Classification of Benign and Malicious Applications using Machine Learning Techniques

ABSTRACT

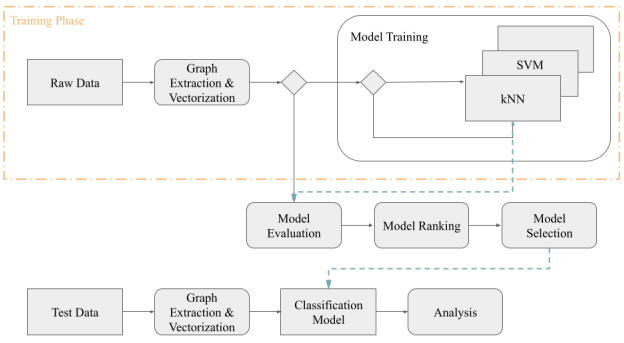
The number of applications (commonly called “apps”) has been growing exponentially over the recent years, with more and more usage of Android Operating System. Some of these apps try to violate the privacy of the user or use the user’s data to do something they are not supposed to do. Such apps are termed as “malicious.” This is highly concerning as it is getting more difficult to differentiate them from non-malicious (benign) apps each day. In this study, we have created a malicious and benign classifier tool using some well-known Machine Learning (ML) methods. We first procure the dataset consisting of both types of apps, extract the manifest file from each of them, extract some features by static analysis, train the models, and finally test them on some apps. Then, we compare all the models based on the f-1 score, accuracy, precision, and recall.

KEYWORDS

Malware detection, Android, Permission grouping

1 INTRODUCTION

Along with malware, the malware detection tools are also growing in number. Behavior-based and pattern-based methods are some such tools in the market, and many more data analysis methods are being implemented nowadays with the advancement in Artificial Intelligence (AI). Malicious software detection has become a major challenge for app developers, app-hosting companies, and users all over the world in the past few decades. There are many pieces of research conducted on malware detection since it first appeared in the early 1970s.



Not everyone has access to tools that can detect whether the app they just downloaded is malicious or not. Our motivation to conduct this research is to produce a recommending tool that can be easily accessed by the general public for detecting malware. Optimistically, we want to reduce the chance of people downloading malicious apps and potentially prevent their devices from being hacked. To achieve that, we will be classifying applications using Logistic Regression and different similarity-based methods including k-nearest neighbors (kNN) as well as Decision Tree classifier to see if different methods can detect certain types of malware or any specific features. We are interested in analyzing whether one classifier has better performance in detecting certain types of malware or specific features, and designing a framework for recommending a method with a specific set of parameters for a certain type of malware and provide users a more friendly interface. With the similarity-based approach, we believe that it will detect malware with much higher accuracy and will be more flexible for applications that evolved over time as they become more complicated.

The contributions of this work in the following way:

• Feature grouping based Android malware detection tool is a low runtime and highly efficient machine learning model.

• The model groups permission-based features with a unique methodology and obtains only 13 static features for each application.

• The model has 97% classification success in the tests and only needs 0.063 s for analysis per application. This value is one of the best values among the models with similar classification success. This value was obtained without using the GPU.

• As a result, a model with high classification success and low analysis time with few features has been revealed thanks to the proposed unique grouping.

2 BACKGROUND

A Malware is any program that intends to harm the system. It typically is disguised as a benign piece of software which causes a system to misbehave. There are multiple types of malware, including but not limited to trojans, virus and spyware. Each malware has a specific purpose of breaching the system by either collecting customer data, or blocking the system or deleting key pieces of code from the system making it unusable. A Malware attack is a lot more serious when attack target is a private organization, rather than individuals. A successful malware attack on a private organisation breaches customer trust, making the organization vulnerable to failure. The ideal state of a system is to avoid a malware attack altogether, although is not feasible to achieve always. Hence arrives the need for anti-virus or anti-malware softwares. Unfortunately, as anti-malware softwares grow, so do the malwares, which makes it a constant battle of cat and mouse. There are a number of Malware detection techniques that are used to detect and block malware attacks on the system, the most famous ones include: Signature-Based, Behavior-Based,

