Modelling A Data Sniffing Malware Detector For Apks

Oyinloye Oghenerukevwe Elohor[#1], Olatomide Awoyomi [#2]

Department of Computer Science, Ekiti State University, Nigeria

Abstract — *Smartphone has experienced rapid growth over the years. Android being the most popular operating system (according to* https://gs.statcounter.com, 2019 *constituting 76.24% of the statistics of the entire mobile usage) has witnessed a dramatic increase in malwares targeted at the platform as malware creators leverage on its popularity to exhibit malicious activity. As such, Android app marketplaces (googleplay and other third parties) remain at risk of hosting malicious apps that could be dangerous to the users. This calls for a need to pay attention to security issue in order to ensure that users can use their desired application without having a fallback on their privacy or any other means that attackers use hence, in this paper we present an effective approach to alleviate this problem based on machine learning approach using Linear Regression and Support Vector Machine (SVM) for classification. The models was trained with 70% and tested with 30% of the collected dataset and results of experiments are presented to demonstrate the effectiveness of the proposed approach nailing 85.7% accuracy.*

Keywords — *Malware, Android, Machine Learning, Linear Regression, Support Vector Machine (SVM)*

I. INTRODUCTION

Smart phones have been widely used in people's daily life, such as online banking, automated home control, and entertainment. Due to the mobility and ever expanding capabilities, the use of smart phones has experienced an exponential growth rate in recent years. It is estimated that 77.7% of all devices connected to the Internet will be smart phones in 2019 (Hou *et. al.*, 2016). Android, as an open source and customizable operating system for smart phones, is currently dominating the smart phone market by 76.24%.

However, due to its large market share and open source ecosystem of development, Android developers are not only producing legitimate Android applications (apps), but also hackers to disseminate malware (malicious software) that deliberately fulfills the harmful intent to the smart phone users. To protect legitimate users from the attacks of Android malware, currently, the major defense is mobile security products, such as Norton, Lookout and Comodo Mobile Security, which mainly use the signature-based method to recognize threats. However, hackers can easily use techniques, such as code obfuscation and repackaging, to evade the detection. The increasing sophistication of Android malware calls for new defensive techniques that are robust and capable of protecting users against novel threats. To be more resilient against the Android malware's evasion tactics.

The nature of Android apps makes it difficult to rely on standard, traditional, malware analysis systems as is.

While Android apps are generally written in the Java programming language and executed on top of the Dalvik virtual machine (VM), native code execution is possible, for instance, via the Java Native Interface (JNI). This mixed execution model seems to suggest the need to reconstruct, and keep in sync, different semantics through virtual machine introspection (VMI) for both the OS and Dalvik views. More recently, Zhang *et al.* (2012) stressed this concept further in and pointed out that traditional system call analysis is ill-suited to characterize the behaviors of Android apps as it misses high-level Android-specific semantics and fails to reconstruct inter-process communications (IPC) and remote procedure call (RPC) interactions, which are essential to understanding Android application behaviors. In a significantly different line of reasoning from, we observed that system call invocations remains central to both low-level OS-specific and high-level Android-specific behaviors. However, a naive analysis of system calls would miss the rich semantic of Android-specific behaviors.

This is where the novelty of our approach lies; our techniques will improve the android detection method by using machine learning approach we will train a model with Linear Regression and Support Vector Machine (SVM) for the classification and detection of malware from an application package to predict either it is clean or infected.

II. RELATED WORKS

Burguera *et. al.*, (2011) proposed an approach to analyze the behavior of Android applications, providing a framework to distinguish between applications who, have the same name and version but behave differently aimed at detecting anomalously behaving applications, thus detecting malware in the form of trojan horses.

Sato *et. al.*, (2013) proposed a method for detecting Android malware by analyzing only the manifest files. Their method runs on a low cost analysis using only the manifest files to detect malware.

Yerima *et. al.*, (2014) proposed a machine learning based zero-day Android malware detector. Their approach leveraged the strengths of supervised learning algorithms (a function-based, tree-based, probabilistic, and two rule based algorithms) to produce a single classification verdict for new applications and has extensive empirical evaluation of the approach by means of real malware samples and benign applications, demonstrating its real-world applicability and capacity.

Narudin *et. al.*, (2014) proposed an alternative solution to evaluating malware detection using the anomaly-based approach with machine learning classifiers. They implemented their anomaly detection with machine learning tools such as Random forest**,** J48**,** MLP (Multi-layer perceptron), Bayesian Net and KNN(K-nearest neighbours).

