HamDroid: Permission-based harmful Android anti-malware detection using neural networks

Saeed Seraj
School of Architecture, Technology and Engineering, University of Brighton, Lewes Road, Brighton, BN2 4GJ, United Kingdom
S.Seraj@brighton.ac.uk

Siavash Khodambashi
Department of Computer Engineering, Yadegar-e-Imam Khomeini (RAH) Shahr-e-Ray Branch, Islamic Azad University, Tehran, Iran
siavashyp@yahoo.com
ORCID 0000-0001-9968-7364

Michalis Pavlidis
School of Architecture, Technology and Engineering, University of Brighton, Lewes Road, Brighton, BN2 4GJ, United Kingdom
M.Pavlidis@Brighton.ac.uk
ORCID 0000-0001-9135-2340

Nikolaos Polatidis*
School of Architecture, Technology and Engineering, University of Brighton, Lewes Road, Brighton, BN2 4GJ, United Kingdom
N.Polatidis@Brighton.ac.uk
ORCID 0000-0003-4249-4953

Abstract

Android platforms are a popular target for attackers, while many users around the world are victims of Android malwares threatening their private information. Numerous Android anti-malware are fake and do not work as advertised because they have been developed either by amateur programmers or by software companies that are not focused on the security aspects of the business. Such applications usually ask for and generally receive non-necessary permissions which at the end collect sensitive information. The rapidly developing fake anti-malware is a serious problem and there is a need for detection of harmful Android anti-malware. This article delivers a dataset of Android anti-malware, including malicious or benign, and a customized multi-layer perceptron neural network that is being used to detect anti-malware based on the permissions of the applications. The results show that the proposed method can detect with very high accuracy fake anti-malware while it outperforms other standard classifiers in terms of accuracy, precision, and recall.

Keywords: Android, malware detection, fake anti-malware, neural networks

*Corresponding author

1. Introduction

The number of Android devices has grown exponentially in the last few years. The Android market share is very high, while such devices provide computational power and connectivity, which allow users to store personal, sensitive, and confidential data, such as messages, credit/debit card details, files, and photos. The popularity of Android devices has made them an attractive target for cyber attackers. To protect devices from malware, security firms are usually pushing a virus-scanning app of some sort. However, according to a report from AV-Comparatives, an Austrian organization specialized in testing antivirus products, two-thirds of all Android antivirus apps are fake and don't operate as advertised [1]. The report was the result of an exhaustive testing process in which 250 Android antivirus apps available on the official Google Play Store were examined. Antivirus apps detecting themselves as malware showed the infirmity of the Android antivirus industry which appears to be filled with non-real cyber-security vendors. The company researchers searched for and downloaded 250 anti-malware security apps by various developers from the Google Play Store. Researchers installed each antivirus app on a separate device, having the test device download 2000 of the most common Android malwares automatically. As the report mentions, only 80 apps detected over 30% of malicious apps and had zero false alarms during individual tests meaning that most of the antiviruses were unable to find malwares. Most of the antiviruses appear to have been developed either by amateur programmers or by software manufacturers that are not security focused from a business point of view. In addition to that, tens of apps displayed the same user interface and were more interested in showing ads, rather than having a fully running malware scanner. They usually ask for and usually receive enough permissions that allow for the collection of personal user data such as the model of the phones, live GPS polling, phone numbers and other personally identifiable information. Many antivirus apps use a whitelist/blacklist approach instead of looking at the code. They would mark any installed apps as malicious if the app's package name was not included in its whitelist. According to studies and research that was done on android malware detection, useful solutions have been provided so far mainly based on machine learning techniques. Many datasets have been prepared and based on them; various approaches have been presented to identify android malwares [2]. However, the current free and commercial anti-malware are mainly based on signature, which is symptomatic that a threat must be widespread to be recognized. Existing methods usually treat all types of android applications in the same way to identify its harmfulness without considering the role of them. Thus, their approach is not usually feasible and useful for the end user.

Given the vital role of anti-malware applications, in this article we prevent a method for the detection of rogue android anti-malware applications using a Multi-Layer Perceptron neural network and creating a dataset of android anti-malware applications for training and testing the MLP network. We define as rogue android anti-malware application an application that is pretending to be anti-malware, but in fact its purpose is to deceive Android users and damage their security and privacy. To achieve this, we first hypothesize that a rogue anti-malware can be identified based on the requested permissions. We created a dataset of 1200 anti-malware applications with 328 specific permissions which might be asked during the installation and identified the harmful anti-malware applications in our dataset based on the recognition of many reputed antivirus companies using the report from VirusTotal [3]. Moreover, we define fake antimalware as rogue apps pretending to be antimalware but are malware instead and deliver a multi-layer perceptron neural network optimized for identifying fake anti-malware applications. The experimental results show that our proposed neural network outperforms other well-known classifiers in accuracy, precision and recall. In addition, the proposed method is feasible, straightforward to implement and fast due to limited number of nodes in the hidden layer of the neural network. The classification is performed fast with reasonable computational resources, since decisions made are only based on the permissions that an anti-malware asks for. The

proposed method can be used to accurately detect harmful android anti-malware applications before these are installed on a user's Android device.

This article delivers the following contributions:

- We use a new dataset of android anti-malware based on application permissions which is available in [4].
- We trained and customized a multi-layer perceptron (MLP) neural network to identify harmful android anti-malware.
- We evaluated our proposed method using well-known metrics with the results showing that fake anti-malware can be detected with very high accuracy.

