CSE 3501 - INFORMATION SECURITY ANALYSIS AND AUDIT

PROJECT REPORT

ANDROID MALWARE ANALYSIS

SUBMITTED TO

PROF Umadevi K S

By

Harshit Mishra (19BCE0799)
Kartik Goel (19BCE2002)
Alokam Nikhitha (19BCE2555)

Abstract

Android is an open source operating system with more than a billion users. The amount of sensitive information produced by these technologies are rapidly increasing, which attracts a large number of audiences to develop tools and techniques to acquire that information or to disrupt the device's smooth operation. Despite several solutions being able to guarantee an adequate level of security, day by day the hacker's skills continued to grow, so it remains a permanent challenge for security tools developers.

As a response, several members of the research community are using artificial intelligence tools for Android security, particularly machine learning techniques to classify between healthy and malicious apps.

In this project, we implemented a static analysis framework and machine learning to do that classification.

Introduction

Android is an open-source operating system for mobile devices, televisions, automobiles and smart watches with more than a billion users. Therefore it opens a wide array of attack vectors targeting the user information.

For the protection of the information and devices, Android has several security mechanisms; the most relevant are: a sandbox environment at the kernel level to prevent access to the file system and other resources; an API of permissions that controls the privileges of the applications in the device; security mechanisms at the applications development level; and a digital distribution platform (Google Play), where the processes are implemented to limit the dissemination of malicious code.

Each application is compiled in an Android Application Package [APK] file, which includes the code of the application (".dex" files), resources, and the AndroidManifest.xml file. This latter is an important element, since it provides most of the information of the security features and configuration of each application. It also includes the information of the API regarding permissions, activities, services, content providers, and the receiving broadcasts.

There are several tools and techniques for the analysis of threats for this operating system. Between the most representative, we have static and dynamic analysis.

Static Analysis

Static analysis is a technique that assesses malicious behavior in the source code, the data, or the binary files without the direct execution of the application. Its complexity has increased due to the experience that cybercriminals have gained in the development of applications. However, it has been demonstrated that it is possible to avoid this using obfuscation techniques.

Dynamic Analysis

Dynamic analysis is a set of methods that studies the behavior of the malware in execution through gesture simulations. In this technique, the processes in execution, the user interface, the network connections and sockets opening are analyzed. Alternatively, there already exist some techniques to avoid the processes performed by dynamic analysis, where the malware has the capacity to detect sandbox-like environments and to stop its malicious behavior.

Literature Survey

A. Machine learning with dynamic analysis

DroidDolphin(2014) is a dynamic analysis framework that uses the GUI, big data, and machine learning for the detection of malicious applications in Android. Its analysis process consists of the extraction of information of the calls to the API and 13 activities, whilst the application executed in virtual environments. The SVM machine learning algorithm with the LIBSVM public library was used, a training dataset of 32,000 benign and malign applications, and a testing dataset of 3,000 healthy applications and 1,000 malicious. The preliminary results showed a precision of 86.1% and an F-score of 0.875.

Wu, Wen-Chieh & Hung, Shih-Hao. (2014). DroidDolphin. 247-252. 10.1145/2663761.2664223.

B. Machine learning with static analysis

Sahs and Khan (2012) propose a machine learning system to detect malware in Android devices with the One-Class SVM classification algorithm and static analysis as a technique to obtain the information from the applications. During

the development, they used the Androguard tool to extract the information of the APK and the Scikit-learn framework. From the AndroidManifest.xml, they developed a binary vector that contains the information of each permission used for every application and a Control Flow Graph [CFG], which corresponds to an abstract representation of a program. Conversely, the algorithm was implemented with a test set of 2,081 clean apps and 91 malicious, with the information of 5 kernels - about binary vectors, strings, diagrams, sets, and uncommon permissions - and another one for each application. As a result, they were able to obtain a low rate of false negatives but a high rate of false positives.

J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," 2012 European Intelligence and Security Informatics Conference, Odense, 2012, pp. 141-147, doi: 10.1109/EISIC.2012.34.

