0.16
		MLP	83.2714 %	0.833	0.835	25.09
	PSO	IBK	80.1115 %	0.801	0.806	0.01
		Decision Tree (J48)	80.2974 %	0.803	0.808	0.12
		MLP	80.9944 %	0.810	0.815	11.55
	CFS	IBK	75.2788 %	0.753	0.773	0.76
		Decision Tree (J48)	75.2788 %	0.753	0.773	0.02
		MLP	75.2788 %	0.753	0.773	1.28
10%	No Optimizer	IBK	81.4591 %	0.815	0.816	0
		Decision Tree (J48)	82.6208 %	0.826	0.828	0.17
		MLP	82.4349 %	0.824	0.827	24.32
	PSO	IBK	78.4387 %	0.784	0.786	0
		Decision Tree (J48)	77.881 %	0.779	0.780	0.18
		MLP	78.7639 %	0.788	0.791	16.57
	CFS	IBK	67.6115 %	0.676	0.732	0
		Decision Tree (J48)	67.6115 %	0.676	0.732	0.02
		MLP	67.6115 %	0.676	0.732	1.26
20%	No Optimizer	IBK	82.1097 %	0.821	0.822	0
		Decision Tree (J48)	82.4349 %	0.824	0.826	0.16
		MLP	83.5502 %	0.836	0.838	22.57
	PSO	IBK	80.2974 %	0.803	0.808	0
		Decision Tree (J48)	81.4591 %	0.815	0.821	0.13
		MLP	81.2732	0.813	0.820	11.29
	CFS	IBK	75.3717 %	0.754	0.772	0
		Decision Tree (J48)	75.3717 %	0.754	0.772	0.02
		MLP	75.4182 %	0.754	0.773	1.18

At 0% ratio and PSO optimizer, MLP has the best accuracy than IBK and finally DT(J48). At a 5% ratio and PSO optimizer accuracy, IBK has the worst performance than DT(J48) where performance is less and MLP still has the best accuracy. At a 10% ratio and PSO optimizer, MLP has the worst accuracy in all runs, the same as for DT(J48) performance getting worse, and IBK not doing better where the accuracy was less than 5% and 0% and it is considered as a common issue for all classifiers in this run were they have the worst accuracy in all runs. At 20% ratio and PSO optimizer, DT(J48) accuracy is the best. It has a higher gap than previous runs, so both DT(J48) and MLP have the same performance. Moreover, all classifiers' accuracies are getting better as shown in Fig. 8.