The rest of the paper has been organized as follows: section 2 presents the related work, section 3 explains the dataset, section 4 describes the proposed method, section 5 contains the experimental evaluation and section 6 is the conclusion.

2. Related work

Various machine learning based Android malware analysis techniques which have been proposed in the literature and are described in this section. Particularly, three types of approaches have been proposed to identify android malwares: static, dynamic and hybrid.

2.1 Static approaches

Static analysis may be based on signature [5], permission [6] or Dalvik bytecode [7]. For better accuracy, a combination of these methods is possible [8]. A static approach examines malicious behavior in an application without executing it. The application is disassembled to its source code and analyzed by reverse engineering tools such as Apktool, dex2jar, dexdump, baksmali, dedexer. Specific patterns or signatures are generated through feature extraction of the source code and get compared with the signatures which previously recognized as harmful. Signature based analysis exploits either cryptographic or similarity measurement algorithms. However, cryptographic hashes are exact and suffer from code obfuscation. Fuzzy hashing [9] and SDHash [10] attempt to measure the similarity level between two files. SDHash is mostly used for image and video files, while fuzzy hashing performs well for text files [11]. Unlike permission-based approaches, a Dalvik bytecode-based analysis consumes power and storage space and fails on native code execution. DroidMOSS [12] implements a fuzzy hashing to defeat application repackaging changes. Faruki et al. [11] used fuzzy hash to create variable length signatures, comparing them with malware signatures in the database. However, this method is unreliable to detect zero-day malwares especially when the database is limited. To overcome this problem, the YARA project provides a public repository fed with malware signatures by a community of people [13]. Further, YARA rules help researchers to classify an application [14]. Wang et al. [15] extracted high risk permissions from the manifest file to identify malwares. Li et al. [16] introduced the permissions related to both malicious and benign characters. Talha et al. [17] and Sanz et al. [18] classified the apps as malware or benign based on the permissions. Verma and Muttoo [19] presented a malware detection framework based on machine learning which used Android permissions of an application. Milosevic et al. [20] proposed a machine learning Android malware detector based on permissions and source code analysis. Kang et al. [21] introduced a machine learning antimalware based on n-gram opcode feature extraction. ScanDal [22] and Uranine [23] analyzed the Dalvik bytecode and monitored privacy leakage to any remote server. It is sometimes hard to find the source of leakage and its target, plus that manipulation of sensitive

information by a malware is also possible. MLDroid is a recent framework for Android malware detection using machine learning techniques, API calls and app ratings. It is a very efficient way to detect malwares of unknown family type with very accurate results [33]. Another recent work that is based on permissions is delivered and as well that is based on linear regression. The algorithm relatively accurately detects malware in general but considers only precision and recall results [34]. GDdroid is an android malware detection methodology which is the first one that is based on graph neural networks and is able to detect malware in general with high accuracy [35].

Static methods are fast and secure with low resource consumption. Nevertheless, they are unable to analyze encrypted and obfuscated malwares. Most of static methods are usually incapable of dealing with unknown malwares and result to false positives. Hence, static analysis may need to be coupled with other security models for an efficient malware detection.

2.2 Dynamic approaches

In dynamic approaches the application is executed on an Android platform and all related system calls and network traffics are monitored. Malwares are identified through their runtime behavior and interactions with the system. Dynamic methods can deal with obfuscated and encrypted malwares. They can detect both known and unknown malwares with better accuracy than static analysis. However, they are slow, resource consuming and vulnerable due to the limitation of code reachability. Hence, they may be unsafe sometimes.

Programs are executed in a sandbox environment and their malicious behaviors are observed [14]. Dynamic analysis can be based on Andromaly [24], Taint [25], emulation [26] or memory [27]. Yet, dynamic approaches suffer from runtime detection methods [28] while they require a certain expertise and considerable amount of time. Andromaly [24] used machine learning methods to recognize malwares by monitoring system resources including CPU usage, network traffic, battery usage, number of active processes etc. Crowdroid [29] fed learning algorithms with system calls data collected by people in the cloud. Monitoring system calls requires a lot of resources and may lead to false alarm particularly when a legitimate application invokes too many system calls. Taintdroid [25] tracked data leakage by monitoring real-time data accesses and labeling sensitive data, although it is hard to determine a malicious outgoing flow. DroidScope [30] is a virtual machine introspection framework which detected attacks by monitoring OS and Dalvik semantics.

2.3 Hybrid approaches

In a hybrid approach, first the application is analyzed through a static method followed by dynamic analysis to enhance the accuracy and overcome both of static and dynamic limitations [8]. In general, the highest accuracy is usually obtained by hybrid methods. However, they are very time and resource consuming due to their complexity. Andrubis [31] was a hybrid method used both Dalvik and system monitoring to detect malwares. EspyDroid [32] was a hybrid method which tackled drawbacks of static approaches as well as runtime dependent parameters.

2.4 State of the art

Android applications are developed in Java mostly using Android SDK, compiled into dex (Dalvik executable) and packaged into .apk for installation. So far, various methods have been proposed for Android malware detection. However, they deal with all types of malwares the same way, including harmful anti-malware. To the best of our knowledge, no efficient approach has been proposed, for detecting fake harmful Android anti-malware threatening Android users.

The novelty of our proposed approach relies in the security aspect and more particularly it uses permissions of android malware to detect fake anti-malware and shows how a neural network should be trained and tuned to maximize accuracy. It is the first method in the literature that is based on app permissions and neural networks to identify fake anti-malware. While other works that are based on permissions exist, these are generally used for all kinds of android malware detection. Therefore, by having a tuned neural network based on permissions it is a fast and accurate way to detect fake antimalware only which is a very important sub-category of Android malware. While other methods can detect malware they do it either for all types of malware which require large datasets and the methods applied make it computationally expensive.

