Static Analysis on Disassembled Files: A Deep Learning Approach to Malware Classification

Dhiego Ramos Pinto[1], **Julio Cesar Duarte**[1]

[1]Seção de Ensino de Engenharia de Computação – Instituto Militar de Engenharia
Praça Gen. Tibúrcio, 80 - Praia Vermelha, Rio de Janeiro - RJ, 22291-270

{dhiegorp,duarte}@ime.eb.br

Abstract. *The cybernetic environment is hostile. An infinitude of gadgets with access to fast networks and the massive use of social networks considerably raised the number of vectors of malware propagation. Deep Learning models achieved great results in many different areas, including security-related tasks, such as static and dynamic malware analysis. This paper details a deep learning approach to the problem of malware classification using only the disassembled artifact's code as input. We show competitive performance when comparing to other solutions that use a higher degree of knowledge.*

1. Introduction

The ascending market acceptance of cloud computing, the widespread use of social networks, smartphones and other devices connected to the Internet, considerably increased the vectors of malware propagation throughout the years [Jang-Jaccard and Nepal 2014]. With the expansion of the number of users and devices, it is possible to observe an increase on the number of attacks [Verison 2016, Symantec 2016, McAfee 2015, McAfee 2014]. Generally speaking, malware can be categorized as a class of threats, like viruses, trojans, ransomware, spyware, worms, and bots, among others, whose objective, in a nutshell, is the subversion of a system, or part of a system, in which it was inserted in a non-authorized manner [McGraw and Morrisett 2000]. Those threats employ all sorts of techniques to self-propagate and attack, exploiting vulnerabilities on hardware, operational systems, networks, and applications, thus becoming the primary weapon for on-line illicit activities [Jang-Jaccard and Nepal 2014, Damshenas et al. 2013].

Malware analysis is the process by which malicious code is detected and classified, and its potential impact discovered. Detection of malicious code is a challenging task that usually is long and meticulous, requiring a specialist that uses all sorts of tools and techniques to analyze a program. It's typical to malware developers to augment their codes with evasion procedures that can emulate innocuous program's behaviors or creates difficulties to the investigation process, like code obfuscation, polymorphic code, encryption and more [Mangialardo and Duarte 2015, Marpaung et al. 2012]. Besides that, the growing volume of programs to be investigated surpasses the human limits, creating a delay between the analysis procedure and definitive treatment. Hence, the demand for autonomous and intelligent tools becomes evident.

Machine learning algorithms have been widely used to solve malware's classification and detection problems with a noted rate of success. Traditional machine learning methods are heavily dependent on feature engineering, currently requiring a lot of experimentation and a strong knowledge about the applied domain. However, Deep Learning models are known to learn hierarchical representations directly from data, achieving success superior to traditional methods in different areas of application [LeCun et al. 2015]. This work presents a methodology to employ Deep Learning Autoencoders to learn representative features directly from operational codes (opcodes) extracted from disassembled Windows Portable Executable (PE) files from Microsoft Malware Classification Challenge dataset [Microsoft 2015]. Our best deep neural network model achieved an overall accuracy of 99,55%, a competitive result in relation to the Challenge's winner performance.

2. Malware Analysis

Malware Analysis is an investigation process that seeks to detect and classify malicious codes and identify its goals. It can be characterized as Static or Dynamic depending on the objectives and the set of techniques used to perform the task. The Static Analysis does investigations on source code or file structure level. To accomplish this, an analyst can use anti-viruses tools, file identification hashes, string information, analysis of API calls or file headers and, ultimately, the use of tools for reverse engineering – analyzing the disassembled code. It is important to notice that during this process, the program is not executed; the objective is to extract information directly from the file and the data available within it. Otherwise, on Dynamic Analysis, the focus is on the behavior of a running program. In this approach, a program is executed in a controlled virtual environment, a sandbox, where its actions are monitored [Mangialardo and Duarte 2015]. This work focuses on Static Analysis, which is simpler and quicker to achieve.

3. Deep Learning Autoencoders

An Autoencoder or Autoassociator is a feed-forward neural network, trained in unsupervised manner, with the goal to approximate an identity function, in other words, given the input data, the network will try to learn how to copy the input values to its output. It is divided into two functions: an encoder, $h = f(x)$, that is responsible for codifying the input x into a new representation h, and a decoder, $r = g(h)$, which tries to reconstruct the original data from the hidden layer.

