applied sciences

Article

Deep Learning Based Android Anomaly Detection Using a Combination of Vulnerabilities Dataset

Zakeya Namrud [1,*], Sègla Kpodjedo [1], Chamseddine Talhi [1], Ahmed Bali [1] and Alvine Boaye Belle [2]

[1] Department of Software and IT Engineering, École de Technologie Supérieure, Montreal, QC H3C 1K3, Canada; segla.kpodjedo@etsmtl.ca (S.K.); chamseddine.talhi@etsmtl.ca (C.T.); ahmed.bali.1@ens.etsmtl.ca (A.B.)

[2] Department of Electrical Engineering and Computer Science, York University, Toronto, ON M2J 4A6, Canada; alvine.belle@lassonde.yorku.ca

* Correspondence: zakeya.namrud.1@ens.etsmtl.ca

Abstract: As the leading mobile phone operating system, Android is an attractive target for malicious applications trying to exploit the system's security vulnerabilities. Although several approaches have been proposed in the research literature for the detection of Android malwares, many of them suffer from issues such as small training datasets, there are few features (most studies are limited to permissions) that ultimately affect their performance. In order to address these issues, we propose an approach combining advanced machine learning techniques and Android vulnerabilities taken from the AndroVul dataset, which contains a novel combination of features for three different vulnerability levels, including dangerous permissions, code smells, and AndroBugs vulnerabilities. Our approach relies on that dataset to train Deep Learning (DL) and Support Vector Machine (SVM) models for the detection of Android malware. Our results show that both models are capable of detecting malware encoded in Android APK files with about 99% accuracy, which is better than the current state-of-the-art approaches.

Keywords: android security; deep neural network; machine learning; support vector machine

check for updates

Citation: Namrud, Z.; Kpodjedo, S.; Talhi, C.; Bali, A.; Boaye Belle, A. Deep Learning Based Android Anomaly Detection Using a Combination of Vulnerabilities Dataset. *Appl. Sci.* **2021**, *11*, 7538. https://doi.org/10.3390/app11167538

Academic Editor: Giancarlo Mauri

Received: 14 July 2021
Accepted: 10 August 2021
Published: 17 August 2021

1. Introduction

The adoption of mobile applications in a wide range of domains has made many activities, from banking to education or gaming, simpler, faster, or more convenient. The dominant mobile operating system is Android, thanks in part, to the high number of freely available apps accessible through its official market (Google Play Store [1]).

The reach of the Android system goes even beyond that official market since the open source OS allows users to install unofficial (e.g., third-party) apps. A key security feature of Android is its permission system; permissions sought by an Android application must be granted manually by the user of the mobile device before the app is installed (on older OS versions) or before the app can perform some operations (on newer OS versions). However, users are generally uneducated about the risks of the permissions they can be asked to grant. They may grant permissions allowing malicious apps to exploit security breaches [2] and to monitor a mobile device without the user's consent [3]. These malwares can cause severe malfunction, steal sensitive personal information (e.g., banking information, passwords), corrupt files, display unwanted advertisement, and even lock the device unless a ransom is paid.

According to Haystack [4], 70% of mobile apps fetch users' personal data and hand it over to third-party companies. Furthermore, a report published by AV-TEST security Institute [5] states that there is an exponential increase of new malicious program (malware) samples every year. In 2020, Kaspersky [6] detected around 5.7 million malicious installation packages for mobile devices, which was an increase of 2.1 million over 2019 (see Figure 1). Given this increasing influx of new malwares, typical signature-based malware detection approaches, which, in short, rely on databases of specific characteristics of known malwares are not up to the task of effectively safe-guarding Android devices from the malware threats. Malwares may go undiscovered if their signature is not identified in the database, and the databases must be continuously updated to stay relevant.

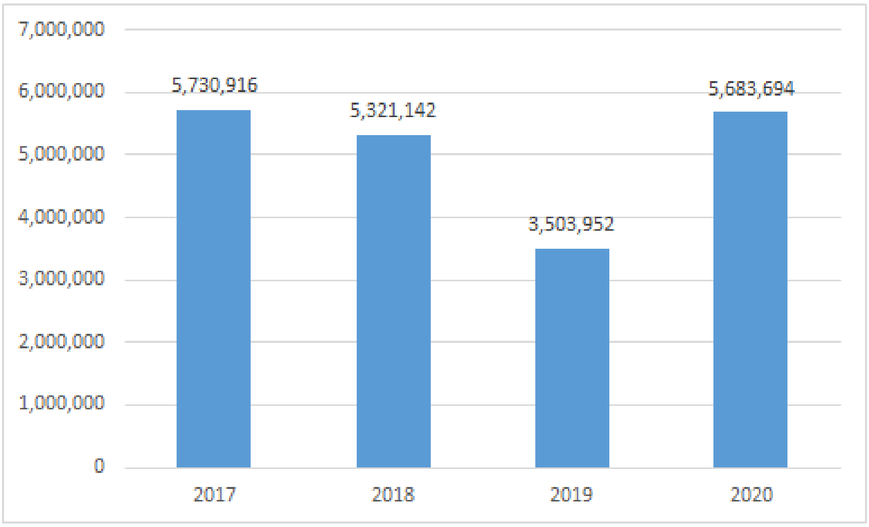


Figure 1. Installation of mobile malicious packages in Android from 2017 to 2020.

Research literature on malware detection (e.g., [2,7,8]) includes advanced proposals using machine learning techniques to detect with a higher accuracy unknown Android malware embedded in APK files. Such work typically extracts features (e.g., permissions, and API calls in the code) from known benign apps and malware, then uses machine learning algorithms (e.g., decision tree, Random Forest) to uncover ways to detect malicious apps.

The present work builds on AndroVul [9], our previous research work centered on the proposal of a dataset of vulnerability features of Android apps. Our current study, not only adds around 6 K apps to that dataset, but most importantly explored the use of advanced techniques such as Deep Learning (DL) and Support Vector Machine (SVM) to achieve the highest possible malware classification performances. Overall, we started from reverse-engineering an Android application APK file into a set of vulnerabilities features that can be used to reflect the application's behaviors. As a result, we obtain a dataset of more than 18 K apps (about 6 K more than in the original paper) and 74 vulnerabilities features, which we use to experiment on DL and SVM models. Thus, our contributions can be summarized as follows:

1. We developed a malware detection model based on deep learning and we investigated several node architectures in hidden layers in order to get the highest possible performance. The proposed model outperforms the state-of-the-art.
2. We developed a malware detection model based on SVM and investigated different parameter settings to identify which were the best for our malware detection task.
3. We provide comparison of the performance of our DL and SVM classifiers, with respect to state-of-the-art approaches and even some commercial anti-viruses and results show that our classifiers are the most effective in identifying malicious applications. As such, our models establish a new, important reference point in the current state-of-the-art when it comes to malware detection.

The remainder of this paper is organized as follows: Section 2 introduces some background concepts. Section 3 describes the overall design of our Android malware detection system and how it operates. Section 4 shows the experimental results obtained when assessing the performance of our models. Section 5 presents related work. Section 6 concludes the paper and outlines future work.

2. Background

In this section, we define some concepts needed to better grasp our approach.

2.1. Android Vulnerabilities

Vulnerabilities, commonly referred to as security-sensitive defects, can be found statically using rules that describe vulnerable code patterns. They are typically diverse in terms of the components involved, the attack vector necessary for exploitation, and so forth. We focus on common vulnerabilities that have a severity level that warrants their inclusion in security reports and earlier Android security research in this study. We briefly list the vulnerabilities that we have taken into consideration as features in our work.