C. DroidMat: Android Malware Detection through Manifest and API Calls Tracing

Although understanding Android malware using dynamic analysis can provide a comprehensive view, it is still subjected to high cost in environment deployment and manual efforts in investigation. They have proposed a static feature-based mechanism to provide a static analyst paradigm for detecting the Android malware. The mechanism considers the static information including permissions, deployment of components, Intent messages passing and API calls for characterizing the Android applications behavior. In order to recognize different intentions of Android malware, different kinds of clustering algorithms can be applied to enhance the malware modeling capability. They have proposed a mechanism and developed a system, called Droid Mat. First, the Droid Mat extracts the information (e.g., requested permissions, Intent messages passing, etc) from each application's manifest file, and regards components (Activity, Service, Receiver) as entry points drilling down for tracing API Calls related to permissions. Next, it applies K-means algorithm that enhances the malware modeling capability. The number of clusters are decided by Singular Value Decomposition (SVD) method on the low rank approximation. Finally, it uses KNN algorithm to classify the application as benign or malicious. The experiment result shows that the recall rate of our approach is better than one of well-known tool, Androguard, published in Black hat 2011, which focuses on Android malware analysis. In addition, Droid Mat is efficient since it takes only half as much time than Androguard to predict 1738 apps as benign apps or Android malware.

D.Wu, C. Mao, T. Wei, H. Lee and K. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," 2012 Seventh Asia Joint Conference on

Information Security, Tokyo, 2012, pp. 62-69, doi: 10.1109/AsiaJCIS.2012.18 D. Machine learning aided Android malware classification

In this paper they have proposed two machine learning aided approaches for static analysis of Android malware. The first approach is based on permissions and the other is based on source code analysis utilizing a bag-of-words representation model. The permission-based model is computationally inexpensive, and is implemented as the feature of OWASP Seraphimdroid Android app that can be obtained from Google Play Store. Our evaluations of both approaches indicate an F-score of 95.1% and F-measure of 89% for the source code-based classification and permission-based classification models, respectively.

Milosevic, N., Dehghantanha, A., & Choo, K. K. R. (2017). Machine learning aided Android malware classification. Computers & Electrical Engineering, 61, 266-274.

E. Droid permission miner: Mining prominent permissions for Android malware analysis

In this paper, they have proposed static analysis of android malware files by mining prominent permissions. The proposed technique is implemented by extracting permissions from 436 .apk files. Feature pruning is carried out to investigate the impact of feature length on accuracy. The prominent features that give way to lesser misclassification are determined using Bi-Normal Separation (BNS) and Mutual Information (MI) feature selection techniques. Results suggest that Droid permission miner can be used for preliminary classification of Android package files.

A. M. Aswini and P. Vinod, "Droid permission miner: Mining prominent permissions for Android malware analysis," The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014), Bangalore, 2014, pp. 81-86, doi: 10.1109/ICADIWT.2014.681467

Proposed work

Other artificial intelligence techniques have also been explored for safety in Android; one is so-called machine learning. Here, the capacity of the systems to learn how to identify malware without being programmed in an explicit way is presented. A large number of proposals use classification algorithms like SVM, Bagging, Neural Network, Decision Tree, and Naive Bayes. Conventional security software requires, on its identification process, human effort that implies time and resources. This might be improved through the use of intelligent tools. One of the more promising paths is the application of machine learning algorithms, which entails an increase in the efficiency in the identification of new threats.

At the present time, some proposals have arisen related to the application of classifiers for the analysis of malicious software in Android; a large number use tools for the download of applications and technologies; hence, the comparison of their results is more complicated.

Having as a foundation a framework of static analysis, we propose an adaptation that uses application databases in published articles with different open-source tools. Additionally, we assessed the framework through the application of several classification algorithms: Naive Bayes, KNeighbors, Random Forest , and Decision Tree.

The proposed framework is composed of four phases: gathering, extraction, features generation, and training and testing.

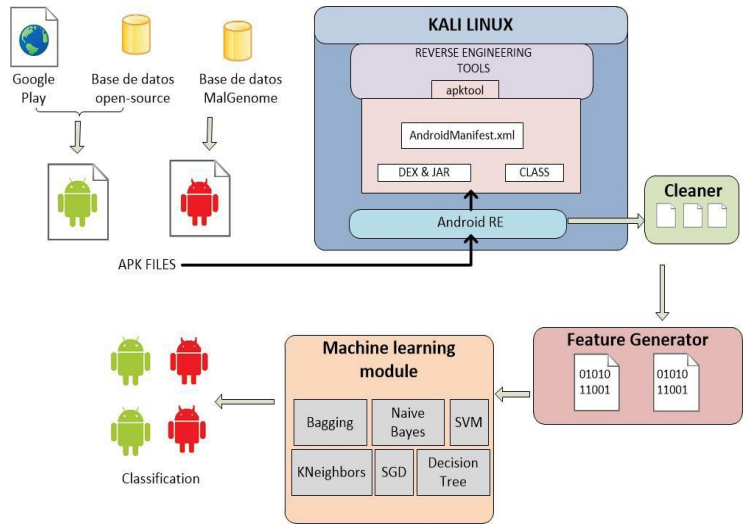


Fig. Proposed framework

Gathering

We collected a total of 558 APK. The collection is composed of 279 applications with low privileges of the free access dataset and a random selection of 279 malwares of the MalGenome.