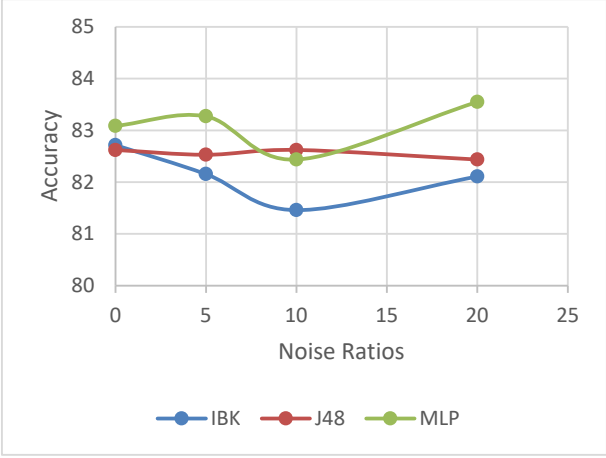

Fig. 7. ML algorithm accuracy at no optimizer in binarized datasets

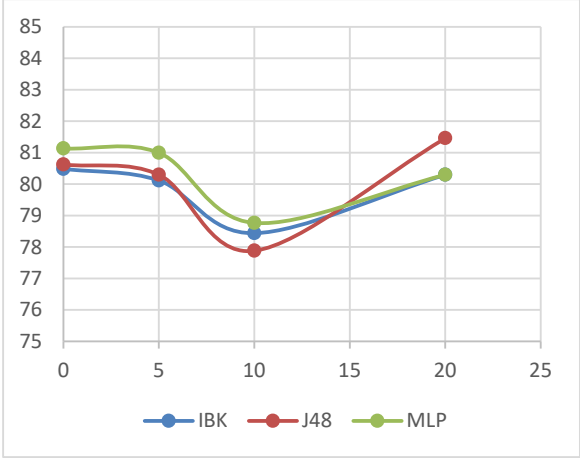

Fig. 8. ML algorithm accuracy at PSO optimizer in binarized datasets

For statistical runs, the accuracy was nearly the same but it is not doing well with statistical optimization as shown in Fig. 9. The reason behind this is that equal runs have statistically chosen the same feature for all noise ratios and at 5% noise ratio which choose one more feature than already chosen by other runs on noise ratios. At 10% run accuracy is the worst and in general classifiers do not work accurately with statistical optimization.

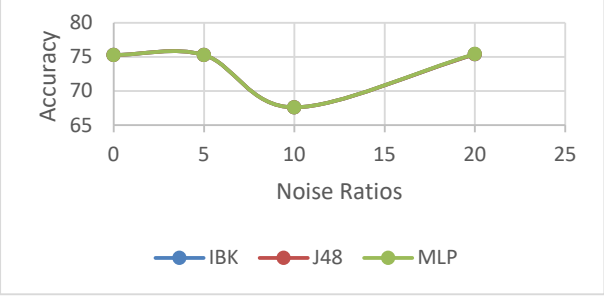

Fig. 9. ML algorithm accuracy at CFS optimizer in binarized datasets

The results for the classification of optimizers in non-binarized datasets are reported in Table 12. Average accuracy, recall, Precision, and Time to build, were reported at different noise ratios and different optimizers. At a 0% ratio and no optimizer, DT(J48) has better accuracy than IBK and MLP. At a 5% ratio and no optimizer, the accuracy of DT(J48) has better performance than IBK and MLP has the same performance. At a 10% ratio and no optimizer, DT(J48) has lower accuracy than a 5% ratio, but it is looking to have robust performance. At 20% ratio and no optimizer, DT(J48) accuracy is the best and it seems to have a robust performance. The gap between DT(J48) is more than 30% which has a considerable effect that leads to DT(J48) being the best, as shown in Fig. 10.

For PSO optimizers, the classifiers have good performance. At a 0% ratio, DT(J48) has the best accuracy than IBK and finally, MLP has the worst performance. At a 5% ratio, the accuracy of DT(J48) was affected but still performed way better than IBK and MLP which have the same performance. At 10% ratio and no optimizer, DT(J48) has better accuracy than 5% ratio but it is looking to have robust performance. At 20% ratio and no optimizer, DT(J48) accuracy is the best and it seems to have a robust performance in all noise ratios. The gap between DT(J48) is more than 30% which has a considerable effect that leads to that DT(J48) is the best, as shown in Fig. 11.

For the Statistical optimizer (CFS), the classifiers have good performance in general, as shown in Fig. 12. At a 0% ratio, DT(J48) has the best accuracy than IBK and finally, MLP has the worst performance. At a 5% ratio, the accuracy of DT(J48) was affected but still performed better than both IBK and MLP who nearly have the same performance. At a 10% ratio and no optimizer, DT(J48) accuracy is affected where it is going lower but it looks to have a robust performance. At a 20% ratio and no optimizer, DT(J48) accuracy is the best among all classifiers. It seems to have a robust performance in all noise ratios with a small effect of noise data. The gap between DT(J48) is more than 30% which has a considerable effect that leads to DT(J48) being the best.

The non-binarized dataset is the best dataset to help in detecting malware and the PSO optimizer is the best optimizer that selects features that affect the classifier where DT(J48) is robust to noise or tolerant to noise whereas MLP and IBK were not affected by noise. Moreover, the time to build the model in DT(J48) is less than 1 second in all runs. IBK has the best time where it is 0 in all but it has bad accuracy.

