2018 42nd IEEE International Conference on Computer Software & Applications

Towards Light-weight Deep Learning based Malware Detection

Zeliang Kan[1], Haoyu Wang[1,2*], Guoai Xu[1*], Yao Guo[3,4], Xiangqun Chen[3,4]
[1] Beijing University of Posts and Telecommunications, Beijing, China, 100876
[2] Beijing Key Lab of Intelligent Telecommunication Software and Multimedia
[3] Key Laboratory of High-Confidence Software Technologies (Ministry of Education)
[4] School of Electronics Engineering and Computer Science, Peking University, Beijing, China, 100871
Email: {kanzeliang945, haoyuwang, xga}@bupt.edu.cn, {yaoguo, cherry}@pku.edu.cn

***Abstract*—The explosive amount of malware continues threating the security of operating systems and networks. Traditional malware detection approaches fail to meet the requirements of detecting polymorphic and new samples. Existing neural network based detection approaches performs better, but consuming much more time in both feature extraction and training. In this paper, we propose a light-weight PC malware detection system which is based on deep convolutional neural network (CNN). The raw inputs of our system are sequences of grouped instructions, which were generated by our Instruction Analyzer in according to different functionalities of the instructions. The network will automatically learn features of malware from the grouped instruction sequences. The experiment results suggest that in a large dataset which contains roughly 70,000 samples, our detection system can achieve an overall accuracy of 95%. The training time of our system with single convolutional layer was only about 10 hours, which is one order of magnitude less than traditional methods.**

***Index Terms*—malware detection, deep learning, machine learning, neural network, Windows platform**

I. Introduction

The proliferation of malware has presented a long-lasting and serious threat to the security of computer systems and internet. For example, the well-known WannaCry ransomware attack [15] has affected millions of devices and caused billions of dollars damage. The number of malware has increased greatly every year, and it is reported that every 4.6 seconds a new malware specimen emerged in 2017 [9]. Windows is the predominant platform for malware, and covers more than 99% of the malware variants [41].

A wide range of malware detection approaches have been proposed by many researchers and anti-virus companies. Signature based approach [34], [38] is widely used in malware detection, in which the rules are extracted from the binary program that can be obtained without running it to represent the malware samples uniquely. However, the malwares usually disguise themselves using various evasion techniques [26], thus the zero-day malware cannot be detected by traditional static feature based approaches due to their polymorphic and/or metamorphic nature. Besides, the manual heuristic inspection of malware analysis to generate the rules is no longer considered effective and efficient compared against the high spreading rate of malware. As a result, dynamic approaches [42], [43] are proposed to cope with malware evasion techniques, in which the runtime behaviors (e.g., system calls, API sequences) are monitored dynamically. However, dynamic analysis faces the inherent challenges such as high performance overhead [42], low execution path coverage and high false positives [43], which is not easily scale to large numbers of samples.

Deep learning has been proven to have a good effect on images classification and natural language processing [12], [23], [37]. It automatically learns high-level representation of data by constructing a deep architecture. Hence, deep learning based technique is considered a profound solution in malware detection, and some recent studies [20], [31] have applied neural networks to malware detection.

For example, the most recent work [31] proposed to learn a malware detection system from raw bytes of entire executable files, which shows promising detection results. However, it introduces greatly performance overhead to learn malware detection model from raw bytes, which is a sequence classification problem on the order of two million time steps that contains too much redundant information. All the raw bytes of the executable file, including resource data such as pictures and font, are used as training data, which could be potentially eliminated. Therefore, they chose a big convolution filter and long stride. For the work which depends on raw byte data, there are usually two main ways to improve the time efficiency. One is to select the raw byte data of a specific segment as the training data, thereby reducing the dimension of input vectors. The other is to increase the size of convolution filter and the length of stride, which may increase the training loss.

Consequently, we face the research question: ***How to reduce the redundant information in the training data as much as possible while ensuring the accuracy of the deep learning malware detection system?***

To maximize and complete the extraction of malware features from low-level data, and inspired by the byte n-gram method [27], [32], [34], in this paper, we propose a light-weight deep learning based malware detection approach, which relies on the disassembled instructions as the raw data to be learned. The features could be automatically learned from low-level data, and hence eliminate the limitations in-

* co-corresponding authors

0730-3157/18/$31.00 ©2018 IEEE
DOI 10.1109/COMPSAC.2018.00092

troduced by manually labeling features [38]. Furthermore, to achieve computationally efficient malware detection, we perform instruction analysis on the decompiled code to classify assembled instructions into different groups, and our malware detection system is learnt based on the abstraction of the grouped instructions. Hence, it could greatly reduce the complexity of the classification problem, so as to realize the dimension reduction of embedding layer. Moreover, the training and testing time of our approach linearly proportional to the number of malware examples.

We have applied our malware detection system to a dataset of around 70,000 samples, and the experiment results suggest that, our detection system could achieve an accuracy of 95%, and the training time of our system with single convolutional layer is only about 10 hours. We have released all the dataset and experiment results to the research community at:

https://github.com/deep-learning-malware/Dataset

II. Related Work

A. Traditional Malware Detection

Traditional malware detection methods mainly include static analysis [36], [38], [39], dynamic analysis [8], [16], [28], [42], [43] and feature-based machine learning methods [17], [22], [33], [40]. In general, static or dynamic methods are used to extract high-level features (e.g., API calls, strings, commands and behavior logs in sandbox, etc.) from original samples, and then various machine learning techniques (e.g., SVM, Naive Bayes, KNN, etc.) are used to explore representative features that can distinguish benign and malicious programs.

Schultz et al. [36] statically extracted features such as DLL list and API calls to represent information contained within each binary. Then Naive Bayes method was applied to detect previously unknown malicious code. Similarly, Kolter et al. [22] used features like DLLs and API calls as the basis and examined the performance of different classifiers such as Naive Bayes, SVM and Decision Tree on the same dataset. Firdausi et al. [17] automatically generated samples' behavior reports from an emulated (sandbox) environment. And they used different kind of classifiers in this research.

Most of the aforementioned approaches used high-level hand-designed or artificial selected features, which require a significant amount of domain knowledge for feature extraction. Malware developers can easily evade these feature-based methods through techniques such as string encryption, polymorphism, code obfuscation [38] and virtual machine environment monitoring [11], etc.