2.1.1. Dangerous Permissions

The permission system in Android is a critical security feature since it regulates the rights granted to apps, requiring them to request particular permissions in order to execute specific operations. This approach necessitates the declaration by app developers of which sensitive resources will be utilised by their applications. When installing or using the apps, app users must consent to the requests made by the developers. According to Android, there are several categories of permissions, among which are "dangerous" ones, which are deemed more critical and privacy sensitive because they grant access to system features such as cameras and internet access as well as personal contact information and SMS messages, among other things [10].

2.1.2. AndroBugs Vulnerabilities

AndroBugs is a well-known security testing tool for Android applications, and it is used to evaluate them for vulnerabilities and possibly critical security issues. APKs are reverse engineered using the tool, which searches for a variety of concerns, ranging from a lack of adherence to best practices to the usage of potentially dangerous shell commands or the exposure to vulnerabilities via third-party libraries. It has a demonstrated track record of uncovering security flaws in some of the most popular applications and software development kits (SDKs). It is run as a command line utility and generates reports with four severity levels. Critical: Confirmed vulnerability that should be solved (except for testing code), Warning: Possible vulnerability that should be checked by developers, Notice: Low priority issue, and Info: No security issue detected.

2.1.3. Code Smell

Code smells refer to code source items that may suggest more serious issues in the code [11]. The term "security code smells" refers to "symptoms in the code that signal the possibility of a security vulnerability" in Android applications, according to Ghafari et al. [12]. Following a review of the literature, they identified 28 security code smells [12] that they categorized into five categories, including Insufficient Attack Protection, Security Invalidation, Broken Access control, Sensitive Data Exposure, and Lax Input Validation.

2.2. Machine Learning (ML)

Machine Learning (ML) refers to a class of methods for automatically creating models from data. These methods allow solving complex problems such as anomaly detection, classification, clustering, and regression [11]. As Verbraeken et al. [11] point out, a problem can be solved with ML through two phases: training and prediction. The training phase results in a trained model, after which the trained model is deployed in practice at the prediction phase. During that phase, the trained model is fed with new data and generates predictions by inferring these new data. Different ML algorithms (e.g., supervised, unsupervised, classification, regression) have been proposed depending on the kind of feedback that the algorithm receives while learning [11]. Machine learning techniques have been deployed in related proposals by some other security researchers in articles such as [13,14].

In the current work, we investigated two of the most powerful families of machine learning techniques: Deep Learning (DL) and Support Vector Machines (SVM).

2.2.1. Deep Learning (DL)

Deep learning [15] is a subfield of Artificial Neural Networks (ANNs) and ML. The DL approach is rapidly gaining traction and is widely utilised in computer vision, speech recognition, and natural language processing. At the same time, DL-based malware detection for Android has become a major trend. A typical DL model for data processing is an extremely deep neural network with numerous hidden layers of many linked neurons. Each layer consists of several different neurons, each with its own weights and likely activation mechanism. When data are fed into a neural network, the loss function computes the prediction error. The optimizer is used to progressively change the weights in order to reduce the loss function error and increase the accuracy. It trains the data and assesses its accuracy on the test set. One of the dominant models in deep learning is ANNs [16] which have been widely used for image recognition and have shown promising results in contextual categorization in DL. An ANN algorithm can learn hidden patterns from data on its own, combine them, and create much more powerful decision rules [8]. Figure 2 shows the overall DL definition, which is composed of three layers, namely the input layer, the hidden layer, and the output layer.

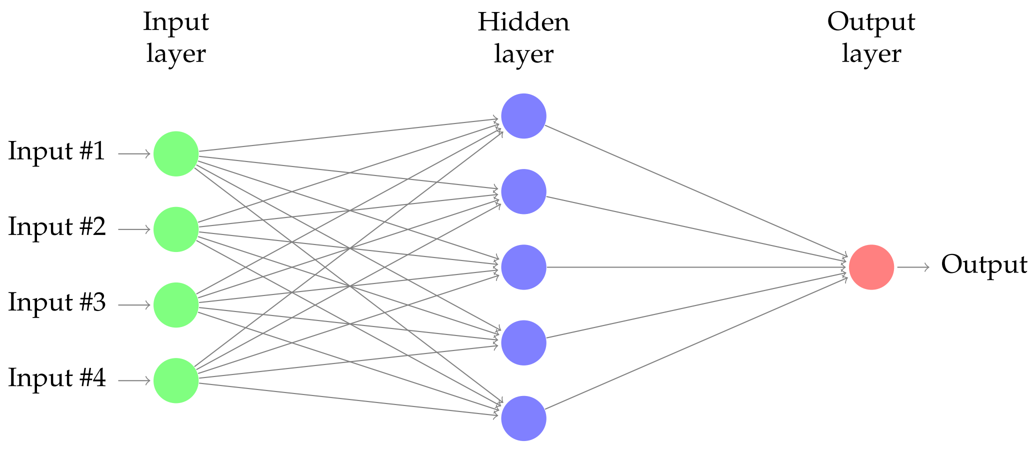


Figure 2. General architecture of a Deep Learning model.

2.2.2. Support Vector Machines (SVM)

A Support Vector Machine (SVM) is a machine learning technique that, in its most basic version, consists in finding a line that separates two classes of (training) data points, in such a way that future data points can be accurately classified, depending on which side of the line they end up on. In most cases, the line will be an hyperplane because the data will often be in an N-dimensional (N denotes the number of features) space [17]. Additionally, among the possible hyperplanes that could separate the data into two classes, the hyperplane which is the furthest from the two data classes it is separating is preferred because it reduces risks of overfitting the model to the current training data. Furthermore, it is not always possible (or even desirable) to neatly separate all the data into two perfectly clean classes. A certain amount of mis-classification can be allowed in order to account for outliers or erroneous data. Finally, the boundary between the two classes of data may not be linear, in which case, there is a need to involve mathematical kernels that can handle those situations. In our study, we use a nonlinear SVM with a Gaussian radial basis function (rbf) kernel, which is a well established and robust version of SVM. When constructing such a classifier, two parameters must be passed as arguments. The parameter C accounts for the intolerance to mis-classification of the training data; the higher it is, the more the training data points will have to be correctly classified. The other parameter gamma can be understood as controlling the influence of a single training data point; the higher it is, the lower the reach of a single data point. The parameters C and gamma work together and have to be carefully chosen.

3. Methodology

In this section, we present the sample of apps on which we performed our experiments, the features we extract from a given apk, and information about the machine learning techniques we selected.

3.1. Dataset

Our evaluation was carried out with the AndroVul [9] dataset, which core is a sample of 18,780 Android apps collected from the AndroZoo [18] repository. The Androzoo project proposes along with the apks of its apps, metadata that includes the number of antiviruses from the website Virus Total [19], compiles a vast range of antivirus products and web virus scanners. that flagged the app as a malware. For our study, and consistent with [9], we considered as benign apps the apps with zero flags and as malicious apps, the apps with two or more antivirus flags. To this core set of apps, we added malwares gathered from VirusShare [20], a malware repository intended to help security analysts and malware researchers. However, the VirusShare repository is not dedicated to Android malwares and does not propose any mechanisms or metadata to quickly identify which programs are apks and which are not. It simply hosts a variety of files without even a specified extension. To recover the Android malwares in that repository, we had to download Giga Bytes worth of potentially harmful files and figure out a simple procedure to identify which files were Android apks. As seen in Algorithm 1, we renamed all the files by adding the extension ".apk", then tried to apply our reverse engineering scripts and tools. Files that return empty folders after the reverse engineering are discarded; the others are saved as apks.