Table 12

Classification result for optimizers in non-binarized datasets

Noise Ratio	Optimizer	ML Algorithm	Accuracy	Recall	Precision	Time
0%	No Optimizer	IBK	50.1394 %	0.501	0.650	0
		Decision Tree (J48)	82.3885 %	0.824	0.828	0.3
		MLP	50.3717 %	0.504	0.592	25.9
	PSO	IBK	53.9498 %	0.539	0.705	0
		Decision Tree (J48)	82.4814 %	0.825	0.827	0.22
		MLP	50.5576 %	0.506	0.585	16.45
	CFS	IBK	50.1859 %	0.502	0.750	0
		Decision Tree (J48)	81.1338 %	0.811	0.814	0.22
		MLP	49.7677 %	0.498	0.467	16.66
5%	No Optimizer	IBK	50.1394 %	0.501	0.650	0
		Decision Tree (J48)	82.5743 %	0.826	0.829	0.26
		MLP	50.4182 %	0.504	0.619	24.91
	PSO	IBK	50.0929 %	0.501	0.750	0.21
		Decision Tree (J48)	81.7379 %	0.817	0.819	16.98
		MLP	49.9535 %	0.500	0.488	0
	CFS	IBK	50.0465 %	0.500	0.750	0
		Decision Tree (J48)	78.7639 %	0.788	0.809	0.11
		MLP	54.2286 %	0.542	0.642	5.95
10%	No Optimizer	IBK	50.1859 %	0.502	0.667	0
		Decision Tree (J48)	81.7379 %	0.817	0.821	0.23
		MLP	50.2788 %	0.503	0.651	23.93
	PSO	IBK	50.1859 %	0.502	0.750	0
		Decision Tree (J48)	82.0167 %	0.820	0.823	0.18
		MLP	50.0465 %	0.500	0.508	16.57
	CFS	IBK	50.0929 %	0.501	0.750	0
		Decision Tree (J48)	77.277 %	0.773	0.797	0.1
		MLP	50.2323 %	0.502	0.560	4.65
20%	No Optimizer	IBK	50.1394 %	0.501	0.650	0
		Decision Tree (J48)	81.4591 %	0.815	0.820	0.22
		MLP	50.1859 %	0.502	0.600	20.44
	PSO	IBK	50.0465 %	0.500	0.750	0
		Decision Tree (J48)	80.0651 %	0.801	0.810	0.16
		MLP	50.1394 %	0.501	0.750	11.22
	CFS	IBK	50.0929 %	0.501	0.750	0
		Decision Tree (J48)	77.277 %	0.773	0.797	0.07
		MLP	50.2323 %	0.502	0.560	4.76

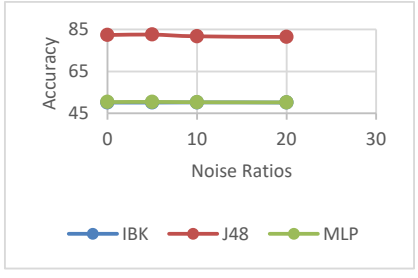

Fig. 10. ML algorithm accuracy at no optimizer in non-binarized datasets

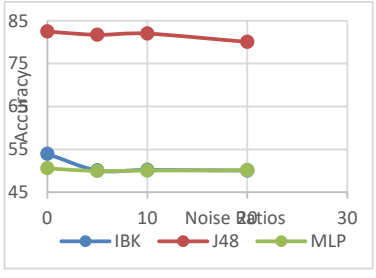

Fig. 11. ML algorithm accuracy at PSO optimizer in non-binarized datasets

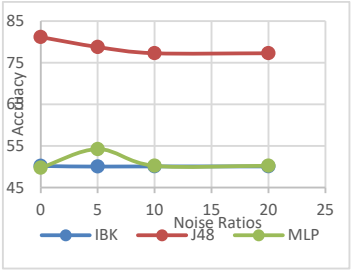

Fig. 12. ML algorithm accuracy at CFS optimizer in non-binarized datasets

6. Conclusions and Future Work

After reviewing all APIs from the first release of Android OS until API level 23. An approach was conducted to build an AI model to detect malware by analyzing API calls in Android applications. The used dataset has been prepared through different phases which are noise insertion and filtering, evolutionary and statistical feature selection, and classification. This analysis was aimed to examine AI model robustness through different noise ratios on datasets. Additionally, the investigation of using optimizers to select the feature that may be considered as security threats that may open the OS to various attacks.

