Deep Neural Networks for Android Malware Detection

Abhilash Hota and Paul Irolla

Laboratoire de Cryptologie et de Virologie Operationnelles, École Supérieure d'Informatique, Électronique, Automatique, Laval, France

Keywords: Machine Learning, Deep learning, Android, Malware.

Abstract: In this paper we present a study of the application of deep neural networks to the problem of pattern matching in Android malware detection. Over the last few years malware have been proliferating and malware authors keep developing new techniques to bypass existing detection methods. Machine learning techniques in general and deep neural networks in particular have been very successful in recent years in a variety of classification tasks. We study various deep neural networks as potential solutions for pattern matching in malware detection systems. The effectiveness of the different architectures is compared and judged as a potential replacement for traditional approaches to malware detection in Android systems.

1 INTRODUCTION

Smartphone use has been rising steadily over the past few years. Surveys (Deloitte, 2017) show anywhere from 80 to 82% smartphone penetration in the tested markets. Of the surveyed users, about a third conduct regular financial transactions on their smartphones. The share of consumers using mobile payments increases to about 90% in markets like India and China. A significant interest in FinTech has led to an even greater increase in adoption of mobile platforms, not just for payments but also as a substitute for regular banking. Globally, in terms of mobile operating systems, 99% of the market share is owned by Android and iOS. Since 2015, Android has consistently managed to hold on to the majority of this market share ranging from 70 to 85% in different markets.

This popularity has, however, also attracted the attention of malware authors. The number of android malware samples detected in the marketplaces has risen from about 500,000 in 2013 to more than 3.5 million in 2017 (SophosLabs, 2017). Different types of malware are currently active in the Android ecosystem including banking bots, ransomwares, adwares, spywares, etc. The problem is compounded by the existence of multiple unofficial marketplaces that are largely unregulated and improperly monitored. As such, there is a distinct need to provide a reliable system to protect these endpoint devices from these malware.

In this paper, we study the application of various deep neural network networks to the problem of android malware detection. The paper is organized as follows. We start off by discussing existing approaches to android malware detection. Section 3 discusses the creation of the datasets. Section 4 discusses the preprocessing of the android application samples and the neural network models we trained for malware detection. Section 5 presents the results followed by the conclusion in section 6.

2 RELATED WORK

There exists a significant body of work on the problem of detecting malware on mobile devices. Generally speaking, these approaches can be broadly categorized as based on static analysis or dynamic analysis. In this paper we are primarily concerned with static analysis.

2.1 Static Analysis based Approaches

Static program analysis refers to the analysis of software without actually executing the program. Static analysis has the advantages of low computation cost and low RAM consumption . However, methods like code obfuscation of even byte padding can often be effective in bypassing these approaches.

These approaches are based primarily on extracting features by inspecting application manifests and code disassembly. The most commonly used features are permissions and API calls.

Hota, A. and Irolla, P.
Deep Neural Networks for Android Malware Detection.
DOI: 10.5220/0007617606570663
In *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP 2019)*, pages 657-663
ISBN: 978-989-758-359-9

(Jiang and Zhou, 2012) discusses the characterization of various android malware including their installation methods, activation mechanisms as well as the nature of carried malicious payloads.

(Feng et al., 2014) discuss a new semantics-based approach for identifying a prevalent class of Android malware that steals private user information. They incorporate a high-level language for specifying signatures that describe semantic characteristics of malware families and static analysis for deciding if a given application matches a malware signature.

(Wu et al., 2012) discuss a static feature-based mechanism for detecting Android malware. They consider static information including permissions, deployment of components, intent messages passing and API calls for characterizing the Android applications behavior. Clustering algorithms are deployed to characterize malware intent.

(Schmidt et al., 2009) perform static analysis on the android applications to extract function calls. Function call lists are compared with malware executables for classifying them with Prism and Nearest Neighbor Algorithms. Further they present a collaborative malware detection approach to extend these results.