Thus, this study investigates how a permission-based analysis can provide a robust method that is able to identify malicious anti-malware apps in Android. To achieve this, we collected anti-malwares from official online stores and analyzed them using VirusTotal. This work provides a dataset of 1200 anti-malware including permissions plus HamDroid, a tuned neural network which identifies harmful anti-malware. HamDroid is fast, secure, and reliable and can identify harmful anti-malware and uses an optimized neural network.

3. The proposed dataset

We started our research based on the hypothesis that fake android anti-malware are recognizable through the special privileges that they require to be installed on the device. Hence, we downloaded over 1200 apk files of android anti-malware mostly from Google Play and other websites such as androidapksfree.com. We analyzed all apk files using VirusTotal [3] to extract all their features including internet access and other required app permissions. Moreover, we have used over 70 reputed anti-malware detection engines to classify the apk files into two groups, i.e., the regular anti-malware apps or the fake ones pretending to be regular but harmful for the device. Assessments showed that there are 869 regular apps out of the downloaded anti-malware while 331 of them are harmful. Besides, there are 328 specific permissions an anti-malware may ask for during installation on a device.

To create the dataset, we have put all the information in a file with .csv format which can be opened by many software applications. There are 329 columns in the dataset including 328 specific permissions plus the risk score column determining the harmfulness of the entries. The first row is column titles, and the rest are 1200 apk features extracted from android anti-malware apps. All values are in binary format i.e., 0 or 1. When an app requires a specific permission, the value in the related entry of the dataset is 1, accordingly unnecessary permissions of an app are zero. The last column i.e., risk score has also binary values. A downloaded app which is recognized malware by most of antivirus companies based on VirusTotal report, is risky and obtains 1 as its risk score. However, other safe android anti-malware have zero risk score. The complete dataset is accessible on [4] for further research. Figure 1 shows a small part of the dataset as a sample. The dataset is reliable since the classifications have been made through VirusTotal, a popular web-based antimalware scanning tool which relies on several antivirus engines and website scanners to identify malicious patterns.

1	INTERNET	CLEAR APP CACHE	GET TASKS	CHANGE WIFI STATE	READ PHONE STATE	SYSTEM ALERT WINDOW	WRITE EXTERNAL STORAGE	CALL_ PHONE	CAMERA	READ CALL LOG		USES POLICY FORCE LOCK	WAKE UP STOKER	Risk score
2	1	1	0	1	0	1	1	0	0	0		0	0	0
3	1	1	1	1	1	1	1	0	1	0		0	0	0
4	1	0	0	0	0	0	0	0	0	0		0	0	1
5	1	1	1	1	1	1	1	0	0	0		0	0	0
6	1	1	1	1	1	1	1	0	1	0		0	0	0

Fig. 1 An illustration of a small part of the proposed dataset

In the next section, we have proposed a multilayer perceptron neural network as a classifier for detecting fake android anti-malware. Our proposed dataset has been applied for training and verification of the MLP neural network. Other classification algorithms can be used as well for distinguishing original anti-malware, with the results presented in section 5.

4. Proposed method

We have used an MLP neural network as the proposed classifier. An MLP includes an additional layer of nodes i.e., more than just the input and output layer. Regardless of the input dimensionality, it turns out that a single-layer perceptron can solve a problem only if the data are linearly separable. The math performed by a multilayer perceptron allows for immense flexibility in the overall function and making it a good estimator in our case. It is a purely mathematical system approximating complex input-output relationships gradually. Hence, large amounts of data help the network to continue refining its weights and thereby achieve greater overall efficacy.

4.1 The proposed MLP neural network structure

According to our android antimalware dataset, we have proposed an MLP neural network for fake antimalware detection illustrated in Figure 2. The dimensionality of the permissions must match the dimensionality of the input layer. Each sample in our dataset includes 328 different privileges which decisions made based on them. Hence, there are 328 input nodes in the neural network structure. Furthermore, the classification here is a yes or no decision making. Accordingly, only one output node is needed even for so many input nodes and one hidden layer is enough for extremely powerful classification. The number of nodes within the hidden layer can be variable and we extensively search to find the optimal number through trial and error. According to our experimental results in section 5, for 328 input nodes and an output node, the optimal number of hidden nodes was obtained 16.

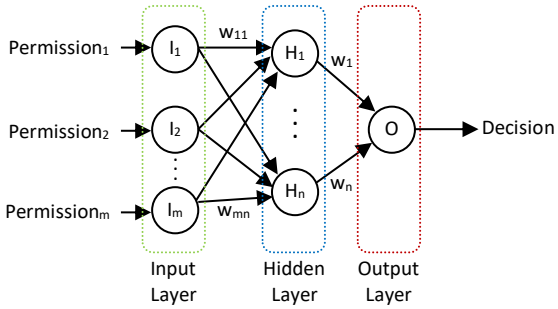

Fig. 2 the proposed multilayer perceptron neural network

Data that move from one node to another are multiplied by weights. Numerical data are summed as they arrive at computational nodes, then they are subjected to an activation function. We intended to train the neural network using gradient descent and we needed a differentiable activation function. We applied the standard logistic sigmoid as the activation function for both hidden and output nodes in the MLP structure (K=1, L=1) as shown in equation 1. Figure 3 shows the output value of the activation function versus the input.

$$f(x) = \frac{L}{1 + e^{-kx}} \quad \underset{\rightarrow}{L = 1, K = 1} \quad f(x) = \frac{1}{1 + e^{-x}} \tag{1}$$

