International Journal of Informatics and Applied Mathematics
e-ISSN:2667-6990 Vol. 4, No. 1, 1-14

A Deep Neural Network Model for Android Malware Detection

Fatima Bourebaa[1] and Mohamed Benmohamed[2]

[1] Abd El Hamid Mehri University, constantine, Algeria
fatima.bourebaa@univ-constantine2.dz
[2] Abd El Hamid Mehri University, constantine, Algeria
mohamed.benmohammed@univ-constantine2.dz

Abstract. Parallel to the adoption of mobile technology in our daily lives, there is a growing and increasing proliferation of cyber frauds and malicious content. Mobile malware can exploit the vulnerabilities of the device, modify, disclose or erase confidential data, such as credit card numbers, passwords, medical data, contacts, or even block the device asking for a ransom. In this paper, we leverage the possibilities of deep fully-connected neural networks, using permissions and Application Programming Interfaces APIs as features, to automatically and efficiently detect Android malware. We achieved a score of 88.9% using a feed-forward of 128x128x1, 2-hidden layers configuration.

Keywords: Neural Networks · Android Malware Detection · Smartphone Security.

1 Introduction

The market of smartphones, tablets, personal digital assistants, and similar handheld devices is prospering since the turn of the century. One main reason behind this success is the ability to download and execute rich functionality applications from app stores, such as access to medical information, weather forecasts, bus schedules, and bank accounts. Indeed, the insertion of these devices into our daily lives is increasing, but unfortunately, the number of frauds and mobile malware is also steadily increasing at boring speeds. In 2019, security experts counted more than 4.18 million malicious applications, with an average of 11,500 new malicious android applications [1].

By definition, mobile malware is a program inserted into a mobile device with the intent of compromising the confidentiality, integrity, or availability of the user's data, applications, operating system, or annoying or disrupting the user. Recent malware, distributed via fraudulent advertising, request ad traffic from their C&C servers and then simulate clicks to generate ad revenue fraudulently. No icon or shortcut is displayed, which makes it difficult to find and remove them. They use fake warnings to get the user to activate the accessibility services and then abuse of this activation to automate graphical interface actions in the background. Besides, the malware operator can instruct it to download additional malicious programs as well as to open the phone up to the remote control to allow for more attacks.

In the early stage of the mobile malware life since its first appearance in 2004, the number of malware threats was relatively small, and handcrafted rules were often enough to detect most of them. But recently, the phenomenal growth of malware makes it impractical for the anti-malware industry to rely on the manual specification of detection rules. To address this problem, results from machine learning based-approaches, such as Support Vector Machine [4], [11] Random Forest [7], K-Nearest Neighbors [10]), show that they are robust and scalable malware detection systems. Additionally, Recent published works reveal that deep learning approaches outperform conventional machine learning models and work much better [16], [12].

In this paper, the primary question we are trying to answer is what topology of fully connected neural networks is well suited to address the problem of android malware detection and produce the best score. More precisely, we make the following contributions:

- Apply Natural language processing techniques to extract and vectorize permissions and call to APIs as features.
- Implement and evaluate different deep fully connected neural networks to efficiently detect android malware.

The paper begins by examining the related works in Section 2. A review of the required background is presented in Section 3 including Android Security Model (Section 3.1), Multilayer Neural Network (Section 3.2). Next, we detail the workflow of our approach, in section 4. Quantitative results of all our model's topologies are provided in Section 5 and the paper is concluded in Section 6.

2 Related Work

Machine learning and deep learning based approaches to malware detection may be characterized according to two discriminative attributes: i) the types of features extracted or learned from an Android Package (APK), which is the file format used by the Android system to distribute and install apps. These features include: APIs/System calls, application components and Intent, opcodes, logs, APK resource file and strings, (ii) the machine learning algorithms they use. DroidAPIMiner [15] relies on the semantic information within the bytecode of the applications ranging from critical API calls to package level information. DroidAPIMiner may be considered as a a state-of-the-art system and a reference in term of performance (99% Accuracy), but this result is obtained by training a kNN algorithm on a small dataset (3987 malware). MAMADroid [9] builds a behavioral model, in the form of a Markov chain, from the sequence of abstracted API calls performed by an application, then uses it to extract features and perform classification. MAMADroid achieves a F1-score of 73%-99% on a set of 8.5K of malicious applications and 35.5K of benign applications. This show that there is no guarantee that classification models built on the basis of machine learning will give the same results under different parameters or with different datasets.