(Arp et al., 2014) propose a method for detection of Android malware directly on endpoint devices. Their system performs a broad static analysis gathering as many features of an application as possible. The features are embedded in a joint vector space from which characterizing patterns maybe be identified.

The works described in this section all use satic analysis methods to build up a dataset by extracting features like API calls, installation methods, etc, and using that dataset to classify or cluster applications.

3 THE DATASETS

The work described in this paper was carried out using two datasets.

3.1 Dataset 1

This was the first dataset that our models were trained on. It consists of the results of static analysis performed on 3500 malware samples from the Drebin Dataset (Arp et al., 2014) and 2700 benign samples from *Fdroid*(Gultnieks, 2010). We took only unique samples from Drebin Dataset, in order to remove statistical bias in the experiment (Irolla and Dey, 2018). The neural networks were trained on opcode sequence of application source code, which is the sequence of Java bytecode operators. It was clear early on that deeper architectures were overtraining on such a limited dataset very quickly. This led us to create the second dataset.

3.2 Dataset 2

(Allix et al., 2016) present Androzoo, a growing collection of Android applications collected from several sources, including the official GooglePlay app market. It currently contains 7,412,461 different APKs, each of which has been (or will soon be) analyzed by tens of different AntiVirus products to know which applications are detected as Malware. The APKs downloaded from Androzoo were verified against VirusTotal (Total, 2012) to build a dataset of 150000 malware samples and 150000 benign samples. Keeping computational constraints in mind, we restricted ourselves to application samples of size less than 4MB.

4 DATA PREPARATION AND NEURAL NETWORKS

This section describes the data preparation and the neural network models developed.

4.1 Data Preparation

Preprocessing steps involved in machine learning applied to the security domain are very similar to those involved in machine learning applied to the *Natural Language Processing* (NLP) domain. In fact, the source code of a program is a language — even if it is not natural. It has words that assemble themselves into syntax to form a meaning. In the NLP domain, it is common to pretrain Deep Learning with a dedicated neural network that learns a numerical representation of words.

Bytes obtained from the dataset were used as words to generate document vectors for the samples using Doc2Vec (Le and Mikolov, 2014).

4.1.1 Doc2Vec

Doc2vec (Le and Mikolov, 2014) is an unsupervised algorithm to generate vectors for sentences, paragraphs or documents. Many neural network architectures require inputs to be fixed-length vectors. The bag-of-words technique is one approach to this end, but has some drawbacks. It loses the ordering of the words and ignores semantics. Paragraph Vector, or doc2vec, was proposed as an unsupervised approach to learning fixed-length feature representations from variable-length texts, such as sentences, paragraphs or documents.

Every paragraph and every word are mapped to unique vectors which are averaged or concatenated to predict the next word in a given context. The paragraph label acts as a context memory. The paragraph vector stays consistent across all contexts from the same paragraph but not across paragraphs. The word vectors, however, are shared across paragraphs.

This approach was designed to resolve certain weaknesses in traditional bag-of-words models. It retains the semantics of the text and does not lose information about word order. A bag-of-words n-gram model would create a very-high-dimensional feature space with poor generalization.

The algorithm itself has two key stages:

1. Training: Compute word vectors W , softmax weights U, b and paragraph vectors D on already seen paragraphs
2. Inference: Compute paragraph vectors D for new paragraphs by adding more columns in D and gradient descending on D while W, U, b are fixed.

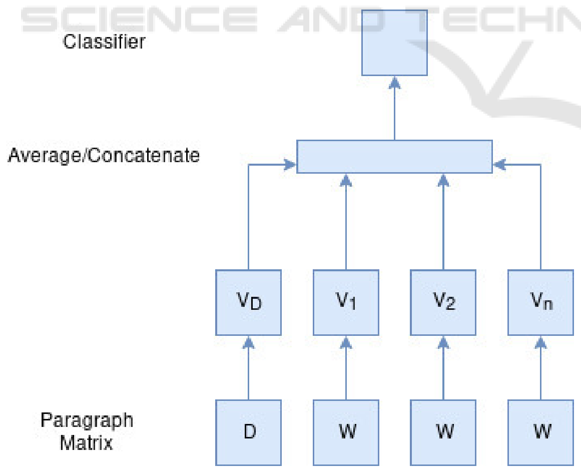

Figure 1: Doc2Vec.