An approach to learning a useful representation is restricting the hidden layer dimension to be smaller than the size of the input dimension. With a constrained h, the network would learn the most salient features in order to reconstruct x with minimal error. An autoencoder with these characteristics is called undercomplete. The opposite approach, the overcomplete, in which h is not constrained by the network's input dimension can be used with the risk that, given the extended capacity, the network could copy the input data without learning useful features. In this case, other regularization techniques must be applied, such as a sparsity penalty function applied upon h (sparse autoencoders) or corrupting x and training the autoencoder to undo the corruption (denoising autoencoders) [Goodfellow et al. 2016].

In order to pre-initialize a MLP, a Deep Autoencoder can be assembled in two ways: The first, like a shallow autoencoder version, but not restricted to use just one hidden layer; or, second, using the stacking technique, in which shallow autoencoders, with just one hidden layer, are trained individually and their results' representations are used as the input to another autoencoder to be trained on and so forth [Vincent et al. 2010, Bengio et al. 2013, Goodfellow et al. 2016]. The union of these learned representations forms a deeper autoencoder.

4. Related Work

In recent years we have seen the use of Deep Learning architectures applied to Malware detection and classification, generally benefiting from previous knowledge that helps to achieve the task. [Hardy et al. 2016] developed a framework for malware detection. The system is divided into two phases: Extraction and conversion of Windows API calls from PE files into 32-bit signatures and pre-training Stacked Autoencoders to further classification. The method showed superior performance in relation to traditional machine learning methods. [Yuan et al. 2014] used features acquired via static and dynamic analysis applied to Deep Belief Networks to detect malware on Android applications. The approach achieved the best accuracy over traditional methods. [Dahl et al. 2013] developed a large-scale malware detection system, based on Deep Belief Networks and Random Projections for dimensionality reduction on selected features. The best result was attained with an ensemble of nine neural networks.

5. Autoencoders for Malware Classification

Here, we present details about an approach to malware static analysis employing a deep autoencoder to pre-initialize a deep multilayer perceptron network for classification of malwares into families. This process can be seen as a pipeline, from the extraction of raw data of the disassembled files until the execution of the trained neural network for prediction's evaluation.

5.1. "Bag of Opcodes", Vectorization and TF-IDF weighting

After extracting the byte opcodes from each disassembled file, a bag-of-opcodes (bag-of-words like) for unigrams and bigrams was generated and its terms were subsequently counted and stored as vectors.

The *Term Frequency - Inverse Document Frequency (TF-IDF)* weighting is then applied to each vector. This technique provides a way to select and analyse the opcodes by its frequencies, which can be related to multiple kinds of malware, acting like some sort of "signature" for malicious code. The TF-IDF is computed for a term t by the formula $\mathit{tfidf}\,(d,t) = \mathit{tf}(t) \times \mathit{idf}\,(d,t)$, where IDF is calculated as $\mathit{idf}(d,t) = \log\left[\frac{(1+n)}{1+\mathit{df}(d,t)}\right] + 1$, n is the total number of documents and $\mathit{df}(d,t)$ is the number of documents that contains the term l [Pedregosa et al. 2011, Manning et al. 2008]. This new vector representation is used to train and validate our networks.

Following a hold-out cross-validation strategy to evaluate the models, the generated vectors are randomly assigned to a particular subset, training or validation, guaranteeing that 25% of the samples from each malware class are present on validation subset.

5.2. Models

On the next phase, Deep Autoencoder networks are configured, trained and evaluated upon the experimentation set. The pre-trained Autoencoder's parameters are used to produce a new network, a Multilayer Perceptron (MLP), which is trained and evaluated on the same dataset with the objective of classification. In this approach, we choose to implement deep autoencoders due to it's architectural similarities with a MLP, thus becoming simple to use the trained weights from the former to initialize the parameters of the latter. All autoencoders produced in the experiments falls in the category of undercomplete autoencoders. Undercomplete autoencoders compress highly dimensional data into lower dimensional representations, forcing the network to learn high quality features directly from input data [Goodfellow et al. 2016]. The choices for the number of layers and number of neurons per layer in each model were based on previous experimentations in which we choose to detail in this study only the deepest networks that achieved an interesting result.

6. Experiments

This section describes the neural networks architecture and hyper-parameters used over the experiments. All experiments were programmed using Keras Framework with Tensorflow as backend [Chollet et al. 2015, Abadi et al. 2015]. When not mentioned, it is assumed that the neural network layers' parameters used default values initialized by the framework. Scikit-learn library was employed on the n-grams extraction, terms counting, vectorization and TF-IDF weighting [Pedregosa et al. 2011].

