CHAPTER 1

INTRODUCTION

In our daily life Mobile Applications have become an essential part since countless facilities are providing to us by using Mobile Apps. It will change the way of communication, as the apps are installed on most of the smart devices. Mobile devices have refined sensors like cameras, gyroscopes, microphones and GPS. These several sensors open up entire innovative world of applications for the users and create massive quantities of data containing highly complex data.

Security solutions are therefore needed to defend operators from malicious applications that exploit the complexity of smart devices and their complex data. Android OS physically grows through the power of a wide range of smart devices. In mobile computing industry, it has largest part with 85% in 2017 due to its vulnerable source distribution [1].

Currently on Android platforms to defend against malware is a risky communication system that notifies users for the required permissions earlier each application is installed. This system is slightly ineffective because it offers permissions on its personal. To distinguish malware from benign applications, the user want excessively much methodical knowledge. The same permissions are required for the both benign and malicious application, consequently we cannot be distinguished by this permission based system. Generally, the permission based methodologies are largely not developed for the detection of malware, but it is used for the risk assessment.

The Android Operating System make malware more difficult for the installation and execution, because of the Android itself provide a several security solution for example Android permission and Google’s Bouncer to address the progressively widespread security threats. Every Android application need to ask the user for the permission to execute certain task on Android devices, such as transfer SMS message, during the installation process.

Most of the users are allow the permission without even considering what kinds of permissions they demand thus the Android permission system is knowingly weaken. Accordingly, the Android permission system spread the malicious apps itself and it is very challenging in training

1.1 AIM OF THE PROJECT

- The main advantage of the image conversion is that it can handle the packed malware without even unpacking it.
- Deep Neural Network and Artificial Neural Networks can be used.DNN is used on images.
- ANN can be used after flattening the last layer of DNN.
- ANN is used to detect recurring patterns.

1.2 SCOPE OF THE PROJECT

- First malware binary is converted into image. Apply DNN on the image.
- As last layer in DNN is in form of n-dimensional matrix we will flatten it.
- Flattening means we convert n-dimensional matrix into 1 dimensional column matrix.
- Converting Malware binary to Image, Applying DNN to Malware image, Converting DNN into Flatten.
- Applying BiLSTM to classify the input are the input data set.

CHAPTER 2

LITERATURE SURVEY

TITLE 1:Automatic framework for android malware detection using deep learning

AUTHORS: Elmouatez Karbab, Mourad Debbabi, Abdelouahid Derhab and Djedjiga Mouheb

YEAR: 2018

Android OS experiences a blazing popularity since the last few years. This predominant platform has established itself not only in the mobile world but also in the Internet of Things (IoT) devices. This popularity, however, comes at the expense of security, as it has become a tempting target of malicious apps. Hence, there is an increasing need for sophisticated, automatic, and portable malware detection solutions. In this paper, we propose MalDozer, an automatic Android malware detection and family attribution framework that relies on sequences classification using deep learning techniques. Starting from the raw sequence of the app's API method calls, MalDozer automatically extracts and learns the malicious and the benign patterns from the actual samples to detect Android malware. MalDozer can serve as a ubiquitous malware detection system that is not only deployed on servers, but also on mobile and even IoT devices. We evaluate MalDozer on multiple Android malware datasets ranging from 1 K to 33 K malware apps, and 38 K benign apps. The results show that MalDozer can correctly detect malware and attribute them to their actual families with an *F1-Score* of 96%–99% and a *false positive* rate of 0.06%–2%, under all tested datasets and settings.

TITLE 2:Android malware characterization and detection using deep learning

AUTHORS: Zhenlong Yuan, Yongqiang Lu and Yibo Xue

YEAR: 2016

Smartphones and mobile tablets are rapidly becoming indispensable in daily life. Android has been the most popular mobile operating system since 2012. However, owing to the open nature of Android, countless malwares are hidden in a large number of benign apps in Android markets that seriously threaten Android security. Deep learning is a new area of machine learning research that has gained increasing attention in artificial intelligence. In this study, we propose to associate the features from the static analysis with features from dynamic analysis of Android apps and characterize malware using deep learning techniques. We implement an online deep-learning-based Android malware detection engine (DroidDetector) that can automatically detect whether an app is a malware or not. With thousands of Android apps, we thoroughly test DroidDetector and perform an indepth analysis on the features that deep learning essentially exploits to characterize malware. The results show that deep learning is suitable for characterizing Android malware and especially effective with the availability of more training data. DroidDetector can achieve 96.76% detection accuracy, which outperforms traditional machine learning techniques. An evaluation of ten popular anti-virus softwares demonstrates the urgency of advancing our capabilities in Android malware detection.

TITLE 3:Identifying Android malware using deep learning

AUTHORS: Zi Wang, Juecong Cai, Sihua Cheng and Wenjia Li

YEAR: 2016

With the proliferation of Android apps, encounters with malicious apps (malware) by mobile users are on the rise as vulnerabilities in the Android platform system are exploited by malware authors to access personal or sensitive information with ill intentions, often with financial gain in mind. To uphold security integrity and maintain user confidence, various approaches have been studied in the field of malware detection. As malware become more capable at hiding its malicious intent through the use of code obfuscation, it becomes imperative for malware detection techniques to keep up with the pace of malware changes. Currently, most of the existing malware detection approaches for Android platform use semantic pattern matching, which is highly effective but is limited to what the computers have encountered before. However, their performance degrades significantly when it comes to identifying malicious apps they have never tackled before. In this paper, we propose DroidDeepLearner, an Android malware characterization and identification approach that uses deep learning algorithm to address the current need for malware detection to become more autonomous at learning to solve problems with less human intervention. Experimental results have shown that the DroidDeepLearner approach achieves good performance when compared to the existing widely used malware detection approaches.

TITLE 4:Deep learning-based malware detection by mining Android application for abnormal usage of sensitive data

AUTHORS: Dali Zhu, Hao Jin, Ying Yang and Di Wu

YEAR: 2017

The open nature of Android allows application developers to take full advantage of the system. While the flexibility is brought to developers and users, it may raise significant issues related to malicious applications. Traditional malware detection approaches based on signatures or abnormal behaviors are invalid when dealing with novel malware. To solve the problem, machine learning algorithms are used to learn the distinctions between malware and benign apps automatically. Deep learning, as a new area of machine learning, is developing rapidly as its better characterization of samples. We thus propose DeepFlow, a novel deep learning-based approach for identifying malware directly from the data flows in the Android application. We test DeepFlow on thousands of benignware and malware. The results show that DeepFlow can achieve a high detection F1 score of 95.05%, outperforming traditional machine learning-based approaches, which reveals the advantage of deep learning technique in malware detection.

CHAPTER 3

SYSTEM ANALYSIS

3.1 EXISTING SYSTEM

The Bouncer can scan the Android application for a limited period of time, allowing a malicious app too effortlessly bypass because of during the scan phase it doing nothing malicious.

At the second step, when scanned by the Bouncer, no malicious code must be included in the initial installer. In this case, the malicious app may have a higher chance of avoiding the detection of Bouncer.

The same permissions are required for the both benign and malicious application, consequently we cannot be distinguished by this permission based system. Generally, the permission based methodologies are largely not developed for the detection of malware, but it is used for the risk assessment.

MalDozer is a Convolutional Neural Network based Android Malware Detection System. MalDozer have a simple design in which minimal preprocessing is used to obtain the assembly processes. These are based on the concrete neural network in terms of extraction and detection / attribution of the features.

3.1.1 Drawback Of Existing System

1. The reported user will get the result, containing complete information from the integrity check and both analyses. Since new types of applications are constantly emerging, two crawler modules have been designed.

2. For crawling the benign apps from the Google Play Store they used one crawler and the other crawler is used to scroll malware from known sources of malware.

3. Droid Deep Learner is an Android Malware categorization and identification method. Droid Deep Learner uses deep learning method to report the present requirement for malware detection. In this method, they required a set of features for the detection.

4. The features like permissions, APIs, Actions, Intents, IP addresses and URLs are encrypted in the apk file. Based on source recompilation tool, they construct a decoder to decode the apps into readable format.

5. The user can identify the Android app is malware infected or not by using Droid Detector and it is available online as an open source. At first user have to submit the .apk file in the system, Droid Detector will check its reliability and defines whether an Android application is truthful, complete and appropriate.

3.2 PROPOSED SYSTEM

The general architecture of the proposed Droid Deep Learner method. We target to get two kinds of features, namely permission and API function calls from Android apps.

The Android apps would first be examined to obtain their equivalent manifest files (*.xml) and source files (*.java) to achieve this objective.

Then, the API function calls are extracted from java source files and from the manifest files the permissions can be extracted.

After that, they will add both of them collected into the feature set of an Android app, which works as an input for both training and testing purpose for the DBN based deep learning model.

Based on the deep learning model the classification result can separate malicious apps from the benign apps.

The first one is used for crawling malware from identified malwares and from Google Play Store we crawling the benign application by using other crawler. Deep Flow can declare its precision in detecting the frequently developing different malware.

3.2.1 Advantage Of Proposed System

(i) malicious or novel for the recognition task, and

(ii) Malicious relations for the attribution task. In deployment phase, the embedding model is used to produce the vector sequence and mine the sequence of techniques.

(iii) They use the vector sequence for detect the android app is malware infected or not.