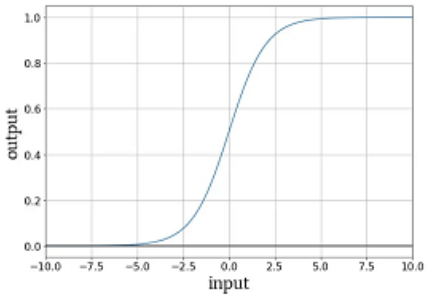

Fig. 3 The standard logistic sigmoid function

The logistic activation function is an excellent improvement upon the unit-step function because the general behavior is equivalent, but the smoothness in the transition region ensures that the function is continuous and therefore differentiable. The shape of the logistic curve as shown in Figure 3 with high derivative near the middle and low variations near the maximum and minimum, promotes successful training with contribution to the stability of the learning of the system. Equation 2 shows that the derivative of the logistic function is related to the original function. Hence, there is no need to use the derivative expression when we have already calculated the output of the logistic function for a given input value.

$$f'(x) = \frac{e^x}{(1+e^x)^2} = f(x)(1-f(x)) \tag{2}$$

Two pre-node and post-node signals are assumed for each computational node. A pre-node signal is computed by performing a dot product i.e., the corresponding elements of two arrays are multiplied and then all the individual products are summed. The first array holds the post-node values of the preceding layer to the current layer, and the second array includes the weights. The pre-node signal calculation is performed according to equation 3 where N denotes the post-node array of the preceding layer, n is number of nodes in the preceding layer, w denotes the weight vector and $preN_i$ denotes the computed value of pre-node signal for node N_i.

$$preN_i = w.N = w_1N_1 + w_2N_2 + \ ... + \ w_nN_n \tag{3}$$

Since there is no direct path to the output node from input-to-hidden weights, the relationship between these weights and the network's output is very complex as shown in Figure 4. For the sake of simplicity, we have shown the pre-node signal by the word pre plus the node's name and the post-node signal by the node's name. The pre-node signal is input to activation function of the node and the result would be assigned to the post-node signal according to Equation 1.

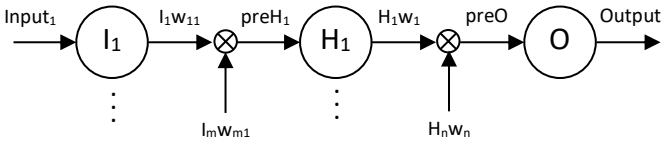

Fig. 4 Data-path of the proposed neural network

A gradient gives us information about how to modify weights. We need partial derivatives to make each weight modification proportional to the slope of the error function with respect to the weight being modified as shown in equation 4 in which α is the learning rate, *target* is the expected

output, *f'* is derivative of the logistic activation function, *input* and *output* are pre-node and post-node signals respectively.

$$weight_{new} = weight_{old} + a\ x\ (target - output)\ x\ f'(input) \quad (4)$$

We also need to update input-to-hidden weights based on the difference between the network's generated output and the target output values, but these weights influence the generated output indirectly. We send an error signal back toward the hidden layer and scale that error signal using both output weights of a hidden node as well as the derivative of that hidden node's activation function. This process is called backpropagation whereby weights are updated based on the weight's contribution to the output error. The steps are shown in equations 5, 6, 7 and 8.

$$FE = output - target \quad (5)$$

$$S_{error} = FE = f'(preOutput) \quad (6)$$

$$\delta_{HO} = S_{error}\ x\ H \quad (7)$$

$$weight_{HO} = weight_{HO} - \alpha\ x\ \delta_{HO} \quad (8)$$

FE is the final error value which is the difference between the post-node signal of the Output node and the correct output value, *f'* is the derivative of the activation function applied to the pre-node signal delivered to the Output node, error signal (S_{error}) is the final error propagated back toward the hidden layer through the activation function of the Output node, δ_{HO} represents the contribution of a given weight (hidden layer to output) to the error signal which is finally subtracted from the current weight in order to calculate the new value of that weight, α is the learning rate for changing the step size. For the input-to-hidden weights, the error must be propagated back through an additional layer as shown in Equations 9, 10 and 11.

$$\delta_{IH} = FE\ x\ f'(preOutput)\ x\ weight_{HO}\ x\ f'(preH)\ x\ ipnut \quad (9)$$

$$\rightarrow \delta_{IH} = S_{error}\ x\ weight_{HO} x\ f'(preH)\ x\ input \quad (10)$$

$$weight_{IH} = weight_{IH} - \alpha\ x\ \delta_{IH} \quad (11)$$

According to equation 9, the error signal is multiplied by hidden-to-output weight connected to the hidden node of interest as well as the derivative of activation function on that hidden node's pre-node signal multiplied by the input value. The input value can be thought of as the post-node signal from the input node.

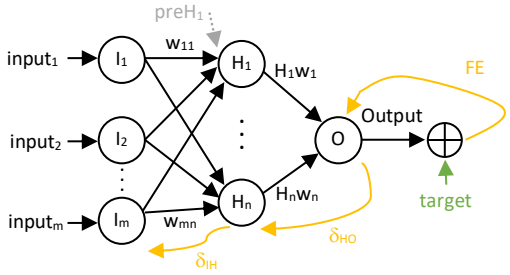

Fig. 5 Propagating error back to correct weights

At the next step, training takes place which is a process that allows a neural network to create a mathematical pathway from input to output. A neural network can perform classification because it automatically finds and implements a mathematical relationship between input data and output values via the training. The goal of the training is to provide data that allow the neural network to converge upon a reliable mathematical relationship between input and output. In this paper, we selected a portion of the dataset as training samples and gave the neural network input values and the corresponding output values. Training process applies a fixed mathematical procedure to gradually modify the network's weights such that the network will be able to calculate correct output values even with input data that it has never seen before. The MLP can approximate the true, generalized relationship between input and output only if we incorporate variety of anti-malware into our training samples, unless a deficient and oversimplified relationship would be found by the network. To avoid the neural network of being negatively affected by the order of training samples, we shuffled them after each epoch.