and Heuristic-Based. Signature based Malware detection algorithms are the most commonly used for anti-virus and anti-malware softwares. The mechanism used here is to find a series of bytes to identify a malware. This is not as effective since it needs the latest list of all malicious signatures to compare the system against. The algorithm is quite stable, predicts a low error rate, but takes longer to read the series of bytes throughout the system, thus us not time-efficient. Behavior based Malware detection works on the principle that software behavior does not change and a change to the core functionality implies a Malware attack. While this method does not need to read all the bytes, it still is a slow method and creates a high number of false positives for cases where behavior change is frequent. Heuristic based Malware detection learns malware behavior using ML techniques, so the shortcomings of Signature or Behavior based detection techniques are alliviated. Being an ML based model adds to the benefit of higher accuracy with high efficiency. The ML technique here needs to build a classifier, which could be done with good datasets and labelled samples, improved overtime with an intelligent learning process.

3 APPROACH

3.1 Data Collection

The datasets that we have used for the malicious apps are Drebin (https://www.sec.tu-bs.de/%C2%A0danarp/drebin/) and Genome (http://www.malgenomeproject.org/) containing 5560 and 1000 samples (APKs) respectively. Arslan (https://doi.org/10.1142/S0218194019500037) dataset has been used for the benign apps. It has around 960 samples of benign apps.

3.2 Application Structure

Apps for Android come in a compressed file with the APK extension (.apk), which can then be run and installed on the phone. These files include the source code, manifest.xml, libraries, resources, etc., as shown in Figure-1. Apps are programmed in Java using the Android SDK. After the completion of the source code, they are converted into Dalvik bytecode (DEX).

What is most crucial to this study inside the compressed apk is the manifest file. It is an XML containing basic cookies, links to external files, and libraries like activities, receivers, content providers, permissions required to access device resources, and target platform data. External resources like videos, sounds, audios, images, and text files needed by the app are also packaged in the apk file. Hence, this whole package is necessary for an app to work. For Android, Google Play Store and other third party platforms offer such prepared apk packages for various apps.

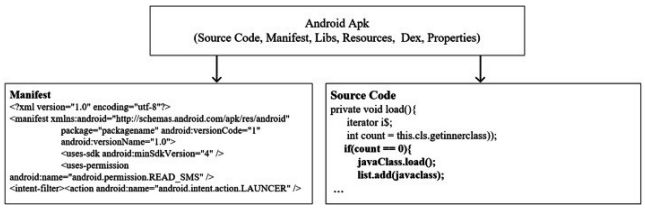


Figure-1 (https://doi.org/10.1016/j.jisa.2021.102794)

3.3 Pre-processing

3.3.1 Manifest Extraction

For extracting the manifest file from each apk, we used an XML parser called AXMLPrinter2 (https://github.com/flyfei/ApkDecompile/tree/master/Tools) that can not only extract the manifest from an apk, but also make it readable.

3.3.2 Feature Selection

We have only worked on the static features of the apps. The three feature sets we chose to include and experiment on in our work are described below. Finally, it is time to label each vector row as benign or malicious, so we can train the models. Hence, we append 0, if benign, and 1, if malicious, to each row of the vector. With this, our feature selection and grouping is over and we have prepared a file with the vector data. We can now pass this to our algorithms.

3.3.2.1 Permissions

The first feature on which we trained our models was the permissions requested by the apps. There are around 395 different permissions an app can ask for to the user. So, we first extracted all the permissions an app can need or ask for from a sample manifest file and listed them in a text document. Then, for each app (apk), we extracted their required permissions from their manifests, compared them with our list of total permissions, and formed a vector containing serial number followed by 395 0s or 1s (0 = permission not asked, 1 = permission needed). The unique step in our work is that after making a vector of 7622 (total apks from all datasets) x 396 (395 permissions, 1 serial id) dimensions, we performed feature grouping for all the permissions. 11 groups of permissions were made according to their category (see figure-2). For example, if an app used 3 of the permissions from the ACCESS group, the element in the vector after the serial number became 3. Hence, the vector dimensions were now 7622 x 12