4.1.2 Dataset 1

Dataset 1 consists of the results of static analysis performed on android applications. The bytes from these files were treated as words and used to generate document vectors of length 300 for all the samples. The document vectors served as inputs for the convolutional neural networks. The bytes were the inputs for the long short-term memory networks.

4.1.3 Dataset 2

Dataset 2 consists of multiple android applications, both malware and benign. Ground truth for training was obtained by checking the samples against VirusTotal (Total, 2012). Android applications are packaged as APK files. Figure 2 shows the general structure of an APK file.

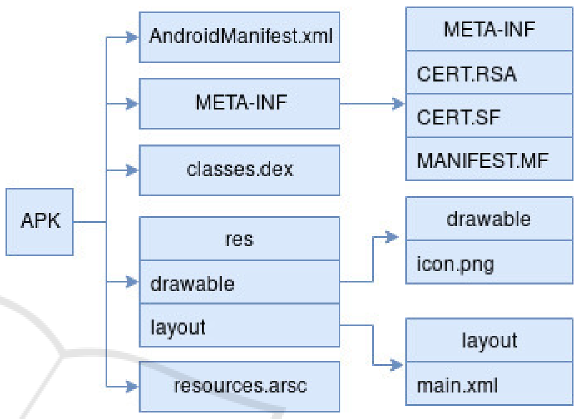

Figure 2: APK File Structure.

For the purposes of the work shown in this paper, we focus on the classes.dex file. Java source code is compiled by the Java compiler into .class files. These files are then processed by the dexer tool in the Android SDK into the .dex (Dalvik Executable) file format. This eliminates all the redundant information that is present in the classes. In DEX all the classes of the application are packed into one file.

We use APKTool (Tumbleson and Wisniewski, 2016) to extract the dex files from the APKs. The bytes from the extracted dex files was then used to generate document vectors of length 300 representing each sample. The document vectors served as inputs for the convolutional neural networks. The bytes were the inputs for the long short-term memory networks.

4.2 Deep Neural Networks

Artificial neural networks (ANNs) are machine learning systems, vaguely inspired by biological neural networks. They can broadly be seen as consisting of four parts:

1. Processing units called neurons, each of which has a certain activation level at any point in time.
2. Weighted interconnections between the units which determine the input to subsequent units given the activation of previous units.

3. An activation rule which a new output signal or activation from the inputs to the units.
4. A learning rule to adjust the weights for a given input/output pair.

ANNs have been around in various forms since the 1940s (McCulloch and Pitts, 1943). Recently, however, the significantly decreasing cost of computational power has led to a massive revival of interest in neural networks in general and deep neural networks in particular.

Deep neural networks, which have shown remarkable successes in recent years in a multitude of fields, are essentially neural networks, distinguished primarily by their depth. The data passes through multiple layers of neurons. Each layer trains on a distinct set of features based on the output of the previous layer. As we go deeper into the network, the layers are able to recombine features from previous layers, leading to increased complexity of features learned.

The models presented in this paper are based on convolutional neural networks and long short-term memory networks.

4.2.1 Convolutional Neural Networks

Convolutional neural networks (CNNs) are a type of neural network that have shown excellent results in various classification tasks and image recognition(Schmidhuber, 2015).

CNNs are based on the assumption of spatially-local correlations. This is enforced by maintaining a local connectivity between neurons of adjacent layers. Stacking many such layers leads to filters that can learn more global features. In CNNs there is also the idea of shared weights. Using the same weights across multiple hidden units allows for features to be detected regardless of position. It is also an effective way of increasing learning efficiency by reducing the number of free parameters to be learned.