Extraction

By using ApkTool 2.0.3 tool, we can obtain the AndroidManifest.xml file of the gathered APK.

Features generator

We developed some tools in Python that create the binary vectors that correspond to the permissions for each application; also, we added a layer regarding the classification of the application (either benign or malign). With these results, we created a new dataset containing the granted permissions for each application in binary.

Training and testing

We created two partitions, one with 71% and the other with 29% of the data for the training and testing phases, respectively. Finally, we trained and verified each machine learning algorithm.

Dataset

Static Analysis

Earlier, a malicious dataset software known as Android Genome Project - MalGenome – was published. The development has a sample of 1,260 malicious applications with 49 different families. The features description of its dataset was also published and, as a part of the results, 1,083 malicious codes (86%) were re-packaged versions of legitimate applications, 36.7% had the capacity to raise privileges, and 45.3% had at their end the subscription of premium message systems. They assessed four security software that in the best case could detect 79.6% of the malicious applications; in the worst case, 20.2%.

Krutz et al. (2015) analyzed a free-access dataset containing more than 1,000 Android applications, more than 4,000 of their versions, and a total of 430,000 commits of online repositories. In addition to their results, it is possible to obtain information regarding the applications under and over privileges per commit category in the repositories and versions.

Dynamic Analysis

For this approach, we used a set of pcap files from the DroidCollector project integrated by 4705 benign and 7846 malicious applications. All of the files were processed by our feature extractor script, the idea of this analysis is to answer the next question, according to the static analysis previously seen a lot of applications use a network connection, in other words, they are trying to communicate or transmit information, so.. is it possible to distinguish between malware and benign application using network traffic.

Implementation

We used Python as a development language, and the machine learning tools of Scikit-Learn 0.17. The selected classification algorithms for the experiment were: Naive Bayes, Bagging, KNeighbors, Support Vector Machines (SVM), Stochastic Gradient Descent (SGD), and Decision Tree. During the application of the methodology, we assessed some algorithms with different configurations

Importing the libraries

```
#Importing all libraries required
from sklearn.naive_bayes import GaussianNB, BernoulliNB
from sklearn.metrics import accuracy_score, classification_report
from sklearn.ensemble import BaggingClassifier
from sklearn.neighbors import KNeighborsClassifier
from sklearn.linear_model import SGDClassifier
from sklearn.model_selection import train_test_split
from sklearn.metrics import cohen_kappa_score
from sklearn.metrics import confusion_matrix
from sklearn.ensemble import RandomForestClassifier

from sklearn import preprocessing
import torch
from sklearn import svm
from sklearn import tree
import pandas as pd
from sklearn.externals import joblib
import pickle
import numpy as np
import seaborn as sns
```

Static Analysis

```
# Doing Static Analysis
import pandas as pd
df = pd.read_csv("train.csv", sep=";")
```

```
df = df.astype("int64")
df.type.value_counts()
```

```
1    199
0    199
Name: type, dtype: int64
```

```
df.shape
```

```
(398, 331)
```

Sorting the dataset by different types: Type 1 means malware

```
pd.Series.sort_values(df[df.type==1].sum(axis=0), ascending=False)[1:11]
```

```
android.permission.INTERNET                 195
android.permission.READ_PHONE_STATE         190
android.permission.ACCESS_NETWORK_STATE     167
android.permission.WRITE_EXTERNAL_STORAGE   136
android.permission.ACCESS_WIFI_STATE        135
android.permission.READ_SMS                 124
android.permission.WRITE_SMS                104
android.permission.RECEIVE_BOOT_COMPLETED   102
android.permission.ACCESS_COARSE_LOCATION    80
android.permission.CHANGE_WIFI_STATE         75
dtype: int64
```