DroidMat[13] extracts the requested permissions and Intent messages passing from the manifest file, and regards android components (Activity, Service, Receiver) as entry points drilling down for tracing API Calls related to permissions. Next, it applies K-means algorithm to enhance the malware modeling capability. StormDroid[5] demonstrated its accuracy and efficiency in classifying malicious applications on a set of 7,970 Android app samples, including 3,620 malicious samples. Evaluation of the results showed that StormDroid is able to achieve 94% accuracy. Martinelli [6] tests CNN on 7100 real-world mobile applications, and obtains an accuracy ranging between 0.85 and 0.95. Yakura et al. [14] proposed a method to reduce the overhead in the investigation of samples by extracting the essential byte sequences in malware samples. Along with CNN, they have applied an attention mechanism to an image. Attention mechanism is a technique to dynamically select important features which improves the performance. The approach is based on region distinction which extracts the characteristic byte sequences mainly related to a malware family. The treated information proves to be very useful in case the malware samples were packed. The authors have used 147803 samples belonging to 542 families from VX Heaven as the dataset. The 2D- CNN achieves an accuracy of 50.97%.

3 Background

3.1 Android Security Model

Before we elaborate on our approach, we briefly describe the architecture of Android applications and its security mechanisms. Android applications, mostly written in the Java programming language, are composed of code, resources and

data. The Android system compiles the code with any data and resource files, then puts the results into an archive file with an .apk suffix. By default, the system assigns each application a unique user identifier or ID. This identity remains constant for the duration of the APK's life on that device. The system sets permissions for all the files in an application so that only the user ID assigned to that application can access them. The notion of an application component is central in the android programming paradigm, it corresponds to a building block of an Android application. The communication scheme is by message passing. There are four different types of application components: activities, services, broadcast receivers and content provides. Each component exists as its own entity and plays a specific role. Activities are single screens for user interaction. Services are a components running in the background to process long-time operations. Broadcast receivers deliver events to the application outside. Lastly, content providers are some kinds of lightweight databases that can be stored on local files. There is no one single entry point, each component may be a different point through which the system can enter the application.

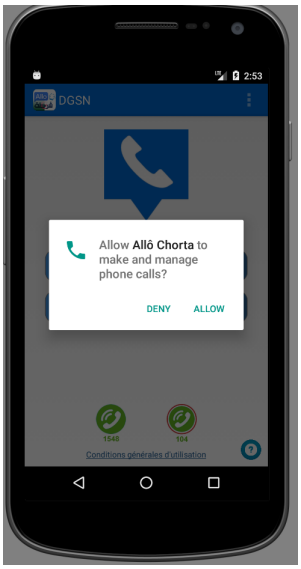

Fig. 1: Example of a dangerous permission (phone calls).

Android is built on a variant of Linux system, and thus inherits its privilege separation security mechanisms. The idea is to ensure that no application can read or write code or data of other applications, the device user, or the operating system itself. Additionally, Android requires that all applications be digitally signed with a certificate before they can be installed. The certificate is used to identify the author of an application, and thus to establish the trust relationship among developers and users. It is often self-signed and does not need to be issued

by a certificate authority. If communicating to another application is dictated by the application's semantics, this later has to explicitly request permission to do so. Android permission is a string expressing the authorization to access sensitive user data such as contacts and SMS, or certain system features such as camera and GPS. An application must declare the permissions it requires by including $< uses-permission >$ tags in the app manifest. For example, an app that needs to send SMS messages would have the following line:

```
<uses-permission android:name="android.permission.SEND_SMS"/>
```

Before Android version 6 (Application Programming Interface API 23), i.e. version 5.1.1 or lower, the user is asked to grant access to all required dangerous permissions at install-time. Recent versions include a dynamic checking mechanism where the user is prompted with a dialog each time any dangerous access is required. Fig. 1 shows how the android system interrupts a legitimate application to ask the user for approving a phone call permission.