Algorithm 1: VirusShare Android apps collection.

```
   Input: Execution files
1  for all files in folder do
2      $file \leftarrow rename$(file.apk)          ▷considering all files Android apps
3      $APK_{file} \leftarrow Open$(file)          ▷Reverse engineering APK with Android tools
4      $Package \leftarrow Get$(files) if ($Package \leftarrow empty$) then
5          $Package \leftarrow delete\_it$
6      else
7          $Package \leftarrow App\_android$
8      end
9                  ▷ App_android save it in Android folder
10 end
```

Table 1 contains a description of the datasets, and Figure 3 depicts a visualisation of the datasets.

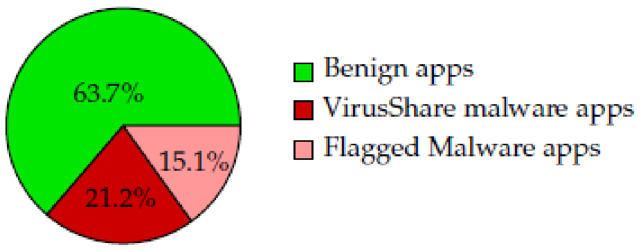


Figure 3. Dataset Visualization.

Table 1. Dataset description.

App	Samples
Benign	11,971
Flagged Malware	2831
VirusShare Malware	3978
Total	18,780

3.2. Feature Extraction

We introduced the AndroVul dataset in [9] and the interested reader can find full details in that publication. In this section, we propose a brief overview of the feature extraction process and output as applied to an app. In short, we used well-known static analysis tools (Apktool, AngroBugs) to extract three kinds of features, i.e., dangerous permissions from the app's manifest file, code smells from the app's Smali code representation, and AndroBugs vulnerabilities from the APK files. In Algorithm 2, we describe the general process for extracting the vulnerability features from an apk file. It starts with the reverse engineering process (line 2), followed by the extraction of the desired features from their respective files (lines 3–5). The extracted vulnerabilities features are then mapped to values and written into a single csv file. The values mapped to the features are determined as shown in the equations below:

$$permissions \rightarrow \begin{cases} Requested\_permission & 1 \\ Other\_permissions & 0 \end{cases}$$

$$CodeSmells \rightarrow \Big\{ Weight(security\,code\,smell)$$

$$AngroBugs\_vulnerabilities \rightarrow \begin{cases} Critical & 1 \\ Worning & 0.5 \\ Other & 0 \end{cases}$$

Algorithm 2: Feature Extraction Algorithm.

```
Input: Apk files; apps
Output: Dataset in CSV_file
1  for all apps in Dataset do
2      $APK_{file} \leftarrow Open(\text{file})$    ▷ Reverse engineering APK
3      $Permissions_{list} \leftarrow Get\_Distinct\_Permissions(manifest_{File})$    ▷ Extracting permissions from manifest file
4      $Code\_Smell_{list} \leftarrow Get\_CodeSmell(Smali_{Files})$    ▷ Extracting code smell from Smali files
5      $AngroBugs\_vulnerabilities_{list} \leftarrow Get\_AngroBugs\_vulnerabilities(AndroBugs\_report)$    ▷ Extracting
        AngroBugs_ vulnerabilities from AndroBugs_report
6      foreach app do
7          $Permission \leftarrow App[i].Permission$    ▷ Mapping permissions
8          $Code\_Smell \leftarrow App[i].CodeSmell$    ▷ Mapping code smells
9          $AndroBugs\_vulnerabilities \leftarrow App[i].AngroBugsvulnerabilities$    ▷ Mapping AndroBugs_
            vulnerabilities
10     end
11 end
12 $CSV_{(file)} \leftarrow Append(CSV_{(file)}, Concat(Vector_{(Permission)}, Vector_{(Code\_Smell)}, Vector_{(AngroBugs\_vulnerabilities)}))$
        ▷ Concatenating all vulnerabilities features in CSV file
13 return ($CSV_{(file)}$)
```

3.3. *General Architecture of Our Machine Learning Approach*

Figure 4 presents the general architecture of our Android malware detection approach, which is divided into three stages: pre-processing, training, and detection.

In the preprocessing phase, the original feature dataset is standardized by reducing the mean and scaling to unit variance. The following formula is used to compute the standard score of sample x:

$$z = (x - u)/s \tag{1}$$

where u denotes the mean of the training samples, and s denotes the standard deviation of the training samples.

As for training and testing, we opted for K-fold cross-validation. This validation approach consists in splitting, after random shuffling, the dataset into K groups, after which each group is used as a test group, while the other K − 1 groups are used for training. More specifically, we chose, in accordance to many similar studies (e.g., [21]), K = 20 for a fold cross-validation study, in which 80% of the data is used for training and 20% for testing (prediction).

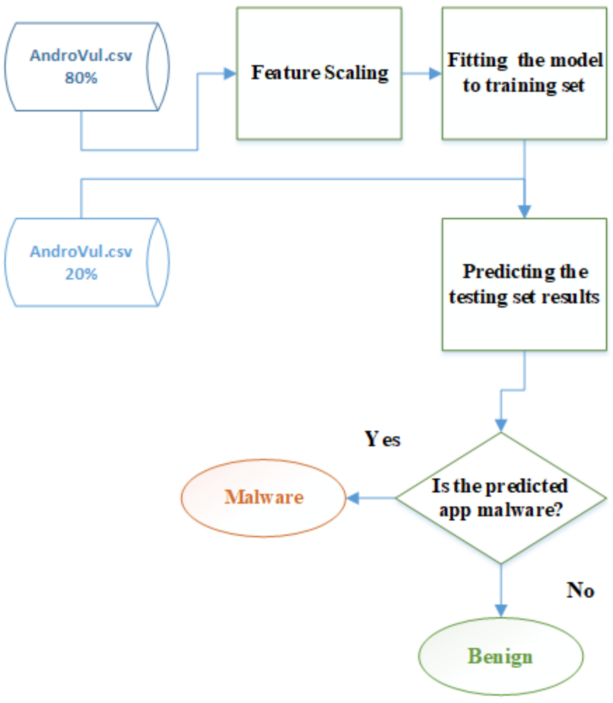

AndroVul.csv 80%
Feature Scaling
Fitting the model to training set
AndroVul.csv 20%
Predicting the testing set results
Yes
Is the predicted app malware?
Malware
No
Benign

Figure 4. Design Methodology for malware detection in Android.

3.4. *Android Malware Detection Based on Deep Learning*

Figure 5 presents our system architecture for Android malware detection using Deep Learning.

In the training phase, the malware and benign behavior patterns are learned by the ANN [22]. The DL model was designed for learning the pattern with four hidden layers, a single input layer and a single output layer. Fully connected feed-forward deep neural network architecture with four hidden layers was utilised to implement the suggested approach. The rationale for limiting the number of hidden layers to four is the complexity of the design. Figure 5 shows the architecture of the DL algorithm and the number of nodes in each layer, and Figure 4 shows the overall model design. The description of mapping layers is as follows. Seventy-four neurons are used in the input layer to read the 74 features. The neurons are then linked to the first hidden layer, known as the dense layer, where a mathematical computation is performed using activation functions [22]. The ReLU activation mechanism was used in this case. Another non-linear activation feature that has become common in the field of DL is the ReLU function. ReLU stands for Rectified Linear Unit. The key benefit of using the ReLU mechanism is that it does not simultaneously activate all neurons. The neurons are therefore disabled only if there is less than 0 in the output of the linear transformation. In addition, the 74 features were also mapped into 74 dimensions, and 74 dimensions are mapped into 32 dimensions in the second hidden layer. Thirty-two dimensions were mapped in 32 dimensions of the third hidden layer. Finally, their values were mapped to a single-dimensional output layer. Equations (2) and (3) define the activation function Y.

$$Y = \max(0, x) + bias \tag{2}$$

where Y denotes the output, x denotes the data, and bias is used to train the neural network on malware and benign patterns.

$$Weighted_{sum} = \sum_i W_i X_i \tag{3}$$

where W_i denotes the weight applied to each input node and X_i denotes the input applied to each node.