CHAPTER 4

SYSTEM REQUIREMENTS

4.1 HARDWARE REQUIREMENTS

• System	: Pentium IV 2.4 GHz.
• Hard Disk	: 40 GB.
• Monitor	: 15 inch VGA Color.
• Mouse	: Logitech Mouse.
• Ram	: 512 MB
• Keyboard	: Standard Keyboard

4.2 SOFTWARE REQUIREMENTS

• Operating System	: Windows XP.
• Platform	: PYTHON TECHNOLOGY
• Tool	: Python 3.6
• Front End	: Python anaconda script
• Back End	: Spyder

CHAPTER 5

SYSTEM DESIGN

5.1 SYSTEM ARCHITECTURE

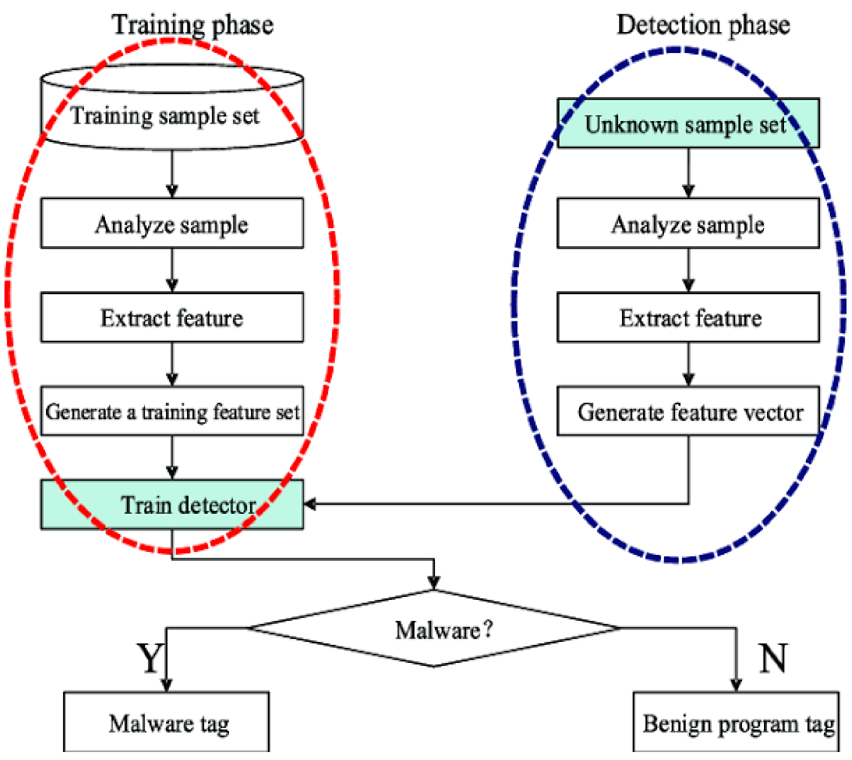


Fig 5.1 SYSTEM ARCHITECTURE

The machine learning-based malware detection process mainly includes two stages: training and detection.

- In the former stage, the analysts usually extract features from the samples set and then employ the features to train the automatic classifier.
- In the latter stage, the features will first be extracted from the samples to be detected, and then input into the trained classifier to obtain a decision result.

5.2 DATA FLOW DIAGRAM

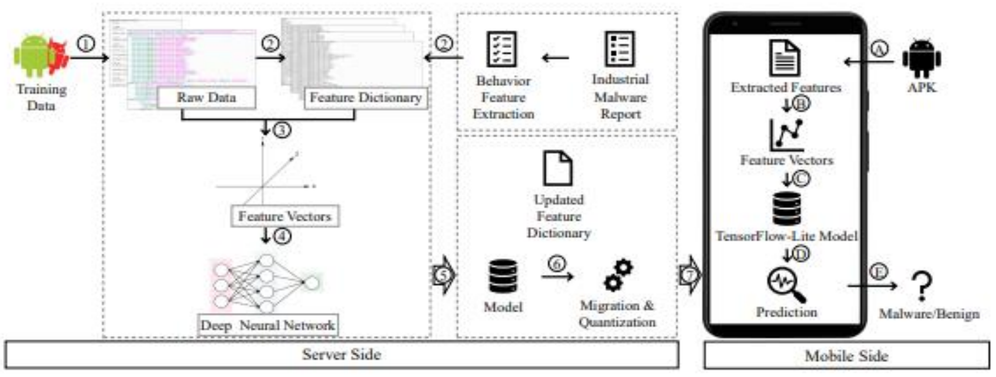


Fig 5.2 DATA FLOW DIAGRAM

A data flow diagram (DFD) is pictographical representation of the "flow" of data through modeling its process aspects. Instead of decompiling APK into source code, like smali code, we extract and vectorize the manifest properties and API calls from binary code directly (step 1). We combine a performance-based feature selection mechanism and behavior-based feature updating method to generate the feature dictionary (step 2). With the customized deep neural networks and extracted feature vectors (steps 3 and 4), the first part allows to provide a trained DL model and a feature dictionary for the second part (step 5). To make the model adaptive to mobile devices, we then migrate the pre-built DL model to a TensorFlow Lite model. Also, a quantization phase [30], which is a general technique to reduce model size while also providing lower latency with little degradation in accuracy, is presented as a performance optimization for the mobile devices (step 6). Fig. 5.2 shows that the second part loads the quantized DL model and feature dictionary into mobile devices. After that, when an application is downloaded from market or thirdparty market, that can extract feature vectors from it and deliver the result (steps →A C). After predicting with the loaded DL model, we obtain a certain level of confidence based on predictive output to know whether the downloaded Android app is a malware or not. (steps →D E).

5.3 UML DIAGRAM

Unified modelling language (UML) is a standardized general-purpose modelling language. UML is a accepted by the Object Management Group (OMG) as the standard for modelling object oriented programs.

5.3.1 Class Diagram

The class diagram is the main building block of object oriented modelling. It is used both for general conceptual modelling of the systematic of the application, and for detailed modelling translating the models into programming code. Class diagrams can also be used for data modelling. The classes in a class diagram represent both the main objects and or interactions in the application and the objects to be programmed.

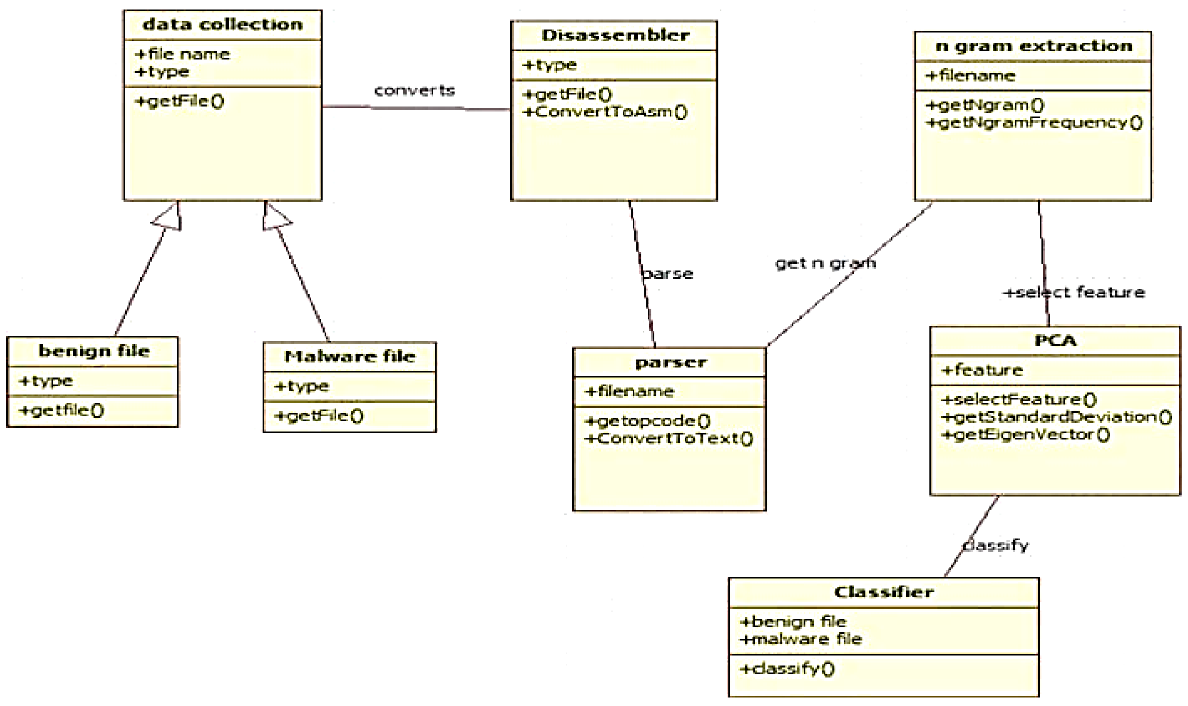


Fig 5.3.1 Class Diagram

5.3.2 Activity Diagram

Activity diagrams are graphical representation of workflows of stepwise activities and actions with support for choice, iteration and concurrency. In the Unified Modelling Language, activity diagrams can be used to describe the business and operational step-by-step workflows of components in a system.