Yerima *et. al.*, (2015) investigated how the power of ensemble learning can be applied to improve Android malware detection.

Pimentel (2015) proposed a method to detect known and unknown Android malware by using a machine learning ensemble method. To build the models, he extracted features from the Android .apk by getting the permissions and API calls. As a classifier he used Troika, an ensemble method that uses several classifiers to improve performance over a single classifier.

Duc *et. al*,. (2015) introduced a method to evaluate the security level of Android applications based on their permission. The method, which is called PAMD: Permission analysis for Android malware detection, analyses the Android Manifest file by understanding the protection level of Android permission and investigating malicious characteristics.

Kang *et. al.*, (2015) proposed an Android malware detection and classification system based on static analysis by using serial number information from the certificate as a feature. Their method mainly checks a serial number, checks suspicious behavior of SMS hiding, detects the malicious system commands in the code, and analyzes the suspicious permission requests.

Dimjaševic *et. al.*, (2016) proposed a novel dynamic Android malware detection techniques based on tracking system calls, all of which they implemented as a free and open-source tool called MALINE. Their work was inspired by a similar approach proposed for desktop malware detection albeit they provide simpler feature encodings and an Android-specific tool flow. They analyzed how the quality of malware classifiers is affected across several dimensions, including the choice of an encoding of system calls into features, the relative sizes of benign and malicious data sets used in experiments, the choice of a classification algorithm, and the size and type of inputs that drive a dynamic analysis.

Qian *et. al.*, (2016) analyzed the Android applications accurately and comprehensively based on combining static and dynamic method to reveal the malicious behaviors of applications leaking user's privacy data. They use some tools such as Eclipse, JDK6, JRE6, Android SDK and Python2.7 will be installed. APK static decompiler and permissions filtering module were implemented with Java. The method is to insert some monitoring Smali byte code, and the performance influence can be ignored but the system is more time consumption.

Liang *et. al.*, (2016) developed an online malware detection tool named Droid Sentinel, which can detect and block background SMS messages and phone calls initiated by malware. Droid Sentinel sends an alarm to the user when background SMS messages, background phone calls, or premium SMS messages or phone calls are detected.

Rahman *et. al.*, (2017) Proposed a system Fairplay that will be able to detect and filter out fraudulent reviews and also Identify malware and fraud indicative feedback from the remaining reviews..

Chaba *et. al.*, (2018) defined an approach that created a dataset using system call log information. Their method implements a dataset on three algorithms namely, Naive Bayes algorithm, Random Forest Algorithm and Stochastic Gradient Descent algorithm. The strength lies in the use of system call but there is limited code coverage.

Karbab *et. al.*, (2018) proposed MalDozer; an automatic Android malware detection and family attribution framework that relies on sequences classification using deep learning techniques. Their method involve the creation of a MalDozer by disassembly a classes.dex to produce the Dalvik VM assembly to formalize the assembly and keep the maximum raw information with minimum noise

Huang *et. al.*, (2019) examined the potential vulnerabilities of MLbased malware detectors in adversarial environments. In aid to improve detection effectiveness while they discovered that machine learning (ML) classifiers are vulnerable to adversarial examples. They designed the white-box, grey-box and black-box attack experiments. In white-box attack, the attacker has complete knowledge of the system, including training data, features, and ML models (i.e. DNN architecture and parameters). They conducted white-box & grey-box attacks to an MLbased malware detector with thorough evaluations.

III. METHODOLOGY

This chapter discusses the steps, concepts and the operations of which this project is planned upon. The credibility of findings and conclusions depend largely on the quality of the research design, data collection of application packages, data management, and data analysis.

Therefore, this chapter is dedicated to the description of the methods and procedures done for implementation. The section is further broken down into two steps:

i. The steps involved in the logical design of creating our machine learning model which includes the following:

- The concept of mathematically applying Linear Regression to find out the cause and effect relationship between the independent and dependent variables in our data. Where the slope of a linear regression given as $Y = a + bX$ …. Equation 1 becomes $Y = a\_0 + a\_1 * X$ …. Equation 2

  (emphasizing the independent variable)