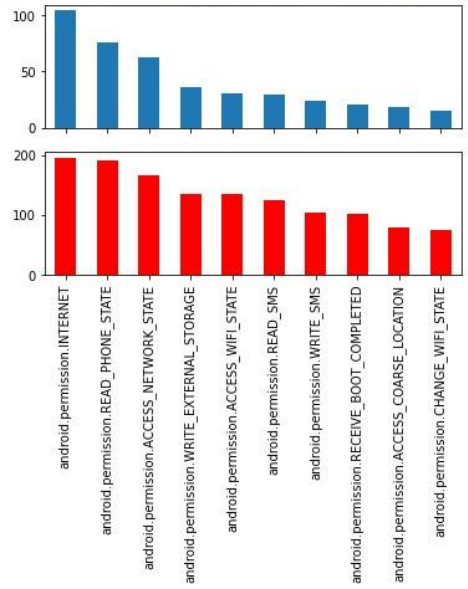
```
pd.Series.sort_values(df[df.type==0].sum(axis=0), ascending=False)[:10]
```

```
android.permission.INTERNET                 104
android.permission.WRITE_EXTERNAL_STORAGE    76
android.permission.ACCESS_NETWORK_STATE      62
android.permission.WAKE_LOCK                 36
android.permission.RECEIVE_BOOT_COMPLETED    30
android.permission.ACCESS_WIFI_STATE         29
android.permission.READ_PHONE_STATE          24
android.permission.VIBRATE                   21
android.permission.ACCESS_FINE_LOCATION      18
android.permission.READ_EXTERNAL_STORAGE     15
dtype: int64
```

```
import matplotlib.pyplot as plt
fig, axs =  plt.subplots(nrows=2, sharex=True)

pd.Series.sort_values(df[df.type==0].sum(axis=0), ascending=False)[:10].plot.bar(ax=axs[0])
pd.Series.sort_values(df[df.type==1].sum(axis=0), ascending=False)[1:11].plot.bar(ax=axs[1], color="red")
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f02246756a0>
```



Modelling and Naive Bayes

```
#Modelling
X_train, X_test, y_train, y_test = train_test_split(df.iloc[:, 1:330], df['type'], test_size=0.20, random_state=42)
```

```
# Naive Bayes algorithm
gnb = GaussianNB()
gnb.fit(X_train, y_train)

# pred
pred = gnb.predict(X_test)

# accuracy
accuracy = accuracy_score(pred, y_test)
print("naive_bayes")
print(accuracy)
print(classification_report(pred, y_test, labels=None))
```

```
naive_bayes
0.8375
              precision    recall  f1-score   support

           0       0.91      0.76      0.83        41
           1       0.78      0.92      0.85        39

    accuracy                           0.84        80
   macro avg       0.85      0.84      0.84        80
weighted avg       0.85      0.84      0.84        80
```

K Neighbors Algorithm

```
# kneighbors algorithm

for i in range(3,15,3):

    neigh = KNeighborsClassifier(n_neighbors=i)
    neigh.fit(X_train, y_train)
    pred = neigh.predict(X_test)
    # accuracy
    accuracy = accuracy_score(pred, y_test)
    print("kneighbors {}".format(i))
    print(accuracy)
    print(classification_report(pred, y_test, labels=None))
    print("")
```

```
kneighbors 3
0.8875
              precision    recall  f1-score   support

           0       0.94      0.82      0.88        39
           1       0.85      0.95      0.90        41

    accuracy                           0.89        80
   macro avg       0.89      0.89      0.89        80
weighted avg       0.89      0.89      0.89        80


kneighbors 6
0.85
              precision    recall  f1-score   support

           0       0.94      0.76      0.84        42
           1       0.78      0.95      0.86        38

    accuracy                           0.85        80
   macro avg       0.86      0.85      0.85        80
weighted avg       0.87      0.85      0.85        80
```

```
kneighbors 9
0.8375
              precision    recall  f1-score   support

           0       0.94      0.74      0.83        43
           1       0.76      0.95      0.84        37

    accuracy                           0.84        80
   macro avg       0.85      0.85      0.84        80
weighted avg       0.86      0.84      0.84        80


kneighbors 12
0.825
              precision    recall  f1-score   support

           0       0.91      0.74      0.82        42
           1       0.76      0.92      0.83        38

    accuracy                           0.82        80
   macro avg       0.84      0.83      0.82        80
weighted avg       0.84      0.82      0.82        80
```

Decision tree

```
#Decision Tree
clf = tree.DecisionTreeClassifier()
clf.fit(X_train, y_train)

# Read the csv test file

pred = clf.predict(X_test)
# accuracy
accuracy = accuracy_score(pred, y_test)
print(clf)
print(accuracy)
print(classification_report(pred, y_test, labels=None))
```

```
DecisionTreeClassifier(ccp_alpha=0.0, class_weight=None, criterion='gini',
                       max_depth=None, max_features=None, max_leaf_nodes=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, presort='deprecated',
                       random_state=None, splitter='best')
0.95
              precision    recall  f1-score   support

           0       0.97      0.92      0.94        36
           1       0.93      0.98      0.96        44

    accuracy                           0.95        80
   macro avg       0.95      0.95      0.95        80
weighted avg       0.95      0.95      0.95        80
```