5. Experimental evaluation

Validation is a crucial aspect of neural-network development because the training dataset is inherently limited and therefore the network's response to this dataset is also limited. By validation, we perform to ensure that the trained neural network meets classification accuracy by running the trained network on new data and assessing the overall performance. According to Figure 2, our proposed neural network is limited to one output node. Hence, all we needed to do was performing a true/false type of classification. The inputs were also binary values, each representing a required permission. We have used the standard feedforward procedure to calculate the Output's signal value. Then applying a threshold that converts the signal value into a true/false classification result.

5.1 Evaluation metrics

To calculate the classification accuracy which shows the overall performance, we compared the classification result to the expected value for the current verification sample, counting the number of correct classifications, and dividing by the number of verification samples as illustrated in equation 12. Another important metric is Precision which describes what portion of predicted fake anti-malware are truly harmful and is calculated by equation 13. Equation 14 explains Recall metric which is the portion of actual harmful anti-malware that are correctly classified. The F1-score is a number between 0 and 1 and is the harmonic mean of precision and recall which is

computed according to equation 15. The value of 1 indicates perfect precision and recall, and the value of 0 means that either the precision or the recall is zero. In all cases TP stands for True Positive predictions, TN for True Negative predictions, FP for False Positive predictions, and FN for False Negative predictions. Accuracy, precision, and recall are widely used evaluation metrics in the literature [33, 36].

$$Accuracy = \frac{TP + TN}{TP + FP + FN + TN} \tag{12}$$

$$Precision = \frac{TP}{TP + FP} \tag{13}$$

$$Recall = \frac{TP}{TP + FN} \tag{14}$$

$$F1 = \frac{2\ x\ Precision\ x\ Recall}{Precision + Recall} \tag{15}$$

5.2 Experimental results and analysis

The proposed dataset can be applied for training and verification of our MLP neural network classifier. To simulate the MLP neural network, we developed the code in the Python programming language, using Numpy and Pandas libraries which are necessary for array operations as well as reading data from files. The code was made up of four phases: definition of the network's parameters including node numbers, learning rate etc., reading the dataset, training the neural network with a portion of dataset and at the last step, verifying the neural network with the whole dataset.

As mentioned before, the number of nodes in the hidden layer can be variable and there are also several parameters that can affect the classification result. An important question is that what the optimum value for those parameters is, which is a complex problem. To overcome this problem, we assumed that all the parameters with fixed initial values and smoothly change only one of them during several runs will obtain the best result. In the training phase, weights were tuned many times in which all the training samples were applied to the neural network in a random sequence and the weights were adjusted by comparing the classification result to the risk score of that sample. To visualize the simulation progress, the quality of classification was evaluated after each epoch. Although so many evaluations made the whole process slow, however we could find that when the result varied negligibly, and the simulation could be terminated. The optimum learning rate obtained 1.0 after several simulations, preventing very long runs and achieving the acceptable result after reasonable number of epochs.

Another important issue is how much of the dataset is acceptable to be used for training the neural network as well as for the verification. We tried the simulation with four different ratios: 1/8, 1/4, 1/2 and 1 for training in which samples were selected randomly from the dataset without any repeat and the final verification was evaluated by the whole dataset. It is obvious that the smaller training set would make the simulation faster. Hence, we started with training size of 1/8 to discover optimum learning rate as well as number of hidden layer's node of the MLP neural network by measuring accuracy of the classification according to equation (12). Table 1 shows the accuracy percentage achieved after enough epochs with different number of nodes in hidden

layer of the MLP neural network. The number of input nodes are 328 and the learning rate is 1.0. According to our experiences, as the epoch number expanded further than 90, overtraining happened and surprisingly the accuracy decreased.

Table 1. Simulation results with training size of 1/8 of dataset

No. hidden nodes	No. epochs	Accuracy %	No. hidden nodes	No. epochs	Accuracy %
2	15	77.52	16	30	85.96
3	15	83.39	16	60	87.06
4	15	79.91	16	90	87.52
5	15	81.01	16	120	86.97
6	15	77.71	32	15	80.27
7	15	83.49	33	15	70.55
8	15	81.93	34	15	73.39
9	15	82.48	64	15	82.94
10	15	82.11	128	15	81.38
11	15	81.19	168	15	77.71
15	15	80.01	329	15	76.88
16	15	77.34	329	328	79.27
16	15	85.78			

Table 2 shows the simulation results for different size of training sets with different epoch numbers. Here, the number of hidden nodes is 16 and the learning rate is 1.0.

Table 2. accuracy percentage obtained with different epochs and size of training sets

No. epochs	Accuracy % (size=1/8)	Accuracy % (size=1/4)	Accuracy % (size=1/2)
1	77.34	78.90	82.84
15	85.78	89.72	89.91
30	85.96	92.29	94.77
60	87.06	92.66	97.71
90	87.52	92.38	97.80

According to the above-mentioned results, it can be inferred that the best accuracy achieved when the learning rate was 1.0, the number of hidden nodes were 16 and training epochs were 90. To make sure that the classification results are stable, we have repeated the simulation for 5 runs and presented the results in detail in Table 3 where σ^2 is the variance. The simulation was executed on a Pentium core i5 @ 2.4 GHz, 4GB of RAM using the Microsoft Windows 10 operating system. The simulation software was a single threaded python application.

