Empowering Convolutional Networks for Malware Classification and Analysis

Bojan Kolosnjaji[1], Ghadir Eraisha[1], George Webster[1], Apostolis Zarras[1], and Claudia Eckert[1,2]

[1] Technical University of Munich [2] Fraunhofer AISEC

***Abstract*—Performing large-scale malware classification is increasingly becoming a critical step in malware analytics as the number and variety of malware samples is rapidly growing. Statistical machine learning constitutes an appealing method to cope with this increase as it can use mathematical tools to extract information out of large-scale datasets and produce interpretable models. This has motivated a surge of scientific work in developing machine learning methods for detection and classification of malicious executables. However, an optimal method for extracting the most informative features for different malware families, with the final goal of malware classification, is yet to be found. Fortunately, neural networks have evolved to the state that they can surpass the limitations of other methods in terms of hierarchical feature extraction. Consequently, neural networks can now offer superior classification accuracy in many domains such as computer vision and natural language processing.**

In this paper, we transfer the performance improvements achieved in the area of neural networks to model the execution sequences of disassembled malicious binaries. We implement a neural network that consists of convolutional and feedforward neural constructs. This architecture embodies a hierarchical feature extraction approach that combines convolution of n-grams of instructions with plain vectorization of features derived from the headers of the Portable Executable (*PE*) files. Our evaluation results demonstrate that our approach outperforms baseline methods, such as simple Feedforward Neural Networks and Support Vector Machines, as we achieve 93% on precision and recall, even in case of obfuscations in the data.

I. Introduction

Malicious software (malware) detection and analysis is becoming more and more difficult as the number of newly detected malware samples grows with an increasing rate. As such, the amount of new malware variants identified by security companies has exponentially increased throughout the years. This is an overwhelming volume of data for malware analysts as they need to extract pertinent information out of these extremely large-scale datasets. The statistics page of VirusTotal shows that the number of newly submitted samples for analysis can easily reach, or even surpass, the one million files per day [1]. This surge of samples makes reverse engineering a challenging task. Although there exist efforts to automate the reverse engineering and malware analysis process, manual signature-based or heuristics-based detection and analysis procedures are still very prominent. The samples not only increase in sheer numbers but also in variety, which is usually caused by advances in malware development that utilize polymorphic and metamorphic algorithms to generate different versions of the same malware. This confines signature-based systems to correctly detect, classify, and analyze malware. Furthermore, it encumbers existing reverse engineering processes to scale up to the order of millions of samples.

To aid malware analysts in retrieving useful information from such a large amount of executables, we need to solve the problem of performing large-scale automatic detection and classification under the existing statistical variance. Assigning malicious examples to preexisting families helps malware analysts to focus their efforts and automate the system protection countermeasures. In order to improve this malware triage procedure, we need a more robust alternative that can abstract away the noise and capture the essential information from static or behavioral malware properties.

Static analysis tools, such as PEInfo [2] and Yara [3], offer extraction of different properties and metadata (e.g., entropy, histograms, section length) from malware code. This data is crucial for characterizing malware samples, under the condition that the binary is not obfuscated to the extent that the majority of properties and metadata are significantly changed. On the other hand, behavioral analysis tools are less sensitive to code obfuscation, as they only record traces of activity retrieved from the execution of malware samples. However, the behavioral traces provide information about only one execution path out of many possible cases.

Many machine learning-based systems have been developed as a solution to the problems of processing a large number and variety of malware samples. As a matter of fact, previous works attempted to tackle the problem of modeling malware behavior [4]–[7]. Apart from behavioral data, static code properties have also been used as data sources for statistical analysis [8]. Furthermore, there are works trying to combine static and dynamic approaches [9]. Yet, albeit neural networks' increasing popularity in machine learning applications, as they have caused performance improvements in many areas, these approaches have not been extensively used in malware analysis. Nevertheless, there exist efforts trying to introduce neural networks to this application area. For instance, feedforward networks have been recently utilized to analyze malware code [10], while recurrent and convolutional networks have been employed for modeling system call sequences in order to construct a "*language model*" for malware [11], [12].

The previous works proved valuable in malware classification, but they are weak for capturing features out of assembly instructions, as for example in cases where the data has hierarchical structure. The reason is that standard machine learning

methods do not take into account the feature hierarchies. Using the assumption that the sequence of instructions in the program actually represents a manifestation of high level actions, we hypothesize the existence of such feature hierarchy. As a matter of fact, we show that by using the convolutional network layers it is possible to obtain clear benefits in terms of discovering valuable information and increasing classification performance.

In this paper, we attempt to improve the feature extraction and classification methodology for malware datasets. To achieve this improvement, we use a neural network that combines convolutional and feedforward layers. More specifically, we provide as input to the feedforward layers various *Portable Executable* (*PE*) metadata. Additionally we feed the sequences of one-hot encoded assembly instructions into convolutional layers. Using this approach we achieve 93% on precision and recall with a 92% on F1-measure and outperform the standard machine learning approaches, even in the case of obfuscated code. Overall, we show that our proposed approach constitutes a valuable asset in the fight against malware.