B. Byte N-gram Approaches

To overcome the limitation of manually selected features, various approaches [30], [32], [35] proposed to build the malware detection system based on low-level data (e.g., raw byte sequences, PE-heads, etc). Byte n-gram is one of the more popular approaches, which extracts features from the context of inputs. Although byte n-gram approach shows promising malware detection results in previous work [25], [30], the process of n-gram feature selection for a large dataset consumes huge time and space resources due to the large amount of different n-grams [19], which limits the application of the n-gram method to large datasets.

Pektas et al. [30] proposed to classify malware instances by using n-gram features of its disassembled code. They used only one vector for classification to reduce the dimension of the n-gram space. However, due to the scalability limitation of n-gram, they were only able to experiment on a dataset of 1,056 samples. Lin et al. proposed to [25] combine dynamic analysis and n-gram method. They extracted n-gram features from behavior logs and built a SVM classifier for malware classification on a dataset contained 4,288 samples. In order to address the challenge of n-gram's huge time consumption, they proposed a two-stage feature reduction method, which they believed could be used to reduce the time cost of re-training. However, it is also time-consuming to monitor the runtime behavior of malware, and previous work suggested that the malware developers tried to evade the detection [21].

C. Neural Network Based Malware Detection

Due to the success of deep learning in research fields such as images classification and natural language processing [12], [23], [37], some recent work [20], [29] proposed to apply neural networks to code analysis and malware detection. Deep neural network is able to automatically learn high-level representation of data by constructing a deep architecture.

Huang et al. [20] extracted data of malicious and benign samples from dynamic analysis, and used up to four hidden layers of feedforward neural network[1] to detect. Regardless of the time cost of dynamic analysis, they were mainly focusing on evaluating multi-task learning ideas. Pascanu et al. [29] used recurrent networks for learning from system call sequences, in order to construct a "language model" for malware. The high-level events like API calls formed the input sequences of their model. They tested Long Short-Term Memory and Gated Recurrent Units and achieved good classification performance. However, they did not experiment with deep learning approaches.

All of the aforementioned approaches used high-level features like API calls and behavior logs as input of the neural network. As we mentioned before, high-level data of the malware may have been modified by authors, and the low-level data keeps the sample's original appearance. So our key idea is to use the low-level features as learning materials for neural networks. Thus, some recent work [31], [35] tried to build neural network based on the raw byte data. Saxe and Berlin et al. [35] used a histogram of byte entropy values as features, which combined ASCII string lengths, PE imports, and other meta-data together. But in data processing, they discarded most information about the actual content of the binary, because they wanted to create a fixed length feature vector as input to the network. Similarly, Raff et al. [31] used raw byte sequences as inputs of deep learning. In their work, they noted that detection from raw bytes presents a sequence

[1] https://en.wikipedia.org/wiki/Feedforward_neural_network

problem with over two million time steps. But actually, their work is more focused on how to get the deep learning system to receive more raw data and longer sequences, which may contain lots of redundant information (e.g. resource files).

Combined with deep learning, we try to build a lightweight malware detection system based on CNN, which learns to detect the characteristics of malware automatically from instruction sequences. Compared with previous approaches, our approach has several advantages. First, in the selection of training data, we try to keep the context of samples as much as possible rather than discarding the partial information of samples. This allows for a greater degree of storage of the integrity of the execution process. Also, compared with deep learning approaches based on high-level hand-designed features, our detection system is more efficient in the time of training data extraction. Furthermore, take advantage of instruction grouping analysis, the dimension of the embedding layer in our neural network is greatly reduced, which allows the training process to be more effective. Moreover, the features extracted by our system may be different from those features extracted manually, which may be understood by the machine only, and makes it difficult for malware authors to create evasion techniques that could bypass our detection system.

III. System Architecture

In this section, we introduce the overall architecture of our system. Our deep learning based malware detection system is performed on low-level features extracted from original executable files. As shown in Figure 1, our system consists of two major components: *Instruction Analyzer* and *Convolutional Neural Network (CNN) Based Classifier*.

- **Instruction Analyzer**: This component is used for data preprocessing, including IDA disassembler [2] and Instruction Processor. It is worth mentioning that, while dealing with instructions, we grouped all the instruction sets of the X86 platform by their functionalities and attributions. Then we generated a dictionary with 205 key-value pairs. After the step of instruction analysis, each instruction in the original file is then mapped to an instruction group. (See Section 3.1 for details.)
- **CNN-based classifier**: We used convolutional neural network (CNN) [24] in this component. And in this work, the inputs of neural network are instruction's group number sequences. A deep learning architecture using CNN with optional multi-convolutional layers is designed to perform unsupervised feature learning, fine-tuning, and thus malware detection. (See Section 3.2 for details.)

A. Instruction Analyzer

In our system, we disassemble Windows executable files firstly, transform it into .asm files, and then extract instruction sequences from .asm files. A Windows software is often an .exe file, which is a common filename extension denoting an executable file. Apart from the executable program, many .exe files contain other components called resources, such as bitmap graphics and icons which the executable program may use for its graphical user interface [13]. We use Interactive Disassembler (IDA) to disassemble each executable file and save as an .asm file. The .asm file contains instructions and each instruction consists of operation codes and operands [14].

Considering that there are many instruction sets on the x86 platform, and many of them have similar functionalities, such as basic move operation, there are operation codes such as MOV, MOVQ, MOVD and so on. But they only exist to distinguish the length and type of operands. If each instruction with similar functionality is treated as a separate class, the training space will be very large in the training process. Thus, for the reduction of dimensionality, we choose to divide instructions into groups according to their operation code's functionality and property.

Here a dictionary file is used to record the group number of instructions. Then we extracted all the instruction sequences from the .asm file. These sequences were then mapped into sequences of instruction group number, which consists of many double-hexadecimal-digit (0x00-0xff). In this work, the dictionary we used contains actually 206 groups of instruction sets[2]. After the pre-processing phase, every executable sample is represented as a double-hexadecimal-digit vector.

It is important to note that we build the dictionary based on all the operation codes in Intel® 64 and IA-32 Architectures Software Developers Manual [18]. It consists of most the assembly instructions on Windows platform. We have collected all of the 663 operation codes listed in this manual. Then we divided instructions into 205 groups based on their operation codes functionality and property. Other instructions based on unrecorded operation codes are considered to be the same group, which is group 206. In the subsequent training process, instructions of the same group are treated as same.

In order to better illustrate the benefits of instruction grouping analysis, we compared the results of instruction grouped situation with ungrouped one in the follow-up experiments. It turned out that when the same number of convolutional layers were used in our system, the grouped situation got the better result. We think this may because many instructions have similar functionality on x86 platform. Neural network may amplify this subtle difference in the training process, treating them totally different in the ungrouped case.