Table 3. Evaluation of the proposed MLP neural network with four well-known classification metrics

Training size ratio & Time		Accuracy	Precision	Recall	F1-score
1/8	Run1	0.8633	0.737327189	0.634920635	0.682302772
18 Minutes	Run2	0.86972	0.723577236	0.706349206	0.714859438

	Run3	0.85505	0.719626168	0.611111111	0.660944206
	Run4	0.87614	0.762331839	0.674603175	0.715789474
	Run5	0.84954	0.705607477	0.599206349	0.64806867
	Average	0.862751358	0.729693982	0.645238095	0.684392912
	σ^2	0.0000924302	0.0003684862	0.0015973796	0.0007575228
1/4 34 Minutes	Run1	0.93211	0.870833333	0.829365079	0.849593496
	Run2	0.93578	0.85546875	0.869047619	0.862204724
	Run3	0.913761	0.831932773	0.785714286	0.808163265
	Run4	0.9513761	0.878326996	0.916666667	0.897087379
	Run5	0.93578	0.864	0.857142857	0.860557769
	Average	0.93376142	0.860112371	0.851587302	0.855521327
	σ^2	0.000144234	0.0002554956	0.0018808264	0.000815139
1/2 67 Minutes	Run1	0.97431	0.934108527	0.956349206	0.945098039
	Run2	0.975229	0.931034483	0.964285714	0.947368421
	Run3	0.969725	0.904059041	0.972222222	0.936902486
	Run4	0.979817	0.956349206	0.956349206	0.956349206
	Run5	0.972477	0.947580645	0.932539683	0.94
	Average	0.9743116	0.93462638	0.956349206	0.94514363
	σ^2	0.0000112	0.0003174	0.0001764	0.000045
1 150 Minutes	Run1	0.99633	0.988188976	0.996031746	0.992094862
	Run2	0.9963302	0.984375	1	0.992125984
	Run3	0.99633	0.984375	1	0.992125984
	Run4	0.99633	0.988188976	0.996031746	0.992094862
	Run5	0.997248	0.988235294	1	0.99408284
	Average	0.99651364	0.986672649	0.998412698	0.992504906
	σ^2	0.0000002	0.0000036	0.0000038	0.0000006

We have compared our proposed neural network to other classifiers including SVM with a dot kernel type, convergence epsilon of 0,001, L pos 1 and L neg 1, Random-Forest with 100 trees and depth of 10, K-NN with k=5 and another neural network MLP baseline with two hidden layers of 50 nodes in each. Table 4 shows the comparison results in Accuracy, Precision, Recall and F1-Score metrics. Results show that HamDroid classifies Android anti-malware more accurately than other classifiers.

Table 4. Comparison of our approach to other well-known classifiers

	HamDroid (proposed)	SVM	Random Forest	Naive Bayes	K-NN	Neural network
Accuracy %	98.62	88.53	80.73	55.05	93.12	94.95
Precision %	95.56	86.84	100	34.90	81.67	83.87
Recall %	97.73	62.26	20.75	98.11	92.45	98.11
F1-Score %	96.63	72.52	34.36	51.48	86.72	90.43

Finally, we have used the area under the curve (AUC) and plotted the curve as shown in figure 6 below.