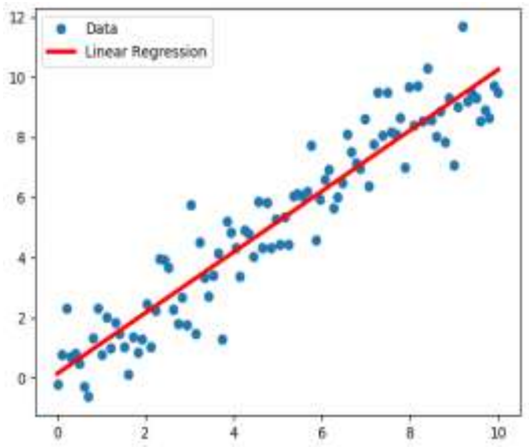


Figure 3.1: Linear Regression Slope

Fig. 3.1 shows a simulated linear regression slope describing how the safest path line (in red) separates two sets of data upon training and classification.

- The process of determining the cost function which will help us helps us to figure out the best possible values for a_0 and a_1 which would provide the best fit line for the data points
- The concept of applying Linear Regression's Gradient Descent to help reduce the cost function or MSE (mean standard error).
- Support Vector Machine's data structure concept for holding features and labels during training.
- The process of testing our model after training, as well as making new predictions.
- The new predictions to test the model will be achieved with an Android application to show the performance rate of the learning model

ii. The algorithms and methods needed to perform the steps in the logical design.

- Guided by the above insight, we will write code to train a model that will be able to detect and classify android malware using linear regression along with some major support vector machine algorithms.
- Note that our data are hundreds of downloaded application packages, including as much that have malwares in them.

3.2 SYSTEM DESIGN

3.2.1 System Architecture

Our input data is a collection of independent variables under two specific **labels** namely: Infected and Clean. The proposed system will collect this data for both training and testing, as well as allow to make new predictions via an Android application. The figure below displays this methodology in the most abstract form.

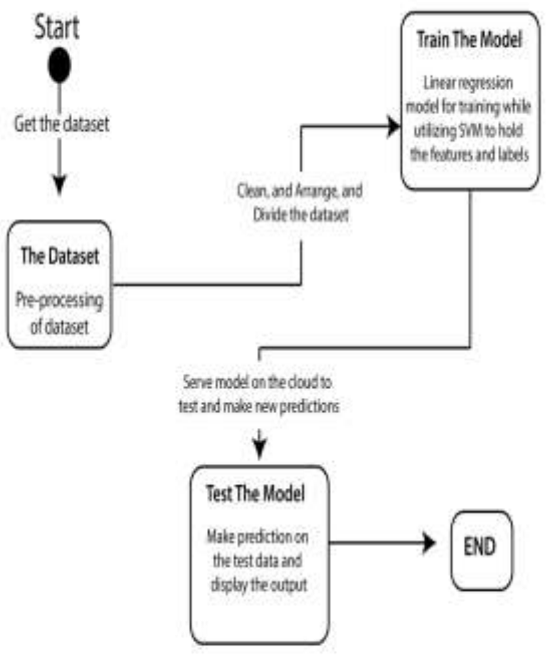

Start
Get the dataset
The Dataset
Pre-processing of dataset
Clean, and Arrange, and Divide the dataset
Train The Model
Linear regression model for training while utilizing SVM to hold the features and labels
Serve model on the cloud to test and make new predictions
Test The Model
Make prediction on the test data and display the output
END

Figure 3.2.1.1: Logical diagram showing the architectural concept of the system.

3.2.2 Architectural Description

The architecture of the system, as abstractly shown in the above figure starts from the collection of data, until training and testing.

The figure below however explains in a less abstract form the process of creating the machine learning model.

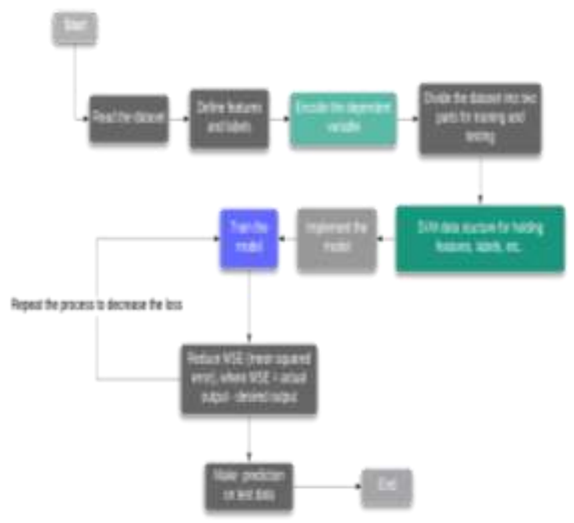

Start
Read the dataset
Define features and labels
Encode the dependent variable
Divide the dataset into two parts for training and testing
data structure for holding features, labels, etc.
Implement the model
Train the model
Repeat the process to decrease the loss
Reduce MSE (mean squared error), where MSE = actual output - desired output
Make prediction on test data
End

Figure 3.2.2.1: Diagram showing the process of creating our ML model

In this phase, we'll emphasize on the training model and algorithms to be used.