In summary, we make the following main contributions:

- We construct a deep neural network and apply it for the classification of malicious executables.
- We combine convolutional and plain feedforward approaches to neural networks for optimizing malware classification.
- We investigate neural unit activation patterns and explain the performance improvement of our models by showing the inner workings of our neural network.

II. Background

This work suggests the use of convolutional networks for improving malware detection and analysis. To facilitate the understanding of this paper, we introduce the concepts of *PE* files and convolutional networks.

A. PE Files

In malware analysis and classification, it is important to know the *PE* format of binaries to properly parse and interpret the malware samples. In detail, the *PE* format is used as a typical way of structuring windows executable files [13]. More precisely, this format prescribes the *PE* File Header, which contains metainformation about the executable files. For example, in the *PE* File Header we can find which *Dynamic Link Libraries* (DLLs) are used by the considered executable and which functions are imported. In addition, we can see the timestamp from the compilation, names of different code sections, their sizes on disk and in memory.

After the *PE* File Header we can find the actual program. This program is divided into multiple sections, for example `.text`, `.rdata`, `.data`, and `.rsrc` section. The `.text` section contains the executable code of the program. This section provides more in-depth insight on how a program operates. Therefore, this code can be disassembled into assembly representation by analysts to inspect it and extract the exact possible features of the given program. The `.rdata` section holds the import and export information, whereas the `.data` section stores the data that is globally accessible throughout the executable. Finally, the `.rsrc` section contains the resources required by the executable, such as images, strings, icons, and menus.

As we already mentioned, it is possible to use the aforementioned sections to characterize the program and enrich this with the additional data using tools like *objdump* and retrieve a series of assembly language instructions. Furthermore, in some cases, it is also possible that this code can be further decompiled to get the source code in a higher level programing language. Although this could simplify the analysis, it does come with its own limitations. For instance, frequently there is a lack of information regarding all the compilation requirements of the binary and therefore an attempt to decompile the binary in a higher level programing language can lead to a loss of valuable information.

B. Convolutional Networks

Neural networks are biologically motivated machine learning constructs. They consist of multiple layers of nonlinear feature transformation, where parameters of this transformation are trained using a gradient descent procedure. In feedforward networks the transformation looks like the following:

$$y = f(\sum_{j=1}^{M} w_j x_j + w_0) \tag{1}$$

where w_i are parameters of the neural network (weights), and f is a nonlinear function that adds complexity to the model.

Multiple layers can be connected by using the output of one layer as an input to the next one. This series typically ends with a *softmax* function in case of classification, which transforms activation values of the last layer into vectors of values between 0 and 1. These vectors indicate the classification labels. Neural networks have long been present in the machine learning community. Some theoretical results show that even the 3-layer network (with one nonlinear layer) can be a universal approximator for smooth functions [14]. Yet, in practice, it has been shown that increasing the number of layers further can improve performance as it enables hierarchical feature extraction [15]. This paradigm of using a high number of layers has been called *deep learning* and it brings many improvements and promising results in the literature, as for example in image processing [16] and text classification [17].

Many successful approaches in deep learning come also from using convolutional networks. This type of networks have been successfully used in image and text processing. In general, convolutional networks often consist of multiple layers of *convolution* and *pooling*. On the one hand, convolution layer uses a nonlinear function to extract features from data samples by moving the convolution filter in a predefined

window [18]. Discrete convolution with filter K executes the following transform on the input I:

$$(I * K)_{r,s} = \sum_{u=-h_1}^{h_1} \sum_{v=-h_2}^{h_2} K_{u,v} I_{r+u,s+v} \quad (2)$$

where the filter is given by:

$$K = \begin{pmatrix} K_{-h_1,-h_2} & \cdots & K_{-h_1,h_2} \\ \vdots & K_{0,0} & \vdots \\ K_{h_1,-h_2} & \cdots & K_{h_1,h_2} \end{pmatrix} \quad (3)$$

The output of a convolutional layer (Y) consists of a series of feature maps. The ith feature map (Y_i) is computed as:

$$Y_i = B_i + \sum K_{i,j} * X_j \quad (4)$$

On the other hand, pooling layer takes the results of a convolutional layer as input and extracts the most salient features. This layer effectively executes subsampling by taking an average or maximum value in a given window. A sequence of convolutional and pooling layers alternates feature extraction and dimension reduction, both of which are trained by executing gradient descent on filter parameters. Gradient is computed using backropagation, i.e., by propagating the errors from the output layer back to the hidden layers.

In image processing, the convolutional filter is used to recognize features in the image, which is very useful for object recognition. This way the objects can be detected almost invariant from their position. Similar to images, in text processing (e.g., sentence classification, search, recommendation) we can extract information and detect high level features for short text using the flexible convolutional filters. Since logs containing instructions from malicious executables consist of sequences of words from a predefined dictionary, there is an obvious analogy with text documents when selecting modeling methods.

III. Methodology

Our methodology consists of gathering and preprocessing appropriate input data, feature extraction, and classification using a neural network.