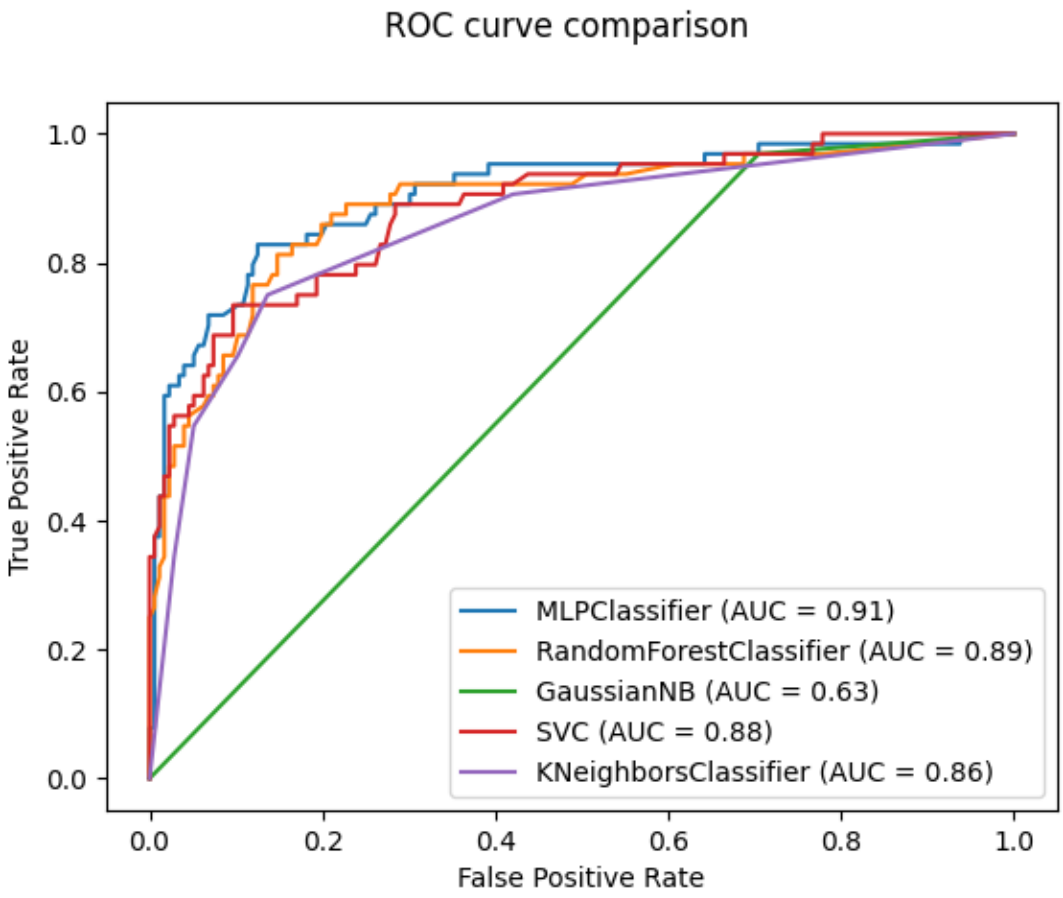

Fig. 6 AUC evaluation results

6 Conclusions

In this article, we examined Android malware detection methods including static, dynamic and hybrid. We showed the importance of identifying fake anti-malware which threaten many Android users. With regards to limited Android device resources, we came up with the idea that a static permission-based approach could provide effective and accurate results for classification of anti-malware, while performs well and in a reasonable time. Afterwards, we provided a dataset of all harmful and benign Android anti-malware, scanned by VirusTotal and other reputed antiviruses and their risk as well as permissions were identified. Moreover, we delivered an optimized neural network that can be used on the dataset to classify anti-malware with reasonable resource usage. We trained and verified our proposed method using the dataset and compared the results using well known metrics and classifiers. In the past there isn't any work published specifically on Android anti-malware. While previous works are based on complex or ensemble classifiers, HamDroid uses a straightforward, customized neural network which provides very accurate results. HamDroid is feasible because permissions can be extracted from the manifest file prior

to the installation of an Android anti-malware and classification would be done quickly by the customized neural network. The advantage of this method is due to the isolation and examination of anti-malware on a separate basis.

In the future we plan to collect data about other platforms and also investigate how deep learning neural networks will work on this and other platforms such as iOS.

Declarations

Conflict of interest The authors declare that they have no conflict of interest.

References

1. https://www.av-comparatives.org/tests/android-test-2019-250-apps/

2. A. Razgallaha, R. Khourya, S. Hallé, K. Khanmohammadi, A survey of malware detection in Android apps: Recommendations and perspectives for future research, Computer Science Review 39(2021), https://doi.org/10.1016/j.cosrev.2020.100358

3. www.virustotal.com

4. https://www.kaggle.com/saeedseraj/a-dataset-for-fake-android-antimalware-detection

5. V. Sihaga, M. Vardhan, P. Singh, A survey of android application and malware hardening, Computer Science Review 39(2021), https://doi.org/10.1016/j.cosrev.2021.100365

6. A. Mathur, L. Mounika, P. Ahmad, Y. Javaid, NATICUSdroid: A malware detection framework for Android using native and custom permissions, Journal of Information Security and Applications, 58(2021), 102696, https://doi.org/10.1016/j.jisa.2020.102696

7. V. Sihaga, M. Vardhan, P. Singh, BLADE: Robust malware detection against obfuscation in android, Forensic Science International: Digital Investigation 38(2021), 301176, https://doi.org/10.1016/j.fsidi.2021.301176

8. Arshad, S., Ali, M.,Khan, A.,Ahmed,M.: Android malware detection & protection: a survey. Int. J. Adv. Comput. Sci. Appl. 7(2), 466 (2016). https://doi.org/10.14569/IJACSA.2016.070262

9. Kornblum, J.: Identifying almost identical files using context triggered piecewise hashing. Digital Investigation 3(SUPPL.), 91–97 (2006). https://doi.org/10.1016/j.diin.2006.06.015

10. Roussev, V.: Data fingerprinting with similarity digests. In: IFIP Advances in information and communication technology, vol. 337 AICT, pp. 207–226. Springer, Berlin, Heidelberg (2010). https://doi.org/10.1007/978-3-642-15506-2_15

11. Faruki, P., Ganmoor, V., Laxmi, V., Gaur, M.S., Bharmal, A.: AndroSimilar: robust signature for detecting variants of android malware. In: Proceedings of the 6th International Conference on Security of Information and Networks—SIN ’13, pp. 152–159. ACM Press, New York, New York, USA (2013). https://doi.org/10.1145/2523514.2523539

12. [droidmoss?] Zhou, Y., Jiang, X.: Dissecting android malware: characterization and evolution. In: Proceedings - IEEE Symposium on Security and Privacy, pp. 95–109. IEEE (2012). https://doi.org/10.1109/SP.2012.16

13. YaraProject: YaraRules Project (2019). https://yararules.com/. Accessed 28 July 2019

14. YaraRules: yara-rules/rules (2019). https://github.com/Yara-Rules/rules

15. Wang, W., Wang, X., Feng, D., Liu, J., Han, Z., Zhang, X.: Exploring permission-induced risk in android applications for malicious application detection. IEEE Trans. Inform. Forensics Security 9(11), 1869–1882 (2014). https://doi.org/10.1109/TIFS.2014.2353996