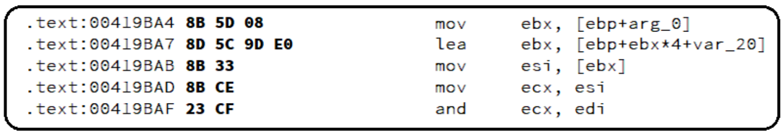
6.1. Dataset

In 2015, Microsoft sponsored a challenge - Microsoft Malware Classification Challenge (BIG 2015) - through Kaggle website where the goal was to classify the provided malware samples into nine different families [Microsoft 2015]. The samples were divided into two sets: the training set, in which every sample was labeled and a validation set containing unlabeled samples. The training set contains two files per sample ID: a ".byte" that contains a hexadecimal representation of the malware's binary without the PE header and a ".asm" with disassembled code and various information extracted from the binary by a professional disassembler program - IDA Pro Disassembler [Hex-Rays 2017].

The distribution of samples throughout the malware families in the original set is as follows (Class – Number of Samples): 1) *Ramnit* – 1541, 2) *Lollipop* – 2478, 3) *Kelihos_ver3* – 2942, 4) *Vundo* – 475, 5) *Simda* – 42, 6) *Tracur* – 751, 7) *Kelihos_ver1* – 398, 8) *Obfuscator.ACY* – 1228 and 9) *Gatak* – 1013, accumulating 10868 samples in total. For the sake of experimentation, only samples of three classes in the original labeled dataset were selected to create our experimentation set, composed only of the disassembled ".asm" files. The criteria adopted to choose these classes were to use the samples from the two most frequent families, *Lollipop and Kelihos_ver3*, and from the rarer one, in this case, *Simda*. Given this selection, for each file, we execute an extraction procedure where only the hexadecimal bytes that represent the *opcodes* and their parameters are separated to another file (Figure 1).

6.2. Performance Measures

The Performance Measures selected to evaluate the experiments are: number of **True Positive (TP)** samples, number of **True Negative (TN)** samples, number of **False Positive (FP)** samples, number of **False Negative (FN)** samples , **Accuracy**, **Precision**, **Recall** and **F1 Score**.



```
.text:00419BA4 8B 5D 08              mov     ebx, [ebp+arg_0]
.text:00419BA7 8D 5C 9D E0           lea     ebx, [ebp+ebx*4+var_20]
.text:00419BAB 8B 33                 mov     esi, [ebx]
.text:00419BAD 8B CE                 mov     ecx, esi
.text:00419BAF 23 CF                 and     ecx, edi
```

Figure 1. Excerpt of a disassembled file from the original dataset [Microsoft 2015]. Byte opcodes and parameters are in bold.

6.3. Networks' topologies and parameters

In respect to the topological aspect, each experiment had a particular configuration for the number of layers and the number of neurons, or units, per layer. Three autoencoder networks were created considering the opcodes vectors, two for unigrams and one for bigrams vectors, respectively (**Figure 2**). In the experiment ***"Unigram 1"***, the autoencoder has 19 hidden layers with input and output layers having 96 units. The MLP was built using the trained encoder parameters adding another layer for classification, resulting in a network with 12 layers in total (96, 86, 76, 66, 56, 46, 36, 26, 16, 9, 3 and 3 units). For ***"Unigram 2"***, the autoencoder has 59 hidden layers with input and output layers having 96 units. The MLP built has 32 layers (96, 93, 90, 87, 84, 81, 78, 75, 72, 69, 66, 63, 60, 57, 54, 51, 48, 45, 42, 39, 33, 30, 27, 24, 21, 18, 15, 12, 9, 6, 3 and 3 units) . The last architecture, for experiment ***"Bigram"***, the autoencoder has 19 hidden layers, with input and output layers having 9216 units. The MLP built has 31 layers (9216, 8216, 7216, 6216, 5216, 4216, 3216, 2216, 1216, 600, 3 and 3 units). Different setups were also evaluated on smaller subsets of the dataset, but these were the ones that showed best initial results.

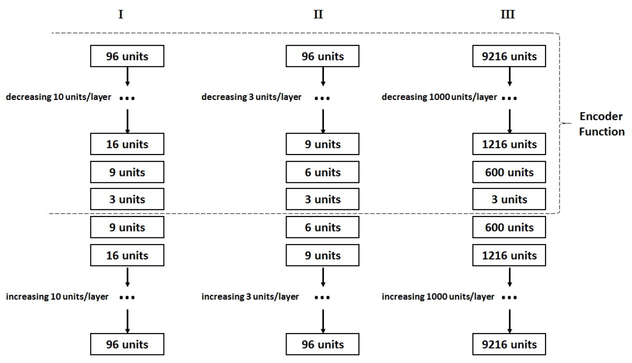


Figure 2. Autoencoders' Topologies for I. Unigram 1; II. Unigram 2 and III. Bigram.