At the final output layer, the sigmoid function is applied to provide output values ranging from 0 to 1. A function of activation is defined by Equation (4).

$$Sigmoid = \left(\frac{1}{1 + e^{-x}}\right) \tag{4}$$

Binary Cross Entropy loss function was used to compile the neural network model.

$$BinaryCrossEntropy = Error(y, f(X)) \tag{5}$$

where y = actual values, $f(X)$ = predicted values. Furthermore, the weights are changed using the gradient descent optimizer [23]. It changes the parameters in such a way that the loss function can be reduced using Equation (6).

$$X = X - \alpha\left(\frac{\sigma}{dX} j(X)\right) \tag{6}$$

where X is the new updated weight, α denotes the rate of learning, and $f(X)$ denotes the cost function, which is a quadratic equation based on the 74 features extracted from Android applications. There are two primary hyperparameters that govern the network's architecture or topology, which are the number of layers and nodes found within each hidden layer. Systematic experimentation allows configuring these hyperparameters when solving a specific predictive modeling problem. We have increased the number of epochs until the model was able to correctly classify the inputs. In the test phase, the DL model is tested using 20% of the dataset. After training the model, we tested it using the remaining 3706 samples. The trained neural network model determines whether the provided APK file is malicious or benign based on the pattern. In the first experiment, the training phase results were significantly higher than the testing phase's, causing overfitting in the model. However, when we increased the number of samples in the dataset for both benign samples and malware samples, the overfitting was solved and the performance in the training phase was almost the same as that in the testing phase.

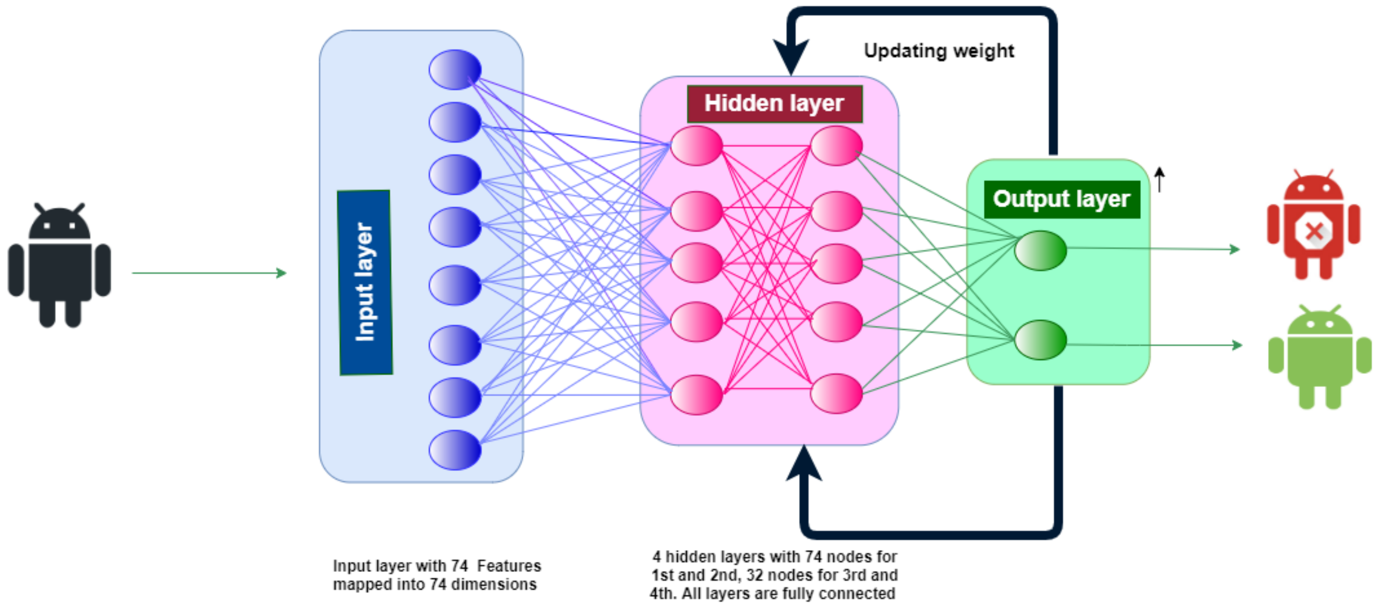
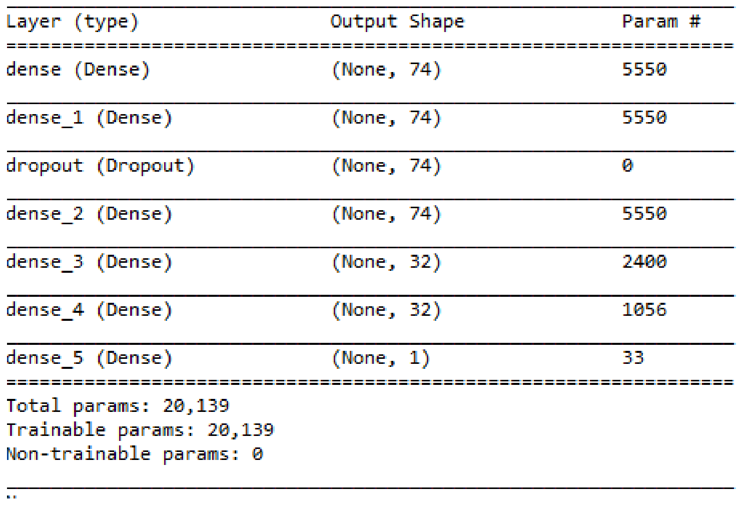

Input layer
Hidden layer
Output layer
Updating weight
Input layer with 74 Features mapped into 74 dimensions
4 hidden layers with 74 nodes for 1st and 2nd, 32 nodes for 3rd and 4th. All layers are fully connected

Figure 5. The architecture of DL layers using Sequential neural network.

In Algorithm 3, we describe the general process of our classifier generation phases namely, data processing (lines 1–3), model building (line 4), and model fitting (line 5) as well as using the model for prediction (lines 6–13). For the complexity discussion, we focus on the prediction phase because once the model is trained, it can be reused as much as there is a need. The time and space complexity of the model's prediction phase is $O(p \times n_{l1} + ... + n_{li-1} \times n_{li} + ... + n_{ln-1} \times o)$, where p is the number of features, n_{li} is the number of neurons at layer i in a neural network, and o is the number of outputs. Therefore, the complexity is asymptotically quadratic, $O(\max(n_{li-1} \times n_{li}))$, in the size of the network layers. The architecture and parameters of our DL model were determined experimentally and tuned for best performance; they are described in Table 2 and Figure 6.