Figure 2 shows an example of instruction analysis in our work. Firstly, sample.exe is compiled by IDA to the sample.asm file, which consists of numerous assembly instructions. Taking instruction "**.text:0040131A mov [esp+8+Format], offset Format ; "hello world"**" as an example, this is a basic move operation that writes the string "hello world" to the specified word unit. We recorded the group number in the dictionary. Then all instructions for the MOV and similar operations are attributed to the same group. Then operation code 'mov' is preserved in sample.tmp but the rest part of this instruction is discarded. Also, 'shr eax,4' and 'shl eax,4' are all shift operation instructions, we divided them

[2]The dictionary is released at: *https://github.com/deep-learning-malware/Dataset*.

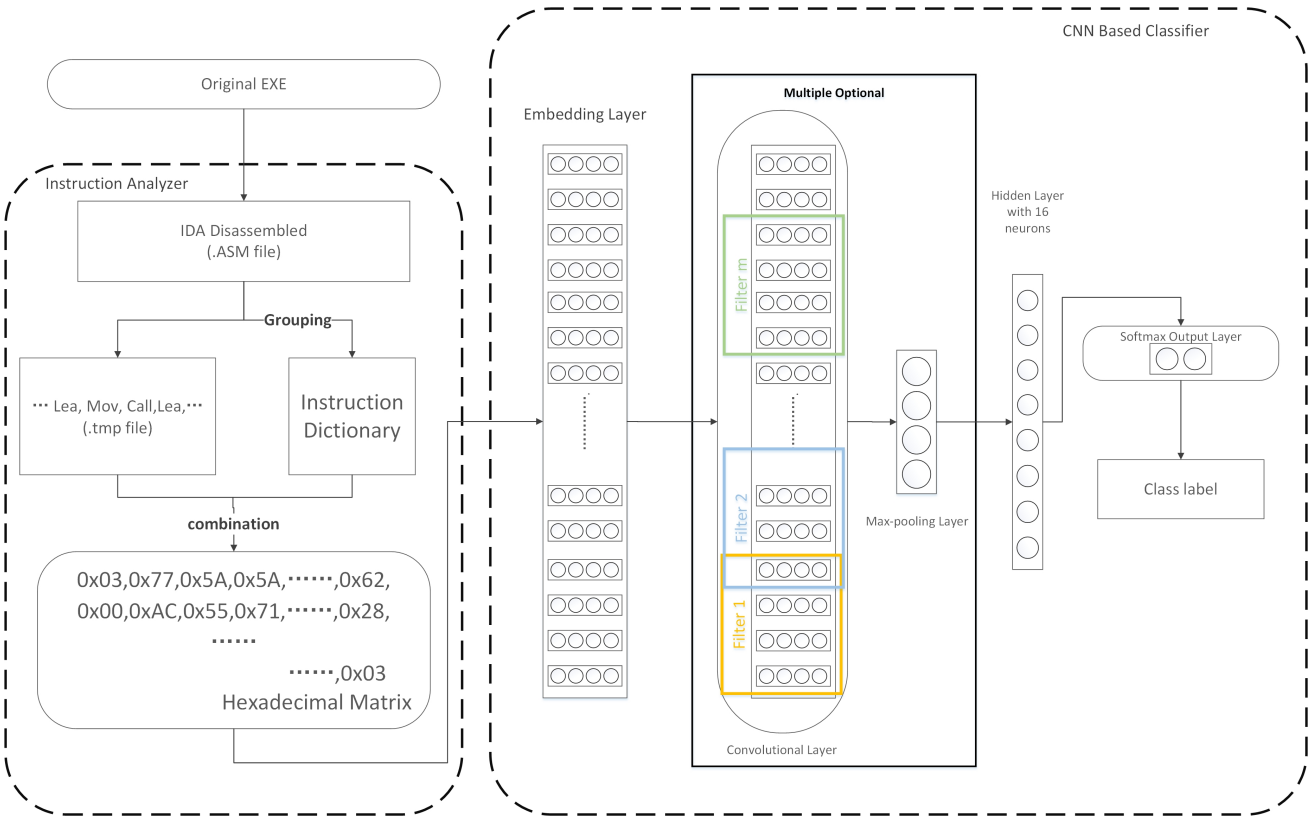

Fig. 1. The overall architecture of our system.

into the same group too. We saved all the operation codes from sample.asm. The previous dictionary were used to generate a double-hexadecimal-digit vector.

B. CNN-Based Classifier

Previous work [44] has suggested that the convolution neural network has a good effect in recognizing the displacement, scaling and other distortions of the image. Similarly, the malicious features of malware, for example, might be placed in any position of the software. Our goal is to find out the existence of malicious feature but not to detect the exact address of it. To better capture such high level location invariance, we choose to use a convolution network architecture.

Fig. 1 illustrates the high-level architecture of our neural network. We chose group number sequences as the raw input of the neural network. The original input firstly passed through an embedding layer, where each byte converted to a feature vector. Then these vectors went into the convolutional part. The convolutional part consists of convolutional layer and max-pooling layer. The activation function we chose in our system is Rectified Linear activation function (ReLU). In convolutional layer, features were extracted from these feature vectors. In fact, the features extracted from the first convolutional layer can be seen as n-grams of instructions. (Here the value of n depends on the width of the convolution kernel.) After each convolutional layer we use max-pooling to reduce the dimensionality of features. Also, in our detection system, the layer of convolutional part can be multiple. Furthermore, we use a hidden fully connected layer after the last convolutional layer, which allows high-order relationships between the features to be detected [10]. Finally a softmax layer is used to output the label probabilities. It is worth noting that most of parameters in the neural network are updated automatically by back propagation during training.

a) Embedding Layer: The purpose of the embedding layer is to project those instructions with similar functionality and semantic information to close points in the embedding space, and making a distinction between instructions of very different functionalities.

In the embedding layer, we firstly encode each instructions group number as a one hot vector $\boldsymbol{xn}$. Then we use $\boldsymbol{X} = \{\boldsymbol{x1}, \boldsymbol{x2}, \ldots, \boldsymbol{xn}\}$ on behalf of all one hot vectors in a sequence. The index of instruction groups is in the range 1 to 206, which determines each one hot vector to be a 206-element-vector. Here we use $\mathbf{K}$ to represent the length of one hot vector, 206. The original one hot coded vector then multiply by a weight matrix, $\boldsymbol{W_E}$ of size $\mathbf{D*K}$, which aimed to reduce the length of new feature vector.