Training Phase – we will train our model with a list of permissions in android application packages gotten from https://www.kaggle.com/ (the list will be highlighted in the next chapter), we will feed the model with permission often requested by a clean apk and an infected one in order to be able to make predictions when new apk is tested.

Linear regression is a way to model the relationship between two variables, The equation of the **slope formula**. The equation has the form *Y= a + bX*, where Y is the dependent variable (that's the variable that goes on the Y axis), X is the independent variable (i.e. it is plotted on the X axis), b is the slope of the line and a is the y-intercept.

$$a = \frac{(\Sigma y)(\Sigma x^2) - (\Sigma x)(\Sigma xy)}{n(\Sigma x^2) - (\Sigma x)^2}$$ ………...

Equation 3

$$b = \frac{n(\Sigma xy) - (\Sigma x)(\Sigma y)}{n(\Sigma x^2) - (\Sigma x)^2}$$ ………….

Equation 4

Simple linear regression is a type of regression analysis where the number of independent variables is one and there is a linear relationship between the independent(x) and dependent(y) variable.

The straight line in **fig 3.1** is referred to as the best fit straight line. Based on the given data points, we try to plot a line that models the points the best. The line can be modelled based on the linear equation shown below.

$$Y = a\_0 + a\_1 * x$$ …………. Equation 5

Whereas multiple linear regression is given as:

$$Y = a\_0 + a\_1 * x\_1 + \ldots + a\_n * x\_n$$ ………….

Equation 6

The motive of the linear regression algorithm is to find the best values for a_0 and a_1. Before moving on to the algorithm, let's have a look at two important concepts for a better understand of linear regression.

Cost Function in Linear regression

The cost function helps us to figure out the best possible values for a_0 and $\boldsymbol{a\_1}$ which would provide the best fit line for the data points. Since we want the best values for a_0 and $\boldsymbol{a\_1}$, we convert this search problem into a minimization problem where we would like to minimize the error between the predicted value and the actual value.

$$minimize\ 1/n\, \Sigma^n{}_{i=1}(pred_i - y_i)^2$$

………….

Equation 7

$$J = \frac{1}{n\Sigma^n{}_{i=1}(pred_i - y_i)^2}$$

………….

Equation 8

We choose the above function to minimize. The difference between the predicted values and ground truth measures the error difference. We square the error difference and sum over all data points and divide that value by the total number of data points. This provides the average squared error over all the data points. Therefore, this cost function is also known as the Mean Squared Error (MSE) function. Now, using this MSE function we are going to change the values of a_0 and $\boldsymbol{a\_1}$ such that the MSE value settles at the minima.

Gradient Descent

A very important concept in linear regression is the gradient descent. Gradient descent is a method of updating a_0 and $\boldsymbol{a\_1}$ to reduce the cost function

(MSE). We start with some values for a_0 and a_1 and then we change these values iteratively to reduce the cost. Gradient descent helps us on how to change the values.

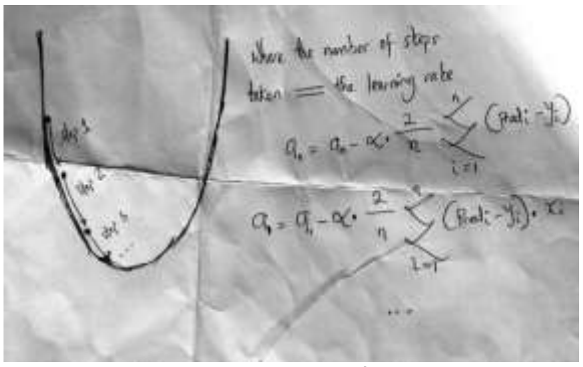
$$a_0 = a_0 - \alpha * \frac{2}{n\Sigma^{n}{}_{i=1}(pred_i - y_i)} \quad \ldots$$

Equation 9

$$a_1 = a_1 - \alpha * \frac{2}{n\Sigma^{n}{}_{i=1}(pred_i - y_i) * x_i}$$

………… Equation 10

and so on …



Where the number of steps taken = the learning rate
$a_0 = a_0 - \alpha \cdot \frac{2}{n} \sum_{i=1}^{n} (pred_i - y_i)$
$a_1 = a_1 - \alpha \cdot \frac{2}{n} \sum_{i=1}^{n} (pred_i - y_i) \cdot x_i$
…

Figure 3.2: Gradient descent of linear regression.