Some weaknesses of the android security model are due to the market business model which allows upload of third party apps with little check on their issuers, or even from unknown non-market sources. Malware authors, therefore, often repackage these legitimate applications, easily reverse-engineered compared to native applications, with a malicious payload. Besides, the implementation of the permission mechanism facilitates the delegation of dangerous permissions to the calling process, which could result in privilege escalation.

3.2 Neural networks

We start our presentation of neural networks with the perceptron, which is the basic model of neural network architectures. A perceptron is a mathematical model mimicking a biological neuron, usually represented graphically by a circle having multiple inputs and a single output. The mathematical function is depicted inside the circle. A biological neuron is an electrically excitable cell, which consists of a body cell or a soma, dendrites, and an axon. The soma receives information as electrical and chemical signals through its dendrites, processes them and then, retransmits them through its axon to other neurons. Electrical signals are modulated in various amounts at special connection points, between axons and dendrites, called synapses.

A neuron fires an output signal only when the total strength of the input signals exceeds a certain threshold. This behavior is mimicked as follows: the dendrites correspond to the inputs, the axon corresponds to the output, and the soma to some mathematical function. Signal modulation is mimicked by calculating the weighted sum of the inputs to denote the total strength of the input signals. To determine the output, an activation function is applied to the obtained sum. A Neural network consists of neurons or basic computing units, organized in interconnected layers. The first layer receives the input and the last one produces the desired result. The layers between the input and the output layers are called hidden layers. Each connection holds a weight. Fig. 2 shows one

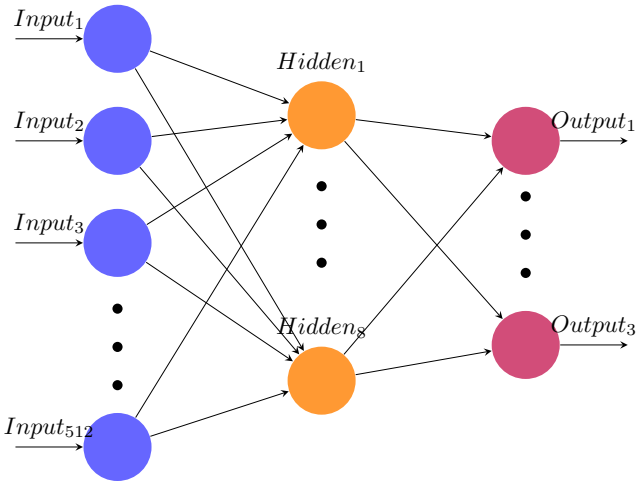

Fig. 2: A Neural Network

hidden layer neural network, with 512 neurons in the input layer, 8 neurons in the hidden layer, and 3 output nodes.

The output of neuron j at layer l is given by equation 1. For simplicity, biases are not considered:

$$a_j^l = \sigma(\sum_k (w_{jk}^l . a_k^{l-1})) \tag{1}$$

where σ is an activation function, and w_{jk}^l is the weight connecting neuron j from layer $l-1$ to neuron k from layer l. The aim of training the neural networks is to calculate the weights that optimize a certain loss function $\mathcal{C}$. This later measures how the model's output is good with respect to a given label. The loss function that is commonly used is the mean squared error (MSE). Nevertheless, the use of a binary cross entropy (BCE) is preferred within binary classifiers. For N samples, the cost function is calculated using the formula of equation 2 :

$$\mathcal{C} = -\frac{1}{N} \sum_{i=1}^{N} (\hat{y}_i \log y_i + (1 - \hat{y}_i) \log(1 - y_i) \tag{2}$$

where $\hat{y}_i$ is the binary class label for sample i, y_i is the predicted output and log is the *log* function. Approaches to machine learning focusing on learning only one or two layers of the data representations are sometimes called shallow learning. Those with more than two layered and hierarchical representations learning are called deep learning, they often involves many hidden successive layers of representations.

4 Method

We conduct our experiences on a set of 10000 malware and 10000 benign applications taken from Koodous [2]. Koodous is an online collaborative platform that provides analysis tools together with social interactions between researchers on Malware Analysis. It provides a similar service to that of Total virus with the difference of being totally dedicated to android malware. Koodous academic dataset is about $0.5To$ size, more recent than the DREBIN [4] one, and provides more samples. The dataset is not specific to a class of malware but includes different categories such as ransomware, trojans, adware, and spyware.