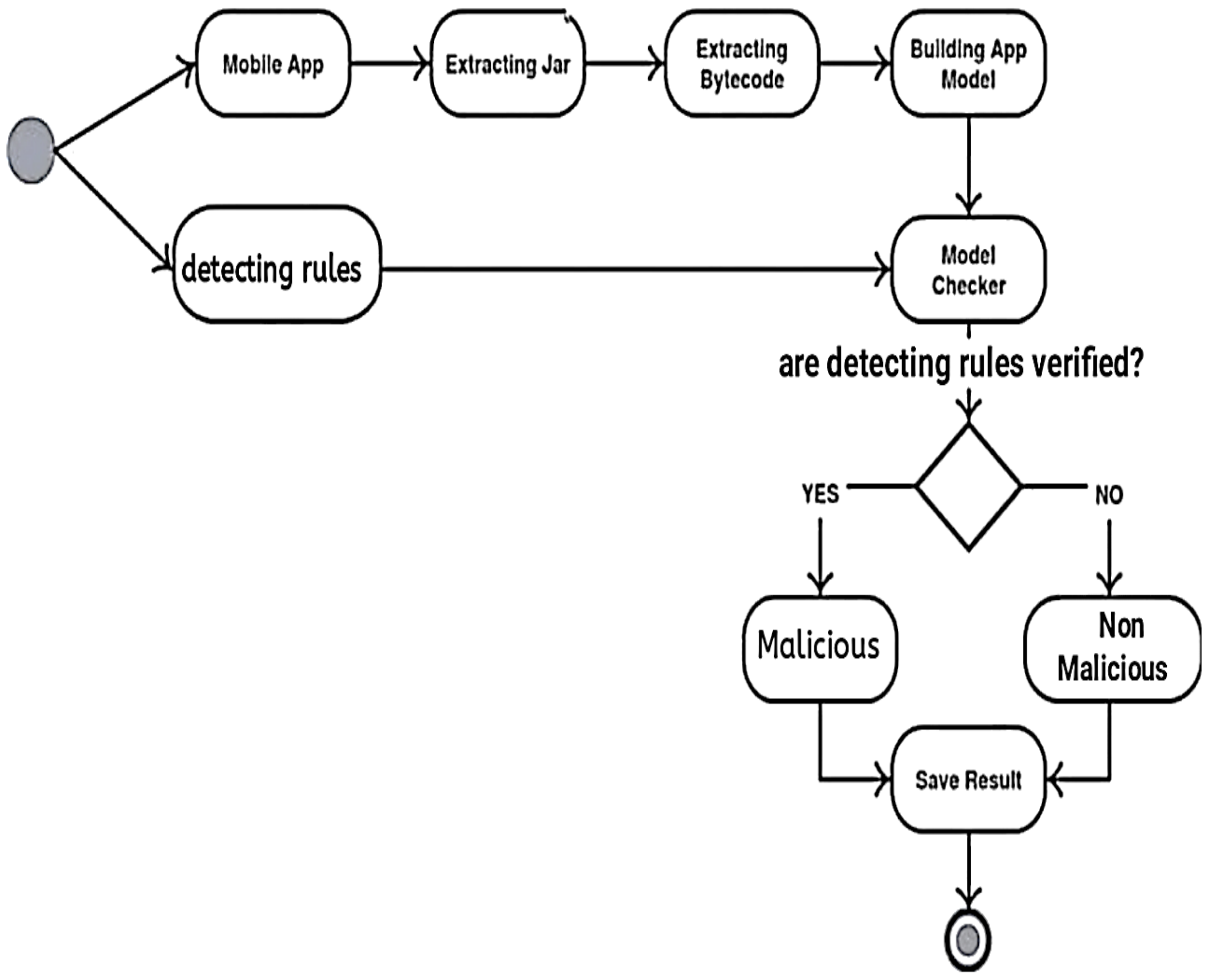


Fig 5.3.2 Activity Diagram

5.3.3 Use Case Diagram

A use case diagram in the Unified Modeling Language (UML) is a type of behavioral diagram defined by and created from a Use-case analysis. The main purpose of use case diagram is to show what system functions are performed for which actor. Roles of the actors in the system can be depicted.

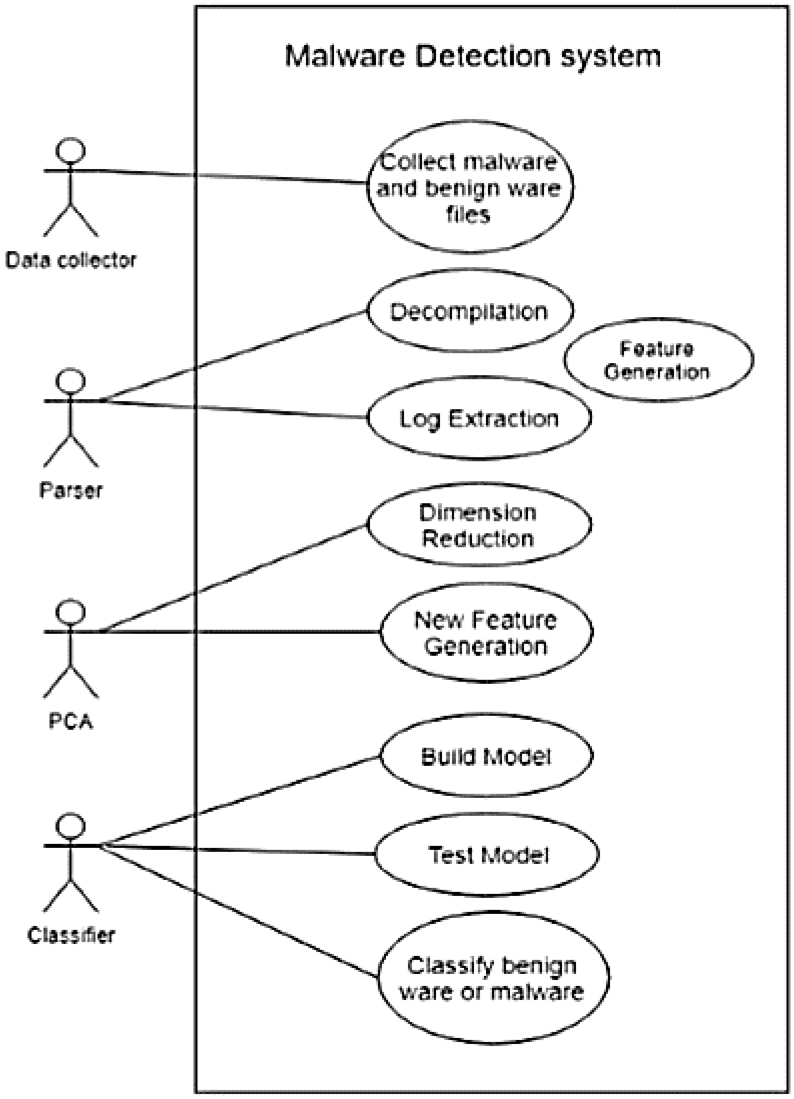


Fig 5.3.3 Use Case Diagram

5.3.4 Sequence Diagram

A sequence diagram is a kind of interaction diagram that shows how processes operate with one another and in what order. It is a construct of a message Sequence Chart. A sequence diagram shows object interactions arranged sequence. Sequence diagrams typically are associated with use case realizations in the Logical view of the system under development.

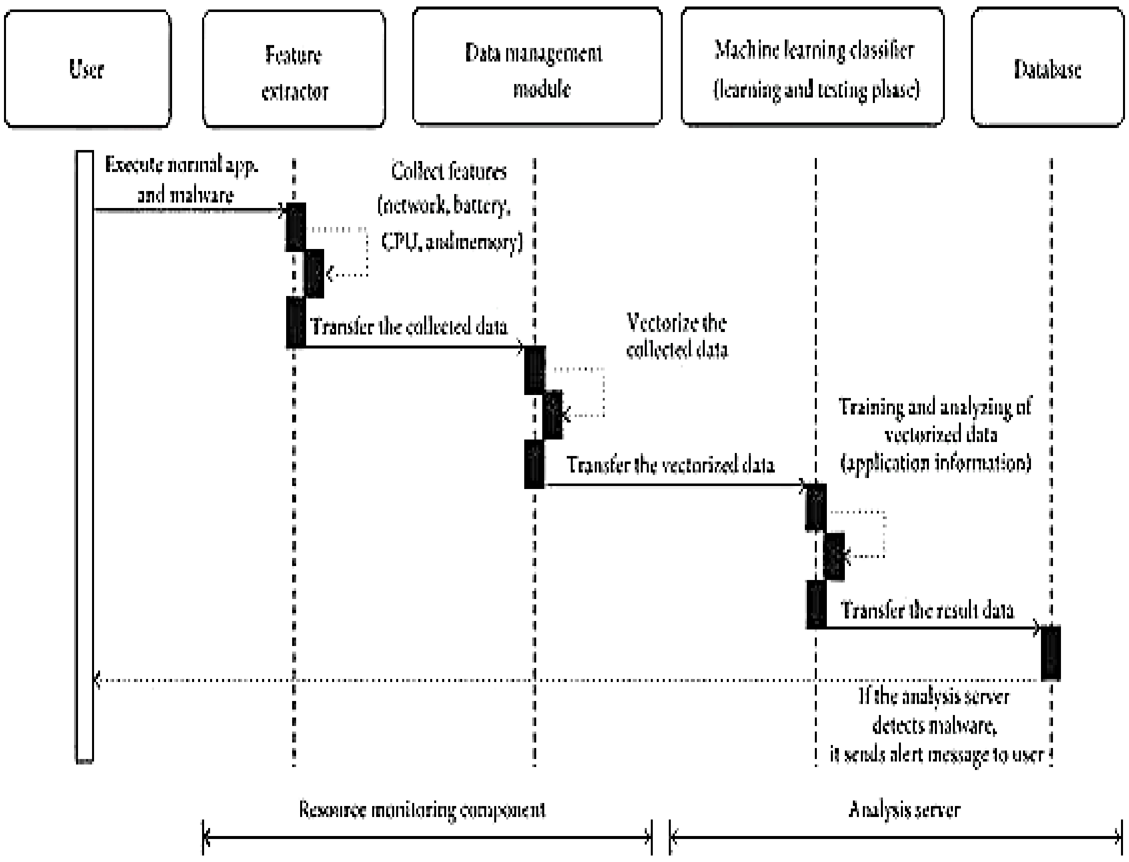

User
Feature extractor
Data management module
Machine learning classifier (learning and testing phase)
Database
Execute normal app. and malware
Collect features (network, battery, CPU, and memory)
Transfer the collected data
Vectorize the collected data
Transfer the vectorized data
Training and analyzing of vectorized data (application information)
Transfer the result data
If the analysis server detects malware, it sends alert message to user
Resource monitoring component
Analysis server

Fig 5.3.4 Sequence Diagram

5.3.5 Collaboration Diagram

Collaboration diagrams belong to a group of UML diagrams called Interaction diagrams. Collaboration diagrams, like Sequence diagrams, show how objects interact over the course of time. However, instead of showing the sequence of events by the layout on the diagrams, collaboration diagrams show the sequence by numbering the messages on the diagram.