Dynamic Analysis

```
#Dynamic Analysis
import pandas as pd
data = pd.read_csv("android_traffic.csv", sep=";")
data.head()
```

	name	tcp_packets	dist_port_tcp	external_ips	vulume_bytes	udp_packets	tcp_urg_packet	source_app_packets	remote_app_packets	source_app_bytes	remote_
0	AntiVirus	36	6	3	3911	0	0	39	33	5100	
1	AntiVirus	117	0	9	23514	0	0	128	107	26248	
2	AntiVirus	196	0	6	24151	0	0	205	214	163887	
3	AntiVirus	6	0	1	889	0	0	7	6	819	
4	AntiVirus	6	0	1	882	0	0	7	6	819	

```
data.columns
```

```
Index(['name', 'tcp_packets', 'dist_port_tcp', 'external_ips', 'vulume_bytes',
       'udp_packets', 'tcp_urg_packet', 'source_app_packets',
       'remote_app_packets', 'source_app_bytes', 'remote_app_bytes',
       'duracion', 'avg_local_pkt_rate', 'avg_remote_pkt_rate',
       'source_app_packets.1', 'dns_query_times', 'type'],
      dtype='object')
```

```
data.shape
```

```
(7845, 17)
```

```
data.type.value_counts()
```

```
benign       4704
malicious    3141
Name: type, dtype: int64
```

Data Cleaning and Processing

```
#Data Cleaning and processing
data.isna().sum()
```

```
name                      0
tcp_packets               0
dist_port_tcp             0
external_ips              0
vulume_bytes              0
udp_packets               0
tcp_urg_packet            0
source_app_packets        0
remote_app_packets        0
source_app_bytes          0
remote_app_bytes          0
duracion               7845
avg_local_pkt_rate     7845
avg_remote_pkt_rate    7845
source_app_packets.1      0
dns_query_times           0
type                      0
dtype: int64
```

```
#dropping the columns with high null values
data = data.drop(['duracion','avg_local_pkt_rate','avg_remote_pkt_rate'], axis=1).copy()
```

Statistical Analysis of data

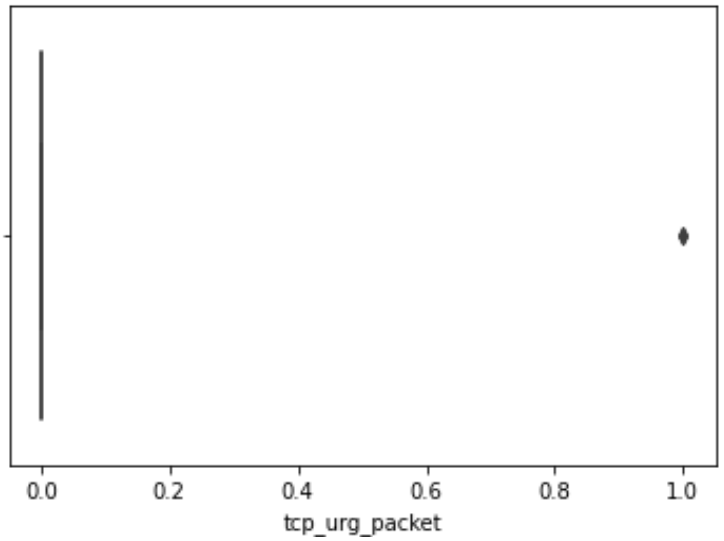
```
#Show all the statistical Analysis of the data
data.describe()
```

	tcp_packets	dist_port_tcp	external_ips	vulume_bytes	udp_packets	tcp_urg_packet	source_app_packets	remote_app_packets	source_app_bytes	remote_app_
count	7845.000000	7845.000000	7845.000000	7.845000e+03	7845.000000	7845.000000	7845.000000	7845.000000	7.845000e+03	7.84500
mean	147.578713	7.738177	2.748502	1.654375e+04	0.056724	0.000255	152.911918	194.706310	2.024967e+05	1.69226
std	777.920084	51.654222	2.923005	8.225650e+04	1.394046	0.015966	779.034618	1068.112696	1.401076e+06	8.23818
min	0.000000	0.000000	0.000000	0.000000e+00	0.000000	0.000000	1.000000	0.000000	0.000000e+00	6.90000
25%	6.000000	0.000000	1.000000	8.880000e+02	0.000000	0.000000	7.000000	7.000000	9.340000e+02	1.04600
50%	25.000000	0.000000	2.000000	3.509000e+03	0.000000	0.000000	30.000000	24.000000	4.090000e+03	3.80300
75%	93.000000	0.000000	4.000000	1.218900e+04	0.000000	0.000000	98.000000	92.000000	2.624400e+04	1.26100
max	37143.000000	2167.000000	43.000000	4.226790e+06	65.000000	1.000000	37150.000000	45928.000000	6.823516e+07	4.22732