Having had the thorough understanding of how the machine learning works, we will implement using Tensor flow.

TensorFlow

TensorFlow is a core open source library to help develop and train ML models.

TensorFlow data is represented in the form of tensors. Now, Tensors are multidimensional arrays, an extension of two-dimensional tables (matrices) to data with higher dimension. We may now need to incorporate the concepts of the Scalar, Vector, and Matrix.

A scalar is a number like 8, -6, 0.56, etc.

A vector is a list of numbers (could be a row or a column)

A matrix is an array of numbers (one or more rows and columns)

The more general existence of tensors expresses the essential features of the scalar, vector, and matrix — making it sometimes very necessary to make use of tensors with orders that exceed two in both physical sciences and machine learning.

In machine learning, vectors often represent the feature vectors, with their individual components specifying how important a particular feature is. Such feature could include a relative importance in how specific the malware file is, or the intensity of a set of malware expression, etc.

In **linear regression**, there is always a need to list our data in x-y format (i.e. two columns of data—independent and dependent variables). Hence, the feature inputs are sure to be stored as vectors.

TensorFlow allow not only to build and train ML models easily with their intuitive high-level APIs like Keras with eager execution, which makes for immediate model iteration and easy debugging, but also with the help of the Tensorboard, allows us to be able to visualize our dataflow, our TensorFlow graph, and plot quantitative metrics about the execution of our graph.

Figure 3.3: Sample diagram of the Tensorboard showing quantitative metric of a loss function

IV. TEST AND RESULT

The implementation of this project begins by defining firsthand, the data, the tools, frameworks and technologies used in executing the methodology. These include what tools were used, the processes, and their requirements. This chapter then shows the development process of the system, with an understanding of organizing, training and managing data for the system, followed by the interactive process between the user, the system, and its data. All of these are layered out in the different sections and sub-sections; the detailed design, experimental testing, result and discussion, documentation, and the entire operation of the system.

TOOLS REQUIRED

- TENSORFLOW
- PYCHARM
- ANDROID STUDIO

- PROGRAMMING LANGUAGES: Python, Java
- A Computer with a minimum configuration of 4GB RAM, 2GHz processor's speed Windows OS / MAC OS / LINUX OS

4.3 SYSTEM OPERATION AND IMPLEMENTATION

For our data, we'll use a list of all the available Android permissions, which will be a result of our research production in ML and Android security. The data is obtained by a process that consisted to create a binary vector of permissions used for each application analyzed {1=used, 0=no used} — where the samples of malware/benign were divided by "Type"; 1 malware and 0 non-malware.

We obtained a total of 324 android permissions such as:

- android.permission. ACCESS_NETWORK_STATE - Allows applications to access information about networks.
- android.permission.ACCESS_FINE_LOCATION - Allows an app to access precise location from location sources such as GPS, cell towers, and Wi-Fi.
- android.permission.READ_LOGS - Allows an application to read the low-level system log files. Log entries can contain the user's private information.
- android.permission.INTERNET - Allows applications to open network sockets
- android.permission.RECEIVE_DATA_ACTIVITY_CHANGE
- android.permission.RECEIVE_SMS - Allows an application to monitor incoming SMS messages, to record or perform processing on them.
- android.permission.SEND_RESPOND_VIA_MESSAGE - Allows an application (Phone) to send a request to other applications to handle the respond-via-message action during incoming calls. Not for use by third-party applications
- android.permission.SEND_SMS_NO_CONFIRMATION - Allows an application to send SMS messages via the Messaging app with no user input or confirmation
- android.permission.WRITE_SMS - Allows an application to write SMS messages.
- android.permission.ACCESS_WIFI_STATE - Allows applications to access information about Wi-Fi networks

Then, we load the list of permissions into a datasheet:

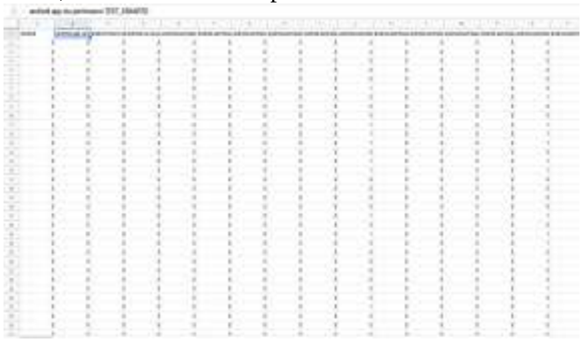


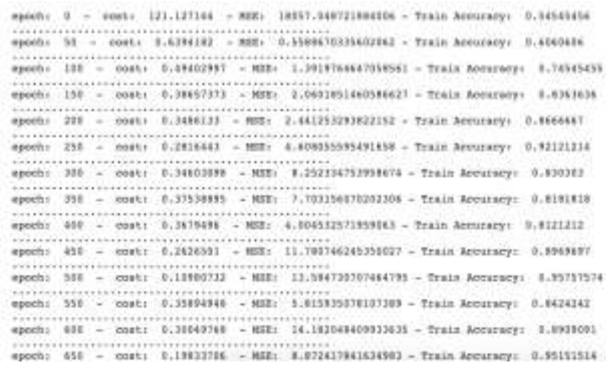
Figure 4.1: Data table

Where their types, noted by malware/benign were divided by "Type"; 1 malware and 0 non-malware:

Figure 4.2: Data table showing TYPE as 1 or 0

Using the datasheet, we load it into a CSV file in order to make it useable in training our model to be able to recognize the permission requested by a clean application package and an infected one by using python programming language to import the CSV file and using SVM feature in tensorflow to hold the label features in as vectors and pass into the model in form of an array. Following the trained pattern from the dataset, the model is able to recognize and predict when an application package is likely to be clean or infected at an instance of a new application package being fed in.

Below shows the epoch values by the difference of 50's and their training accuracy:



```
epoch:  0    -  cost:  121.127144   - MSE:  18557.348721884106 - Train Accuracy:  0.54545454
-------------------------------------------------------
epoch:  50   -  cost:  0.6394182    - MSE:  0.5508670335602862 - Train Accuracy:  0.6060606
-------------------------------------------------------
epoch:  100  -  cost:  0.49402997   - MSE:  1.3919766647058561 - Train Accuracy:  0.74545455
-------------------------------------------------------
epoch:  150  -  cost:  0.38657373   - MSE:  2.0692051460586627 - Train Accuracy:  0.6363636
-------------------------------------------------------
epoch:  200  -  cost:  0.3486133    - MSE:  2.441253293822152 - Train Accuracy:  0.8666667
-------------------------------------------------------
epoch:  250  -  cost:  0.2818443    - MSE:  4.608355595491858 - Train Accuracy:  0.92121214
-------------------------------------------------------
epoch:  300  -  cost:  0.34803898   - MSE:  8.252334753938674 - Train Accuracy:  0.830303
-------------------------------------------------------
epoch:  350  -  cost:  0.37538895   - MSE:  7.703356870202306 - Train Accuracy:  0.8181818
-------------------------------------------------------
epoch:  400  -  cost:  0.3679496    - MSE:  4.004532571959863 - Train Accuracy:  0.8121212
-------------------------------------------------------
epoch:  450  -  cost:  0.2626501    - MSE:  11.780746245358027 - Train Accuracy:  0.8969697
-------------------------------------------------------
epoch:  500  -  cost:  0.18900732   - MSE:  13.584720707484795 - Train Accuracy:  0.95757574
-------------------------------------------------------
epoch:  550  -  cost:  0.35894946   - MSE:  5.815835078107389 - Train Accuracy:  0.8424242
-------------------------------------------------------
epoch:  600  -  cost:  0.38049768   - MSE:  14.182048409333635 - Train Accuracy:  0.8939091
-------------------------------------------------------
epoch:  650  -  cost:  0.19833786   - MSE:  8.072417841634903 - Train Accuracy:  0.95151514
```