Feature Group	Included Features	Feature Group Details
1- Access	ACCESS_ALL_DOWNLOADS;ACCESS_BLUETOOTH_SHARE;ACCESS_CACHE_FILESYSTEM;ACCESS_CHECKIN_PROPERTIES;ACCESS_CONTENT_PROVIDERS_EXTERNALLY;ACCESS_DOWNLOAD_MANAGER; ETC.	It is the group that contains the permissions used to ACCESS various information such as downloads, Bluetooth, location information, internet network usage information, wi-fi status.
2- Modify	MODIFY_APP;MODIFY_AUDIO_ROUTING;MODIFY_AUDIO_SETTINGS;MODIFY_CELL_BROADCASTS;MODIFY_DAY_NIGHT_MODE;MODIFY_NETWORK_ACCOUNTING;MODIFY_PARENTAL_CONTROLS;MODIFY_PHONE_STATE	The group contains the permissions used to MODIFY settings such as sound settings, access status, and network account information.
3- Set	SET_ACTIVITY_WATCHER;SET_ALWAYS_FINISH;SET_ANIMATION_SCALE;SET_DEBUG_APP;SET_INPUT_CALIBRATION;SET_KEYBOARD_LAYOUT;SET_ORIENTATION;SET_POINTER_SPEED;SET_PREFERRED_APPLICATIONS;SET_PROCESS_LIMIT;SET_SCREEN_COMPATIBILITY;SET_TIME;SET_TIME_ZONE;SET_WALLPAPER;SET_WALLPAPER_COMPONENT;SET_WALLPAPER_HINTS	The group contains the permissions used to SET information such as activity monitoring, checking whether the activities are finished when the applications are running in the background, and animation scaling.
4- Update	UPDATE_APP_OPS_STATS;UPDATE_CONFIG;UPDATE_DEVICE_STATS;UPDATE_LOCK;UPDATE_LOCK_TASK_PACKAGES	The group contains the permissions used to UPDATE configurations, information such as device statistics.
5- Write	WRITE_CALENDAR;WRITE_CONTACTS;WRITE_EXTERNAL_STORAGE;WRITE_HISTORY_BOOKMARKS; WRITE_CALL_LOG	The group contains the permissions used for WRITING calendar information, contacts, storage information.
6- Read	READ_CALL_LOG; READ_CALENDAR; READ_CONTACTS; READ_EXTERNAL_STORAGE;READ_HISTORY_BOOKMARKS;READ_PHONE_STATE;READ_SMS	The group has permissions used to READ calendar information, contacts, storage information.
7- Get	GET_ACCOUNTS; GET_ACCOUNTS_PRIVILEGED; GET_APP_GRANTED_URI_PERMISSIONS;GET_APP_OPS_STATS;GET_DETAILED_TASKS;GET_INTENT_SENDER_INTENT;GET_PACKAGE_IMPORTANCE;GET_PACKAGE_SIZE;GET_PASSWORD;GET_PROCESS_STATE_AND_OOM_SCORE;GET_TASKS;GET_TOP_ACTIVITY_INFO	The group contains the permissions used to GET account information, authorization status of accounts, and employee tasks.
8- Manage	MANAGE_ACCOUNTS;MANAGE_ACTIVITY_STACKS;MANAGE_APP_OPS_RESTRICTIONS;MANAGE_APP_TOKENS;MANAGE_CA_CERTIFICATES;MANAGE_DEVICE_ADMINS;MANAGE_DOCUMENTS;MANAGE_FINGERPRINT;MANAGE_MEDIA_PROJECTION;MANAGE_NETWORK_POLICY;MANAGE_NOTIFICATIONS;MANAGE_PROFILE_AND_DEVICE_OWNERS;MANAGE_SOUND_TRIGGER;MANAGE_USB;MANAGE_USERS;MANAGE_VOICE_KEYPHRASES	The group contains the permissions required for the MANAGEMENT of account information, authorization status of the accounts, and access to running tasks.
9- Bind	BIND_ACCESSIBILITY_SERVICE;BIND_APPWIDGET;BIND_CARRIER_MESSAGING_SERVICE;BIND_CARRIER_SERVICES;BIND_CHOOSER_TARGET_SERVICE;BIND_CONDITION_PROVIDER_SERVICE;BIND_CONNECTION_SERVICE;BIND_DEVICE_ADMIN; ETC.	The group allows components to BIND to services send requests, receive responses, and communication between them.
10- Broadcast	BROADCAST_CALLLOG_INFO;BROADCAST_NETWORK_PRIVILEGED;BROADCAST_PACKAGE_REMOVED;BROADCAST_PHONE_ACCOUNT_REGISTRATION;BROADCAST_SMS;BROADCAST_STICKY;BROADCAST_WAP_PUSH	The group contains the permissions used to BROADCAST network authorization and removed packages status.
11- Control	CONTROL_INCALL_EXPERIENCE;CONTROL_KEYGUARD;CONTROL_LOCATION_UPDATES;CONTROL_VPN;CONTROL_WIFI_DISPLAY	The group contains the necessary permissions to provide functionality to CONTROL phone calls or to CONTROL critical situations such as that all activities can continue to run even when the phone is locked.