The different steps of our method are articulated in the rest of this section. First, we extract the set of permissions and calls to APIs for each Android application, which constitute our set of features. After that, we proceed to the cleaning and padding of these features. As neural network classifiers accept tensors as input, the padded words are vectorized before training the neural network model. More details are presented bellow.

4.1 Features extraction and vectorization

To properly capture Android Application semantics, we consider, as previously mentioned, both permissions and calls to API methods. As for the API calls, we consider the Android API name from the hole API signature, which also includes the class to which the API call belongs, the set of parameters types, and the returned value type. For example, from the piece of bytecode listed below, we only consider the name *sendMultipartTextMessage* of the API call.

```
invoke-virtual/range v0 ... v5, Landroid/telephony/SmsManager;->
sendMultipartTextMessage(Ljava/lang/String;
Ljava/lang/String;
Ljava/util/ArrayList; Ljava/util/ArrayList;
Ljava/util/ArrayList;)V
```

In order to clean the corpus and eliminate the useless lexical units such as points, dashes, and capital letters. First, we split the sequence of permissions into tokens. Then, we remove the remaining tokens that are not alphabetic. After that, we filter out stop words in the set stop_words = set (”, ”, ”[”, ”]”, ” < ”, ” > ”). Finally, we filter out short tokens. We do the same cleaning to the set of APIs. For each sample, we extract the list of concatenated APIs and Permissions and calculate its length. When looping over all samples, we update the max length and set it to the maximum one encountered. Sequences that are shorter than max length are padded with value 0 until they are max length long.

The result of this padding stage is mapped to a vector space. We represent each Permission or API call using a small and dense vector of 100 dimensions. This choice reduces the vector dimension and highlights the similarities between API calls having close semantics. Let's consider an example of an application that uses three permissions: android.permission.CAMERA, android.permission.READ_SMS, and android.permission.CALL_PHONE, and an

API call *getapplicationinfo*. These permissions and APIs will be mapped as illustrated in equation 3.

$$\begin{pmatrix} \vdots \\ CAMERA \\ READ\_SMS \\ \vdots \\ CALL\_PHONE \\ \vdots \\ getapplicationinfo \\ \vdots \end{pmatrix} = \begin{pmatrix} \vdots \\ -0.00658598 \ldots\ldots -0.0273192 \\ 0.02044574 \ldots\ldots -0.0433061 \\ \vdots \\ 0.00844907 \ldots\ldots 0.04540039 \\ \vdots \\ 0.00933907 \ldots\ldots 0.01240039 \\ \vdots \end{pmatrix} \tag{3}$$

4.2 Building initial models

We choose densely connected layers as a predictive model. This type of neural networks are stacks of dense layers, where the neurons of each layer are connected to every other neuron in the next layer. Training a densely connected network is a highly iterative process. This includes decisions about the number and size of layers, the choice of activation function within layers, the loss function, and the number of epochs. The aim is to find these parameters and hyper parameters that optimize the loss function on both the training and test samples. To achieve this goal, we design a first multilayer network and then empirically evaluate its performance using the set of (10000 Malware, 10000 Goodware) .

We start our experiments by the architecture : (164,)x(12, Relu)x(8, Relu)x (1, sigmoid). The first hidden layer is composed of 12 neurons with the Relu function as activation function, the second layer has 256 neurons and uses the same activation function. The last layer is the output layer and uses the sigmoid function as activation function. The algorithm chosen for optimization is Adam, the reasons behind this choice are its simplicity and efficiency in terms of calculations. The weights are initialized to numbers generated randomly between 0 and 0.05.

4.3 Implementation, Metrics & Tests

In this section, we present the implementation and the evaluation metrics used to mesure the performances of the proposed architectures. We also present the testing method.

Implementation. We implement the extraction of features, the vectorization, and the deep models using Python language. We map each application (malware or goodware) to a list of permissions and APIs that we clean, pad and vectorized as described in section 4.1 and 4.2. Strings and Regular expressions are employed

to extract API calls. Code in listing 1.1 corresponds to the function *clean_corpus* and shows how the extracted features are cleaned.