We use Formula 1 to represent this operation

$$\boldsymbol{Y} = \boldsymbol{X} * \boldsymbol{W_E} \tag{1}$$

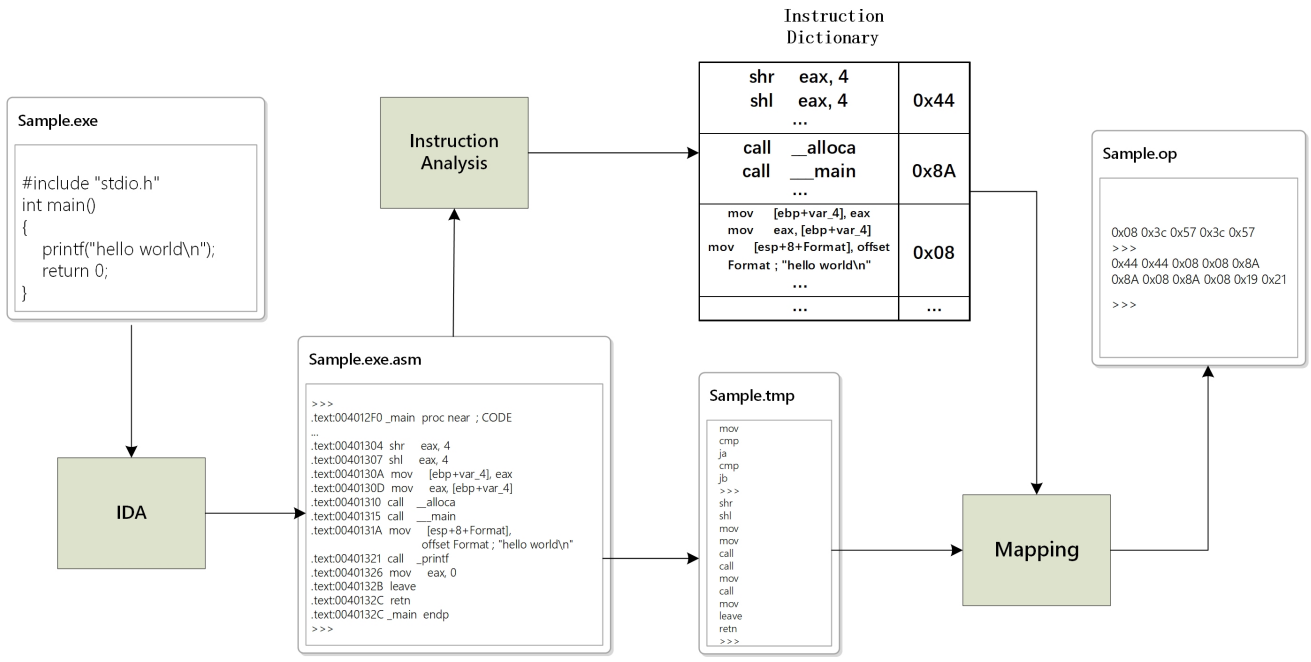

Fig. 2. An example of Instruction Analyzer

We then get a matrix, **Y**, which represented the whole program. The size of **Y** is **n*D**, where **D** is the dimensionality of the embedding space. After the embedding layer, instructions with similar functionality and semantic information will be projected to close points in the embedding space, and instructions of very different functionalities will be projected to distant points. It is worth noting that the **D** is an important parameter associated with the classification results. The selection of dimensions is related to specific datasets. In general, the more training data, the higher the dimension required [10].

b) Convolutional Layers: The purpose of convolution is to extract different features of input. More convolutional layers of network can iterate extracting more complex features from low-level features. In our proposed architecture, there can be multiple convolutional layers being used. These convolutional layers are numbered from 1 to **L**. The first convolutional layer's input is the embedding matrix **Y**, of size **n*D**. We use $\boldsymbol{m_l}$ to represent the number of convolution filters in the **l' th** convolutional layer. In the first convolutional layer, the size of filter is $\boldsymbol{s_1 * D}$, where $\boldsymbol{s_1}$ is the length of instruction sequence that can be detected. The deeper convolutional layers receive the output of the previous convolutional layer as input. And the size of filter in deeper convolutional layer is $\boldsymbol{s_l * m_{l-1}}$. After convoluting **Y** with convolution filters, we use the activation function to preserve features and remove redundancies in data. Here we choose the rectified linear activation function (ReLU),

$$\boldsymbol{ReLU = max(0, x)} \tag{2}$$

As the Formula 3 shows, each of the $\boldsymbol{m_l}$ convolutional filters produces an activation map $\boldsymbol{a_{l,m}}$ of size n * 1, where $\boldsymbol{W_{l,m}}$ and $\boldsymbol{b_{l,m}}$ are the weight and bias parameters of the **m' th** convolutional filter of convolution layer **l**. Conv represents the mathematical operation of convolution of the filter with the input matrix **Y** (Take the first Convolutional layer as an example).

$$\boldsymbol{a_{l,m} = ReLU(Conv(Y)_{W_{l,m}, b_{l,m}})} \tag{3}$$

All activation maps that generated by $\boldsymbol{m_l}$ convolution filters can be combined to get an output matrix, $\boldsymbol{A_l}$ of size $\boldsymbol{n * m_l}$. For the last convolutional layer, we use $\boldsymbol{A_L}$ to represent its output matrix. Following the last convolutional layer is a max-pooling layer. The max-pooling operation is to generate a vector which consists of maximum value in each activation map in $\boldsymbol{A_L}$. The max-pooling layer is introduced for two purposes. One is keeping the invariance of samples. The other is to get fixed length output and reduce the input size of the next layer. After the max-pooling layer we got a vector $\boldsymbol{\beta}$ of length $\boldsymbol{m_L}$ as follows,

$$\boldsymbol{\beta = (max(a_{L,1}) | max(a_{L,2}) | \ldots | max(a_{L,m}))} \tag{4}$$

c) Hidden Layer and Output Layer: Fig.3 illustrates the last part of the neural network, which consists of a fully-connected hidden layer and a softmax output layer. After the max-pooling operation, we got a vector $\boldsymbol{\beta}$ of length $\boldsymbol{m_L}$. We then pass the vector to a fully-connected hidden layer, where relationships between the features extracted by the convolutional layer are able to be detected [10]. ReLU is still used as the activation function in the hidden layer. We use $\boldsymbol{W_h}$ and $\boldsymbol{b_h}$ to stand for the weight and bias of the fully-connected