Figure-2

3.3.2.2 App size

Our second feature was the app size. We calculated the size of an app from the size of its resources folder (.apk/res/). The res folder stores the resources that are used by the application, namely, features of color, styles, dimensions, etc. We got the sizes in bytes; thus, we converted them into KBs and appended them to the vector of each app. Now, the size of the vector is 7622 x 13.

3.3.2.3 Number of icons of an app

We now look at the number of icons used by an app as our third and last feature. Malicious apps try to use more icons to fool the users and make them click it thinking it is a different app, though it is the same one. We extracted this from the manifest file itself, resulting in a 7622 x 14 vector.

3.3.3 Algorithms

The following are the three existing ML models (using scikit-learn in Python) we have implemented in our study.

3.3.3.1 Logistic Regression

A Logistic regression is used to predict a binary outcome of an event. An example could be if a person could be sick or not. The possible outcomes of the question here are only Yes and No, which could be predicted using a logistic regression.

The prediction of a Logistic regression is based on a Logistic function, or the Sigmoid function, or known as an S-curve, which converts all data to a value between 0 and 1. The mathematical formula is

$$\frac{1}{1 + e^{-val}}$$

The most common applications of a Logistic regression are healthcare industry to model if a person has Covid-19, financial industry to determine a fraudulent transaction, and advertisement to determine if a customer is likely to buy a product.

The examples above are for a Binary logistic regression, which could be expanded to Multinomial Logistic regression to increase the number of options available, say for E.g.: if the person is sick, what disease do they have: Flu, Cold or Covid.

3.3.3.2 K-Neighbors

While the Logistic Regression model is a regression model, The K-Nearest Neighbors (henceforth KNN) can be used to solve both Regression and Classification problems. The premise that KNN is based on is that similar things stay near to each other. For e.g.: Given a movies data set, predict the movie that I would like. The definition of "near" is dependent on the type of dataset you use: Could be Euclidean distance, Manhattan distance, Hamming distance, and Minkowski distance. The algorithm depends on the value of K. A low value of K will make the prediction unstable since it takes only the nearest datapoints not giving enough good answers, a higher value of K has an averaging issue which could lead to incorrect predictions owing to statistical mode. The mechanism to determine the right value of K is to run the model with various values and try on a dataset. The most common usage of KNN is in the credit system to determine the Credit rating of an individual, pattern recognition, and recommendation systems. Although KNN could get slow with more data, the algorithm might not be feasible for a real-time analysis.

3.3.3.3 Decision Tree

As the name suggests, a Decision Tree model uses a tree-like model of decisions. A decision tree can be used for both regression or classification modeling, thus also referred as CART (Classification and Regression Trees). Decision Trees are best for a visual representation of how a certain decision was reached based on the input parameters. All possible solutions of an algorithm are modeled as a Leaf Node of the decision tree, and all conditions modeled as a Decision Node. A general decision tree is based on a Yes-No modeling, and the traversal of the tree is based

on outcome of each Yes-No decisions. Decision trees are best used if a visual representation of an algorithm is required.