We list bellow The Python's packages used in the implementation, so that readers can easily reproduce our results.
Numpy and Pandas. Numpy provides a support for large, multi-dimensional arrays and matrices, along with a large library of high-level mathematical functions to operate on these arrays. Pandas is built on top of numpy and offers data structures and operations for manipulating numerical tables and time series. *Matplotlib* is the python package used for visualizing the results. *Keras* is the python API for building neural networks. Although other packages may be used, such as tensorflow, theano or torch, we have chosen keras for its user-friendliness, modularity, and easy extensibility.

Metrics . We evaluate our model using the following performance metrics:

- **Confusion Matrix**: For a binary classification problem, it's a $2X2$ matrix containing the following values: TP, TN, FP, FN. TP represents the number of malwares correctly detected. TN are properly classified benign application. FN: number of malware not detected, considered as benign applications. FP: number of benign application predicted as malware.
- **Precision (or exactness)**: also called the Positive Predictive Value (PPV), it is the number of True Positives divided by the number of True Positives and False Positives.
- **Recall**: also called sensitivity, it is the number of True Positives divided by the number of True Positives and the number of False Negatives.
- **Accuracy**: given by the following formula:

$$Acc = (TP + TN)/(TP + TN + FP + FN).$$

- **F1-score** : this score is given by the formula:

$$F1 - score = 2 * ((precision * recall)/(precision + recall))$$

Testing . Testing is the validation method of machine learning and deep models. In fact, the main challenge for the trained model is to perform well on new, previously unseen data, and not just applications on which it was trained. This ability to perform well on previously unseen samples is called generalization. To test the fully connected architectures regarding generalization, we split the dataset into 80% for training and use 20% for testing. The obtained results are presented in the next section.

```
from pickle import dump
from string import punctuation
# - to clean the dataset -#
def clean_corpus(corpus):
# split into tokens
tokens = corpus.split()
# remove punctuation
```

```
table = str.maketrans('', '', punctuation)
tokens = [w.translate(table) for w in tokens]
# remove non alphabetic tokens
tokens = [w for w in tokens if w.isalpha()]
# filter  stop words
stop_words = set({",", "[", "]", '<' '>'})
tokens = [w for w in tokens if not w in stop_words]
# filter  short tokens
tokens = [w for w in tokens if len(w) > 1]
tokens = ' '.join(tokens)
return tokens
```

Listing 1.1: Python code of the cleaning function

5 Results

The output of a model for a given input is an approximation or a probability that this input is a malware. The evaluation of the (164,)X(12, Relu)X(8, Relu)X(1, sigmoid) architecture according to the generated confusion matrix is 0.92% for the training data and 83% for the test data.

After evaluating this initial architecture, we turn to the hyperparameters tuning. Hyperparameters are those parameters that are arbitrarily set by the user before the training phase. We may tune hyperparameters in different manners. First, we choose, empirically, some model hyperparameters based on our previous experience. Then, train the model, evaluate its accuracy, and restart the process again. We repeat this process until reaching a satisfactory accuracy.

We choose to adjust three hyperparameters: the number of layers and the nodes within each layer, the activation function, and the number of epochs. Further information on these parameters with their corresponding possible values are summarized in table 1.

Table 1: Hyperparameters and their corresponding values .

Abreviations	Definition	Possible values
$\#Layers$	Number of layers	2, 3, 4
A	Activation Function	Relu, Segmoid, Softmax
$\#EPOCHS$	Number of Epochs	Min: 30, Max : 120, step : 20

The result of training several topologies using manual tuning is recapitulated in table 2. The topology that outputs the best score is 164x128x128x1. Fig. 3 represents the associated normalized confusion matrix. Results of architecture NN_8 show that the use of Softmax as the activation function in hidden layers decreases the performance significantly. To compare scores, we plotted Roc curves for architectures NN_1 and NN_3 (see Fig. 4). ROC curve is a performance measurement for classification problems at various thresholds settings.

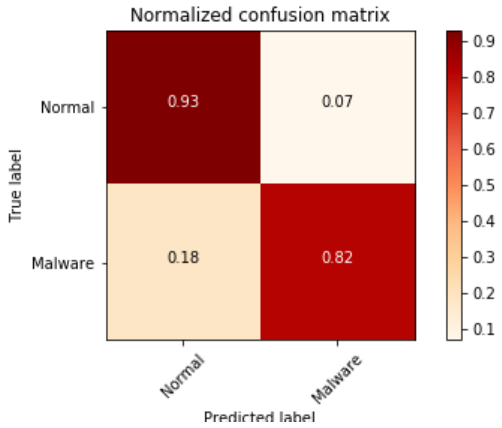

Fig. 3: Confusion matrix for neural network model

it represents the degree or measure of separability, i.e., it shows how much the model is capable of distinguishing between malware and goodware.