CNNs can be described in terms of 4 primary functions:

1. **Convolution.** This step involves the convolution of the input vectors with a linear filter and adding a bias term.

   Let h^k_{ij} be the k^{th} feature map at a hidden layer. Given weights W^k and bias b_k C can be computed as follows
   $$h^k_{ij} = (W^k * x)_{ij} + b_k \qquad (1)$$
   The size of the feature map is affected by three parameters:
   (a) Depth: This refers to the number of filters used for the convolution.
   (b) Stride: This refers to the number of steps the filter slides over the input matrix. Larger strides produce smaller feature maps.
   (c) Zero-padding: Zero-padding the input matrix, or wide convolution (Kalchbrenner et al., 2014) can allow us to apply the filters to border values in the input matrix and control the size of the feature map.
2. **Non-linearity.** The purpose of the non-linearity is to account for the non-linearity in most real-world data. Different types of non-linear functions are used for various models including *tanh*, *sigmoid*, *ReLU*, etc. *Rectified Linear Units* (*ReLU*, $h(x) = max(0,x)$) is an element-wise operation that replaces all negative values in the feature map by zeros. In Deep Learning, ReLU has been shown to outperform sigmoid or tanh (which is essentially a symmetric sigmoid) in most cases (Krizhevsky et al., 2012). Half of the neurons are activated, and sparse activation is preferred on deep architectures for feature selection (to recall, it is the network that learns to detect the best features in Deep Learning) and for processing time. Moreover, when stacking neuron layers it creates a *vanishing gradient* effect with sigmoid-like transfer function. Gradient values are multiplied together and as the derivative of sigmoid-like function tends to zero when their input tends to their extremum values, the gradient magnitude diminishes from a layer to one other. ReLU, having a constant or null derivative, solves the vanishing gradient problem.

   Applying non-linear function f_N to C_{ij} from Eq.1 we get the rectified feature map.
   $$F = f_N(h^k_{ij}) = f_N((W^k * x)_{ij} + b_k) \qquad (2)$$
3. **Pooling or Sub-sampling.** Max-pooling splits the input matrix into a set of non-overlapping spaces and, for each such sub-region, outputs the maximum value. It provides the following advantages:
   (a) It reduces the computational load by eliminating non-maximal values.
   (b) It gives us a sort of translational invariance(Springenberg et al., 2014). In other words, by providing robustness to feature position it allows us to reduce the dimensionality of intermediate representations.
4. **Classification.** Here we employ a fully connected layer, which is a multi layer perceptron with a softmax activation. The convolutional and pooling layers give us high-level features of the input matrix. The purpose of the fully connected

layer is to classify the input based on these features. The layer is also a cheap way to learn non-linear combinations of these features. The output of the softmax activation is equivalent to a categorical probability distribution and makes sure that the sum of output probabilities of the fully connected layer is always 1.

We created three models using CNNs.

Model 1. This network, shown in fig.3, consists of three 2D convolutional layers, each followed by ReLU units and maxpooling. An initial dropout layer is added to avoid overfitting. The final output is obtained through a fully connected layer and softmax activation. The document vectors of length 300 were reshaped into matrices of dimensions (30,10) and used as the input. Essentially the network would be treating it as an input image. Thus the use of 2D convolutions.

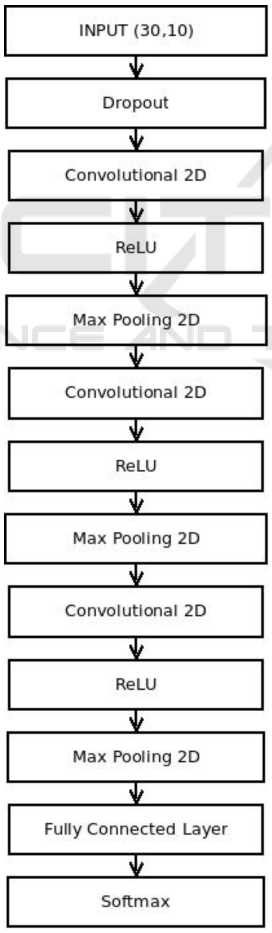

Figure 3: Model 1.