Algorithm 3: Deep Learning based Model.

```
   Input: $X : apps\_features, Y : labels$              ▷ label is benign or malware
   $Building\_params_{list} = (units, activation\_function, input\_dim, dpoint)$
   $fitting\_params_{list} = (X_{train}, Y_{train}, batch\_size, epochs)$
   $dpoint : decision\_point$                            ▷ 0.5 by default
   Output: $predicted\_app$                              ▷benign or malware app
1  $X_{train}, X_{test}, Y_{train}, Y_{test}$            ▷ Splitting the dataset
2  $X_{train} = sc.fit_{transform(X_{train})}$           ▷Feature Scaling using StandardScaler (sc)
3  $X_{test} = sc.transform(X_{test})$
4  $Model = build\_ANN\_model\_architecture(Building\_params)$    ▷ANN model Building
5  $Model \leftarrow ANN\_model\_fit(fitting\_params)$   ▷ANN model fitting
6  $y_{pred} = Model.predict(X_{test})$
7  for $app.pred \in y_{pred}$ do
8      if $(app.pred > dpoint)$ then
9          $app \leftarrow Malware$ ;
10     else
11         $app \leftarrow Benign$ ;
12     end
13 end
```



```
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 74)                5550
_________________________________________________________________
dense_1 (Dense)              (None, 74)                5550
_________________________________________________________________
dropout (Dropout)            (None, 74)                0
_________________________________________________________________
dense_2 (Dense)              (None, 74)                5550
_________________________________________________________________
dense_3 (Dense)              (None, 32)                2400
_________________________________________________________________
dense_4 (Dense)              (None, 32)                1056
_________________________________________________________________
dense_5 (Dense)              (None, 1)                 33
=================================================================
Total params: 20,139
Trainable params: 20,139
Non-trainable params: 0
_________________________________________________________________
..
```

Figure 6. Built DL model.

Table 2. Best hyper-parameters.

Parameters	Value
Number of units	74-74-74-32-32-1
Number of layers	one input, 4 hidden, one output
Activation function	relu, sigmoid
Kernel initializer	uniform
Dropout	0.2
optimizer	adam
epochs	1000
batch_size	200
loss	binary_crossentropy

3.5. Android Malware Detection Based on Support Vector Machine

We opted for a (non linear) Radial Basis Function (RBF) kernel SVM. In Algorithm 4, we describe the general process of our SVM classifier, from the data processing (lines 1–3), to the model building (line 4), the model fitting (line 5) as well as the prediction phase (lines 6–9). The complexity of the training phase is polynomial, $O(n_{sv}^2 \times p + p^3)$, in the size of the model parameters, where p is number of features and n_{sv} is the number of support vectors). That relatively high complexity of the training model is compensated by the low complexity of the prediction model, which is of only $O(n_{sv} \times p)$ and can be reused several times once the model is well trained. Table 3 shows the best hyper-parameters for SVM model.

Table 3. The experimental results for SVM parameters showing the best hyper-parameters for SVM model.

C	Gamma	Accuracy	F1	AUC_score
10	0.1	93.98%	91.4%	94%
100	0.1	94.7%	92.4%	94.75%
1000	0.1	95.1%	93%	95.2%
10	0.01	97.95%	96.95%	97.4%
100	0.01	98.38%	97.6%	97.97%
1000	0.01	98.76%	98.2%	98.5%

Algorithm 4: Support Vector Machine based Model.

```
Input: $X : apps\_features; Y : labels;$          ▷ label is benign or malware
$Building\_params_{list} = (C, kernel, gamma)$
$fitting\_params_{list} = (X_{train}, Y_{train})$ $dpoint : decision\_point$          ▷ 0.5 by default
Output: $predicted\_app$          ▷benign or malware app
1  $X_{train}, X_{test}, Y_{train}, Y_{test}$          ▷ Splitting the dataset
2  $X_{train} = sc.fit_{transform(X_{train})}$          ▷Feature Scaling using StandardScaler (sc)
3  $X_{test} = sc.transform(X_{test})$
4  $Model = build\_SVM\_model\_architecture(Building\_params)$          ▷SVM model Building
5  $Model \leftarrow SVM\_model\_fit(fitting\_params)$          ▷SVM model fitting
6  $y_{pred} = Model.predict(X_{test})$
7  for $app.pred \in y_{pred}$ do
8      if $(app.pred > dpoint)$ then
9          $app \leftarrow Malware$ ;
10     else
11         $app \leftarrow Benign$ ;
12     end
13 end
```

4. Experiments

4.1. Performance Indicators

As it relates to the detection of malwares, we refer to True Positive (TP) as the number of malwares actually classified as such, True Negative (TN) as the number of benign apps classified as such, False Positive (FP) as the number of benign apps wrongly classified as malwares, and finally False Negative (FN) as the number of malwares wrongly classified as benign. More informative measures, widely used in malware detection analysis work, are derived from these simple measures, such as:

- Precision: The ratio of actual malwares in the set of apps classified as such: TP/(TP+FP)
- Recall: The ratio of malwares that were detected as such: TP/(TP+FN)
- Accuracy: The percentage of applications that have been appropriately categorised: (TP+TN)/(TP+TN+FP+FN)
- F1-Measure: A performance indicator that takes into account both the precision and recall of the obtained classification: 2 $\times$ (Recall $\times$ Precision)/(Recall + Precision)
- Area under ROC Curve (AUC): A measure of the predictive power of the classifier that basically informs on how well the model can distinguish between classes (here, benign apps vs. malwares).

For all these measures, the higher, the better, with 1 being the perfect value.

4.2. Experimental Setup

We conducted experiments with both DL and SVM models. All the experiments were carried out using the same dataset. The experiments are done using the Python programming language, and the following are the characteristics of the computer used for the experiments; Windows 10(64 bit), Intel(R) Core(TM)i7-2600 CPU@ 3.40 GHZ, and 16 GB RAM.

4.3. Results

In this section, we present obtained results. The factors below explain why our approach was able to outperform other approaches. These include hyper-parameters tuning as well as the combination of vulnerability features in our dataset.

- Performance of DL model:
  Initially, we used 11,814 apps; in this experiment, an app which has 0 flag labelled as benign, whereas an app which has two or more flags labelled as malware, and an app with one flag are excluded. The training phase performance results were higher than the testing phase, which caused over-fitting in the model. To solve this situation,

we increased the size of the dataset by adding the malware apps from VirusShare and apps with one flag as benign. Experimentally, we observe that the performance improved. From Table 4, we can observe that the size of the dataset has increased from 11,814 to 18,780 to avoid over-fitting and improve the performance.

Table 4. Comparison between the results of datasets with 11,814 samples, and 18,780 samples.

Size of Dataset	Accuracy	F1	AUC_score
11,814 samples	89%	90%	88%
18,780 samples	99.33%	99%	99.15%

Figures 7 and 8 illustrate the history model's accuracy and loss for 11,814 and 18,526 samples, respectively. From the figures, it is clear that when we increase the size of the dataset, the accuracy and loss curve lines in the training phase are very close to the accuracy and loss curve lines in the testing phase. In the previous experiment, the difference between the accuracy and loss curve lines was huge. The performance improved when we increased the size of the dataset and the over-fitting problem was solved. If the output layer's Sigmoid result was greater than or equal to 0.5, the application was categorised as malware. Values below 0.5 were considered benign.

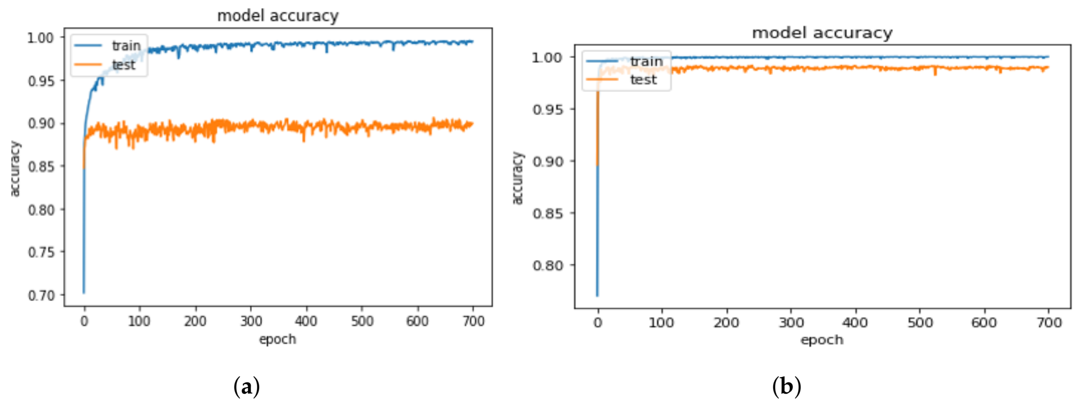


Figure 7. Comparison between history models Accuracy for 11,814 samples and 18,526 samples. (**a**) Accuracy for 11,814 samples. (**b**) Accuracy for 18,526.