3.3.3.4 Adaptive Boosting

Boosting is a mechanism in which we combine the results from multiple models to improve accuracy and the prediction power of the model. It embodies the phrase: convert several weak learners to one strong learner. Ada Boost, also called Adaptive Boost algorithm, uses an ensemble method - using multiple models in series, and in an iterative manner adjusting weights until a low error rate is achieved. Adaboost was the first successful boosting algorithm developed for binary classification. The most common algorithm used with Adaboost is the decision tree algorithm with one level, called decision stump. Unfortunately, Noisy data or outliers will force the ensemble to overcorrect the models and decrease performance of the model. This is one of the biggest disadvantages of using an Adaboost. Real-world applications of Adaboost are customer churn prediction and player performance in big leagues like NFL where multiple datapoints: past performance, athletics, running speed can predict their success in the game.

3.3.3.5 Gradient Boosting

Adaboost was recast in a statistical framework, further developed, and was called Gradient Boosting models. The framework used numerical optimization, with an objective to minimize the loss by using a differentiable loss function. The Gradient boosting involves three elements: a loss function to be optimized, a weak learner to make predictions and an additive model to minimize loss function. The additive model acts as a gradient differential to the unchanged models in step 2. Gradient boosting is a greedy algorithm and can overfit a training dataset quickly. Gradient Boosting algorithms are used where we need to decrease the Bias Error. All Boosting models are considered algorithm independent models since we can use any algorithm to improve the accuracy of the model by using multiple of them in conjunction.

4. RESULT AND ANALYSIS

The results of this experiment show a 97% area under the Receiver Operating Characteristic (ROC) curve (AUC score) with only 13 features and just 0.063 seconds time taken for training and testing per sample apk. Hence, we were successful in implementing an efficient classifier with few feature sets and low analysis time, along with unique feature grouping.

4.1 System Setup and Evaluation Criteria

The table below shows the computer architecture involved in performing this work. It is very useful in analysis time calculations.

Parame	Value
CPU	Intel Core i7-12650H
RAM	16 GB DDR4
Operating	Windows 11 Home
Python	3.11.1
Libraries	scikit-learn, matplotlib, pandas, seaborn, numpy,

Table-1 - System Requirements

4.2 Formulae

These are the formulae for precision, recall, f-1 score, and accuracy we have used for the performance evaluation of each ML model.

$$\text{precision} = \frac{TP}{TP + FP}$$

$$\text{recall} = \frac{TP}{TP + FN}$$

$$\text{f-score} = \frac{\frac{TP}{TP+FN} * \frac{TP}{TP+FP}}{\frac{TP}{TP+FN} + \frac{TP}{TP+FP}}$$

$$\text{accuracy} = \frac{TP + TN}{TP + TN + FP + FN}$$

where,

TP (True Positive)
= number of truly malicious samples from those predicted as malw are
FP (False Positive)
= number of samples that are predicted to be malware that are not tr uly malicious
FN (False Negative)
= number of samples that are predicted to be benign that are not trul y benign
TN (True Negative)
= number of truly benign samples from those predicted as benign

4.3 Hyper-parameter Tuning

After the testing was over for each model, we performed hyper-parameter tuning for all the classifiers. Algorithms used for that were GridSearch and RandomSearch to get the best results.

Classifier	Hyper-parameter tuning range	Selected values
KNeighbors	n_neighbors: (1,20,1) p: (1,5,1) weights: ('uniform', 'distance'),	'n_neighbors': 5 'p': 3 'weights': 'uniform'
GradientBoosting	loss: ['log_loss', 'deviance', 'exponential'] "learning_rate": [0.01, 0.025, 0.2]	loss:log_loss "learning_rate": 1.0
Ada Boost	'learning_rate': [0.01, 0.1, 1, 10], 'n_estimators': [50, 500, 2000]	'learning_rate': 1 'n_estimators': 50
Decision Tree	'criterion': ['gini', 'entropy'],	criterion='gini'
Logistic Regression	'C': [1.e-03, 1.e-02, 1.e-01, 1.e+00, 1.e+01, 1.e+02, 1.e+03], 'penalty': ['l1', 'l2']}	C': 1.0, 'penalty': 'l2'