Checking for Outliers

```
#checking the outliers in the data
sns.boxplot(data.tcp_urg_packet)
```

```
/usr/local/lib/python3.6/dist-packages/seaborn/_decorator
  FutureWarning
<matplotlib.axes._subplots.AxesSubplot at 0x7fe8d2196128>
```




```
data.loc[data.tcp_urg_packet > 0].shape[0]
```

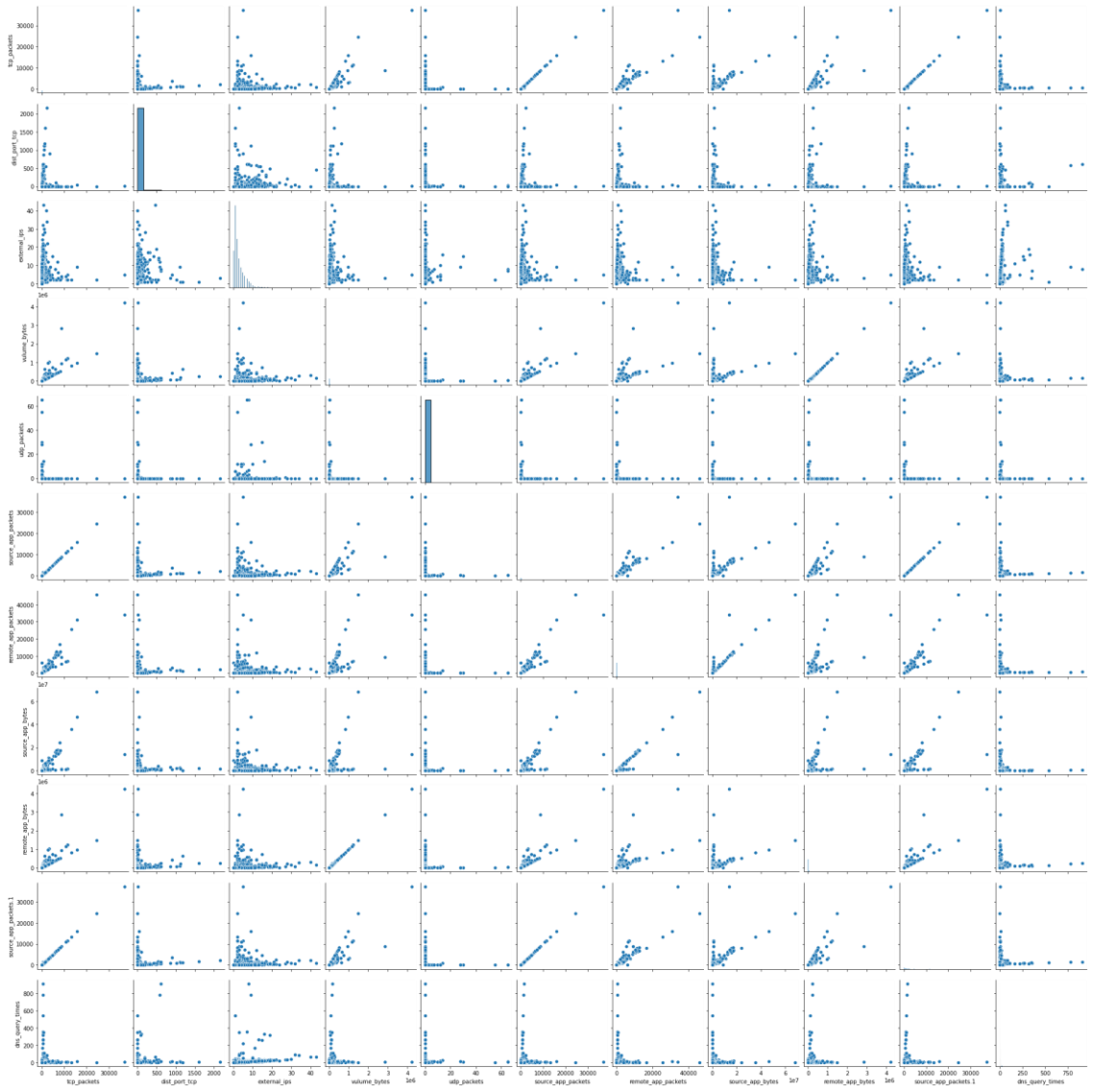
```
2
```

```
#Dropping the column 'tcp_urg_packet' since it has only two columns which are non-zero
data = data.drop(columns=["tcp_urg_packet"], axis=1).copy()
data.shape
```

```
(7845, 13)
```