A. Data Acquisition

We collected a set of malware samples over multiple months from three primary sources: Virusshare Maltrieve and private collections. We selected these sources to provide a large and diverse volume of samples for evaluation. In order to extract useful data out of a large-scale sample set we use a *malware zoo*, where we define and query *RESTful* services to obtain data from multiple malware analysis tools. In our case, we utilize *objdump* [19] to retrieve the opcodes and *PEInfo* [2] to extract information from the *PE* Header. The zoo backend infrastructure is composed of large-scale analysis concepts proposed by Webster et al. [20]. This enables us to distribute and parallelize the data extraction process.

B. Preprocessing

We extract features of *PE* files by preprocessing the fields of the *PE* Header and opcodes from the code section. In order to use this data in the classification process, we need to create numerical feature vectors out of the already existent structure.

PE Metadata. The metadata contained in the *PE* Header offers some potentially useful input. This data is often used in malware analysis, as it is easy to just extract it from the beginning part of the binary. In this work we used the following metadata: (i) compile time stamp, (ii) address offsets to image resources (dialog boxes, menus etc), and (iii) size of image resources. In addition, for each code section we take (i) the entropy, (ii) the virtual address, (iii) the virtual size, and (iv) the size on disk. In total, we extract 27 numerical features.

Our motivation for extracting these features is that the *PE* metadata may help the neural network to detect suspicious aspects of a given *PE* file. For instance, if the virtual size of the `.text` section is much larger than its raw size, then it is implied that the code is packed (i.e., it has been modified using a runtime compression or encryption program). Furthermore, the set of image resources may help the neural network to identify the resources required by malware samples belonging to the same malware family.

PE Import Features. Next, we look at the list of imported functions and their DLL files. We can extract these lists from the *PE* files using the *PE* Header. To vectorize these lists, we encode the presence of unique funictions along with their associated DLLS using the *one-hot* encoding approach. In total, we counted 488 unique functions from the *PE* files in our collection.

Our motivation for extracting these features is that imported functions may help the neural network to identify the purpose of a particular malware sample. For instance, functions imported from *kernel32.dll*—such as *OpenProcess*, *GetCurrentProcess*, and *GetProcessHeap*—imply that malware opens and manipulates processes. This DLL file provides functions for most of the Win32 APIs. Many GUI manipulation imported functions—such as *RegisterClassEx*, *SetWindowText*, and *ShowWindow*—indicate that the malware has a GUI and possibly imitates the appearance of a benign GUI application. Functions imported from *shell32.dll* imply that the malware launches other programs. Subsequently, the neural network classifies malware samples with a semantically similar combination of imported functions to the same family.

Assembly Opcode Features. The last set of features is derived from the assembly instructions of the *PE* files. For each *PE* file, we first disassemble it to generate the assembly instructions with opcode information. Second, we convert the generated sequence of opcodes into a matrix representation, where each row of the matrix is a vector representation of one opcode in the sequence. To convert an opcode into a vector, we represent it by the *one-hot* encoding scheme. Our motivation for extracting these features is that opcodes may help the

Table I: The most frequent malware signatures included in each of the 13 clusters.

Cluster	Signatures
0	Neurevt, QQPass, Keylogger and IRCbot
1	Zapchast and Banload
2	Artemis and IRCbot
3	Neurevt, QQPass and IRCbot
4	Dridex
5	QQPass and Keylogger
6	Banload
7	Generic
8	Genbl
9	Keylogger
10	IRCbot
11	QQPass and IRCbot
12	Androm

neural network to capture the semantics of the instruction usage patterns that a particular malware sample relies upon and thus exhibit the exact behavior of the sample. Consequently, the neural network can learn the discriminative instruction usage patterns among different benign and malware families.

Signature Clustering. In the past, there were many issues with training labels for malware classification [21]. To get more confident labels, we perform a selection procedure that utilizes a simplified version of signature clustering method introduced in VAMO [22]. Specifically, we create boolean label vectors for every malware sample that represent presence or absence of signatures given by different antivirus engines. Our assumption is that the malware samples of the same family will have the same or at least similar boolean feature vector. The sample signature vectors are clustered using DBSCAN [23] and a variant of *cosine distance*, as we do not know the number of clusters in advance. The DBSCAN algorithm enables us to find significant clusters in the signature vector dataset and tune the algorithm to make clusters with high population and compactness in the feature space. It also enables us to detect outliers, as they will not be included in any significant cluster. Finally, we select 13 clusters with the highest number of members as families for classification, since they cover most of our dataset. Table I illustrates the most prominent antivirus signatures included in each of the 13 clusters.

C. Neural Network Architecture

We use a hybrid approach to design our neural network to get an optimal structure for hierarchical feature extraction. Figure 1 illustrates an overview of our architecture. While we transform the *PE* metadata and *PE* Import features using feedforward neurons, in parallel, we leverage convolutional network layers for the malware opcode sequences.