Figure 4.3: The epoch values with intervals of 50 and their training accuracy

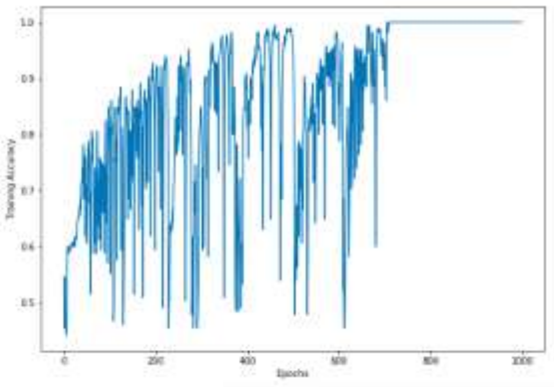
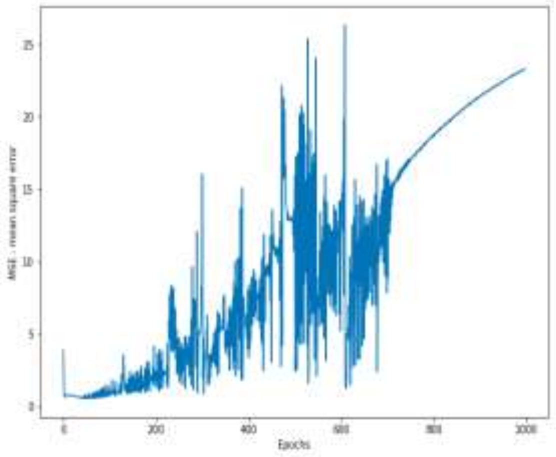
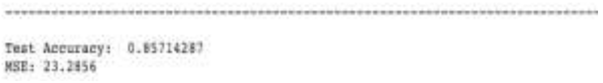


Figure 4.4: Training accuracy leveled at 0.9515

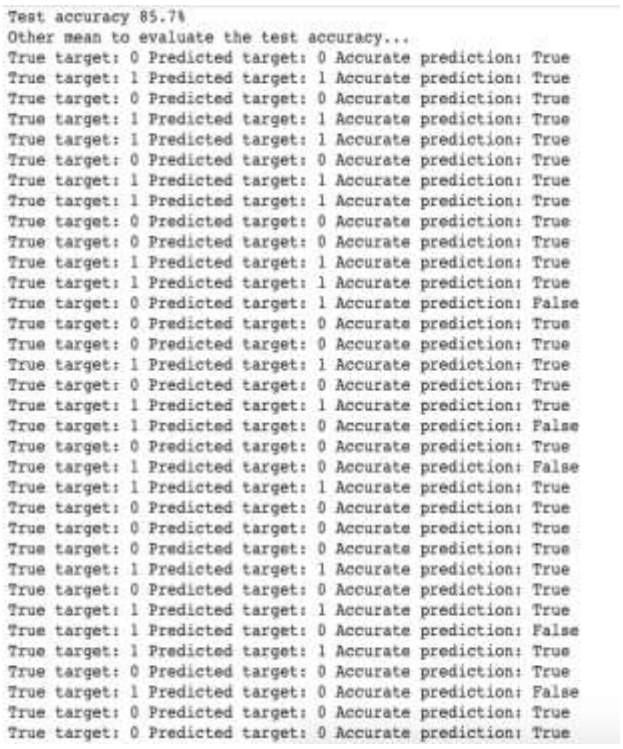




```
--------------------------------------------------------------------
Test Accuracy:  0.85714287
MSE: 23.2856
```

Figure 4.5: Test accuracy levels at 0.8571

While our MSE (Mean Squared Error) levels at 23.2856, our test accuracy levels at 0.8571.
This proves that our model is good enough to be served.



```
Test accuracy 85.7%
Other mean to evaluate the test accuracy...
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 1 Accurate prediction: False
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 1 Predicted target: 0 Accurate prediction: False
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 0 Accurate prediction: False
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 1 Predicted target: 0 Accurate prediction: False
True target: 1 Predicted target: 1 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 1 Predicted target: 0 Accurate prediction: False
True target: 0 Predicted target: 0 Accurate prediction: True
True target: 0 Predicted target: 0 Accurate prediction: True
```

Figure 4.6: Test accuracy reads 85% showing the true target, the predicted target and the prediction accuracy

V. CONCLUSIONS

The system design of this project was tested on a number of unbiased users and the following objectives were achieved:
A methodology has been implemented that is mildly capable of predicting which APKs are safe and which may contain malware. The learning process proved that this by correctly predicting the test data. And it was tested by a matter of how much it identifies the permissions in the manifest file, which is based on a levels of either malicious or not.