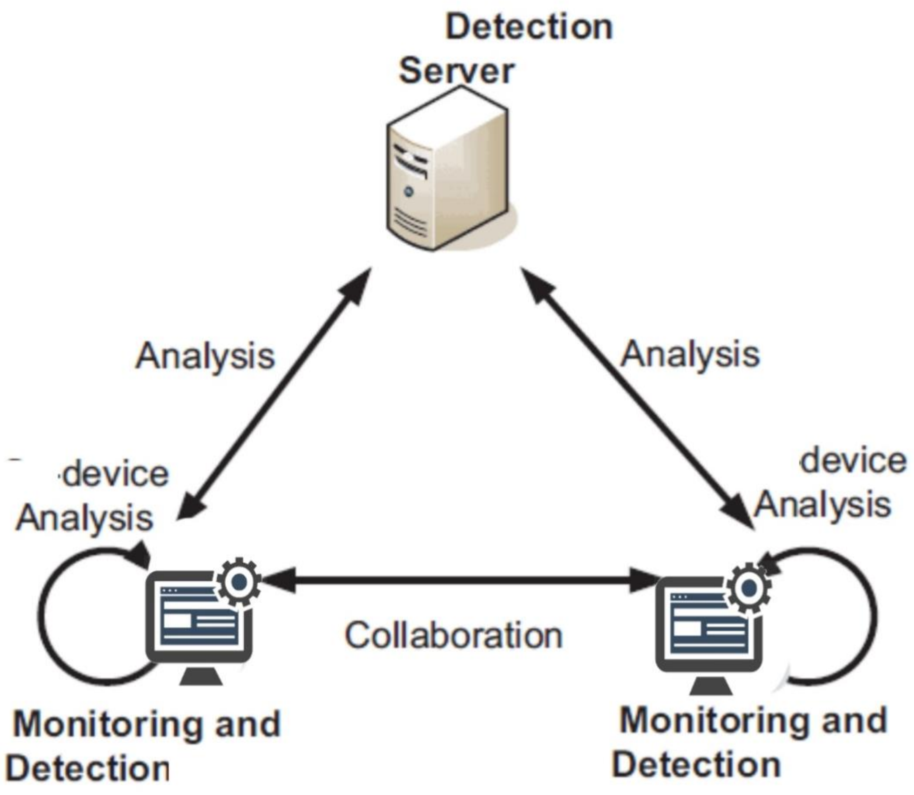


Fig 5.3.5 Collaboration Diagram

CHAPTER 6

METHODOLOGY

Deep Flow is a different deep learning model for classifying malware straight as of the data flows in Android application. Deep Flow architecture use the Deep Belief Network based model in Deep Learning for identifying the Android app is malware or not.

First, the features are extracted from apk file. Deep Flow uses the Flow Droid a static exploration device to categorize complex information flows in application.

To categorize delicate files pours in an Android app, at first, Flow Droid examines for life span and call-back approaches along with call to sources and sinks, to categorize complex data flows in an application.

Flow Droid treats every sensitive sources as the beginning point of the stain exploration procedure based on control flow graph. Flow Droid associate a forward-taint exploration and an on-demand backward alias exploration.

6.1 ALGORITHM

They used the Discretization algorithm to discretize the API method call sequences. The sequence identifier must be designed to fit into our neural network. By representing a vector for each identifier they can solve this.

They used one-hot vectors, where the value of the vector is one in the edge value row and rest is the zero.

From each app, they got the vector sequences that contains the instruction of the unique API calls and every vector has static size K.

Droid Delver used the API method calls, which are used for access the system resources and the functionality in Android app. So that to extract the API calls, first they have to unzip the apk file to get access to the class.dex file, then they used the Aptos for the extraction of API calls.

In this method, they used the Max-Relevance algorithm to select API calls sets. After the API call extraction, they classify the API calls which appropriate to the equivalent process in the smali code into the API call block.

There are multiple hyper-parameters like amount of layers and the model's complication. During the deployment time, they try to have the neural network model as humble as likely. To routinely determine the design in the raw method calls, MalDozer depend on the convolution layers. The vector sequence is used as input to the neural network, i.e. an L×K shaped matrix. In the training phase, depend on the app vector classification and its tags, MalDozer trains neural network parameters.

6.2 TECHNIQUE

The APK Tool is used for extract the APKs and decompile the dex files to smali code. The API call extractor is used to extract the API calls from the smali code.

The API call block generator used for the extracted APIs that will be appropriate to the several technique in smali code will be more classified into a block. Every Android app will be characterized by the API call block.

Deep Flow starts Flow Droid to perform the every complex data pours from all the complex sources to the every complex sinks.

By using, SUSI technique the Deep Flow classifies the extracted flows to catch features and create a feature vector. In DBN deep learning model input the feature vector for classification of malware.

The DBN model has dual crawler modules. The first one is used for crawling malware from identified malwares and from Google Play Store we crawling the benign application by using other crawler.

6.2.1 K-Nearest Neighbors

The result of the K-Nearest Method can be inferred from the cross table. The results outlined there should be understood as follows: rows represent the actual classes of the tested samples, while columns represent the predicted values. Therefore, the cell of the 1st row and 1st column will show thenumber of correct instances for the 1st class. The cell of the 1st row and 2nd column will show the number of 1st class instances, that were marked as 2nd class, etc.

6.2.2 Support Vector Machines

The next algorithm that was tested was Support Vector Machines. The result ofthe predictions can be outlined. The overall accuracy achieved was 87.6% for multi-class classification and 94.6% for binary classification.

6.2.3 Artificial Neural Network (ANN)

An artificial neuron network (neural network) is a computational model that mimics the way nerve cells work in the human brain.

Artificial neural networks (ANNs) use learning algorithms that can independently make adjustments - or learn, in a sense - as they receive new input. This makes them a very effective tool for non-linear statistical data modeling. Deep learning ANNs play an important role in machine learning (ML) and support the broader field of artificial intelligence (AI) technology.

6.2.4 Generative Adversarial Network (GAN)

The core idea of a GAN is based on the "indirect" training through the discriminator, another neural network that is able to tell how much an input is "realistic", which itself is also being updated dynamically. This basically means that the generator is not trained to minimize the distance to a specific image, but rather to fool the discriminator. This enables the model to learn in an unsupervised manner.

CHAPTER 7

MODULE DESCRIPTION

7.1 FEATURE EXTRACTION

In any of the examples mentioned above, we should be able to extract the attributes from the input data, so that it can be fed to the algorithm. For example, for the housing prices case, data could be represented as a multidimensional matrix, where each column represents an attribute and rows represent the numerical values for these attributes. In the image case, data can be represented as an RGB value of each pixel.

Such attributes are referred to as **features,** and the matrix is referred to as feature vector. The process of extracting data from the files is called feature extraction. The goal of feature extraction is to obtain a set of informative and non-redundant data. It is essential to understand that features should represent the important and relevant information about our dataset since without it we cannot make an accurate prediction. That is why feature extraction is often a non-obvious task, which requires a lot of testing and research. Moreover, it is very domain-specific, so general methods apply here poorly.

Another important requirement for a decent feature set is non-redundancy. Having redundant features i.e. features that outline the same information, as well as redundant information attributes, that are closely dependent on each other, can make the algorithm biased and, therefore, provide an inaccurate result.

In addition to that, if the input data is too big to be fed into the algorithm (has too many features), then it can be transformed to a reduced feature vector (vector, having a smaller number of features). The process of reducing the vector dimensions is referred to as feature selection. At the end of this process,we expect the selected features to outline the relevant information from the initial set so that it can be used instead of initial data without any accuracy loss.

heuristics-based analysis in combination with machine learning methods that offer a higher efficiency during detection.

When relying on heuristics-based approach, there has to be a certain thresholdfor malware triggers, defining the amount of heuristics needed for the software to be called malicious. For example, we can define a set of suspicious features,such as “registry key changed”, “connection established”, “permissionchanged”, etc.

Then we can state, that any software, that triggers at least five features from that set can be called malicious. Although this approach providessome level of effectiveness, it is not always accurate, since some features canhave more “weight” than others, for example, “permission changed” usually results in more severe impact to the system than “registry key changed”. In addition to that, some feature combinations might be more suspicious than features by themselves. (Rieck, et al. 2011).

To take these correlations into account and provide more accurate detection, machine learning methods can be used.

7.2 MALWARE DETECTION TECHNIQUE

Malware is a malicious code which is developed to harm a computer or network. The number of malwares is growing so fast and this amount of growth makes the computer security researchers invent new methods to protect computers and networks. There are three main methods used to malware detection.

Signature based, Behavioral based and Heuristic ones. Signature based malware detection is the most common method used by commercial antiviruses but it can be used in the cases which are completely known and documented. Behavioral malware detection was introduced to cover deficiencies of signature based method.

7.3 DEEP LEARNING BASED MALWARE DETECTION

Malware development has seen diversity in terms of architecture and features. This advancement in the competencies of malware poses a severe threat and opens new research dimensions in malware detection. This study is focused on metamorphic malware that is the most advanced member of the malware family.

It is quite impossible for anti-virus applications using traditional signature-based methods to detect metamorphic malware, which makes it difficult to classify this type of malware accordingly. Recent research literature about malware detection and classification discusses this issue related to malware behavior.

We can transform a malware/benign file into a grayscale image using the method described later. Then we can apply these deep learning techniques on the generated images to classify them as malware or benign.

CHAPTER 8

SYSTEM IMPLEMENTATION

8.1 INTRODUCTION TO PYTHON

HISTORY

Python was conceived in the late 1980s by Guido van Rossum at Centrum Wiskunde & Informatica (CWI) in the Netherlands as a successor to the ABC programming language, which was inspired by SETL, capable of exception handling and interfacing with the Amoeba operating system. Its implementation began in December 1989. Van Rossum shouldered sole responsibility for the project, as the lead developer, until 12 July 2018, when he announced his "permanent vacation" from his responsibilities as Python's "benevolent dictator for life", a title the Python community bestowed upon him to reflect his long-term commitment as the project's chief decision-maker. In January 2019, active Python core developers elected a five-member Steering Council to lead the project.

Python 2.0 was released on 16 October 2000, with many major new features. Python 3.0, released on 3 December 2008, with many of its major features backported to Python 2.6.x and 2.7.x. Releases of Python 3 include the 2to3 utility, which automates the translation of Python 2 code to Python 3.

Python 2.7's end-of-life was initially set for 2015, then postponed to 2020 out of concern that a large body of existing code could not easily be forward-ported to Python 3. No further security patches or other improvements will be released for it. With Python 2's end-of-life, only Python 3.6.x and later were supported. Later, support for 3.6 was also discontinued. In 2021, Python 3.9.2 and 3.8.8 were expedited as all versions of Python (including 2.7) had security issues leading to possible remote code execution and web cache poisoning.

In 2022, Python 3.10.4 and 3.9.12 were expedited and so were older releases including 3.8.13, and 3.7.13 because of many security issues in 2022. Python 3.9.13 is the latest 3.9 version, and from now on 3.9 (and older; 3.8 and 3.7) will only get security updates.

8.2 Spyder IDE

Spyder is an open-source cross-platform IDE. The Python Spyder IDE is written completely in Python. It is designed by scientists and is exclusively for scientists, data analysts, and engineers. It is also known as the Scientific Python Development IDE and has a huge set of remarkable features which are discussed below.