Model 2. This network has the same architecture as Model 1, shown in fig.3. However the difference is that larger document vectors of length 10000 were generated and the input was a matrix of dimension (100,100). The intention was to see if using larger document vectors would allow the network to learn better representations and thus be able to classify the samples better.

Model 3. This model is based on the MalConv architecture (Raff et al., 2017). This architecture was developed for malware detection by ingesting PE files. Here, it has been modified to accept raw bytes from the dex files extracted from the APKs as the input sequences. The architecture, shown in fig. 4 is based on CNNs. Here the problem is treated similar to a natural language processing problem. The input bytes are tokenized and an embedding layer learns a representation for the tokens. Since the input here is in the form of row vectors instead of 2D images, we deploy a 1D convolutional layer. The gating layer determines the fraction of information hat is sent to subsequent layers.

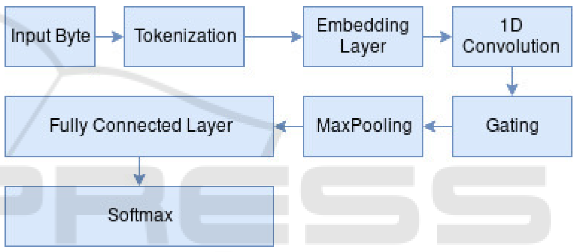

Figure 4: Model 3 - MalConv.

4.2.2 Long Short-Term Memory Networks

Long-short term memory (LSTM) (Gers et al., 1999) is a variant of recurrent neural networks (RNNs). RNNs use recurrent connections within the hidden layer to create an internal state representing the previous input values. This allows RNNs to capture temporal context. However, as the time interval expands, the updated gradient from backpropagation can decay or explode exponentially, referred to as the vanishing and exploding gradient problems respectively. This makes it difficult for RNNs to learn long-term dependencies. LSTM uses constant error carousel (CEC) to propagate a constant error signal through time, using a gate structure to prevent backpropagated errors from vanishing or exploding. The gate structure controls information flow and memory by tuning the value of CEC according to current input and previous context. A gate is essentially a pointwise multiplication operation and a nonlinear transformation that allows errors to flow backwards through a longer time range.

Model 4. This model, shown in fig.5, is based on LSTM cells.

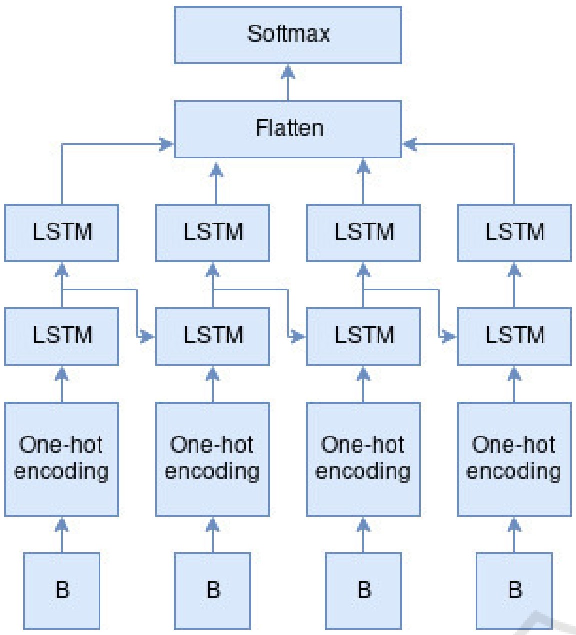

Figure 5: Model 4 - LSTM Network.