hidden layer respectively. This hidden layer can be expressed as Formula 5,

$$\boldsymbol{H_L = ReLU(W_h\beta + b_h)} \tag{5}$$

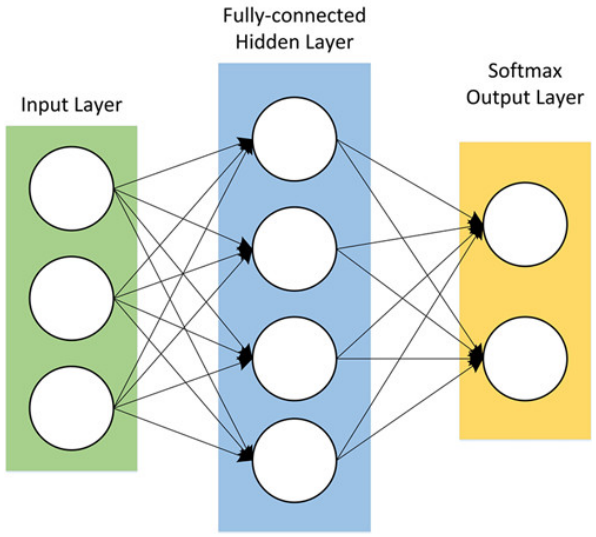

Fig. 3. The hidden layer and softmax output layer.

The output vector $\boldsymbol{H_L}$ of the hidden layer is then passed to a softmax layer, where the vector is normalized. The output vectors dimension of the softmax layer is based on the number of classes. In this work, we focus on determine whether the software is malicious or benign. As a result, output of the softmax layer is a two-dimensional vector. We cite it as **h**.

$$h_\Theta(z) = \frac{1}{e^{\Theta_1^T z+b_1} + e^{\Theta_2^T z+b_2}} \begin{bmatrix} e^{\Theta_1^T z+b_1} \\ e^{\Theta_2^T z+b_2} \end{bmatrix} \tag{6}$$

Each element of **h** shows the probability of the current sample belonging to each class. The probability is computed as follows,

$$\begin{aligned} P(o=1) &= \frac{e^{\Theta_1^T z+b_1}}{e^{\Theta_1^T z+b_1} + e^{\Theta_2^T z+b_2}} \\ &= \frac{1}{1+e^{(\theta_2-\theta_1)^T z+(b_2-b_1)}} \end{aligned} \tag{7}$$

Where $\boldsymbol{\theta_i}$ and $\boldsymbol{b_i}$ represent the parameters of the classifier for class $\mathbf{o}\in\{\mathbf{1,2}\}$. The probability that the origin program is malware denoted as Formula 7 (assume class 1 is malware).

d) Cost Function: The cost function of softmax regression is shown as Formula 8,

$$\boldsymbol{C = -\frac{1}{b}\sum_{i=1}^{n}\sum_{o=1}^{O} 1\{label_i = o\} logp(label_i = o|H_i))} \tag{8}$$

Here $1\{\cdot\}$ is the indicator function, so that $1\{true\} = 1$, and $1\{false\} = 0$. $\boldsymbol{H_i}$ is the $\boldsymbol{i'th}$ sample's output of the hidden layer, and $\boldsymbol{label_i}$ is the correct label of the $\boldsymbol{i'th}$ sample. We use δ to stand for parameters of the neural network(i.e. weights and bias). To minimize the cost, during the training process, with the standard implementation of gradient descent, we would perform the update on each iteration as formula 9,

$$\boldsymbol{\delta := \delta - \alpha\frac{\partial C}{\partial \delta}} \tag{9}$$

The α in formula 9 is the learning rate. During training, the network is repeatedly presented with batches of training samples in randomized order until parameters converge [10].

IV. Evaluation

A. Implementation

We implemented the of CNN based malware detection system on Torch [4], a scientific computing framework for machine learning. The system is implemented in Lua language, with more than 3000 lines of code. The whole neural network computing supports GPU acceleration. The batch size of our model is 16 samples per batch. And the learning rate is 1e-2 for 10 epochs, with RMSProp optimized. All network weights and biases were randomly initialized using the default Torch initialization. We built our network and trained the model on an NVIDIA Titan X GPU. After completing the training process, the trained model will be saved to test the testing set and get the classification label.

We used 10-fold cross validation on the training process, in order to set parameters empirically, such as the number of convolutional filters and the dimension of the embedding space. In this work, we used an 8-dimensional embedding space, 64 convolutional filters of length 8, and 16 neurons in the hidden fully connected layer.

B. Dataset

We evaluate our system on three different datasets. The malware samples in the first two datasets were downloaded from VirusShare [5], and all of the trusted samples were collected from Tencent application store [3] and Baidu application store [1]. All the samples have been verified by VirusTotal [6] to make sure that malware samples were really harmful and trusted samples were highly probable to be malware free.

- Dataset A. This dataset is made up of 4,994 samples. There are 2,859 malware samples and 2,135 trusted samples in this dataset.
- Dataset B. It contains 11,130 samples, in which 6,066 samples are malware and 5,064 samples are trusted.
- Dataset C, which is the largest dataset in our experiment. It has a total number of 71,584 applications, in which all malware samples were provided by National Internet Emergency Center [7]. After discarding empty processed files, the dataset contains 41,346 malware samples and 28,653 trusted samples. It is worth noting that this dataset may have overlapping with the Dataset A and Dataset B, which we did not verify.

For comparison, we evaluate our system on different dataset respectively. In order to get a relatively good result in malware detection, we used single convolutional layer on the first

TABLE I
THE EFFECT OF INSTRUCTION GROUPING ANALYSIS ON TRAINING DATA.

	variety of word	word length	entropy of sequence	size
File with grouped data	41.2	2	3.47	35.8KB
File with ungrouped data	76.7	3	6.59	47.4KB

dataset and five convolutional layers on other two datasets. The reason why we chose such a structure is that too many layers may lead to overfitting on Dataset A, which has fewer samples. On the contrary, we have a relatively large number of samples in other two datasets. The malicious features of samples are more diverse, and multi-layer convolution can more thoroughly fit these features.

C. The effect of instruction grouping

To measure the effect of instruction grouping, we randomly selected 50 samples and preprocessed them. As shown in Table I, in the grouped case, the average word type in each sample is 41, compared with the ungrouped case of 76. Similarly, because of the use of hexadecimal numbers to represent samples, ungrouped data needs to be represented by three digits, but the grouped requires only two digits. In addition, we calculate the average entropy of each sample. There are more entropy in the ungrouped file. The average file size of grouped case is also smaller than that of ungrouped case.