LIMITATIONS

The limitations of this system include its present inability to rightly discern other factors such as the malware family, as permission may not be the only factor to decide the presence of malware or not.

RECOMMENDATIONS

This project proves more advantageous as compared to other existing machine learning research work for sniffing malware in application packages, given its conceptual methodology. It therefore, would be optimistic to recommend that the new design be embraced and adopted. More so, future modification can be done to accommodate training of more rare and complex factors.

REFERENCES

[1] Alain Pimentel (2015). Detecting android malware by using a machine learning ensemble method. https://pdfs.semanticscholar.org/9923/097a3bf70c0642690ac1fd9eb245d367007d.pdf

[2] ElMouatez Billah Karbab, Mourad Debbabi, Abdelouahid Derhab and Djedjiga Mouheb (2018) MalDozer: Automatic framework for android malware detection using deep learning. Digital Investigation 24 (2018) S48 - S59

[3] Fairuz Amalina Narudin, Ali Feizollah, Nor Badrul Anuar and Abdullah Gani (2014). Evaluation of machine learning classifiers for mobile malware detection. https://umexpert.um.edu.my/file/publication/00001293_118859.pdf

[4] Hyunjae Kang, Jae-wook Jang, Aziz Mohaisen and Huy Kang Kim (2015). Detecting and classifying android malware using static analysis along with creator information. International Journal of Distributed Sensor Networks Volume 2015, Article ID 479174

[5] Iker Burguera, Urko Zurutuza and Simin Nadjm-Tehrani (2011). Behavior-based malware detection system for android. https://www.researchgate.net/publication/245022829_Crowdroid_Behavior-Based_Malware_Detection_System_for_Android

[6] Mahmudur Rahman, Mizanur Rahman, Bogdan Carbunar, Duen Horng Chau (2017). FairPlay: Fraud and Malware Detection in Google Play. https://www.researchgate.net/publication/313683480_Search_Rank_Fraud_and_Malware_Detection_in_Google_Play

[7] Marko Dimjaševic, Simone Atzeni, Zvonimir Rakamaric and Ivo Ugrina (2016). Evaluation of android malware detection based on system calls. https://dimjasevic.net/marko/wp-content/papercite-data/pdf/iwspa2016-daur.pdf

[8] Nguyen Viet Duc, Pham Thanh Giang and Pham Minh V (2015). Permission analysis for android malware detection. https://www.researchgate.net/publication/296704790_Permission_Analysis_for_Android_Malware_Detection

[9] Quan Qian, Jing Cai, Mengbo Xie and Rui Zhan (2016). Malicious Behavior Analysis For Android Application. International Journal of Network Security, Vol.18 No.1, PP.182-192, Jan. 2016

[10] Ryo Sato, Daiki Chiba and Shigeki Goto (2013). Detecting Android Malware By Analyzing Manifest Files https://www.researchgate.net/publication/272778915_Detecting_Android_Malware_by_Analyzing_Manifest_Files

[11] Sanya Chaba, Rahul Kumar, Rohan Pant, Mayank Dave (2018). Malware Detection Approach For Android Systems Using System Call Logs https://arxiv.org/ftp/arxiv/papers/1709/1709.08805.pdf

[12] Shuang Liang, Xiaojiang Du, Chiu C. Tan, Wei Yu (2016). An effective online scheme for detecting android malware. International Journal of Distributed Sensor Networks Volume 2015, Article ID 479174

[13] Suleiman Y. Yerima, Sakir Sezer and Igor Muttik (2014).0020Android Malware Detection Using Parallel Machine Learning Classifiers. 8th International Conference on Next Generation Mobile Applications, Services and Technologies, (NGMAST 2014), 10-14 Sept., 2014

[14] Suleiman Y. Yerima, Sakir Sezer and Igor Muttik (2015). High Accuracy Android Malware Detection Using Ensemble Learning. https://www.researchgate.net/publication/276158407_High_Accuracy_Android_Malware_Detection_Using_Ensemble_Learning

[15] Yonghong Huang, Utkarsh Verma, Celeste Fralick, Gabriel Infante-Lopez, Brajesh Kumar and Carl Woodward (2019). Malware Evasion Attack and Defense. https://arxiv.org/pdf/1904.05747

[16] https://gs.statcounter.com/os-market-share/mobile/worldwide