As we mentioned, in order to construct our final neural network-based malware classifier, we combine the feedforward and convolutional neural network architectures along with their corresponding features into a single network. The second hidden layer from the feedforward neural network and the pooling layer from the convolutional neural network are aggregated into one layer. Then, a fully-connected output layer,

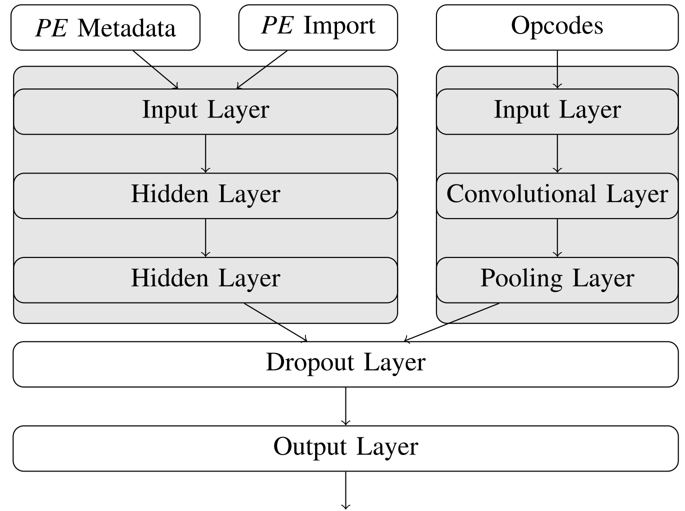

Figure 1: Overview of our neural network architecture.

of 13 *softmax* units is added to generate the final classification output. In the following, we provide more details about the feedforward and convolutional layers.

Feedforward Layers. We use feedforward sublayers to process the data that does not have the structure which would benefit from convolutional filters. Specifically, we use 27 features from *PE* metadata and 488 boolean features from the *PE* Import list. We apply a dropout layer with 20% dropout rate. This is followed by two fully-connected hidden layers, where each of the layers has 500 units, with a *Parametric Rectifier Linear Unit* (PReLU) [24] activation function. Both of the hidden layers apply 20% dropout to the hidden units. The weights of the hidden layers are initialized using Glorot's scheme [25]. Finally, after the last dropout layer, we use a fully connected output layer of 13 *softmax* units to generate the final classification output.

Convolutional Layers. For the assembly opcode sequences we use convolutional layers to make the neural network learn the high level features out of this data. We want to use convolution as a way of capturing the semantics of the instruction usage patterns. Figure 2 reveals the structure of convolutional layers. We assume that as a consequence of the good feature extraction, our convolutional architecture in the neural network will help us to discriminate usage patterns among benign and malware families. Indeed, the convolutional filter helps to discover the higher-order local features that are invariant to small changes in data. This means that our method can also help with small obfuscation done by the malware authors, such as reordering of instructions and adding an amount of unreachable code.

Since the rows of the input matrix represent discrete opcodes, it is reasonable for filters to slide over full rows of the matrix, similar to the applications in Natural Language Processing. Thus, we choose the width of the filters to be 118, which is equal to the dimensionality of the opcode

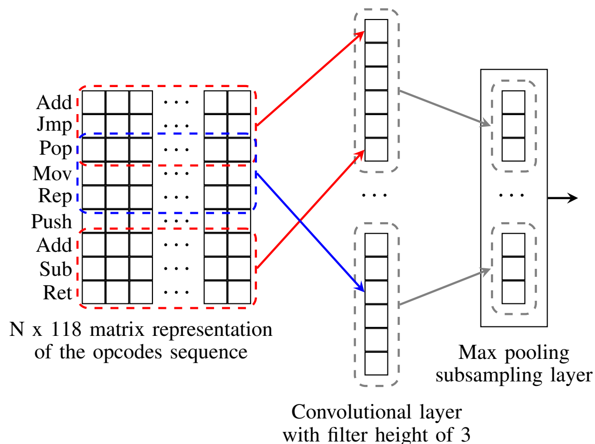

Figure 2: Convolutional layers.

vectors. However, the height of the filters, which represents the number of adjacent opcodes, varies among different variants of our model. We mainly experimented with 2, 3, 4, and 5 filter heights with 10 feature maps each. The dimensions of the filters are chosen by looking at NLP application papers, where convolutional filters can detect specific patterns in the assembly instructions that are quite similar to opcode n-grams. Finally, we choose the height 3 as it gives best results in our experiments.

The convolutional layers are followed by a max-pooling subsampling layer with factor 2 in one dimension, which reduces the output dimensionality while keeping the significant global information captured by the filters. If a specific opcode sequence pattern is reallocated in the code, information regarding whether this pattern occurred in the code will be kept, but information regarding its order in the code will be lost. Finally, we use a fully-connected output layer of 13 *softmax* units to generate the final classification output. Similar to the feedforward neural network design, PReLU was chosen as the activation function, and Glorot's scheme was selected to initialize the network weights.

The neural network is trained over shuffled mini-batches using backpropagation and stochastic gradient descent with Nesterov momentum [26].

IV. Evaluation

In this section we show and comment on the results of our approach. More precisely, we first discuss about our crossvalidation experiments and then present our classification results. Finally, we show how our approach is robust against instruction reordering.