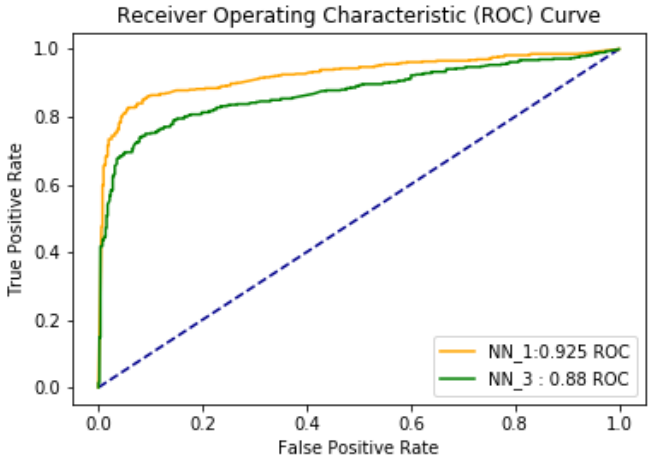

Fig. 4: Roc Curve for Models NN_1 et NN_3

5.1 Comparison with similar work

In this section, we compare our work with recently published papers on android malware detection using koodous as their or part of their dataset. We note that

Table 2: Evaluation metrics for fully connected neural networks architecures

Model topologies	**Evaluation Metrics**			
	Acc	***Prec***	***Rec***	***F1-sc***
NN_1: (,164)x(128,Relu)x(128,Relu)x(1,Sigmoid)	0.86	0.88	0.83	0.85
NN_2(,164)x(12, Relu)x(8,Relu)x(1, Sigmoid)	0.83	0.90	0.74	0.81
NN_3: (,164)x(128,Relu)x(128,Relu)x(1,Sigmoid)	0.88	0.90	0.85	0.87
NN_4: (,164)x(128,Relu)x(64,Relu)x(64,Relu)x(1,Sigmoid)	0.86	0.88	0.84	0.86
NN_5: (,164)x(128,Relu)x(1, Sigmoid)	0.86	0.87	0.83	0.85
NN_6: (,164)x(1024, Relu)x(1, Sigmoid)	0.86	0.87	0.84	0.86
NN_7: (,164)x(128,Sigmoid)x(128,Sigmoid)(1, Sigmoid)	0.85	0.87	0.82	0.85
NN_8: (,164)x(128,Softmax)x(128,Softmax)(1, Sigmoid)	0.81	0.93	0.67	0.78

most researchers focused on MalGenome, or Drebin, collected between 2010-2012, as a reference source of data, and unfortunately, a very small number of them are leveraging recent samples from the Koodous dataset.

In this context, the most similar work to ours is [3], where the authors construct and compare the graphs for malware and normal samples by extracting the permission pairs from the manifest files. It is worth mentioning that they study their method on a set of 7533 malware applications, taken from three different sources (Genome, Drebin, Koudous) with 2944 samples for training, and 3264 samples for testing the accuracy. The authors reported an overall score of 89.28% on Koodous samples. To our knowledge, IPDroid [8] is also based on Koodous, it calculates a set of 20 intents and 17 permissions as features, and applies different machine learning models (SVM, Random Forest & Naive Bayes) on this 37 features. The Random Forest classifier gives the best accuracy of 94.73%. However the authors achieves this score using only 1714 malware apps and 1414 benign apps. We achieved a score of 88% but on more big dataset of 10000 Malware and 10000 goodware. We believe that when the dataset is bigger enough, it would be more representative and consequently the resulting automatic classifier is more effective in detecting malicious content.

6 Conclusion and Future Work

In this paper, we presented a detailed application of feed-forward fully connected deep neural networks to provide automatic Android malware detection. The empirical results show that we can enhance the score by 5% by just dropping or adding one layer. We achieved an accuracy of 88.9% using the configuration (,164)x(128,Relu)x(128,Relu)x(1,Sigmoid) and the Relu function as activation function in hidden layers.