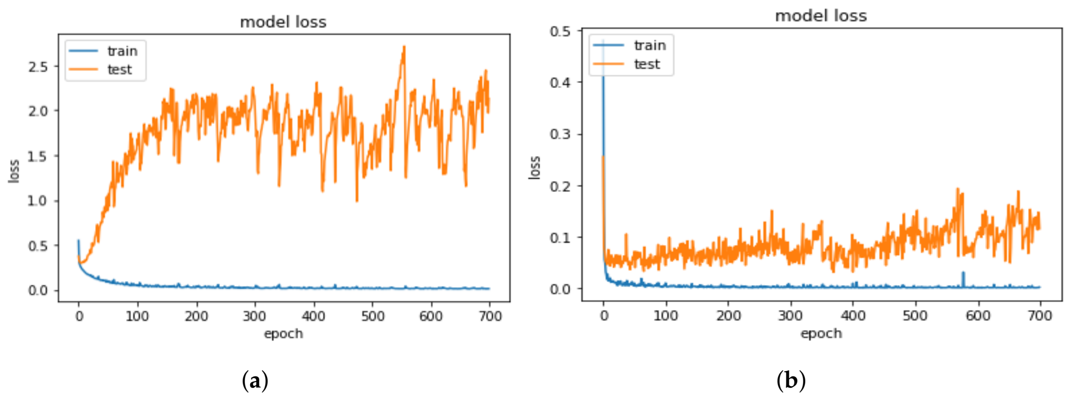


Figure 8. Comparison between history models loss for 11,814 samples and 18,526 samples. (**a**) Loss for 11,814 samples. (**b**) Loss for 18,526.

As Table 5 indicates, the performance improved when we increased the size of the dataset and the over-fitting problem was solved. If the output layer's Sigmoid result was greater than or equal to 0.5, the application was categorised as malware. Values below 0.5 were considered benign. Table 5 thus shows that the confusion matrix correctly classifies the 2406 benign samples as benign and 1275 malware samples as malware. Out of 3706 app samples, 3681 samples were predicted accurately and only 25 samples were wrongly predicted.

Table 5. DL Confusion matrix.

3706		Predicted Class	
		Benign	**Malware**
Sensitivity	B (99.71%)	2406	7
Specificity	M (98.61%)	18	1275

- Performance of the SVM model:
  Table 3 illustrates the experimental results. When we were tuning the hyperplane parameters, we noticed that when gamma is smaller than 0.01 and C is higher than 1000, the results improve, i.e., both parameters increase the values of AUC, F1, and the accuracy. With such parameter values, we can therefore get the correctly separating hyperplane and improve the performance of the model. Table 6 shows that the confusion matrix correctly classifies the 2425 benign samples as benign and 1235 malware samples as malware. Out of 3706 app samples, 3660 were predicted accurately and only 46 were wrongly predicted.
- Challenges and discussion
  Improving the performance of the SVM classifier was challenging and involved some fine tuning with respect to the two parameters: C and gamma. Our results showed that tuning C correctly is a vital step in the use of SVMs for structural risk minimization In RBF kernel, both C and gamma parameters need to be optimized simultaneously. If gamma is large, the effect of C becomes negligible.. When gamma gets smaller, the results improve.
  As for the DL, the configuration of the hyperparameters (the number of layers and nodes in each hidden layer) for our specific predictive modeling problem was done via systematic experimentation. It is worth noting that the time complexity of the DL algorithm is higher than the time complexity of the SVM algorithm.

Table 6. SVM Confusion matrix.

3706		Predicted Class	
		Benign	**Malware**
Sensitivity	B (99.26%)	2425	18
Specificity	M (97.78%)	28	1235

Table 7 shows the results (accuracy, F1 and AUC_score) obtained with our DL and SVM classifiers. It also compares these results to the ones obtained by the best state-of-the-art approach i.e., [24]. The highest accuracy for related work is 95.31%, but our models show better performances in both DL and SVM classifiers. Their accuracies are 99.33% and 98.76% respectively.

Table 7. Comparison between DL, SVM classifiers and the Related work.

The Classifier	Accuracy	F1	AUC_score
Deep Learning	99.33%	99.03%	99.15%
SVM	98.76%	98.2%	98.5%
Best result for State of Art	95.31%	95.31	N/A

4.4. Comparison with Well-Known Anti-Virus Tool

As we mentioned previously, the dataset has two kinds of malwares: the flagged malware apps and the malware apps collected from the VirusShare repository. This repository provides access to live malware and day one malware, motivating us to upload the VirusShare malware apps to the Virus Total tool to scan them. We compared the obtained results using our model to detect VirusShare malware apps, and the obtained results using Virus Total tool to detect VirusShare malware apps (the same samples used in our approach). We observed that our model was able to detect 99.33% of the VirusShare malware apps, while Virus Total tool was able to detect only 75%. Table 8 shows a comparison with a well-known anti-virus tool (Virus Total).

Table 8. Comparison with well known anti-virus tool.

Malware Detection	Accuracy
Our approach	99.33%
Virus Total	75%

5. Related Work

Over the last few years, considerable effort has been devoted to the development of novel methodologies for detecting Android malware anomalies using machine learning techniques (e.g., [25–27]). In the current section, we propose an overview of the different proposals through the lens of the kind of analyses performed to obtain the features used in training the machine learnique: static analysis, dynamic analysis, hybrid analysis.

5.1. Static Analysis

Static analysis is the easiest and least expensive method for obtaining the features that will characterise an application. Permissions are the most commonly used features but some other elements such as intent filters, api calls, etc. have been investigated as well.

Sirisha et al. [2] focused solely on permissions and proposed a a deep neural network model which attained an accuracy of 85%, on a dataset of 398 apps (benign and malware) and 331 features (permissions). Also focused on permissions, Rehman et al. [28] proposed a framework that is both signature- and heuristic-based. They performed experiments using various classifiers such as SVM, Decision Tree, J48 and KNN, and used an existing dataset containing 401 apps and permissions as features. The accuracy of their approach is 85%.

Differently, Kumaran and Li [7] applied different ML algorithms to features extracted from permissions and intent filters found in an app's manifest. They found that permissions performed much better than intent filters but that using both sources yielded a detection accuracy of 91.7% percent (SVM) and 91.4% percent (KNN), which outperforms the classification performance of either feature set individually. More recently, Zhu et al. [29] proposed DroidDet, an Android malware classification approach built on Random Forest. It utilizes various static features derived from permissions and API calls and attained an accuracy of 88.26% on a dataset of 2130 apps. Similarly, Li et al. [8] proposed a Deep Learning algorithm that achieved 90% accuracy on a dataset of 2800 apps (benign and malware) and 237 features (permissions, API calls, and URLs). Also using deep learning, Naway et al. [24] investigated static features (permissions, Intents, API calls, Invalid certificates) on a dataset of 1200 apps and attained an accuracy of 95.31%.