A. Crossvalidation Experiments

We conducted a set of crossvalidation experiments to assess how well each of the feedforward and convolutional neural networks individually perform, and then we evaluated the performance of the hybrid neural network. We ran the experiments on a machine with a *CUDA* graphical processing unit. In addition, we built and trained the neural network

Table II: Classification performance of the feedforward, convolutional, and hybrid neural networks as well as the SVM.

Model	Precision	Recall	F1-score
FFNN	0.90	0.92	0.90
CNN	0.92	0.92	0.91
SVM	0.92	0.92	0.92
Hybrid NN	0.93	0.93	0.92

models using the Lasagne v0.1 [27] and Tensorflow [28] libraries. Finally, we wrote the feature extraction in Python programing language, and extensively use the SciPy and NumPy libraries. Our final dataset after noise filtering contains 22,757 executables, with 22,694 malicious executables and 63 benign executables from ZDNet's download list of most popular programs [29]. ("*benign*" class). One may notice that we ended up with unbalanced datasets. However, since we use multiclass classification, where we have one benign class and multiple classes of malicious executables, this reduces the problem with balance among class size.

In detail, we conducted 3-fold crossvalidation experiments to estimate the results over new data. In each experiment, we randomly split the dataset into three equally sized partitions, where we trained against two partitions and tested against the remaining one. This process was repeated for three times while a different partition is left out for testing each time. Finally, we computed the average results of the three tests to obtain a reliable measure of how well the proposed neural network architecture performs over the entire dataset. We quantitatively assessed the performance of the neural network architecture using three metrics: *precision*, *recall*, and *F1-score*.

To measure the performance gain brought by our combined deep neural network, we trained a reference support vector machine (SVM) classifier using the *PE* metadata, *PE* import, and assembly opcode features. In detail, we used the same set of *PE* metadata and import bag-of words features. However, in the case of the assembly opcodes of a *PE* file we used counts of 3-grams of opcode instructions to get a methodology for comparison with convolutional neural networks.

B. Classification Results

We compared the averages of the classification performance of the fully feedforward, convolutional, and our neural network as well as the support vector machine, as shown in Table II. Our hybrid neural network provides a performance improvement over the feedforward and convolutional neural networks as well as the support vector machine. The hybrid neural network achieves an F1-score value of 0.92 as well as precision and recall values of 0.93.

The experiments demonstrated that our approach accurately classifies malicious executables that behave similarly into the same family. Table III depicts the averages of the classification performance for each of the classes of the hybrid neural network. Table IV, on the other hand, shows the confusion matrix of the hybrid neural network, which illustrates how

Table III: Classification performance of our neural network for each of the classes.

Class	Precision	Recall	F1-score
0	1.00	0.16	0.27
1	1.00	1.00	1.00
2	0.00	0.00	0.00
3	0.41	1.00	0.59
4	0.93	1.00	0.96
5	0.92	0.99	0.95
6	0.87	0.31	0.46
7	0.98	0.96	0.97
8	0.56	1.00	0.72
9	0.78	0.75	0.77
10	0.67	0.10	0.18
11	0.00	0.00	0.00
12	0.50	0.80	0.61
Benign	1.00	0.92	0.96
Average	0.93	0.93	0.92

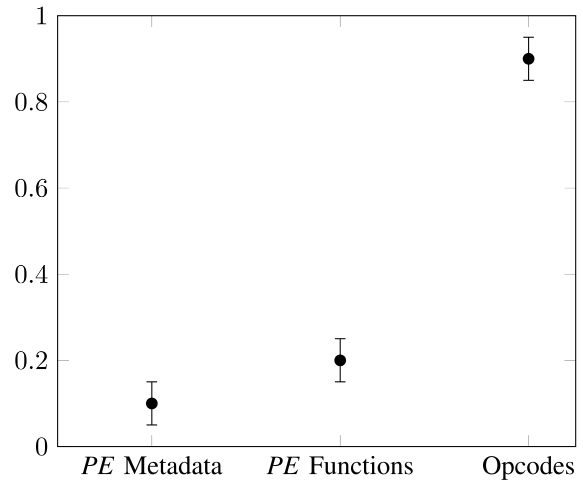

Figure 3: Visualizing the average activation values of the input features for ten malware samples of class 1.

malware samples of a particular family are correctly classified, or misclassified into different families. In detail, Table III reveals that samples belonging to classes 1, 4, 5, and 7 as well as the benign samples are approximately correctly classified. In contrast, samples belonging to classes 0, 2, 10, and 11 are misclassified. Furthermore, the confusion matrix of the deep neural network, shown on Table IV, indicates that malware samples belonging to classes 0, 2, and 11 are particularly confused with class 3. The reason behind this is that these classes share common malware signatures, as demonstrated in Table I.

Next, we examined the activation values of the input neurons at the input layers in order to determine which features have higher activation values after training. For instance, Figure 3 presents the average activation values of the input features for ten malware samples of class 1. It is interesting to see that the most active neurons at the input layer are the assembly opcode features. This indicates that in our tests the assembly opcode features make the most significant contribution to the malware classification.