As future work, we plan to investigate the possibilities of genetic algorithm to find optimized values for hyperparameters and extensively evaluate fine-tuned obtained models on different datasets. It would also be interesting to design the

whole system using the multi-agent paradigm. Additionally, we would like to investigate the model interpretability and explain why a given one is achieving good results by looking at its internal layers.

Acknowledgment

The authors would like to thank the Koodous administrators for their effort in collecting and sharing the academic malware dataset.

References

1. G data mobile malware report 2019: New high for malicious android apps. https://www.gdatasoftware.com/news/2019/07/35228-mobile-malware-report-no-let-up-with-android-malware. (last accessed August 2020).
2. Koodous dataset. Available at https://koodous.com (last accessed July 2020).
3. Arora, A., Peddoju, S.K., Conti, M.: Permpair: Android malware detection using permission pairs. IEEE Transactions on Information Forensics and Security **15**, 1968–1982 (2020)
4. Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K.: Drebin: Effective and explainable detection of android malware in your pocket. In: NDSS. The Internet Society (2014)
5. Chen, S., Xue, M., Tang, Z., Xu, L., Zhu, H.: Stormdroid: A streaminglized machine learning-based system for detecting android malware. In: Proceedings of the 11th ACM on Asia Conference on Computer and Communications Security. pp. 377 – 388. ASIA CCS '16, Association for Computing Machinery, New York, NY, USA (2016), https://doi.org/10.1145/2897845.2897860
6. Fabio, M., Fiammetta, M., Francesco, M.: Evaluating convolutional neural network for effective mobile malware detection. In: KES. vol. 112, pp. 2372 – 2381 (2017)
7. Joshi, S., Upadhyay, H., Lagos, L., Akkipeddi, N.S., Guerra, V.: Machine learning approach for malware detection using random forest classifier on process list data structure. In: Proceedings of the 2nd International Conference on Information System and Data Mining. pp. 98–102. ICISDM '18, Association for Computing Machinery, New York, NY, USA (2018)
8. Khariwal, K., Singh, J., Arora, A.: Ipdroid: Android malware detection using intents and permissions. In: 2020 Fourth World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4). pp. 197–202 (2020)
9. Mariconti, E., Onwuzurike, L., Andriotis, P., Cristofaro, E.D., Ross, G., Stringhini, G.: Mamadroid: Detecting android malware by building markov chains of behavioral models (2017)
10. Papernot, N., McDaniel, P.D.: Deep k-nearest neighbors: Towards confident, interpretable and robust deep learning. CoRR **abs/1803.04765** (2018), http://arxiv.org/abs/1803.04765
11. Sun, J., Yan, K., Liu, X., Yang, C., Fu, Y.: Malware detection on android smartphones using keywords vector and svm. In: 2017 IEEE/ACIS 16th International Conference on Computer and Information Science (ICIS). pp. 833–838 (2017)
12. Vasan, D., Alazab, M., Wassan, S., Naeem, H., Safaei, B., Zheng, Q.: Imcfn: Image-based malware classification using fine-tuned convolutional neural network architecture. Comput. Netw. **171**(C) (Apr 2020)

13. Wu, D.J., Mao, C.H., Wei, T.E., Lee, H.M., Wu, K.P.: Droidmat: Android malware detection through manifest and api calls tracing. In: Proceedings of the 2012 Seventh Asia Joint Conference on Information Security. pp. 62–69. ASIAJCIS '12, IEEE Computer Society, USA (2012)
14. Yakura, H., Shinozaki, S., Nishimura, R., Oyama, Y., Sakuma, J.: Malware analysis of imaged binary samples by convolutional neural network with attention mechanism. In: Proceedings of the Eighth ACM Conference on Data and Application Security and Privacy. pp. 127–134. CODASPY '18, ACM, New York, NY, USA (2018), https://doi.org/10.1145/3176258.3176335
15. Yousra, A., Wenliang, D., Yin, H.: Droidapiminer: Mining api-level features for robust malware detection in android. In: SecureComm. vol. 127, pp. 86–103 (2013)
16. 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. SIGCOMM '14, ACM, New York, NY, USA (2014), https://doi.org/10.1145/2619239.2631434