Pairplots

```
#Pairplots
sns.pairplot(data)
```




```
scaler = preprocessing.RobustScaler()
scaledData = scaler.fit_transform(data.iloc[:,1:11])
scaledData = pd.DataFrame(scaledData, columns=['tcp_packets','dist_port_tcp',
                                               'external_ips','vulume_bytes','udp_packets','source_app_packets',
                                               'remote_app_packets',' source_app_bytes','remote_app_bytes',
                                               'dns_query_times'])
```

```
#Modelling
X_train, X_test, y_train, y_test = train_test_split(scaledData.iloc[:,0:10], data.type.astype("str"), test_size=0.25, random_state=45)
```

```
data=data[data.tcp_packets<20000].copy()
data=data[data.dist_port_tcp<1400].copy()
data=data[data.external_ips<35].copy()
data=data[data.vulume_bytes<2000000].copy()
data=data[data.udp_packets<40].copy()
data=data[data.remote_app_packets<15000].copy()
```

```
data[data.duplicated()].sum()
```

```
name                  AntiVirusAntiVirusAntiVirusAntiVirusAntiVirusA...
tcp_packets                                                       15038
dist_port_tcp                                                      3514
external_ips                                                       1434
vulume_bytes                                                    2061210
udp_packets                                                          38
source_app_packets                                                21720
remote_app_packets                                                18841
source_app_bytes                                                8615120
remote_app_bytes                                                2456160
source_app_packets.1                                              21720
dns_query_times                                                    5095
type                  benignbenignbenignbenignbenignbenignbenignbeni...
dtype: object
```

```
data=data.drop('source_app_packets.1',axis=1).copy()
```

Naive Bayes Algorithm

```
#Naive Bayes Algorithm
gnb = GaussianNB()
gnb.fit(X_train, y_train)
pred = gnb.predict(X_test)
## accuracy
accuracy = accuracy_score(y_test,pred)
print("naive_bayes")
print(accuracy)
print(classification_report(y_test,pred, labels=None))
print("cohen kappa score")
print(cohen_kappa_score(y_test, pred))
```

```
naive_bayes
0.44688457609805926
              precision    recall  f1-score   support

      benign       0.81      0.12      0.20      1190
   malicious       0.41      0.96      0.58       768

    accuracy                           0.45      1958
   macro avg       0.61      0.54      0.39      1958
weighted avg       0.66      0.45      0.35      1958

cohen kappa score
0.06082933470572538
```

KNeighbors Algorithm

```
# kneighbors algorithm

for i in range(3,15,3):

    neigh = KNeighborsClassifier(n_neighbors=i)
    neigh.fit(X_train, y_train)
    pred = neigh.predict(X_test)
    # accuracy
    accuracy = accuracy_score(pred, y_test)
    print("kneighbors {}".format(i))
    print(accuracy)
    print(classification_report(pred, y_test, labels=None))
    print("cohen kappa score")
    print(cohen_kappa_score(y_test, pred))
    print("")
```

```
kneighbors 3
0.8861082737487231
              precision    recall  f1-score   support

      benign       0.90      0.91      0.91      1173
   malicious       0.87      0.85      0.86       785

    accuracy                           0.89      1958
   macro avg       0.88      0.88      0.88      1958
weighted avg       0.89      0.89      0.89      1958

cohen kappa score
0.7620541314671169

kneighbors 6
0.8784473953013279
              precision    recall  f1-score   support

      benign       0.92      0.88      0.90      1240
   malicious       0.81      0.87      0.84       718

    accuracy                           0.88      1958
   macro avg       0.87      0.88      0.87      1958
weighted avg       0.88      0.88      0.88      1958

cohen kappa score
0.7420746759356631
```

```
kneighbors 9
0.8707865168539326
              precision    recall  f1-score   support

      benign       0.89      0.90      0.89      1175
   malicious       0.85      0.83      0.84       783

    accuracy                           0.87      1958
   macro avg       0.87      0.86      0.86      1958
weighted avg       0.87      0.87      0.87      1958

cohen kappa score
0.729919255030886

kneighbors 12
0.8615934627170582
              precision    recall  f1-score   support

      benign       0.88      0.89      0.89      1185
   malicious       0.83      0.82      0.82       773

    accuracy                           0.86      1958
   macro avg       0.86      0.85      0.86      1958
weighted avg       0.86      0.86      0.86      1958

cohen kappa score
0.7100368862537227
```