Furthermore, the high-dimensional activations of the neurons at the last hidden layer were projected to a lower dimensional space for visualization and analysis. As shown in Figure 4, the feature points for classes 1, 3, 4, 5, 8, 12, and 13 are well separated. However, there is confusion among classes 6, 7, 9, and 10. To keep the distances between activation values of different samples while reducing the vectors to 2D we used the T-SNE [30] method for dimension reduction.

C. Robustness Against Instruction Reordering

Attackers often create malware variants by reordering the instructions to bypass signature detection. The instructions are reordered in such a way that preserves the inter-instruction dependencies. Hence, in this experiment, we demonstrate the robustness of the neural network against instruction reordering. Given 150 test samples, we randomly shuffled their corresponding sequences of opcode and measured the neural network performance. Our neural network was still able to accurately classify the malware and benign samples with F1-score values above 0.90 until 50% rate of shuffling. Additionally, we randomly increased or decreased the input frequencies of 3-gram opcodes to the support vector machine for these 150 test samples and measured the support vector machine performance. The support vector machine performance has

Table IV: The confusion matrix of the deep neural network.

	0	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**	**12**	**Benign**
0	26	0	0	119	0	0	0	0	0	0	0	0	0	0
1	0	100	0	0	0	0	0	0	0	0	0	0	0	0
2	0	0	0	267	0	0	0	0	0	0	0	0	0	0
3	0	0	0	504	0	0	0	0	0	0	0	0	0	0
4	0	0	0	0	157	0	0	0	0	0	0	0	0	0
5	0	0	0	0	0	869	0	4	0	2	0	0	0	0
6	0	0	0	0	0	0	70	170	14	0	0	0	12	0
7	0	0	0	200	25	2	14	18572	245	16	10	0	180	0
8	0	0	0	0	0	0	0	0	430	0	0	0	0	0
9	0	0	0	0	4	2	2	20	0	98	0	0	4	0
10	0	0	0	0	0	0	0	112	19	5	12	0	0	0
11	0	0	0	129	0	65	0	0	1	0	0	0	0	0
12	0	0	0	0	0	0	0	66	0	0	0	0	145	0
Benign	0	0	0	0	0	0	0	4	0	0	0	0	0	59

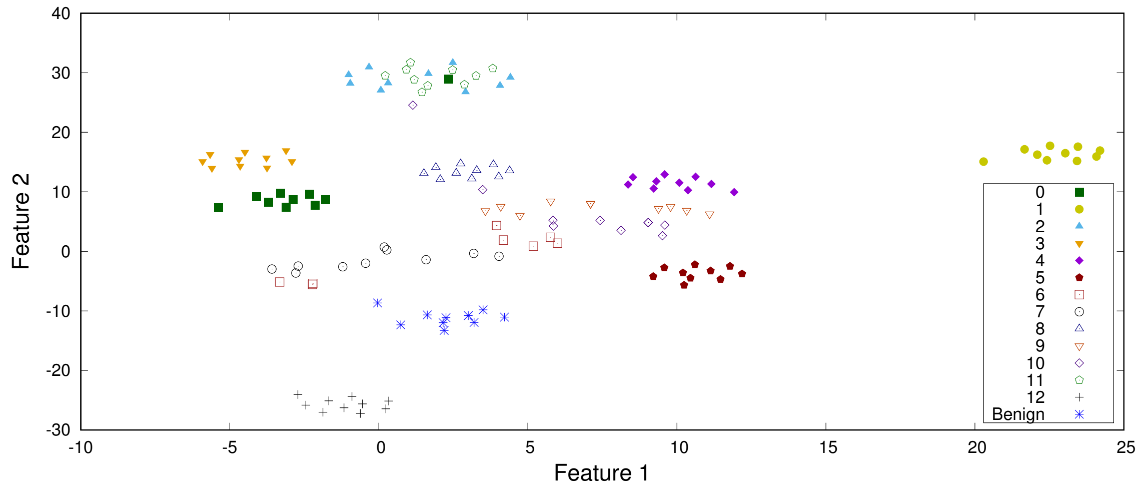

Figure 4: Visualizing ten samples from each class with their corresponding activation values of the last hidden layer neurons.

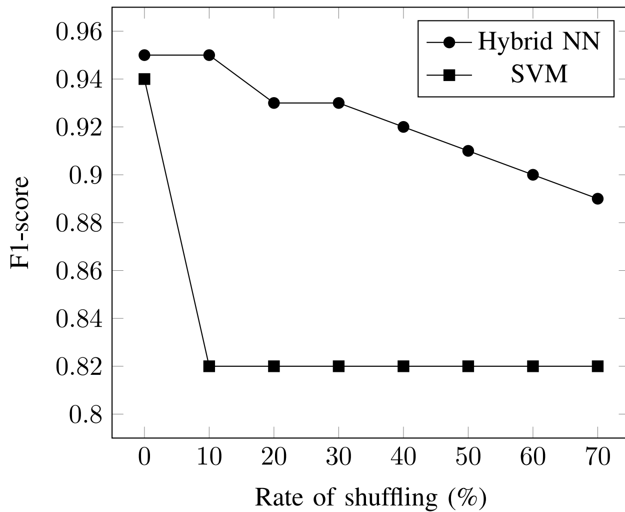

Figure 5: Demonstrating the decrease in the F1-score for the hybrid neural network and support vector machine while increasing the rate of instruction reordering.