The *Rectified linear unit* (ReLU) activation function was used for all layers for the three autoencoders and posteriorly on the MLPs. ReLU overcome the use of traditional sigmoidal activation functions for deep neural networks and helped to achieve better results in several datasets in the most diverse domains [Schmidhuber 2015]. The Sigmoid function was choosed for the MLP's classification layer. The Loss function used for autoencoders was the *Mean Squared Error* and the *Cross-entropy* for the MLPs. With regards to the training and validation phase, we configured a loop of 1,000 epochs, using mini-batches of 32 samples. Early stop criteria was adopted, based on the minimization of the loss value measured on network's validation, with a delta of at least 0.01, for autoencoders, and the maximization of the accuracy value measured on network's validation, with a delta of at least 0.01, for the MLPs. The training stops if the criteria analyzed were not reached in a span of a hundred epochs. The optimizer used was the *Stochastic Gradient Decent* (SGD), with a learning rate value of 0.01.

6.4. Results

After training all deep autoencoders and assembling all classifiers, we executed the MLP training and evaluation, resulting in the collected measurements as shown in Table 1.

Table 1. Performance measurements by class.

Experiment	Classes	TP	TN	FP	FN	Recall	Precision	Accuracy	F1 Score
Unigram 1	Lollipop	617	747	0	3	99,51%	100,00%	99,78%	99,75%
	Kelihos_ver3	736	617	14	0	100,00%	98,13%	98,97%	99,05%
	Simda	0	1356	0	11	0,00%	–	99,19%	0,00%
Unigram 2	Lollipop	0	747	0	620	0,00%	–	54,64%	0,00%
	Kelihos_ver3	736	0	631	0	100,00%	53,84%	53,84%	69,99%
	Simda	0	1356	0	11	0,00%	–	99,19%	0,00%
Bigram	Lollipop	619	747	0	1	99,83%	100,00%	99,92%	99,91%
	Kelihos_ver3	736	623	8	0	100,00%	98,92%	99,41%	99,45%
	Simda	3	1355	1	8	27,27%	75,00%	99,34%	40,00%

The first experiment, that used unigram features, ***Unigram 1***, showed good results in classifying samples from the *Lollypop* and *Kelihos_ver3* families – mistaken three samples from the former and classifying every sample from the latter – the dataset's most representative families, although this network was not capable to classify any of the *Simda* examples provided.

Unigram 2, the deepest architecture tested, achieved the worst performance over all experiments. It classified every sample of the dataset as a member of the *Kelihos_ver3* family.

However, the network trained with bigram features – ***Bigram experiment*** – achieved the best results for all families in the experimentation dataset, being capable of classifying almost every *Lollipop* sample, misclassifying just by one sample, and every *Kelihos_ver3* sample provided. It stills correctly classified three examples of *Simda*, the rarest family of the dataset, as seen in subsection 6.1.

In comparison with the challenge's winner's approach, our 'Bigram' model showed a good performance, with the winner's solution achieving 99,83% of average accuracy against an average accuracy of 99.55% of our model. To the best of our knowledge, it was not possible to proper compare our results by class against the winner's solution, using other performance measure than the accuracy. Despite that fact, our 'Bigram' model showed great rates for the two most frequent classes, *Lollypop* and *Kelihos_v3*, with a recall of 99,83% and 100,00% respectively, showing a diminutive number of false positives and false negatives as can be seen in Table 1. The winner's approach relied heavily on feature selection and engineering and used a 4-fold cross validation over the same labeled dataset used in our experiments. Respecting the differences in our dataset, that used just samples from three classes of the original dataset, our approach showed competitive results and indicates a promising solution to the malware classification problem.

7. Conclusion

This paper presents a methodology using byte opcodes and their parameters as features in the form of a ”bag-of-opcodes” with TF-IDF weighting extracted from disassembled malware files, applied to Deep Autoencoders for pre-training Multi-layer Perceptrons to classify malwares into families. It was shown in a series of experiments that even without any former knowledge about specificities of each malware and even losing semantic content when analyzing opcodes as just counts and frequencies of terms in a document, allied with the high capability of deep learning models in learning complex patterns and hierarchical representations that this method could actually achieve promising results, with our best experiment attaining an overall accuracy of 99.55%.