D. Malware Detection Results

For each dataset, we split malware and trusted samples into 90% as the training and validation set, and the remaining 10% as the testing set. To evaluate the effectiveness of our detection system (which we refer as **EzNet**), we used standard measures of the classification accuracy, precision, recall and F1-score. The training set, validation set and the test set are completely uncrossed. To better reflect the effect and detection capability of our system, we evaluate our malware detection system on different dataset respectively.

Furthermore, in order to illustrate the benefits of instruction grouping analysis on the testing results, we also conducted experiments on ungrouped conditions. It should be noted that the following experimental results, if not specified, are the results of the instruction grouped situation.

a) Dataset A: We compared our system with the traditional bytes n-gram methods (n=2 and n=3) on Dataset A. We chose 6 classical algorithms from the scikit-learn of python 2.7, they are Logistic Regression (LR), Random Forest (RF), Decision Tree Regression (DT), Extra Trees (ET), Gradient Boosting(GB), and SVM (kernel = poly).

As shown in Table II, compared to 3-gram based methods, our model with one convolutional layer is seen the second best result of accuracy in the first experiment. Our system achieve an accuracy of 0.992 and F1-score of 0.991. The 3-gram based RF method got the best result of accuracy, which is 0.993. There is a very small gap between the accuracy of our system and the best result. In term of precision rate, our system got the best result of 1.0.

TABLE II
EXPERIMENT RESULTS ON DIFFERENT DATASETS (THE BEST RESULTS ARE IN BOLD.)

Dataset	Methods	Feature	Accuracy	Precision	Recall	F1-score
Dataset A (2859 malware and 2135 trusted samples)	EzNet	grouped	0.992	**1.0**	0.981	0.991
		ungrouped	0.988	0.986	0.986	0.986
	LR	2-gram	0.981	0.977	0.992	0.984
		3-gram	0.968	0.961	0.987	0.974
	RF	2-gram	0.992	0.989	0.998	**0.994**
		3-gram	**0.993**	0.989	0.998	0.993
	DT	2-gram	0.989	0.991	0.991	0.991
		3-gram	0.982	0.987	0.984	0.985
	ET	2-gram	0.992	0.988	0.998	0.993
		3-gram	0.991	0.987	0.998	0.993
	GB	2-gram	0.991	0.990	0.996	0.993
		3-gram	0.986	0.981	0.996	0.989
	SVM	2-gram	0.971	0.954	**1.0**	0.977
		3-gram	0.965	0.946	**1.0**	0.972
Dataset B (6066 malware and 5064 trusted samples)	EzNet	grouped	**0.990**	**0.994**	0.984	**0.989**
		ungrouped	0.981	0.990	0.968	0.979
	LR	3-gram	0.902	0.874	0.966	0.918
	RF	3-gram	0.983	0.979	0.991	0.985
	DT	3-gram	0.975	0.978	0.978	0.978
	ET	3-gram	0.982	0.977	0.990	0.983
	GB	3-gram	0.963	0.955	0.980	0.967
	SVM	3-gram	0.971	0.954	**0.996**	0.975
Dataset C (41346 malware and 28653 trusted samples)	EzNet	grouped	**0.96**	**0.95**	**0.95**	**0.95**
		ungrouped	0.93	0.92	0.91	0.93

It is worth noting that, most of the 2-gram based methods achieved better results than 3-gram on the Dataset A, although the gap is very small. This may be because Dataset A has less samples and the differences between samples are not obvious enough. The 2-gram methods are good enough to fit the features. When it comes to comparing the accuracy between 2-gram based methods and our approach, we can see that our approach achieved the best result of accuracy, so did some other 2-gram based traditional methods. The best F1-score is achieved by the 2-gram based RF method.

In general, on Dataset A, our method is better than most of the traditional byte n-gram methods on testing results. When n=2, RF and ET methods got the same results as our model on testing set. Although the same results have been achieved, our approach is better in terms of training time and computational consumption. When the number of samples in dataset gets larger, the time and computational resource consumption of byte n-gram based methods will be magnified exponentially, which will be illustrated in section 4.4.

b) Dataset B: In the second experiment, we only conducted 3-gram malware classifications on the Dataset B. As shown in Table II, our system has achieved the best results in terms of accuracy, precision rate and F1-score. The figure of them were 0.990, 0.994 and 0.989 respectively. The SVM approach had the best result on the recall rate, with the figure of 0.996, but it has low accuracy and F1-score.

Combined with the results of the first experiment, we can find that our system achieved high accuracy and F1-score in both Dataset A and Dataset B. Also, among these six

traditional methods, the RF method obtained the best results.

c) Dataset C: Then we did the third experiment on our Dataset C, which contains 41,346 malware samples and 28,653 trusted samples. Due to the computational cost and time consumption of traditional method, it is hard to make comparison between traditional n-gram based approaches and our model on the Dataset C. As shown in Table II, the accuracy on Dataset C is 0.96 and F1-score is 0.95, with 4135 malware samples and 2866 trusted samples being assigned to the testing set. In this part of the experiment, time and computational consumption are relatively few and acceptable, which will be explained in section 4.4.

Experiments on these three datasets suggest that our malware detection system could achieve high accuracy. Although 2-gram based RF method has achieved better F1-score on Dataset A, the limitations of traditional methods become more obvious as the dataset increased. For the traditional n-gram methods, it is difficult to carry out on the Dataset C, because the n-gram feature selection for a large dataset consumes huge time and space resources due to the large amount of different n-grams [19].

E. Performance Evaluation

For the six aforementioned traditional methods, the RF algorithm achieved the best results on the first two datasets. Thus, in this part, we compared the performance between traditional 3-gram based RF method and our system in time consumption. Since preprocessing phases (e.g., extracting ASM and instructions, etc) are same in malware detection, we only compared training time and testing time of RF method and our model.

As shown in Table III, it took 2153.46 seconds for training and only 0.85 seconds for testing on Dataset A, compared to 4782.54 seconds and 21.63 seconds for 3-gram Random Forest. The n-gram based method experiences exponential slow-down with the number of the n-gram features increasing. When it came to the Dataset B, We can see that there is a clear gap between traditional methods and our methods in training time. Our system takes 9810.17 seconds for training and only 5.04 seconds for testing. The traditional RF method used 17513.85 seconds for training, which more than three times longer than the time of our system. The testing time of 3-gram RF method was also much longer than our system.