Bytes from the extracted dex files are fed directly into the network. An initial one-hot encoding of these bytes is used as a representation and fed into the LSTM network. The output of the LSTM layers is flattened and the final output is obtained after a softmax layer.

5 TRAINING AND RESULTS

Model 1: was trained using NADAM (Dozat, 2016). This is a variation of ADAM (Kingma and Ba, 2014) that employs Nesterov momentum instead of classical momentum. Hyperparameters was optimized over 50 trials. A 9:1 split was maintained for training and validation.
Model 2: was trained using ADAM (Kingma and Ba, 2014). Hyperparameter optimization was performed by experi- menting over 50 trials.
Model 3: was trained using ADAM (Kingma and Ba, 2014). Hyperparameter optimization was performed by experi- menting over 60 trials.
Model 4: was trained using ADAM (Kingma and Ba, 2014) over 1000 steps, with validation checks every 20 steps.

Training time over Dataset 1 was shorter. However, due to less data the models would easily overfit. Accuracy over the training set would quickly reach about 98-99% but accuracy over the test set was limited to about 60-80%.

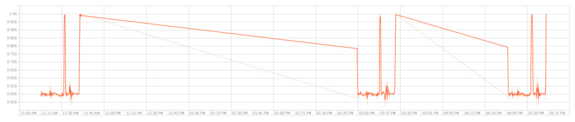
Figure 6: Training Accuracy.

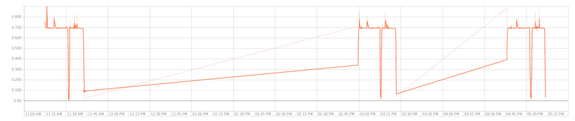
Figure 7: Training loss.

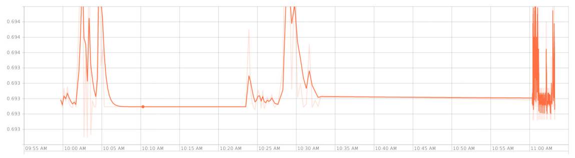
Figure 8: Test Accuracy.

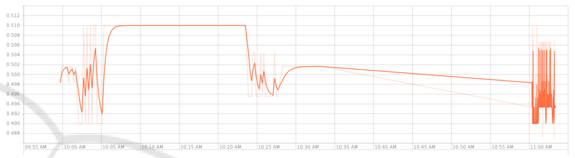
Figure 9: Test loss.

Dataset 2 gives much better results considering the increased amount of data available (Table 1). Accuracy over the training set approaches 1 and over the test set approaches 95.3%. As expected training time is significantly increased to about 160-170 minutes.

Table 1: Malware Detection Accuracy.

	Dataset 1	Dataset 2
Model 1	67%	94.4%
Model 2	70%	95.1%
Model 3	89%	93.7%
Model 4	70%	95.3%

6 CONCLUSIONS

Deep neural networks have shown excellent results in recent years and they show promise in the field of malware detection. In their current state of development, however, they present certain challenges. They require immense amounts of data and processing power to learn the features that make them successful. When it comes to endpoint device security in Android systems, most smartphones running the OS have resource limitations. As such, building a real-time monitoring and malware detection system would currently not be feasible using deep neural networks on the endpoint devices. Moreover treating byte code from the applications as input to the neural networks makes the networks very sensitive to slight modifications of the binary. It requires a fixed length input, so any shift of the binary bytes (like adding bloating code) breaks the detection. Deep neural networks that learn to detect malware based on static analysis would be subject to the same limitations as traditional signature-based approaches. Malware authors have shown a considerable talent for avoiding signature detection systems. Adversarial neural networks have been very successful in fooling neural network based recognition systems (Nguyen et al., 2015).