8.2.1 Features of Spyder

Some of the remarkable features of Spyder are:

- Customizable Syntax Highlighting
- Availability of breakpoints (debugging and conditional breakpoints)
- Interactive execution which allows you to run line, file, cell, etc.
- Run configurations for working directory selections, command-line options, current/ dedicated/ external console, etc
- Can clear variables automatically (or enter debugging)
- Navigation through cells, functions, blocks, etc can be achieved through the Outline Explorer
- It provides real-time code introspection (The ability to examine what functions, keywords, and classes are, what they are doing and what information they contain)
- Automatic colon insertion after if, while, etc
- Supports all the IPython magic commands
- Inline display for graphics produced using Matplotlib
- Also provides features such as help, file explorer, find files, etc

8.3 ANACONDA PROMPT

Anaconda distribution comes with over 250 packages automatically installed, and over 7,500 additional open-source packages can be installed from PyPI as well as the conda package and virtual environment manager. It also includes a GUI, Anaconda Navigator, as a graphical alternative to the command-line interface (CLI).

The big difference between conda and the pip package manager is in how package dependencies are managed, which is a significant challenge for Python data science and the reason conda exists.

Before version 20.3, when pip installed a package, it automatically installed any dependent Python packages without checking if these conflict with previously installed packages. It would install a package and any of its dependencies regardless of the state of the existing installation. Because of this, a user with a working installation of, for example, TensorFlow, could find that it stopped working having used pip to install a different package that requires a different version of the dependent numpy library than the one used by TensorFlow. In some cases, the package would appear to work but produce different results in detail. While pip has since implemented consistent dependency resolution, this difference accounts for a historical differentiation of the conda package manager.

In contrast, conda analyses the current environment including everything currently installed, and, together with any version limitations specified (e.g. the user may wish to have TensorFlow version 2,0 or higher), works out how to install a compatible set of dependencies, and shows a warning if this cannot be done.

Open source packages can be individually installed from the Anaconda repository, Anaconda Cloud (anaconda.org), or the user's own private repository or mirror, using the `conda install` command. Anaconda, Inc. compiles and builds the packages available in the Anaconda repository itself, and provides binaries for Windows 32/64 bit, Linux 64 bit and MacOS 64-bit. Anything available on PyPI may be installed into a conda environment using pip, and conda will keep track of what it has installed itself and what pip has installed.

The default installation of Anaconda2 includes Python 2.7 and Anaconda3 includes Python 3.7. However, it is possible to create new environments that include any version of Python packaged with conda.

CHAPTER 9

CONCLUSION

In this paper we have discussed about different types of Android Malware Detection Techniques using various Deep Learning Methods. Because of open nature on Android, countless malwares are hidden in a large number of benign apps in Android markets. These malwares are seriously threat Android security. The attacker can monitor user’s information like: Messages, Contacts, Bank mTANs, Locations, etc.

Here we survey on different Android Malware Detection Techniques like: MalDozer, Droid Detector, Droid Deep Learner and Deep Flow. MalDozer is used the Convolution Neural Network for Malware Detection. It works on static analysis method and API method calls as a feature to detect the application is malware infected or not.

Droid Detector will use the Deep Belief Network for the detection. They used the static and dynamic analysis with features like: permissions, APIs, Dynamic behavior for malware detection. Droid Deep Learner method is also use the Deep Belief Network for malware detection.

8.1 FUTURE ENHACEMENT

They also use a static analysis method with the features like permissions and APIs for malware detection. Deep Flow also use the Deep Belief Network with the static analysis method. In this method they use the API method calls for Android Malware Detection.

But, these all methods are working after installing the application on device or upload it to their model. To overcome this problem we are trying to implement a Deep Learning model that can automatically identify the application is malicious or not before the installation.