dramatically decreased to an F1-score of value 0.82. Figure 5 shows the small decrease in the F1-score of the hybrid neural network while increasing the rate of instruction shuffling. In addition, it shows the significant decrease in the F1-score of the support vector machine while increasing the rate of 3-grams' distribution alteration. This shows that our methodology contains a defense mechanism against the above mentioned obfuscation approaches.

D. Saliency Maps

In order to visualize and interpret the classification process of our neural network, we use the methodology developed by Symonian et al. [31]. In particular, we use Taylor expansion and compute first partial derivative of the classification results before putting them through the softmax function. This gives us a detector of salient features that contribute the most to attributing samples with a certain class. Out of a saliency map we can draw conclusions about the reasons for classification results. For example, in Figure 6 we can see that the chosen sample should be given a label 7, according to most of the *PEInfo* features. Yet, there is a significant peak at the feature with number 34 that indicates that this sample also has some characteristics of the class 10 (i.e., IRCBot). This feature is the size of a certain *PE RESOURCE* field in the *PE Header*.

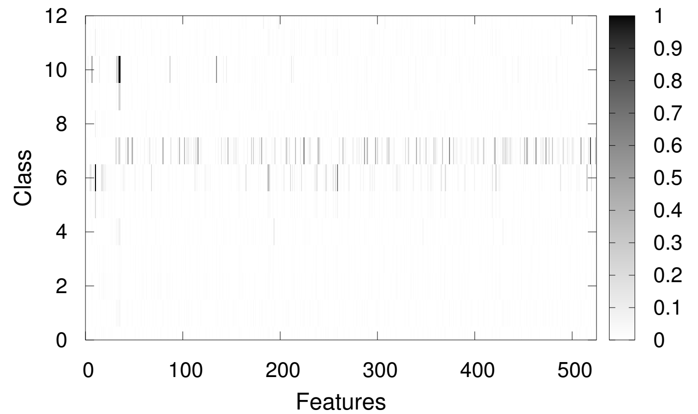

Figure 6: Saliency map for a chosen sample and PEInfo features.

V. Limitations and Future Work

Although we have, in principle, achieved our goal of leveraging convolutional network to improve malware classification, there are some limitations to our methodology. Regarding the dataset, we only use results of static malware analysis. Although we show that our convolutional layers are somewhat robust to code obfuscation, there still are ways to obfuscate the code to an extent that our methodology will not work. For example, the malware samples may be packed with a custom packer or contain code that gets decrypted in execution. In dynamic analysis, malware samples are executed in a protected

environment and traces of actual malware behavior can be extracted. Optimal methodology would be to use both static features and behavioral traces to gain knowledge from both views. Since multiple views on malware provide different aspects of malware characteristics, one machine learning method cannot be optimal for this heterogeneous data. Previous work has shown a multiple kernel learning approach to unifying models based on different aspects of malware characteristics [32].

The second limitation is that we use a closed world assumption, i.e., we do not consider how appearance of malware from a family unknown to our system will affect the performance. Finally, in our methodology we use convolutional filters to capture information from opcode sequences. However, we do not create an explicitly sequential model. One possible improvement would be to create such a model using fully recurrent networks. This may be difficult because recurrent networks are hard to train efficiently. However, future work can be directed to tackle this problem.

VI. Conclusion

In this paper, we presented a novel neural network for detection and classification of malware based on information from static analysis. The neural network consists of convolutional and feedforward layers and uses metadata of *PE* files, imported functions, and series of opcodes to separate malicious executables from benign programs and classify malware into 13 predefined classes. Furthermore, it differentiates benign executable files from the predefined malicious classes. The evaluation results demonstrate that our approach outperforms baseline machine learning methods such as feedforward networks and support vector machines. In detail, we achieved 93% on precision and recall, with an F1-score of 92%. Our neural network also maintains partial resilience to obfuscation done by shuffling instructions and adding bogus code, as a difference from baseline machine learning methods.

Acknowledgements

The research was supported by the German Federal Ministry of Education and Research under grant 16KIS0327 (IUNO) and by the Bavarian State Ministry of Education, Science and the Arts as part of the FORSEC research association.

References

[1] VirusTotal, "File Statistics," https://www.virustotal.com/en/statistics/, Nov 2015.

[2] "PEInfo Service," https://github.com/crits/crits_services/tree/master/peinfo_service.

[3] V. M. Alvarez, "Yara 3.3.0. VirusTotal (Google, Inc)," http://plusvic.github.io/yara/, 2015.