In our previous work [9], we proposed the AndroVul dataset and a preliminary investigation of the dataset as it relates to the detection of malwares. More precisely, we used the well-known machine learning software Weka and selected NaiveBayes (NB) from its bayes category, RBF classifier from its function category, JRip from its rules category, and J48 from its tree category. The selected machine learning approaches were applied under identical settings and with default parameters. The objective of that paper was to demonstrate the potential of the proposed features for the detection of malwares.

In contrast to that work, our key objective in this research work is to propose a finely tuned machine learning appproach able to outperform existing approaches and anti-virus products. The additional work required involved tuning the hyper-parameters of the machine learning approaches, increasing the amount of malware apps, and conducting additional experiments and comparisons with existing literature and antiviruses.

5.2. Dynamic Analysis

Dynamic analysis takes interest into an app's behavior at run-time and may detect malicious activity on an actual execution path. As such, it is resistant to code obfuscation but on the other hand may have minimal code coverage, depending on how extensive and complete are the execution scenarios it considers.

Mas'ud et al. [30] proposed a malware detection system that uses dynamic analysis based on five different sets of features obtained through dynamic analysis. It employs five separate ML classifiers in order to find the optimal combination for efficiently classifying Android malware. The experimental results showed that a multilayer perceptron classifier yielded the highest accuracy 83%. Martinelli et al. [31] developed a method that utilizes a network of neural convolution implemented through dynamic analysis of system calls occurrences. Their work is based on a recent dataset composed of 7100 apps. They created a number of user interface interactions and system events during the duration of the application's execution. The accuracy is 90%.

5.3. Hybrid Analysis

Hybrid analysis techniques (e.g., [15,32]) entails the use of both static and dynamic elements. This dual perspective improve the identification's accuracy but may come with more resource consumption, especially when the analysis is done on a mobile device.

Yuan at al. [33] presented a machine learning-based method for malware detection that makes use of over 200 features collected from both static and dynamic analysis of Android apps. The comparison of modelling results reveals that the deep learning technique is particularly well-suited for Android malware detection, with a high level of 96% accuracy when applied to real-world Android application collections. their dataset contains 250 malware samples from Contagio Mobile and 250 benign apps from Google Play Store.

In a subsequent work, Yuan at al. [34] developed another model based on the DBN: the Droid Detector. The proposed method was validated against a broad unbalanced dataset containing 20,000 benign and malicious samples. The results showed that DBN performed well, with an accuracy of 96.76%. Around the same time, L. Xu et al. [35] proposed an approach for identifying Android malware that relies on autoencoders to analyse the app's features. It then uses an SVM classifier to classify the apps as malicious or trustworthy. They conducted experiments on a dataset of 5888 benign and malware apps, analysing static and dynamic elements separately and found that static features outperformed dynamic features.

Some other security researchers deployed machine learning techniques to propose related approaches. For instance, in [13], authors were mostly concerned with metamorphic malware. The primary objective of this research is to provide a mechanism for classifying malware based on its behaviour. They began their investigation by building a dataset of API calls performed on the Windows operating system that reflects malicious software behaviour. LSTM was utilised to classify the data in this investigation. (Long Short-Term Memory), The classifier's result indicates an accuracy of up to 95% with an F1-score of 0.83. The use of machine learning to handle security vulnerabilities is similar to their approach. However, we have chosen to concentrate on Android platform security issues rather than other platforms.

Table 9 presents the feature, dataset, and classifier used in each related work as well as our approach. In particular, this table allows us to conclude that: (1) our work has been tested on a dataset that includes more sort of features than the ones used by other approaches; and (2) it outperforms existing approaches.

Table 9. Comparison between state of the art research and our approach.

References	Feature Used	Dataset Used	Used Classifier	Accuracy
Paper [2]	permissions	398 samples 331 features	Deep Learning	85%
Paper [8]	permissions, APIs, URLs	2800 samples 237 features	Deep Learning	90%
Paper [24]	permissions, APIs, Invalid certificate	1200 samples	Deep Learning	95.31%
Paper [29]	permissions, APIs	2130 samples	Random Forest	88.26%
Paper [28]	permissions	401 samples	SVM	85%
Paper [36]	permissions, APIs	2130 samples	Random Forest	89.91%.
Our previous work [9]	Permissions, Code smell, AndroBugs vulnerabilities	1600 samples 74 features	Weka (RBF) provides best result	83%
Our approach	Permissions, Code smell, AndroBugs vulnerabilities	18,526 samples 74 features	Deep learning & SVM	99.3% & 98.76% Respectively

6. Conclusions

Android is the most popular smartphone operating system, accounting for 85 percent of the market. However, Android's widespread acceptance and openness make it an ideal target for malicious applications that take advantage of the system's security flaws. Signature-based malware detection present in most antiviruses is vulnerable to new malware, so advanced technologies such as machine learning approaches have been proposed to tackle malware detection. Our current work builds on and extends a previous work in which we collected vulnerability features (e.g., code smells, dangerous permissions, and vulnerabilities identified by the tool AndroBugs) from Android apks and proposed a dataset of almost 12K apps from the AndroZoo repository. A first important contribution was the addition (and reverse engineering of the features) of thousands of malwares from VirusShare, a well-known virus repository. In general, the more data points, the better the prediction models, so it was important and beneficial to our experiments and the research community in general to improve the size of the dataset. The focus of the current paper is on proposing highly efficient machine learning models able to fully leverage the potential of the features we collected. To achieve that goal, we used two different advanced classifiers (Deep Learning and SVM) to learn the malware and benign patterns. We implemented these algorithms and experimented with them to get the best hyper parameters for malware detection using the features we collected. Both of our classifiers achieve an accuracy of around 99% and these results significantly outperform the state-of-art and a collection of antivirus, as proposed on the site VirusTotal.

Short term future work involves the investigation of possible trends in Android malware development (and thus detection); we plan to investigate the data on a multiple year basis to identify whether some features become more relevant in the newest malware. This is especially interesting, considering the relatively rapid pace at which the Android OS changes. Longer term, we plan to apply the lessons learned while experimenting with DL and SVM parameters on an expanded dataset of apps and features. More specifically, we plan to investigate the potential of other features, especially those that can be obtained from an app's manifest file (intent filters, xml data, etc.). Additionnally, we would like to investigate whether the category assigned to an app by a developer (whether a malicious actor or not) should be a factor in the patterns learned by advanced techniques.

Author Contributions: Designing, implementing, investigation, writing the manuscript, Z.N.; reviewing, editing, supervision, S.K.; supervision, C.T.; proposing, reviewing and editing, A.B.; reviewing, A.B.B. All authors have read and agreed to the published version of the manuscript.

Funding: The Libyan-North American Scholarship Program, Canadian Bureau for International Education, provided funding for the work mentioned in this article (CBIE).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: https://github.com/Zakeya/AndroVul (accessed on 14 July 2021).

Acknowledgments: The authors thank the Libyan Ministry of Education, and Canadian Bureau for International Education (CBIE).

Conflicts of Interest: We wish to confirm that there are no known conflict of interest associated with this publication and there has been no significant financial support for this work that could have influenced its outcome.