On Dataset C, our model with one convolutional layer used only 10.39 hours for training and 6.61 seconds for testing, compared to our five-layer system with 19.10 hours for training and 25.89 seconds for testing. We did not compute time consumption of n-gram RF and other n-gram based method on Dataset C due to its predictable huge result. Thus, our model is more reasonable in training time. The training and testing time of our approach linearly proportional to the number of malware examples. The time consumption is only related to the number of samples and convolutional layers. But when it comes to traditional n-gram based approaches, the time of extracting eigenvectors increases exponentially with the number of sample sizes increased, which limited the efficiency of these kind of methods.

F. Benifits of Instruction Grouping Analysis

Comparing the grouped situation with the ungrouped one, we can find that the grouped situation got better results on all of datasets. As shown in Table II, the ungrouped situation achieved the accuracy of 0.988 and the F1-score of 0.986 on Dataset A. Although the gap between these two groups is small, the time consumption of training process is much different. As Table IV shows, our system with single layer and grouped data used 2153.46 seconds for training on Dataset A, compared to the system with ungrouped data used around 4 times longer time for training. The extra time is not worth it.

The same results appear on dataset B. It can be seen that the gap widened, the ungrouped situation achieved the accuracy of 0.981 and the F1-score of 0.979, compared with results of 0.990 and 0.989 of experiments with the grouped instructions. Also, on the Dataset B, the time consumption of ungrouped situation was more than 2 times larger than the grouped one.

In the same way, we also conducted experiments on Dataset C. When using the five-layer convolution, the experiment with grouped instructions still achieved better results. The gap in training time became even greater when figure were analyzed for Dataset C. The ungrouped one used 32.17 hours for training, but the grouped situation took only half the time.

By analyzing the testing time, we can see that in the same dataset, the time consumption of the two methods is close. Because when the structure of neural network is determined, the testing time is mostly dependent on the number of samples.

Combined with the previous experimental results, we can see that the grouped situation has obtained better experimental results on every dataset. We think this may because many instructions have similar functionality on x86 platform. Neural network may amplify this subtle difference in the training process, thinking these instruction are totally different.

In general, the group with instruction grouping analysis has achieved better accuracy and F1-score. Therefore, we believe that the instruction grouping analysis can save the training time, and will not reduce the accuracy of the detection.

V. The Performance of Real-time Detection

In this section, we compared the detection time and resources consumption of our system.

a) Time Consumption: First, we compared the detection time of different sizes of file in our system with traditional RF methods. We chose five sets of .op files with different size. For each group of samples, we conducted 10 experiments and took the average value as final results. As shown in Fig 4, in our system, it is obvious that time consumption of each file increased linearly with the increasing of the number of convolutional layers. And the increasing rate is very small (The line of vertices of each column has a small slope.). For the traditional RF method, the detection time is an order of magnitude more than our system.

TABLE III
THE TIME CONSUMPTION OF DIFFERENT METHODS ON DIFFERENT DATASETS

Method	Training Time(second)			Testing Time(second)		
	EzNet(1-layer)	EzNet(5-layers)	RF(3-gram)	EzNet(1-layer)	EzNet(5-layers)	RF(3-gram)
Dataset A	2,153.46	3,794.79	4,782.54	0.85	2.16	16.69
Dataset B	5,855.28	9,810.17	17,513.85	1.78	5.04	44.51
Dataset C	37,425.86(10.39 hours)	68,756.91(19.10 hours)	-	6.61	25.90	-

TABLE IV
THE TIME CONSUMPTION OF DIFFERENT METHODS ON DIFFERENT DATASETS

Method	Training Time(second)		Testing Time(second)	
	EzNet(grouped)	EzNet(ungrouped)	EzNet(grouped)	EzNet(ungrouped)
Dataset A with single layer	2,153.46	8,135.99	0.85	0.84
Dataset B with 5 layers	9,810.17	21,815.32	5.04	5.72
Dataset C with 5 layers	68,756.91(19.10 hours)	115,812.73(32.17 hours)	25.90	29.86

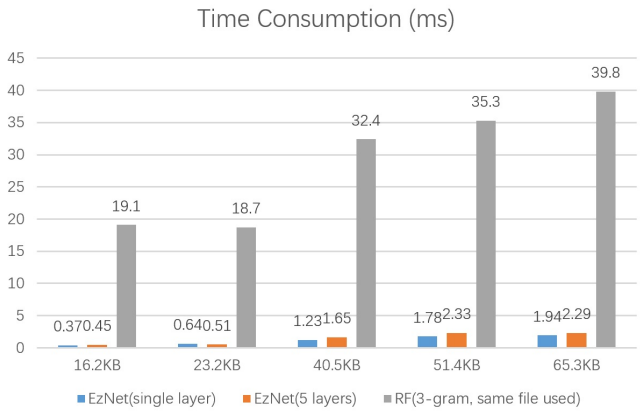

Fig. 4. Comparison of testing time between different size files.

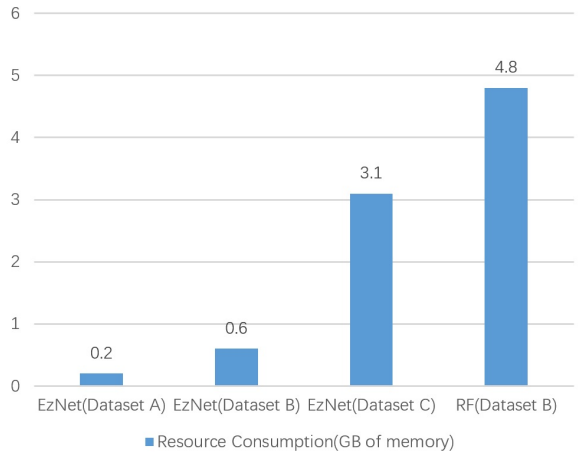

Fig. 5. Resource Consumption (GB of memory)

b) Resource Comparison: As shown in Fig 5, the resource consumption of our system is not high in training process on all of the datasets. With 5 convolutional layers, our system occupied around 600MB of memory during the training process of Dataset B. In the training phase of Dataset C, the occupation is about 3GB of memory. The resource consumption of our system is linearly dependent on the size of dataset. But for traditional methods, the consumption of hardware resources is 8 times more than our system during the training phase of Dataset B, which means our system also more efficient in terms of the usage of hardware resources.