Figure-3

Below are the screenshots of the output we got after running our code consisting of five ML algorithms on three datasets containing samples of benign and malicious applications. As we can see we got as high as 0.982 f-1 score with Adaptive Boosting. We have shown the confusion matrices for each algorithm involving TP, FP, FN, and TN.

```
algorithm is: LogisticRegression(max_iter=1000)
accuracy_score 0.921
confusion matrix
[[  85   92]
 [  28 1320]]
Predicted    0     1   All
True
0           85    92   177
1           28  1320  1348
All        113  1412  1525
Precision: 0.935
Recall: 0.979
F1-measure: 0.957
```

```
algorithm is: KNeighborsClassifier(n_neighbors=8, p=5)
accuracy_score 0.941
confusion matrix
[[  99   78]
 [  12 1336]]
Predicted    0     1   All
True
0           99    78   177
1           12  1336  1348
All        111  1414  1525
Precision: 0.945
Recall: 0.991
F1-measure: 0.967
```

```
algorithm is: DecisionTreeClassifier(random_state=80)
accuracy_score 0.959
confusion matrix
[[ 139   38]
 [  25 1323]]
Predicted    0     1   All
True
0          139    38   177
1           25  1323  1348
All        164  1361  1525
Precision: 0.972
Recall: 0.981
F1-measure: 0.977
```

```
algorithm is: AdaBoostClassifier()
accuracy_score 0.967
confusion matrix
[[ 137   40]
 [  10 1338]]
Predicted    0     1   All
True
0          137    40   177
1           10  1338  1348
All        147  1378  1525
Precision: 0.971
Recall: 0.993
F1-measure: 0.982
```

```
algorithm is: GradientBoostingClassifier()
accuracy_score 0.961
confusion matrix
[[ 126   51]
 [   9 1339]]
Predicted    0     1   All
True
0          126    51   177
1            9  1339  1348
All        135  1390  1525
Precision: 0.963
Recall: 0.993
F1-measure: 0.978
```

Figure-4

Here is a table summarizing the results.

ML model	AUC score	F-1 score (%)	Accuracy (%)	Precision (%)	Recall (%)
Adaptive Boosting	0.955	98.2	96.7	97.1	99.3
Gradient Boosting	0.965	97.8	96.1	96.3	99.3
Decision Tree	0.970	97.7	95.9	97.2	98.1
K-Neighbors	0.965	96.7	94.1	94.5	99.1
Logistic Regression	0.881	95.7	92.1	93.5	97.9

Table-2 - Results

As we can see that all the models have been very successful in predicting the malicious and benign apps, all achieving >95% f-1 scores and >90% accuracies. As a result, a very high AUC score of 97% was also achieved. This proves that it is not necessary for each permission to be an individual feature.

4.4 Cross-validation

"RepeatedStratifiedKFold" function was used for the process of cross-validation on the model to improve the results. We let n_splits = 10, n_repeats = 3, and random state = 123. As anticipated, Gradient Boosting and Decision Tree performed better than others at average classification rate.

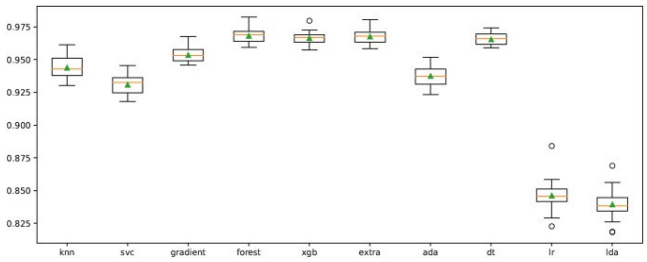


Figure-4

4.5 ROC curves and Time Comparisons

Here are the ROC curves of all the models used. A massive 97.0% was achieved in Decision Tree algorithm, which implies its high capability in malware detection. The effect of the feature grouping approach proposed is strengthened when we get a high classification value in various classifiers.