Furthermore, there are still possibilities where this work could be extended. For future work, we hope to run new experiments with the complete dataset, gathering other ’n-gram’ terms. Moreover, in order to improve our results, we expect to run a specificity analysis to find the best parameters for our network models. In order to better evaluate the generalization capacity of our approach we will execute k-fold cross validation on future experiments. Another possibility is experimenting with word embedding techniques to try to extract semantic content from the disassembled instructions. Other deep learning architectures, like Convolutional Neural Networks or Recurrent Neural Networks, are considered to be employed to solve the malware classification problem.

References

Abadi, M. et al. (2015). TensorFlow: Large-scale machine learning on heterogeneous systems. `http://tensorflow.org/`. Software available from tensorflow.org.

Bengio, Y., Courville, A., and Vincent, P. (2013). Representation learning: A review and new perspectives. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1798–1828.

Chollet, F. et al. (2015). Keras. `https://github.com/fchollet/keras`. [Online; accessed 2-August-2017].

Dahl, G. E., Stokes, J. W., Deng, L., and Yu, D. (2013). Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*, pages 3422–3426. IEEE.

Damshenas, M., Dehghantanha, A., and Mahmoud, R. (2013). A survey on malware propagation, analysis, and detection. *International Journal of Cyber-Security and Digital Forensics (IJCSDF)*, 2(4):10–29.

Goodfellow, I., Bengio, Y., and Courville, A. (2016). *Deep Learning*. MIT Press. `http://www.deeplearningbook.org`.

Hardy, W., Chen, L., Hou, S., Ye, Y., and Li, X. (2016). Dl4md: A deep learning framework for intelligent malware detection. In *Proceedings of the International Conference on Data Mining (DMIN)*, page 61. The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp).

Hex-Rays (2017). IDA Pro Disassembler. `https://www.hexrays.com/products/ida/index.shtml`. [Online; accessed 2-August-2017].

Jang-Jaccard, J. and Nepal, S. (2014). A survey of emerging threats in cybersecurity. *Journal of Computer and System Sciences*, 80(5):973–993.

LeCun, Y., Bengio, Y., and Hinton, G. (2015). Deep learning. *Nature*, 521(7553):436–444.

Mangialardo, R. J. and Duarte, J. C. (2015). Integrating static and dynamic malware analysis using machine learning. *IEEE Latin America Transactions*, 13(9):3080–3087.

Manning, C. D., Raghavan, P., and Schütze, H. (2008). Scoring, term weighting and the vector space model. *Introduction to information retrieval*, 100:2–4.

Marpaung, J. A., Sain, M., and Lee, H.-J. (2012). Survey on malware evasion techniques: State of the art and challenges. In *Advanced Communication Technology (ICACT), 2012 14th International Conference on*, pages 744–749. IEEE.

McAfee (2014). Net losses: Estimating the global cost of cybercrime. intel security (mcafee). `http://www.mcafee.com/us/resources/reports/rp-economic-impact-cybercrime2.pdf`. [Online; accessed 2-August-2017].

McAfee (2015). Previsões do mcafee labs sobre ameaças em 2016. `http://www.mcafee.com/br/resources/reports/rp-threats-predictions-2016.pdf`. [Online; accessed 2-August-2017].

McGraw, G. and Morrisett, G. (2000). Attacking malicious code: A report to the infosec research council. *IEEE Softw.*, 17(5):33–41.

Microsoft (2015). Microsoft Malware Classification Challenge (BIG 2015). `https://www.kaggle.com/c/malware-classification`. [Online; accessed 2-August-2017].

Pedregosa, F., Varoquaux, G., Gramfort, A., Michel, V., Thirion, B., Grisel, O., Blondel, M., Prettenhofer, P., Weiss, R., Dubourg, V., Vanderplas, J., Passos, A., Cournapeau, D., Brucher, M., Perrot, M., and Duchesnay, E. (2011). Scikit-learn: Machine learning in Python. *Journal of Machine Learning Research*, 12:2825–2830.

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

Symantec (2016). Internet Security Threat Report. `https://www.symantec.com/content/dam/symantec/docs/reports/istr-21-2016-en.pdf`. [Online; accessed 2-August-2017].

Verison (2016). 2016 data breach investigations report. `http://www.verizonenterprise.com/verizon-insights-lab/dbir/2016/`. [Online; accessed 2-August-2017].

Vincent, P., Larochelle, H., Lajoie, I., Bengio, Y., and Manzagol, P.-A. (2010). Stacked denoising autoencoders: Learning useful representations in a deep network with a local denoising criterion. *Journal of Machine Learning Research*, 11(Dec):3371–3408.

Yuan, Z., Lu, Y., Wang, Z., and Xue, Y. (2014). Droid-sec: Deep learning in android malware detection. In *ACM SIGCOMM Computer Communication Review*, volume 44, pages 371–372. ACM.