In conclusion, this paper found that Decision Trees (J48) and particle swarm optimizer for non-binarized dataset is the most robust and tolerant classifier in different noise ratios.

In future work, more datasets are to be tested such as the IEEE Dataport dataset. Additionally, building an AI tool to examine API calls in Android apps through testing the apps and returning results reports for (.APK) files and applying higher noise ratios to run more classifiers. Furthermore, a wider selection of optimizers could be used to find the best optimizer that helps in choosing the most important features taking into consideration more than only API calls where permission and intent play a good role in detecting malware.

References

Abd-Alsabour, N. (2014, 21-23 Oct. 2014). A Review on Evolutionary Feature Selection. Paper presented at the 2014 European Modelling Symposium.

Abuthawabeh, M., & Mahmoud, K. W. (2020). Enhanced android malware detection and family classification, using conversation-level network traffic features. *The International Arab Journal of Information Technology, 17*(4A), 607-614.

Agrawal, P., & Trivedi, B. (2019, 20-22 Feb. 2019). A Survey on Android Malware and their Detection Techniques. Paper presented at the 2019 *IEEE International Conference on Electrical, Computer and Communication Technologies (ICECCT).*

Alagarsamy, M., & Sathik, A. S. (2022). Context Aware Mobile Application Pre-Launching Model using KNN Classifier. *International Arab Journal Of Information Technology, 19*(6), 932-941.

Alam, S., & Demir, A. K. (2023). Mining android bytecodes through the eyes of gabor filters for detecting malware. *The International Arab Journal of Information Technology, 20*(2), 180-189.

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.*

Almomani, I. M., & Khayer, A. A. (2020). A Comprehensive Analysis of the Android Permissions System. *IEEE Access, 8*, 216671-216688. doi: 10.1109/ACCESS.2020.3041432

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

Alsoghyer, S., & Almomani, I. (2019). Ransomware Detection System for Android Applications. *Electronics, 8*(8). doi: 10.3390/electronics8080868

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. doi: doi:10.1515/jisys-2020-0061

AVTEST. (2021). Malware. from https://www.av-test.org/en/statistics/malware/

Buczak, A. L., & Guven, E. (2016). A Survey of Data Mining and Machine Learning Methods for Cyber Security Intrusion Detection. *IEEE Communications Surveys & Tutorials, 18*(2), 1153-1176. doi: 10.1109/COMST.2015.2494502

Chandrashekar, G., & Sahin, F. (2014). A survey on feature selection methods. *Computers & Electrical Engineering, 40*(1), 16-28.

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.

Dubey, H., Bhatt, S., & Negi, L.(2023) Digital Forensics Techniques and Trends: A Review. *The International Arab Journal of Information Technology (IAJIT) 20*(4), 644 - 654, doi: 10.34028/iajit/20/4/11.

Faris, H., Habib, M., Almomani, I., Eshtay, M., & Aljarah, I. (2020). Optimizing Extreme Learning Machines Using Chains of Salps for Efficient Android Ransomware Detection. *Applied Sciences, 10*(11). doi: 10.3390/app10113706

François-Lavet, V., Henderson, P., Islam, R., Bellemare, M. G., & Pineau, J. (2018). An introduction to deep reinforcement learning. arXiv preprint arXiv:1811.12560.

Hall, M. A. (1999). Correlation-based feature selection for machine learning (Doctoral dissertation, The University of Waikato).