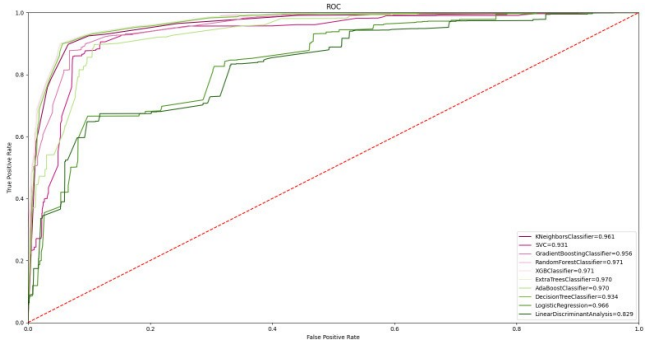


Figure-5

5. RELATED WORK

A lot of work has been done in recent years on Android malware detection. These studies basically used static analysis, dynamic analysis, or hybrid feature extraction methods. While some of these obtained features contributed positively to the classification performance, some may have had no effect at all, and some may have had a deteriorating effect. For this reason, it is beneficial to determine those features that contribute positively to the result and to remove the others from the feature set. In this study, a feature grouping method was proposed for the use of features extracted from static analysis in the classification. We used the feature vector consisting of grouped features and made classification with the ExtraTree algorithm. Instead of selecting and removing features from the feature vector, the approach of evaluating these features within the group was adopted. The Drebin malicious dataset, which has been widely used for many years, was used in the tests.

Paper	Dataset	Feature extraction	Feature selection or grouping algorithms and classification methods	Classification Performance	Sec. for identification each app
Ratibah Tuan Mat et al. (2021)	Androzoo, Drebin	Permissions	Naïve Bayesian	91.10%	–
Mohammed Arif et al. (2021)	Androzoo, Drebin	Permissions	MCDM	90.54% (4 classification levels)	–
Millar et al. (2021)	Drebin, Genome	Opcode and permissions	No feature selection, classification with multi-view deep learning	91.00%	0.008
Zhang et al. (2021)	Drebin	Text sequences of apps	No feature selection. Classification with Text CNN	95.20%	0.28
Arp et al. (2014)	Drebin	Used permissions, sys. Api calls, network address	Machine learning	94%	10
Anastasia (Fereidooni et al., 2016)	Own dataset	Api calls, network address	ML(NB, RF, KNN)	96%	0.29
MamaDroid (Onwuzurike et al., 2019)	Drebin	Api calls, call graphs	SVM, RF, 1-NN, 3-NN	87%	0.7 ± 1.5
Taheri et al. (2020)	Drebin, Genome	Api calls, intents, permissions(21492 features)	FNN, ANN, WANN, KMNN	90%–99%	Very high
Apkauditor (Kabakus, Doğru & Çetin, 2015)	Own dataset	Permissions, services, receivers	Signature based	92.5%	–
Syrris & Geneiatakis (2021b)	Drebin	Static features	ML(6 six classifiers)	99%	–
Droidmat (Wu et al., 2012)	Own dataset	Intents, Api calls	Signature based	91.83%	–
Alazab et al. (2020)	Own dataset	Api calls	ML(RF, J48, KNN, NB)	94.30%	0.2 –0.92
Pektaş & Acarman (2020)	Drebin, AMD, Androzoo	Api calls	SDNE(DNN model)	98.5%	–
Shehata et al. (2020)	Own dataset	Activities, services, receivers, providers, permissions	RF	97.1%	–
Thiyagarajan, Akash & Murugan (2020)	Androzoo	Permissions(113) –>PCA (10)	DT with PCA feature selection	94.3%	–

Figure-6

6. CONCLUSION

In this study, we saw how the number of malicious or malware-containing applications are increasing day by day. For this, we have built a classifier which can predict if an Android application is benign or malicious. We looked at the structure of an app in detail and how it comes in a package with an apk extension. We have used features like permissions, application size, and the number of icons, to make that prediction as accurate as possible. We have also performed feature grouping which proved that it increases the accuracy of the models. Speaking of the models, we implemented some well-known ML techniques, and the best came out to be Adaptive Boosting, with an f-1 score of 98.2%. We also added the ROC curves and calculated the AOC scores, implemented cross-validation, and did time analysis.