APPENDIX 1 SAMPLE CODE

```
# Importing the libraries
import json
import pandas as pd
import time
import numpy as np
import matplotlib.pyplot as plt
from keras.models import Sequential
from keras.layers import Dense
from keras.layers import Dropout
from sklearn.metrics import roc_curve



#LOAD DATA
start_time = time.time()
#load .json datasets for malware and benign
#concatenate them
with open("Mfeatures_jsons.json", "r") as mal:
    data_m = json.load(mal)
with open("Bfeatures_jsons.json", "r") as beg:
    data_b = json.load(beg)

data_m.update(data_b)

with open("M_Bfeatures_jsons.json", "w") as fo:
    json.dump(data_m, fo, indent = 2)
```

```
#POPULATE FEATURES LISTS
#general lists upon which feature presence check will be made
#these are the features under investigation for each app

Glist_of_receivers_actions = []
Glist_of_permissions = []
Glist_of_apis = []
Gruntime_registered_receivers = []
Glist_of_fingerprints = []

#populate lists
with open("M_Bfeatures_jsons.json", "r") as d:
    json_dataset = json.load(d)



    for key, value in json_dataset.items():

        # here I push a list of data into a pandas DataFrame
        # each key(sample) forms a row
        #this function will be transfered to the final script that
        #reads the json file created here

       list_of_receivers_actions = value['list_of_receivers_actions']
       list_of_permissions = value['list_of_permissions']

       apis =   value['apis']
       runtime_registered_receivers =  value['runtime_registered_receivers']
       list_of_fingerprints = value['list_of_fingerprints']



       #POPULATE GENERAL LISTS UPON WHICH FEATURE PRESENCE
CHECK WILL BE DONE
       #list_of_receivers_actions
       for i in range(len(list_of_receivers_actions)):
           r = list_of_receivers_actions[i]
           if  r not in Glist_of_receivers_actions:
               if r.startswith("android.intent.action."):
                   Glist_of_receivers_actions.append(r)

       #list_of_permissions
       for i in range(len(list_of_permissions)):
           s = list_of_permissions[i]
           if s  not in Glist_of_permissions:
               if s.startswith('android.permission.'):
                   Glist_of_permissions.append(s)

       #list_of_api_names
       for key in  apis.keys():
           if key not in Glist_of_apis:
               Glist_of_apis.append(key)

       #registered_receivers
       for i in range(len(runtime_registered_receivers)):
            rt =  runtime_registered_receivers[i]
            if rt not in Gruntime_registered_receivers:
                if rt.startswith("android.intent.action."):
                    Gruntime_registered_receivers.append(rt)

        #list_of_fingerprints
        for i in range(len(list_of_fingerprints)):
            if  list_of_fingerprints[i] not in Glist_of_fingerprints:
                Glist_of_fingerprints.append(list_of_fingerprints[i])



#CREATE PANDAS DATAFRAME
#here I define my pandas Dataframe with the columns I want to get from the json
#dataset for all samples(malware & benign)

others = ['malware']

#create dataset(1527 columns of features)

data = pd.DataFrame(columns = Glist_of_permissions
                      + Glist_of_receivers_actions + Gruntime_registered_receivers+
                       Glist_of_fingerprints + Glist_of_apis +  others)

#initialize  dataset with 0 values for all features
with open("M_Bfeatures_jsons.json", "r") as d:
    json_dataset = json.load(d)
    #add rows with only index values(md5)
    for key  in json_dataset.keys():
        data.loc[key] = 0

#CHECK FOR Features and update corresponding  dataset values
with open("M_Bfeatures_jsons.json", "r") as d:
    json_dataset = json.load(d)

    for key, value in json_dataset.items():

        # here I push append  data to a pandas DataFrame
        # each key(sample) forms a row

       list_of_permissions = value['list_of_permissions']
       list_of_receivers = value['list_of_receivers']
       list_of_receivers_actions = value['list_of_receivers_actions']
       runtime_registered_receivers =  value['runtime_registered_receivers']
       list_of_fingerprints = value['list_of_fingerprints']
       apis =   value['apis']
       malware = value['malware']

       #update presence of given permission for sample(key)
       for i in range(len(list_of_permissions)):
           m =  list_of_permissions[i]
           if m in Glist_of_permissions:
               data.loc[key, m] = 1
```

```
#update presence of given receivers_action for sample(key)
for i in range(len(list_of_receivers_actions)):
    m =  list_of_receivers_actions[i]
    if m in Glist_of_receivers_actions:
        data.loc[key, m] = 1

#update presence of given registered_receiver for sample(key)
for i in range(len(runtime_registered_receivers)):
    m =  runtime_registered_receivers[i]
    if m in Gruntime_registered_receivers:
        data.loc[key, m] = 1

#update presence of given fingerprint for sample(key)
for i in range(len(list_of_fingerprints)):
    m = list_of_fingerprints[i]
    if m in Glist_of_fingerprints:
        data.loc[key, m] = 1

#update presence of a given api call for sample(key)
for api in apis.keys():
    m = api
    if m in Glist_of_apis:
         data.loc[key, m] = 1

#update  others features
data.loc[key, 'malware'] = malware
```

```
#save dataset in  csv format
data.to_csv("MSGmalware_analysis_dataset_all_features.csv",  encoding='utf8')
end_time = time.time()
print('Execution time: '+str(round( end_time -start_time, 3))+'seconds')

#*****POPULATE GENERAL FEATURES LISTS************
#general lists upon which feature selection will be made basing on feature importance


Glist_of_receivers_actions = []
Glist_of_permissions = []
Glist_of_apis = []
Gruntime_registered_receivers = []
Glist_of_fingerprints = []
#populate lists
with open("M_Bfeatures_jsons.json", "r") as d:
   json_dataset = json.load(d)
   for key, value in json_dataset.items():
     list_of_receivers_actions = value['list_of_receivers_actions']
     list_of_permissions = value['list_of_permissions']
     apis =   value['apis']
     runtime_registered_receivers =  value['runtime_registered_receivers']
     list_of_fingerprints = value['list_of_fingerprints']



     #POPULATE GENERAL LISTS UPON WHICH FEATURE PRESENCE CHECK WILL BE DONE
   #list_of_receivers_actions
for i in range(len(list_of_receivers_actions)):
    r = list_of_receivers_actions[i]
    if  r not in Glist_of_receivers_actions:
        if r.startswith("android.intent.action."):
            Glist_of_receivers_actions.append(r)

#list_of_permissions
for i in range(len(list_of_permissions)):
    s = list_of_permissions[i]
    if s  not in Glist_of_permissions:
        if s.startswith('android.permission.'):
            Glist_of_permissions.append(s)

#list_of_api_names
for key in  apis.keys():
    if key not in Glist_of_apis:
        Glist_of_apis.append(key)

#registered_receivers
for i in range(len(runtime_registered_receivers)):
    rt =  runtime_registered_receivers[i]
    if rt not in Gruntime_registered_receivers:
        if rt.startswith("android.intent.action."):
            Gruntime_registered_receivers.append(rt)

#list_of_fingerprints
```

```
    for i in range(len(list_of_fingerprints)):
        if  list_of_fingerprints[i] not in Glist_of_fingerprints:
            Glist_of_fingerprints.append(list_of_fingerprints[i])



others = ['malware']


all_features = (Glist_of_permissions + Glist_of_receivers_actions + Gruntime_registered_receivers+
                Glist_of_fingerprints + Glist_of_apis +  others)



#*GET FEATURE IMPORTANCES *****************
# Importing the dataset
dataset = pd.read_csv("MSGmalware_analysis_dataset_all_features.csv", delimiter=",")
# split into input (X) and output (Y) variables
X = dataset.iloc[:, 1:714].values
y = dataset.iloc[:, 714].values
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state = 0)
## Import the random forest model.
from sklearn.ensemble import RandomForestClassifier
## This line instantiates the model.
rf = RandomForestClassifier()
```

```
## Fit the model on your training data.
rf.fit(X_train, y_train)
## And score it on your testing data.
rf.score(X_test, y_test)

importances = rf.feature_importances_
std = np.std([tree.feature_importances_ for tree in rf.estimators_],
             axis=0)
indices = np.argsort(importances)[::-1]
# Print the feature ranking
print("Feature ranking:")
important_features = []
important_features_scores = []
for f in range(X.shape[1]):
   print("%d. %-*s %f" % (f + 1, 0, all_features[indices[f]],
importances[indices[f]]))
   #store important features
   if importances[indices[f]] > 0.000004:
      important_features.append(all_features[indices[f]])
      important_features_scores.append(all_features[indices[f]]+' >>>
'+str(round(importances[indices[f]],6)))

#new general lists of important features(if)
if_list_of_receivers_actions = []
if_list_of_permissions = []
if_list_of_apis = []
if_runtime_registered_receivers = []
```

```
if_list_of_fingerprints = []
if_others = []

#update if_list_of_receivers_actions
for i in range (len(Glist_of_receivers_actions)):
    if Glist_of_receivers_actions[i] in important_features:
        if_list_of_receivers_actions.append(Glist_of_receivers_actions[i])
#update if_others
for i in range(len(others)):
    if others[i] in important_features:
        if_others.append(others[i])
#update if_list_of_permissions
for i in range(len(Glist_of_permissions)):
    if   Glist_of_permissions[i] in  important_features:
        if_list_of_permissions.append(Glist_of_permissions[i])
#update if_list_of_apis
for i in range(len(Glist_of_apis)):
    if   Glist_of_apis[i] in  important_features:
        if_list_of_apis.append(Glist_of_apis[i])
#update if_runtime_registered_receivers
for i in range(len(Gruntime_registered_receivers)):
    if   Gruntime_registered_receivers[i] in  important_features:
        if_runtime_registered_receivers.append(Gruntime_registered_receivers[i])
#update if_list_of_fingerprints
for i in range(len(Glist_of_fingerprints)):
    if  Glist_of_fingerprints[i] in  important_features:
        if_list_of_fingerprints.append(Glist_of_fingerprints[i])
```

```
start_time = time.time()
#LOAD DATA
#import important feature lists
from significant_features import (if_list_of_permissions, if_list_of_receivers_actions ,
if_runtime_registered_receivers ,if_list_of_fingerprints , if_list_of_apis)



#CREATE PANDAS DATAFRAME
# pandas Dataframe with the columns  from the json
#dataset for all samples(malware & benign)
other = ['malware']
data = pd.DataFrame(columns = if_list_of_permissions
                      + if_list_of_receivers_actions + if_runtime_registered_receivers+
                       if_list_of_fingerprints + if_list_of_apis +  other)

#initialize  dataset with 0 values for all features
with open("M_Bfeatures_jsons.json", "r") as d:
    json_dataset = json.load(d)
    #add rows with only index values(md5)
    for key  in json_dataset.keys():
        data.loc[key] = 0

#check for features and update corresponding  dataset values
with open("M_Bfeatures_jsons.json", "r") as d:
    json_dataset = json.load(d)
    for key, value in json_dataset.items():
```

```
 # append  data to a pandas DataFrame
 # each key(sample) forms a row
list_of_permissions = value['list_of_permissions']
list_of_receivers = value['list_of_receivers']
list_of_receivers_actions = value['list_of_receivers_actions']
runtime_registered_receivers =  value['runtime_registered_receivers']
list_of_fingerprints = value['list_of_fingerprints']
apis =   value['apis']
malware = value['malware']

#update presence of given permission for sample(key)
for i in range(len(list_of_permissions)):
    m =  list_of_permissions[i]
    if m in if_list_of_permissions:
        data.loc[key, m] = 1

#update presence of given receivers_action for sample(key)
for i in range(len(list_of_receivers_actions)):
    m =  list_of_receivers_actions[i]
    if m in if_list_of_receivers_actions:
        data.loc[key, m] = 1

#update presence of given registered_receiver for sample(key)
for i in range(len(runtime_registered_receivers)):
    m =  runtime_registered_receivers[i]
    if m in if_runtime_registered_receivers:
        data.loc[key, m] = 1
    #update presence of given fingerprint for sample(key)
    for i in range(len(list_of_fingerprints)):
        m = list_of_fingerprints[i]
        if m in if_list_of_fingerprints:
            data.loc[key, m] = 1

    #update presence of a given api call for sample(key)
    for api in apis.keys():
        m = api
        if m in if_list_of_apis:
             data.loc[key, m] = 1

    #update  others features
    data.loc[key, 'malware'] = malware

#save dataset in  csv format
data.to_csv("MSGmalware_analysis_dataset_if.csv",  encoding='utf8')

end_time=time.time()
print('Execution time: '+str(round( end_time -start_time, 3))+'seconds')

start_time = time.time()
#*PART1---DATA PREPROCESSING*****************
# Importing the dataset
dataset = pd.read_csv("MSGmalware.csv", delimiter=",")
# split into input (X) and output (Y) variables
```

```
X = dataset.iloc[:, 1:358].values
y = dataset.iloc[:, 358].values
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.2,
                                                    random_state = 0)

#**PART2----BUILDING THE ANN(SmartAM2)*************
# Initialising the ANN
model = Sequential()
# Adding the input layer and the first hidden layer
model.add(Dense(units = 300, kernel_initializer = 'uniform',
                activation = 'relu', input_dim = 357))
# Adding the second hidden layer
model.add(Dense(units = 250, kernel_initializer = 'uniform',
                activation = 'relu'))
# Adding the third hidden layer
model.add(Dense(units = 50, kernel_initializer = 'uniform',
                activation = 'relu'))
# Adding the output layer
model.add(Dense(units = 1, kernel_initializer = 'uniform',
                activation = 'sigmoid'))
# Compiling the ANN
model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
              metrics = ['accuracy'])
# Fitting the ANN to the Training set
model.fit(X_train, y_train, batch_size = 100, epochs = 700)
```

```
#*PART3----MAKING PREDICTIONS & EVALUATING THE MODEL*********
# Predicting the Test set results
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm_org = confusion_matrix(y_test, y_pred)
# evaluate the model
scores = model.evaluate(X_train, y_train)
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))

#ROC
y_org_pred = model.predict(X_test).ravel()
fpr_org, tpr_org, thresholds_org = roc_curve(y_test, y_org_pred)

#AUC value can also be calculated like this.
from sklearn.metrics import auc
auc_org = auc(fpr_org, tpr_org)

#**PART4----VALIDATING WITH UNSEEN ADVERSARIAL SAMPLES*******
dataset_adver = pd.read_csv('test_1.csv')
x_adver = dataset_adver.iloc[:, 1:-1].values
y_adver = dataset_adver.iloc[:, 358].values
# Predicting the Test set results for adver
```