[4] S. Attaluri, S. McGhee, and M. Stamp, "Profile Hidden Markov Models and Metamorphic Virus Detection," *Journal in computer virology*, vol. 5, no. 2, pp. 151–169, 2009.

[5] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, Behavior-Based Malware Clustering." in *ISOC Network and Distributed System Security Symposium (NDSS)*, 2009.

[6] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario, "Automated Classification and Analysis of Internet Malware," in *International Symposium on Research in Attacks, Intrusions and Defenses (RAID)*, 2007.

[7] J. Pfoh, C. Schneider, and C. Eckert, "Leveraging String Kernels for Malware Detection," in *Network and System Security*, 2013.

[8] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data Mining Methods for Detection of New Malicious Executables," in *IEEE Symposium on Security and Privacy*, 2001.

[9] B. Kolosnjaji, A. Zarras, T. Lengyel, G. Webster, and C. Eckert, "Adaptive semantics-aware malware classification," in *Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer, 2016, pp. 419–439.

[10] J. Saxe and K. Berlin, "Deep Neural Network Based Malware Detection Using Two Dimensional Binary Program Features," *arXiv preprint arXiv:1508.03096*, 2015.

[11] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas, "Malware Classification With Recurrent Networks," in *IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2015.

[12] B. Kolosnjaji, A. Zarras, G. Webster, and C. Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer, 2016, pp. 137–149.

[13] M. Pietrek, "An In-Depth Look Into the Win32 Portable Executable File Format," *MSDN magazine*, vol. 17, no. 2, pp. 80–90, 2002.

[14] G. Cybenko, "Approximation by Superpositions of a Sigmoidal Function," *Mathematics of control, signals and systems*, vol. 2, no. 4, pp. 303–314, 1989.

[15] Y. Bengio, "Learning Deep Architectures for AI," *Foundations and trends in Machine Learning*, vol. 2, no. 1, pp. 1–127, 2009.

[16] A. Krizhevsky, I. Sutskever, and G. E. Hinton, "Imagenet Classification With Deep Convolutional Neural Networks," in *Advances in neural information processing systems*, 2012.

[17] Y. Kim, "Convolutional Neural Networks for Sentence Classification," *arXiv preprint arXiv:1408.5882*, 2014.

[18] Y. LeCun and Y. Bengio, "Convolutional Networks for Images, Speech, and Time Series," *The handbook of brain theory and neural networks*, vol. 3361, no. 10, p. 1995, 1995.

[19] "ObjDump UNIX Tool," https://sourceware.org/binutils/docs/binutils/objdump.html.

[20] G. Webster, Z. Hanif, A. Ludwig, T. Lengyel, A. Zarras, and C. Eckert, "SKALD: A Scalable Architecture for Feature Extraction, Multi-User Analysis, and Real-Time Information Sharing," in *International Conference on Information Security (ISC)*, 2016.

[21] A. Mohaisen and O. Alrawi, "Av-Meter: An Evaluation of Antivirus Scans and Labels," in *Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*, 2014.

[22] R. Perdisci and M. C. U, "VAMO: Towards a Fully Automated Malware Clustering Validity Analysis," in *Annual Computer Security Applications Conference (ACSAC)*, 2012.

[23] M. Ester, H.-P. Kriegel, J. Sander, and X. Xu, "A Density-Based Algorithm for Discovering Clusters in Large Spatial Databases With Noise." in *Kdd*, 1996.

[24] K. He, X. Zhang, S. Ren, and J. Sun, "Delving Deep Into Rectifiers: Surpassing Human-Level Performance on Imagenet Classification," in *IEEE International Conference on Computer Vision*, 2015.

[25] X. Glorot and Y. Bengio, "Understanding the Difficulty of Training Deep Feedforward Neural Networks," in *International Conference on Artificial Intelligence and Statistics*, 2010.

[26] Y. Nesterov, *Introductory Lectures on Convex Optimization: A Basic Course*. Springer Science & Business Media, 2013.

[27] "Lasagne Library," http://lasagne.readthedocs.io/en/latest/index.html.

[28] M. Abadi, P. Barham, J. Chen, Z. Chen, A. Davis, J. Dean, M. Devin, S. Ghemawat, G. Irving, M. Isard *et al.*, "Tensorflow: A system for large-scale machine learning," in *Proceedings of the 12th USENIX Symposium on Operating Systems Design and Implementation (OSDI). Savannah, Georgia, USA*, 2016.

[29] "ZDNet Most Popular Downloads," http://downloads.zdnet.com/popular/.

[30] L. Van der Maaten and G. Hinton, "Visualizing Data Using T-Sne," *Journal of Machine Learning Research*, vol. 9, no. 2579-2605, p. 85, 2008.

[31] K. Simonyan, A. Vedaldi, and A. Zisserman, "Deep Inside Convolutional Networks: Visualising Image Classification Models and Saliency Maps," *arXiv preprint arXiv:1312.6034*, 2013.

[32] B. Anderson, C. Storlie, and T. Lane, "Multiple Kernel Learning Clustering With an Application to Malware," in *IEEE International Conference on Data Mining (ICDM)*, 2012.