16. Li, J., Sun, L., Yan, Q., Li, Z., Srisa-An, W., Ye, H.: Significant permission identification for machine-learning-based android malware detection. IEEE Trans. Indu. Inform. 14(7), 3216–3225 (2018). https://doi.org/10.1109/TII.2017.2789219

17. Talha, K.A.,Alper,D.I.,Aydin, C.:APK auditor: permission-based android malware detection system. Digital Investigation 13, 1–14 (2015). https://doi.org/10.1016/j.diin.2015.01.001

18. Sanz, B., Santos, I., Laorden, C., Ugarte-Pedrero, X., Bringas, P.G., Álvarez, G.: PUMA: Permission usage to detect malware in android. In: Advances in Intelligent Systems and Computing, vol. 189 AISC, pp. 289–298. Springer, Berlin, Heidelberg (2013). https://doi.org/10.1007/978-3-642-33018-6_30

19. S. Verma and S. K. Muttoo, “An android malware detection framework-based on permissions and intents”, Defence Science Journal, vol. 66, no. 6, pp. 618–623, 2016.

20. N. Milosevic, A. Dehghantanha, and K. K. R. Choo, “Machine learning aided Android malware classification”, Computers & Electrical Engineering, Elsevier, vol. 61, pp. 266–274, 2017.

21. B. J. Kang, S. Y. Yerima, K. McLaughlin, and S. Sezer, “N-opcode analysis for android malware classification and categorization”, In Proceedings of IEEE International Conference on Cyber Security And Protection Of Digital Services (Cyber Security) , June 2016, pp. 1-7 .

22. Kim, J., Yoon, Y., Yi, K., Shin, J., Center, S.: ScanDal: Static analyzer for detecting privacy leaks in android applications. In: MoST, vol. 12 (2012). https://pdfs.semanticscholar.org/7520/336ec2a08ad4fcbc5073082a8318571d679c.pdf.Accessed 17 Apr 2019

23. Rastogi, V., Qu, Z., McClurg, J., Cao, Y., Chen, Y.: Uranine: Real-time privacy leakage monitoring without system modification for android. In: Lecture Notes of the Institute for Computer Sciences, Social-Informatics and Telecommunications Engineering, LNICST, vol. 164, pp. 256–276. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-28865-9_14

24. Shabtai, A.,Kanonov, U.,Elovici,Y., Glezer,C.,Weiss,Y.: “Andromaly”: a behavioral malware detection framework for android devices. J. Intell. Inform. Syst. 38(1), 161–190 (2012). https://doi.org/10.1007/s10844-010-0148-x

25. Enck,W., Gilbert, P., Han, S., Tendulkar, V., Chun, B.G., Cox, L.P., Jung, J.,McDaniel, P., Sheth, A.N.: TaintDroid. ACMTrans. Comput. Syst. 32(2), 1–29 (2014). https://doi.org/10.1145/2619091

26. Zhang, F., Leach, K., Stavrou, A., Wang, H., Sun, K.: Using hardware features for increased debugging transparency. In: Proceedings—IEEE Symposium on Security and Privacy, vol. 2015-July, pp. 55–69 (2015). https://doi.org/10.1109/SP.2015.11

27. Sylve, J., Case, A., Marziale, L., Richard, G.G.: Acquisition and analysis of volatile memory from android devices. Digital Investigation 8(3–4), 175–184 (2012). https://doi.org/10.1016/j.diin.2011.10.003

28. Vidas, T., Christin, N.: Evading android runtime analysis via sandbox detection. In: Proceedings of the 9th ACM symposium on Information, computer and communications security—ASIA CCS'14, pp. 447–458. ACMPress, New York, New York, USA (2014). https://doi.org/10.1145/2590296.2590325

29. Burguera, I., Zurutuza, U., Nadjm-Tehrani, S.: Crowdroid: behavior-based malware detection system for Android. In: Proceedings of the 1st ACM workshop on Security and privacy in smartphones and mobile devices - SPSM '11, p. 15. ACM Press, New York, New York, USA (2011). https://doi.org/10.1145/2046614.2046619

30. Yan, L.K., Yin, H.: DroidScope: seamlessly reconstructing the OS and Dalvik semantic views for dynamic Android malware analysis. In: Proceedings of the 21st USENIX conference on Security symposium, pp. 1–16. USENIX Association Berkeley, CA, USA, Bellevue, WA (2012)

31. Lindorfer, M., Neugschwandtner, M., Weichselbaum, L., Fratantonio, Y., Veen, V.V.D., Platzer, C.: ANDRUBIS - 1,000,000 Apps Later: A View on Current Android Malware Behaviors. In: Proceedings—3rd International Workshop on Building Analysis Datasets and Gathering Experience Returns for Security, BADGERS 2014, pp. 3–17. IEEE (2016). https://doi.org/10.1109/BADGERS.2014.7

32. Gajrani, J., Agarwal, U., Laxmi, V., Bezawada, B., Gaur, M. S., Tripathi, M., & Zemmari, A. (2020). EspyDroid+: Precise reflection analysis of android apps. Computers & Security, 90, 101688.

33. Mahindru, A., & Sangal, A. L. (2021). MLDroid—framework for Android malware detection using machine learning techniques. *Neural Computing and Applications*, *33*(10), 5183-5240.

34. Şahin, D. Ö., Kural, O. E., Akleylek, S., & Kılıç, E. (2021). A novel permission-based Android malware detection system using feature selection based on linear regression. *Neural Computing and Applications*, 1-16.

35. Gao, H., Cheng, S., & Zhang, W. (2021). GDroid: Android malware detection and classification with graph convolutional network. *Computers & Security*, *106*, 102264.