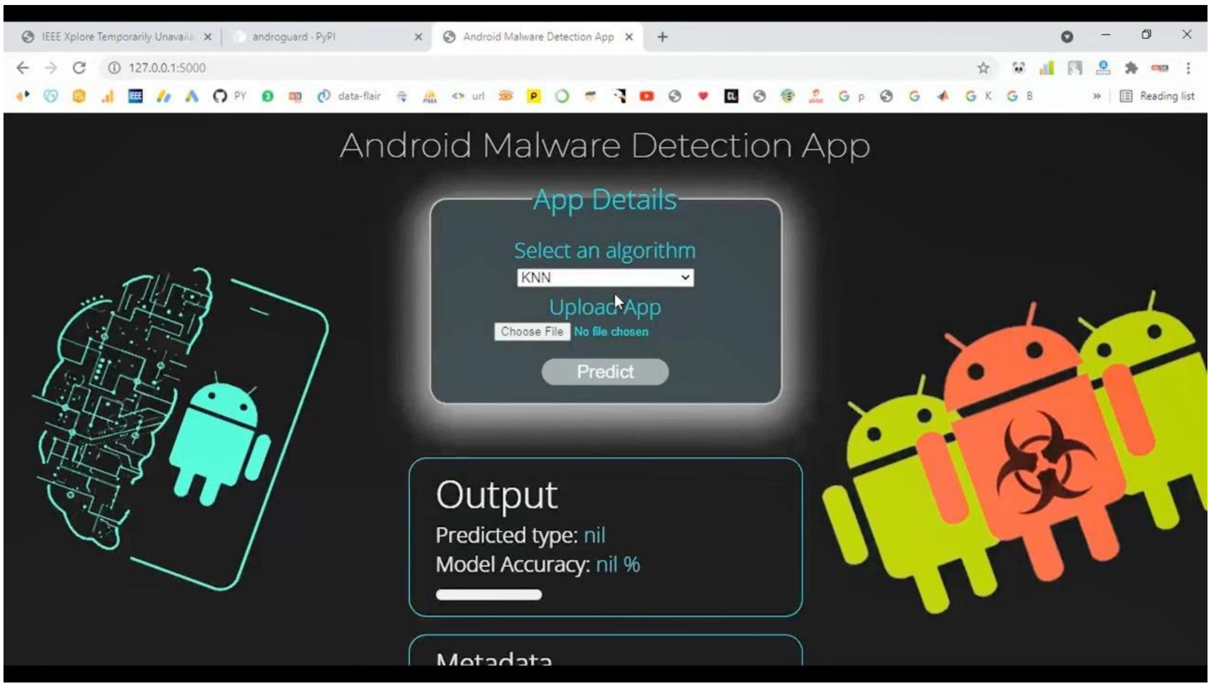
```
y_adver_pred = model.predict(x_adver)
y_adver_pred = (y_adver_pred > 0.5)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm_adver = confusion_matrix(y_adver, y_adver_pred)

#Now, let’s plot the ROC for the MODEL;
plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_org, tpr_org, label='Original samples (area = {:.3f})'.format(auc_org))
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()

end_time=time.time()
print('Execution time: '+str(round( end_time -start_time, 3))+'seconds')

#***SAVING THE MODEL******************
#save json
model_json = model.to_json()
with open("SmartAM1.json", "w") as json_file:
   json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("SmartAM1_1.h5")
```

```
print("Saved model to disk")
""""


#visualize the ann
from ann_visualizer.visualize import ann_viz
ann_viz(model, view=True, filename="SmartAM1.gv", title="SmartAM1")
""""
start_time = time.time()
#*PART1---DATA PREPROCESSING*****************
# Importing the datasets
dataset_org = pd.read_csv("MSGmalware.csv", delimiter=",")
dataset_adver = pd.read_csv('train_1.csv', delimiter=",")
#concatenate original and adversarial samples
dataset_retrain = pd.concat([dataset_org, dataset_adver], axis=0)
# split into input (X) and output (Y) variables
X = dataset_retrain.iloc[:, 1:358].values
y = dataset_retrain.iloc[:, 358].values
# Splitting the dataset into the Training set and Test set
from sklearn.model_selection import train_test_split
X_train, X_test, y_train, y_test = train_test_split(X, y, test_size = 0.3,
                                                    random_state = 0)


#**PART2----BUILDING THE ANN(SmartAM2)*************
# Initialising the ANN
model = Sequential()
# Adding the input layer and the first hidden layer
model.add(Dense(units = 300, kernel_initializer = 'uniform',
                activation = 'relu', input_dim = 357))
# Adding the second hidden layer
model.add(Dense(units = 200, kernel_initializer = 'uniform',
                activation = 'relu'))
# Adding the third hidden layer
model.add(Dense(units = 80, kernel_initializer = 'uniform',
                activation = 'relu'))
# Adding the output layer
model.add(Dense(units = 1, kernel_initializer = 'uniform',
                activation = 'sigmoid'))
# Compiling the ANN
model.compile(optimizer = 'adam', loss = 'binary_crossentropy',
              metrics = ['accuracy'])
# Fitting the ANN to the Training set
model.fit(X_train, y_train, batch_size = 100, epochs = 800)


#*PART3----MAKING PREDICTIONS & EVALUATING THE MODEL*********
# Predicting the Test set results
y_pred = model.predict(X_test)
y_pred = (y_pred > 0.5)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm_org = confusion_matrix(y_test, y_pred)
# evaluate the model
scores = model.evaluate(X_train, y_train)
```

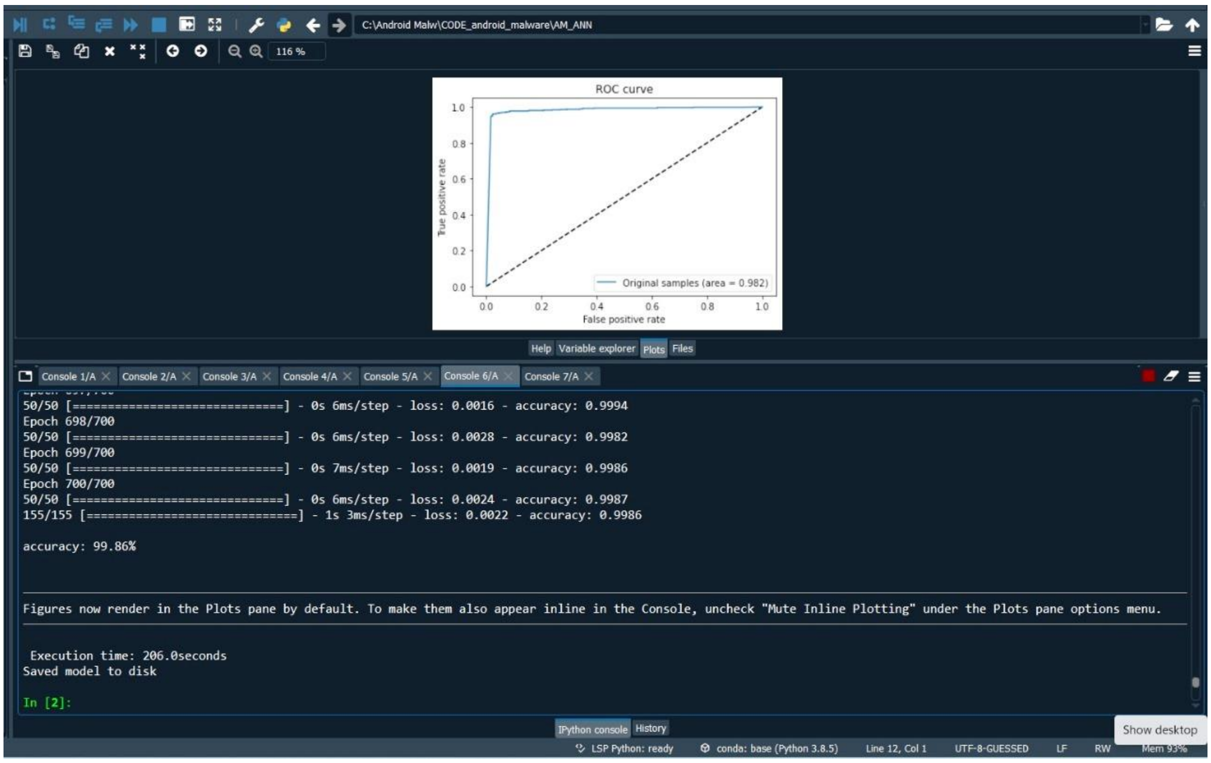
```
print("\n%s: %.2f%%" % (model.metrics_names[1], scores[1]*100))


#ROC
y_org_pred = model.predict(X_test).ravel()
fpr_org, tpr_org, thresholds_org = roc_curve(y_test, y_org_pred)


#AUC value can also be calculated like this.
from sklearn.metrics import auc
auc_org = auc(fpr_org, tpr_org)



#**PART4----VALIDATING WITH UNSEEN ADVERSARIAL SAMPLES*******
dataset_adver = pd.read_csv('test_1.csv')
x_adver = dataset_adver.iloc[:, 1:-1].values
y_adver = dataset_adver.iloc[:, 358].values
# Predicting the Test set results for adver
y_adver_pred = model.predict(x_adver)
y_adver_pred = (y_adver_pred > 0.5)
# Making the Confusion Matrix
from sklearn.metrics import confusion_matrix
cm_adver = confusion_matrix(y_adver, y_adver_pred)



#Now, let’s plot the ROC for the MODEL;
plt.figure(1)
plt.plot([0, 1], [0, 1], 'k--')
plt.plot(fpr_org, tpr_org, label='Original+Adversarial samples (area = {:.3f})'.format(auc_org))
plt.xlabel('False positive rate')
plt.ylabel('True positive rate')
plt.title('ROC curve')
plt.legend(loc='best')
plt.show()


end_time=time.time()
print('Execution time: '+str(round( end_time -start_time, 3))+'seconds')


#***SAVING THE MODEL******************
#save json
model_json = model.to_json()
with open("SmartAM2.json", "w") as json_file:
    json_file.write(model_json)
# serialize weights to HDF5
model.save_weights("SmartAM2.h5")
print("Saved model to disk")
```

APPENDIX 2 SCREENSHOTS

MAIN PAGE:

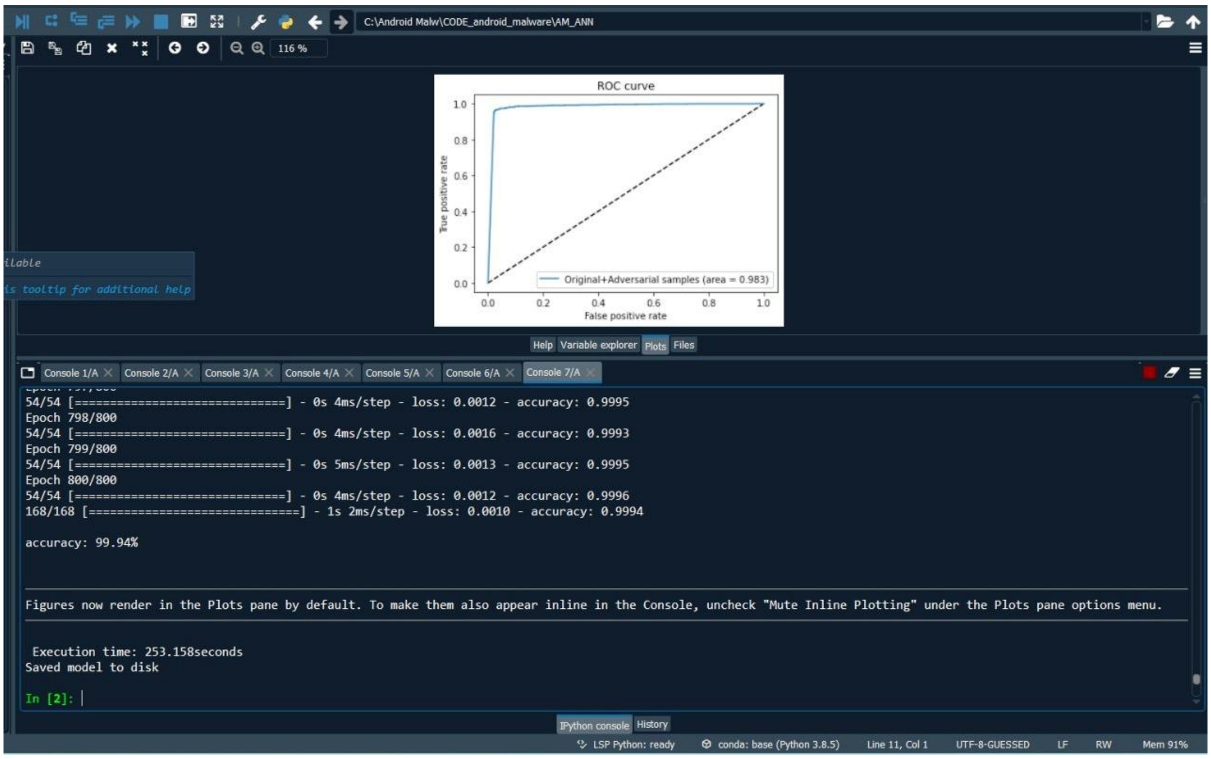


TRAINING A RAW DATA:



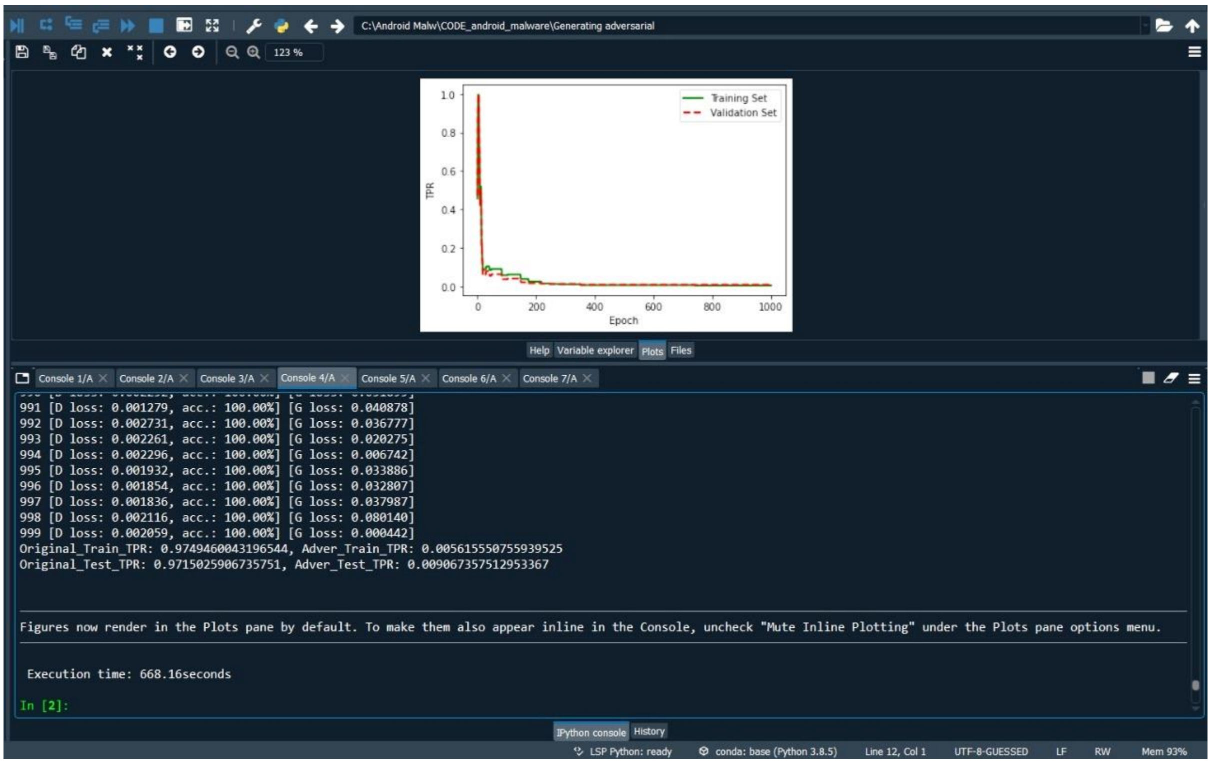
```
50/50 [==============================] - 0s 6ms/step - loss: 0.0016 - accuracy: 0.9994
Epoch 698/700
50/50 [==============================] - 0s 6ms/step - loss: 0.0028 - accuracy: 0.9982
Epoch 699/700
50/50 [==============================] - 0s 7ms/step - loss: 0.0019 - accuracy: 0.9986
Epoch 700/700
50/50 [==============================] - 0s 6ms/step - loss: 0.0024 - accuracy: 0.9987
155/155 [==============================] - 1s 3ms/step - loss: 0.0022 - accuracy: 0.9986

accuracy: 99.86%

Figures now render in the Plots pane by default. To make them also appear inline in the Console, uncheck "Mute Inline Plotting" under the Plots pane options menu.

 Execution time: 206.0seconds
Saved model to disk

In [2]:
```

TRAINING USING FEATURE DICTIONARY:



```
54/54 [==============================] - 0s 4ms/step - loss: 0.0012 - accuracy: 0.9995
Epoch 798/800
54/54 [==============================] - 0s 4ms/step - loss: 0.0016 - accuracy: 0.9993
Epoch 799/800
54/54 [==============================] - 0s 5ms/step - loss: 0.0013 - accuracy: 0.9995
Epoch 800/800
54/54 [==============================] - 0s 4ms/step - loss: 0.0012 - accuracy: 0.9996
168/168 [==============================] - 1s 2ms/step - loss: 0.0010 - accuracy: 0.9994

accuracy: 99.94%

Figures now render in the Plots pane by default. To make them also appear inline in the Console, uncheck "Mute Inline Plotting" under the Plots pane options menu.

 Execution time: 253.158seconds
Saved model to disk

In [2]:
```

TRAINING USING GAN_SMART:



```
991 [D loss: 0.001279, acc.: 100.00%] [G loss: 0.040878]
992 [D loss: 0.002731, acc.: 100.00%] [G loss: 0.036777]
993 [D loss: 0.002261, acc.: 100.00%] [G loss: 0.020275]
994 [D loss: 0.002296, acc.: 100.00%] [G loss: 0.006742]
995 [D loss: 0.001932, acc.: 100.00%] [G loss: 0.033886]
996 [D loss: 0.001854, acc.: 100.00%] [G loss: 0.032807]
997 [D loss: 0.001836, acc.: 100.00%] [G loss: 0.037987]
998 [D loss: 0.002116, acc.: 100.00%] [G loss: 0.080140]
999 [D loss: 0.002059, acc.: 100.00%] [G loss: 0.000442]
Original_Train_TPR: 0.9749460043196544, Adver_Train_TPR: 0.005615550755939525
Original_Test_TPR: 0.9715025906735751, Adver_Test_TPR: 0.009067357512953367

Figures now render in the Plots pane by default. To make them also appear inline in the Console, uncheck "Mute Inline Plotting" under the Plots pane options menu.

 Execution time: 668.16seconds

In [2]:
```

REFERENCES

1. User Interfaces in C#: Windows Forms and Custom Controls by Matthew MacDonald.

2. Applied Microsoft® .NET Framework Programming (Pro-Developer) by Jeffrey Richter.

3. Practical .Net2 and C#2: Harness the Platform, the Language, and the Framework by Patrick Smacchia.

4. Data Communications and Networking, by Behrouz A Forouzan.

5. Computer Networking: A Top-Down Approach, by James F. Kurose.

6. Operating System Concepts, by Abraham Silberschatz.

7. M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. H. Katz, A. Konwinski, G. Lee, D. A. Patterson, A. Rabkin, I. Stoica, and M. Zaharia, “Above the clouds: A berkeley view of cloud computing,” University of California, Berkeley, Tech. Rep. USB-EECS-2009-28, Feb 2009.

8. “The apache cassandra project,” http://cassandra.apache.org/.

9. L. Lamport, “The part-time parliament,” ACM Transactions on Computer Systems, vol. 16, pp. 133–169, 1998.

10. N. Bonvin, T. G. Papaioannou, and K. Aberer, “Cost-efficient and differentiated data availability guarantees in data clouds,”in Proc. of the ICDE, Long Beach, CA, USA, 2010.

11. O. Regev and N. Nisan, “The popcorn market. online markets for computational resources,” Decision Support Systems,vol. 28, no. 1-2, pp. 177 – 189, 2000.

12. A. Helsinger and T. Wright, “Cougaar: A robust configurable multi agent platform,” in Proc. of the IEEE Aerospace Conference, 2005.

13. J. Brunelle, P. Hurst, J. Huth, L. Kang, C. Ng, D. C. Parkes,M. Seltzer, J. Shank, and S. Youssef, “Egg: an extensible and economics-inspired open grid computing platform,” in Proc. of the GECON, Singapore, May 2006.

14. J. Norris, K. Coleman, A. Fox, and G. Candea, “Oncall: Defeating spikes with a free-market application cluster,” in Proc. of the International Conference on Autonomic Computing,New York, NY, USA, May 2004.

15. C. Pautasso, T. Heinis, and G. Alonso, “Autonomic resource provisioning for software business processes,” Information and Software Technology, vol. 49, pp. 65–80, 2007.

16. A. Dan, D. Davis, R. Kearney, A. Keller, R. King, D. Kuebler,H. Ludwig, M. Polan, M. Spreitzer, and A. Youssef, “Web services on demand: Wsla-driven automated management,” IBM Syst. J., vol. 43, no. 1, pp. 136–158, 2004.

17. M. Wang and T. Suda, “The bio-networking architecture: a biologically inspired approach to the design of scalable, adaptive, and survivable/available network applications,” in Proc.of the IEEE Symposium on Applications and the Internet,2001.

18. N. Laranjeiro and M. Vieira, “Towards fault tolerance in web services compositions,” in Proc. of the workshop on engineering fault tolerant systems, New York, NY, USA,2007.

19. C. Engelmann, S. L. Scott, C. Leangsuksun, and X. He,“Transparent symmetric active/active replication for service level high availability,” in Proc. of the CCGrid, 2007.

20. J. Salas, F. Perez-Sorrosal, n.-M. M. Pati and R. Jim´enez-Peris, “Ws-replication: a framework for highly available web services,” in Proc. of the WWW, New York, NY, USA, 2006,