As such,it would be preferable to have deep neural networks running on the cloud and analyzing application behavior for applications available on the marketplaces, instead of running on endpoint devices for real-time detection. Potentially, unsupervised approaches to deep learning could be used to generate representations. These signatures could then be used by pattern-mathcing based detection systems on endpoint devices. There is a lot more scope for work in this field. The study presented here considers only static analysis or raw byte analysis. However the results do show promise and potential for application to dynamic analysis methods.

REFERENCES

Allix, K., Bissyandé, T. F., Klein, J., and Le Traon, Y. (2016). Androzoo: Collecting millions of android apps for the research community. In *Mining Software Repositories (MSR), 2016 IEEE/ACM 13th Working Conference on*, pages 468–471. IEEE.

Arp, D., Spreitzenbarth, M., Hubner, M., Gascon, H., Rieck, K., and Siemens, C. (2014). Drebin: Effective and explainable detection of android malware in your pocket. In *Ndss*, volume 14, pages 23–26.

Deloitte (2017). Global mobile consumer survey. Technical report.

Dozat, T. (2016). Incorporating nesterov momentum into adam.

Feng, Y., Anand, S., Dillig, I., and Aiken, A. (2014). Apposcopy: Semantics-based detection of android malware through static analysis. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 576–587. ACM.

Gers, F. A., Schmidhuber, J., and Cummins, F. (1999). Learning to forget: Continual prediction with lstm.

Gultnieks, C. (2010). F-droid. https://f-droid.org/en/. Accessed: 2018-10-20.

Irolla, P. and Dey, A. (2018). The duplication issue within the drebin dataset. *Journal of Computer Virology and Hacking Techniques*, pages 1–5.

Jiang, X. and Zhou, Y. (2012). Dissecting android malware: Characterization and evolution. In *2012 IEEE Symposium on Security and Privacy*, pages 95–109. IEEE.

Kalchbrenner, N., Grefenstette, E., and Blunsom, P. (2014). A convolutional neural network for modelling sentences. *arXiv preprint arXiv:1404.2188*.

Kingma, D. P. and Ba, J. (2014). Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*.

Krizhevsky, A., Sutskever, I., and Hinton, G. E. (2012). Imagenet classification with deep convolutional neural networks. In *Advances in neural information processing systems*, pages 1097–1105.

Le, Q. and Mikolov, T. (2014). Distributed representations of sentences and documents. In *International Conference on Machine Learning*, pages 1188–1196.

McCulloch, W. S. and Pitts, W. (1943). A logical calculus of the ideas immanent in nervous activity. *The bulletin of mathematical biophysics*, 5(4):115–133.

Nguyen, A., Yosinski, J., and Clune, J. (2015). Deep neural networks are easily fooled: High confidence predictions for unrecognizable images. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 427–436.

Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., and Nicholas, C. (2017). Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*.

Schmidhuber, J. (2015). Deep learning in neural networks: An overview. *Neural networks*, 61:85–117.

Schmidt, A.-D., Bye, R., Schmidt, H.-G., Clausen, J., Kiraz, O., Yuksel, K. A., Camtepe, S. A., and Albayrak, S. (2009). Static analysis of executables for collaborative malware detection on android. In *Communications, 2009. ICC'09. IEEE International Conference on*, pages 1–5. IEEE.

SophosLabs (2017). Sophoslabs 2018 malware forecast. Technical report.

Springenberg, J. T., Dosovitskiy, A., Brox, T., and Riedmiller, M. (2014). Striving for simplicity: The all convolutional net. *arXiv preprint arXiv:1412.6806*.

Total, V. (2012). Virustotal-free online virus, malware and url scanner. *Online: https://www. virustotal. com/en*.

Tumbleson, C. and Wisniewski, R. (2016). Apktool.

Wu, D.-J., Mao, C.-H., Wei, T.-E., Lee, H.-M., and Wu, K.-P. (2012). Droidmat: Android malware detection through manifest and api calls tracing. In *Information Security (Asia JCIS), 2012 Seventh Asia Joint Conference on*, pages 62–69. IEEE.