References

1. Google Play Store. Android Official Store. Available online: https://play.google.com/store/apps (accessed on 27 January 2021).
2. Sirisha, P.; Anuradha, T. Detection of Permission Driven Malware in Android Using Deep Learning Techniques. In Proceedings of the 2019 3rd International conference on Electronics, Communication and Aerospace Technology (ICECA), Coimbatore, India, 12 June 2019; pp. 941–945.
3. Sabhadiya, S.; Barad, J.; Gheewala, J. Android Malware Detection using Deep Learning. In Proceedings of the 2019 3rd International Conference on Trends in Electronics and Informatics (ICOEI), Tirunelveli, India, 23 April 2019; pp. 1254–1260.
4. Haystack. Mobile Issues. Available online: https://safeguarde.com/mobile-apps-stealing-your-information/ (accessed on 14 March 2021).
5. AV-TEST. Security Institute. Available online: https://www.av-test.org/en/statistics/malware/ (accessed on 12 November 2021).
6. Chebyshev, V. Mobile Malware Evolution 2020. Available online: https://securelist.com/mobile-malware-evolution-2020/101029/ (accessed on 1 March 2021).
7. Kumaran, M.; Li, W. Lightweight malware detection based on machine learning algorithms and the android manifest file. In Proceedings of the 2016 IEEE MIT Undergraduate Research Technology Conference (URTC), Cambridge, MA, USA, 4 November 2016; pp. 1–3.
8. Li, W.; Wang, Z.; Cai, J.; Cheng, S. An android malware detection approach using weight-adjusted deep learning. In Proceedings of the 2018 International Conference on Computing, Networking and Communications (ICNC), Maui, HI, USA, 5 Mar 2018; pp. 437–441.
9. Namrud, Z.; Kpodjedo, S.; Talhi, C. AndroVul: A repository for Android security vulnerabilities. In Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering, Markham, ON, Canada, 4 November 2019; pp. 64–71.
10. Tchakounté, F.; Hayata, F. Supervised learning based detection of malware on android. In *Mobile Security and Privacy*; Elsevier: Amsterdam, The Netherlands, 2017; pp. 101–154.
11. Verbraeken, J.; Wolting, M.; Katzy, J.; Kloppenburg, J.; Verbelen, T.; Rellermeyer, J.S. A Survey on Distributed Machine Learning. *ACM Comput. Surv. (CSUR)* **2020**, *53*, 1–33. [CrossRef]
12. Gadient, P.; Nierstrasz, O.; Ghafari, M. *Security in Android Applications*; University of Bern: Bern, Switzerland, 2017.
13. Catak, F.O.; Yazı, A.F.; Elezaj, O.; Ahmed, J. Deep learning based Sequential model for malware analysis using Windows exe API Calls. *PeerJ Comput. Sci.* **2020**, *6*, e285. [CrossRef] [PubMed]
14. Catak, F.O.; Ahmed, J.; Sahinbas, K.; Khand, Z.H. Data augmentation based malware detection using convolutional neural networks. *PeerJ Comput. Sci.* **2021**, *7*, e346. [CrossRef] [PubMed]
15. Naway, A.; Li, Y. A review on the use of deep learning in android malware detection. *arXiv* **2018**, arXiv:1812.10360.
16. Li, Y.; Wang, G.; Nie, L.; Wang, Q.; Tan, W. Distance metric optimization driven convolutional neural network for age invariant face recognition. *Pattern Recognit.* **2018**, *75*, 51–62. [CrossRef]
17. Verma, S.; Sharan, A. Enhancing the performance of SVM based document classifier by selecting good class representative using fuzzy membership criteria. In Proceedings of the 2017 3rd International Conference on Computational Intelligence & Communication Technology (CICT), Ghaziabad, India, 9 February 2017; pp. 1–6.
18. AndroZoo. Android Apps Repository. Available online: https://AndroZoo.uni.lu/ (accessed on 1 March 2018).
19. VirusTotal. Antiviruses Website Scanners. Available online: https://www.virustotal.com/gui/ (accessed on 11 March 2018).
20. VirusShare. Malware Repository. Available online: https://virusshare.com/ (accessed on 25 August 2019).
21. Bhattacharya, A.; Goswami, R.T. DMDAM: Data mining based detection of android malware. In Proceedings of the First International Conference on Intelligent Computing and Communication, Kalyani, West Bengal, India, 2 August 2017; pp. 187–194.
22. Xu, J.; Rahmatizadeh, R.; Bölöni, L.; Turgut, D. A sequence learning model with recurrent neural networks for taxi demand prediction. In Proceedings of the 2017 IEEE 42nd Conference on Local Computer Networks (LCN), Singapore, 9 October 2017; pp. 261–268E.

23. Huang, G.; Liu, Z.; Van Der Maaten, L.; Weinberger, K.Q. Densely connected convolutional networks. In Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition, Honolulu, HI, USA, 21–26 July 2017; pp. 4700–4708.
24. Naway, A.; Li, Y. Using Deep Neural Network for Android Malware Detection. *arXiv* **2019**, arXiv:1904.00736.
25. Baskaran, B.; Ralescu, A. A Study of Android Malware Detection Techniques and Machine Learning. In Proceedings of the 27th Modern Artificial Intelligence and Cognitive Science Conference 2016, Dayton, OH, USA, 22–23 April 2016; pp 15–23.
26. Yerima, S.Y.; Sezer, S.; Muttik, I. Android malware detection using parallel machine learning classifiers. In Proceedings of the 2014 Eighth International Conference on Next Generation Mobile Apps, Services and Technologies, Oxford, UK, 10 September 2014; pp. 37–42.
27. Al Ali, M.; Svetinovic, D.; Aung, Z.; Lukman, S. Malware detection in android mobile platform using machine learning algorithms. In Proceedings of the 2017 International Conference on Infocom Technologies and Unmanned Systems (Trends and Future Directions) (ICTUS), Dubai, United Arab Emirates, 18 December 2017; pp. 763–768.
28. Rehman, Z.U.; Khan, S.N.; Muhammad, K.; Lee, J.W.; Lv, Z.; Baik, S.W.; Shah, P.A.; Awan, K.; Mehmood, I. Machine learning-assisted signature and heuristic-based detection of malwares in Android devices. *Comput. Electr. Eng.* **2018**, *69*, 828–841. [CrossRef]
29. Zhu, H.J.; You, Z.H.; Zhu, Z.X.; Shi, W.L.; Chen, X.; Cheng, L. DroidDet: Effective and robust detection of android malware using static analysis along with rotation forest model. *Neurocomputing* **2018**, *272*, 638–646. [CrossRef]
30. Masud, M.Z.; Sahib, S.; Abdollah, M.F.; Selamat, S.R.; Yusof, R. Analysis of features selection and machine learning classifier in android malware detection. In Proceedings of the 2014 International Conference on Information Science & Applications (ICISA), Seoul, Korea, 6 May 2014; pp. 1–5.
31. Martinelli, F.; Marulli, F.; Mercaldo, F. Evaluating convolutional neural network for effective mobile malware detection. *Procedia Comput. Sci.* **2017**, *112*, 2372–2381. [CrossRef]
32. Muttoo, S.K.; Badhani, S. Android malware detection: state of the art. *Int. J. Inf. Technol.* **2017**, *9*, 111–117. [CrossRef]
33. Yuan, Z.; Lu, Y.; Wang, Z.; Xue, Y. Droid-sec: Deep learning in android malware detection. In Proceedings of the 2014 ACM Conference on SIGCOMM, Chicago, IL, USA, 17 August 2014; pp. 371–372.
34. Yuan, Z.; Lu, Y.; Xue, Y. Droiddetector: Android malware characterization and detection using deep learning. *Tsinghua Sci. Technol.* **2016**, *21*, 114–123. [CrossRef]
35. Xu, L.; Zhang, D.; Jayasena, N.; Cavazos, J. Hadm: Hybrid analysis for detection of malware. In Proceedings of the SAI Intelligent Systems Conference, London, UK, 21–22 September 2016; pp. 702–724.
36. Zhu, H.J.; Jiang, T.H.; Ma, B.; You, Z.H.; Shi, W.L.; Cheng, L. HEMD: A highly efficient random forest-based malware detection framework for Android. *Neural Comput. Appl.* **2018**, *30*, 3353–3361. [CrossRef]