Random Forest

```
#Random Forest Algorithm
rdF=RandomForestClassifier(n_estimators=250, max_depth=50,random_state=45)
rdF.fit(X_train,y_train)
pred=rdF.predict(X_test)
cm=confusion_matrix(y_test, pred)
accuracy = accuracy_score(y_test,pred)
print(rdF)
print(accuracy)
print(classification_report(y_test,pred, labels=None))
print("cohen kappa score")
print(cohen_kappa_score(y_test, pred))
print(cm)
```

```
RandomForestClassifier(bootstrap=True, ccp_alpha=0.0, class_weight=None,
                       criterion='gini', max_depth=50, max_features='auto',
                       max_leaf_nodes=None, max_samples=None,
                       min_impurity_decrease=0.0, min_impurity_split=None,
                       min_samples_leaf=1, min_samples_split=2,
                       min_weight_fraction_leaf=0.0, n_estimators=250,
                       n_jobs=None, oob_score=False, random_state=45, verbose=0,
                       warm_start=False)
0.9172625127681308
              precision    recall  f1-score   support

      benign       0.93      0.94      0.93      1190
   malicious       0.90      0.88      0.89       768

    accuracy                           0.92      1958
   macro avg       0.91      0.91      0.91      1958
weighted avg       0.92      0.92      0.92      1958

cohen kappa score
0.8258206083396299
[[1117   73]
 [  89  679]]
```

Results and Discussion

Here, we are going to compare between all the algorithms we have used for both static analysis and dynamic analysis.

Static analysis

Algorithm	precision		recall		f1-score		accuracy
	0	1	0	1	0	1	
Naive Bayes	0.91	0.78	0.76	0.92	0.83	0.85	0.84
Kneighbors	0.94	0.85	0.82	0.95	0.88	0.90	0.89
Decision tree	0.97	0.93	0.92	0.98	0.94	0.96	0.95

Table 1. Static Analysis

Dynamic analysis

Algorithm	precision		recall		f1-score		accuracy	cohen kappa score
	B	M	B	M	B	M		
Naive Bayes	0.81	0.41	0.12	0.96	0.20	0.58	0.45	0.0608
Kneighbors	0.90	0.87	0.91	0.85	0.91	0.86	0.89	0.762
Random forest	0.93	0.90	0.94	0.88	0.93	0.89	0.92	0.825

Table 2 . Dynamic Analysis

As we can see that in static analysis, the decision tree had the highest result in the classification performance of 97% and 93% respectively for Benign and Malicious apps. Whereas in dynamic analysis, random forest had the highest result in the classification performance of 93% and 90% respectively.

We have successfully implemented the static and dynamic analysis frameworks for classifying between the malware and benign apps by using several machine learning algorithms. By the obtained results and the precision scores we can classify the future apps between benign and malware.

References

[1] Wu, Wen-Chieh & Hung, Shih-Hao. (2014). DroidDolphin. 247-252. 10.1145/2663761.2664223.

[2] J. Sahs and L. Khan, "A Machine Learning Approach to Android Malware Detection," 2012 European Intelligence and Security Informatics Conference, Odense, 2012, pp. 141-147, doi: 10.1109/EISIC.2012.34.

[3] D. Wu, C. Mao, T. Wei, H. Lee and K. Wu, "DroidMat: Android Malware Detection through Manifest and API Calls Tracing," 2012 Seventh Asia Joint Conference on Information Security, Tokyo, 2012, pp. 62-69, doi: 10.1109/AsiaJCIS.2012.18.

[4] Milosevic, N., Dehghantanha, A., & Choo, K. K. R. (2017). Machine learning aided Android malware classification. Computers & Electrical Engineering, 61, 266-274.

[5] A. M. Aswini and P. Vinod, "Droid permission miner: Mining prominent permissions for Android malware analysis," The Fifth International Conference on the Applications of Digital Information and Web Technologies (ICADIWT 2014), Bangalore, 2014, pp. 81-86, doi: 10.1109/ICADIWT.2014.6814679.

[6] D. E. Krutz et al., "A Dataset of Open-Source Android Applications," 2015 IEEE/ACM 12th Working Conference on Mining Software Repositories, Florence, 2015, pp. 522-525, doi: 10.1109/MSR.2015.79.

[7] Wen-Chieh Wu and Shih-Hao Hung. 2014. DroidDolphin: a dynamic Android malware detection framework using big data and machine learning. In Proceedings of the 2014 Conference on Research in Adaptive and Convergent Systems (RACS '14). Association for Computing Machinery, New York, NY, USA, 247–252. DOI:https://doi.org/10.1145/2663761.2664223

[8] Urcuqui López, Christian & Navarro, Andres. (2016). Machine learning classifiers for android malware analysis. 1-6. 10.1109/ColComCon.2016.7516385.

[9] Batyuk, Leonid & Herpich, Markus & Camtepe, Seyit & Camtepe, Karsten & Raddatz, Aubrey- automatic assessment and mitigation of unwanted and malicious activities within Android