Jung, J., Kim, H., Shin, D., Lee, M., Lee, H., Cho, S., & Suh, K. (2018, 26-28 Sept. 2018). Android Malware Detection Based on Useful API Calls and Machine Learning. *Paper presented at the 2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE).*

Jung, J., Lim, K., Kim, B., Cho, S., Han, S., & Suh, K. (2019, 3-5 June 2019). Detecting Malicious Android Apps using the Popularity and Relations of APIs. *Paper presented at the 2019 IEEE Second International Conference on Artificial Intelligence and Knowledge Engineering (AIKE).*

Khayer, A. A., Almomani, I., & Elkawlak, K. (2020, 3-5 Nov. 2020). ASAF: Android Static Analysis Framework. *Paper presented at the 2020 First International Conference of Smart Systems and Emerging Technologies (SMARTTECH).*

Khurma, R. A., Aljarah, I., Sharieh, A., & Mirjalili, S. (2020). *EvoloPy-FS: An Open-Source Nature-Inspired Optimization Framework in Python for Feature Selection*. In S. Mirjalili, H. Faris & I. Aljarah (Eds.), Evolutionary Machine Learning Techniques: Algorithms and Applications (pp. 131-173). Singapore: Springer Singapore.

Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Van Der Veen, V., & Platzer, C. (2014, 2014). Andrubis--1,000,000 apps later: A view on current Android malware behaviors.

Louridas, P., & Ebert, C. (2016). Machine Learning. IEEE Software, 33(5), 110-115. doi: 10.1109/MS.2016.114

Nellaivadivelu, G., Di Troia, F., & Stamp, M. (2020). Black box analysis of android malware detectors. *Array, 6*, 100022. doi: https://doi.org/10.1016/j.array.2020.100022

Priyadarsini, R. P., Valarmathi, M. L., & Sivakumari, S. (2011). Gain ratio based feature selection method for privacy preservation. *ICTACT Journal on soft computing, 1*(4), 201-205.

Rathore, M. M., Ahmad, A., & Paul, A. (2016). Real time intrusion detection system for ultra-high-speed big data environments. *The Journal of Supercomputing, 72*(9), 3489-3510. doi: 10.1007/s11227-015-1615-5

Ray, S. (2019, 14-16 Feb. 2019). A Quick Review of Machine Learning Algorithms. *Paper presented at the 2019 International Conference on Machine Learning, Big Data, Cloud and Parallel Computing (COMITCon).*

Sabar, N. R., Yi, X., & Song, A. (2018). A bi-objective hyper-heuristic support vector machines for big data cyber-security. IEEE Access, 6, 10421-10431.

Sarkar, A., Goyal, A., Hicks, D., Sarkar, D., & Hazra, S. (2019, 12-14 Dec. 2019). Android Application Development: A Brief Overview of Android Platforms and Evolution of Security Systems. *Paper presented at the 2019 Third International conference on I-SMAC (IoT in Social, Mobile, Analytics and Cloud) (I-SMAC).*

Seliya, N., Khoshgoftaar, T. M., & Hulse, J. V. (2009, 2-4 Nov. 2009). A Study on the Relationships of Classifier Performance Metrics. *Paper presented at the 2009 21st IEEE International Conference on Tools with Artificial Intelligence.*

Sharma, S., Challa, R. K., & Kumar, R. (2021). An ensemble-based supervised machine learning framework for android ransomware detection. *The International Arab Journal of Information Technology, 18*(3A), 422-429.

Sharma, T., & Rattan, D. (2021). Malicious application detection in android — A systematic literature review. *Computer Science Review, 40*, 100373. doi: https://doi.org/10.1016/j.cosrev.2021.100373

Van Engelen, J. E., & Hoos, H. H. (2020). A survey on semi-supervised learning. *Machine Learning, 109*(2), 373-440. doi: 10.1007/s10994-019-05855-6

Yang, F. (2019, 5-7 Dec. 2019). An Extended Idea about Decision Trees. *Paper presented at the 2019 International Conference on Computational Science and Computational Intelligence (CSCI).*

Zhang, Z., & Cai, H. (2019, May). A look into developer intentions for app compatibility in android. *In 2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft) (pp. 40-44).* IEEE.

Zhu, X., & Wu, X. (2004). Class Noise vs. Attribute Noise: A Quantitative Study. *Artificial Intelligence Review, 22*(3), 177-210. doi: 10.1007/s10462-004-0751-8