DISCUSSION

Due to differences of datasets and platform configuration, we do not compare our approach with other deep learning systems. Also, shelled softwares may affect our testing system. We did not specifically check the number of shelled samples in datasets. There are more than 70,000 samples in Dataset C, and it is likely that there would be some shelled samples. Some unpacking approaches could be used to build a packing-resilient malware detection approach. In addition, the structure of the neural network in our system is simpler than other work on images and semantics. If the neural network becomes deeper and more complex, for example, more hidden layers used in architecture, we believe the accuracy should be higher.

CONCLUSION

In this work, we design a light-wighted CNN based malware detection system. It is based on disassembled instructions extracted from sample files, which has been proven to perform well on different datasets. Also, benefited from instruction grouping analysis, the accuracy and efficiency of our system are all improved. Compared to other work, our detection system is more lightweight both in aspects of the training data extraction and training time.

VI. Acknowledgement

This work is supported by the science and technology project of State Grid Corporation of China: "Research on Key Technologies of Security Threat Analysis and Monitoring for Power Mobile Terminals" (Grant No. SGRIXTKJ[2017]265).

References

[1] Baidu online application store. http://rj.baidu.com/. [Online; accessed 10-October-2017].

[2] Ida. https://www.hex-rays.com/products/ida/. [Online; accessed 21-November-2017].

[3] Tencent online application store. http://sj.qq.com/. [Online; accessed 10-October-2017].

[4] Torch. http://torch.ch/. [Online; accessed 21-November-2017].

[5] Virusshare. https://virusshare.com/. [Online; accessed 10-October-2017].

[6] Virustotal. https://www.virustotal.com/#/home/upload. [Online; accessed 10-October-2017].

[7] National internet emergency center, cncert/cc, 2018. [Institutions; accessed 20-January-2018].

[8] Blake Anderson, Daniel Quist, Joshua Neil, Curtis Storlie, and Terran Lane. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258, 2011.

[9] Ralf Benzmller. Malware trends 2017. [Online; accessed 2017-10-04].

[10] Christopher M Bishop. *Neural networks for pattern recognition*. Oxford university press, 1995.

[11] Xu Chen, Jon Andersen, Z Morley Mao, Michael Bailey, and Jose Nazario. Towards an understanding of anti-virtualization and anti-debugging behavior in modern malware. In *Dependable Systems and Networks With FTCS and DCC, 2008. DSN 2008. IEEE International Conference on*, pages 177–186. IEEE, 2008.

[12] Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

[13] Wikipedia contributors. .exe — wikipedia, the free encyclopedia, 2017. [Online; accessed 20-January-2018].

[14] Wikipedia contributors. Instruction set architecture — wikipedia, the free encyclopedia, 2017. [Online; accessed 20-January-2018].

[15] Wikipedia contributors. Wannacry ransomware attack — wikipedia, the free encyclopedia, 2018. [Online; accessed 20-January-2018].

[16] Artem Dinaburg, Paul Royal, Monirul Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In *Proceedings of the 15th ACM conference on Computer and communications security*, pages 51–62. ACM, 2008.

[17] Ivan Firdausi, Alva Erwin, Anto Satriyo Nugroho, et al. Analysis of machine learning techniques used in behavior-based malware detection. In *Advances in Computing, Control and Telecommunication Technologies (ACT), 2010 Second International Conference on*, pages 201–203. IEEE, 2010.

[18] Part Guide. Intel® 64 and ia-32 architectures software developers manual. *Volume 3A and 3B*, 2, 2011.

[19] Weiwei Hu and Ying Tan. Partitioning based n-gram feature selection for malware classification. In *International Conference on Data Mining and Big Data*, pages 187–195. Springer, 2016.

[20] Wenyi Huang and Jack W Stokes. Mtnet: a multi-task neural network for dynamic malware classification. In *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 399–418. Springer, 2016.

[21] Jung-Uk Joo, Incheol Shin, Tong-Wook Hwang, and Minsoo Kim. The trigger of malicious behaviors in sandbox. 2014.

[22] Jeremy Z Kolter and Marcus A Maloof. Learning to detect malicious executables in the wild. In *Proceedings of the tenth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 470–478. ACM, 2004.

[23] Florian Krebs, Bruno Lubascher, Tobias Moers, Pieter Schaap, and Gerasimos Spanakis. Social emotion mining techniques for facebook posts reaction prediction. *arXiv preprint arXiv:1712.03249*, 2017.

[24] Yann LeCun, Yoshua Bengio, and Geoffrey Hinton. Deep learning. *Nature*, 521(7553):436–444, 2015.

[25] Chih-Ta Lin, Nai-Jian Wang, Han Xiao, and Claudia Eckert. Feature selection and extraction for malware classification. *J. Inf. Sci. Eng.*, 31(3):965–992, 2015.

[26] Jonathan AP Marpaung, Mangal Sain, and Hoon-Jae Lee. 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, 2012.

[27] Mohd Zaki Mas' ud, Shahrin Sahib, Mohd Faizal Abdollah, Siti Rahayu Selamat, and Choo Yun Huoy. A comparative study on feature selection method for n-gram mobile malware detection. *IJ Network Security*, 19(5):727–733, 2017.

[28] Vinod P Nair, Harshit Jain, Yashwant K Golecha, Manoj Singh Gaur, and Vijay Laxmi. Medusa: Metamorphic malware dynamic analysis usingsignature from api. In *Proceedings of the 3rd International Conference on Security of Information and Networks*, pages 263–269. ACM, 2010.

[29] Razvan Pascanu, Jack W Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE, 2015.

[30] Abdurrahman Pektaş, Mehmet Eriş, and Tankut Acarman. Proposal of n-gram based algorithm for malware classification. In *The Fifth International Conference on Emerging Security Information, Systems and Technologies*, pages 7–13, 2011.

[31] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe. *arXiv preprint arXiv:1710.09435*, 2017.

[32] Edward Raff, Richard Zak, Russell Cox, Jared Sylvester, Paul Yacci, Rebecca Ward, Anna Tracy, Mark McLean, and Charles Nicholas. An investigation of byte n-gram features for malware classification. *Journal of Computer Virology and Hacking Techniques*, pages 1–20.

[33] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security*, 19(4):639–668, 2011.

[34] Igor Santos, Yoseba K Penya, Jaime Devesa, and Pablo Garcia Bringas. N-grams-based file signatures for malware detection. *ICEIS (2)*, 9:317–320, 2009.

[35] Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. In *Malicious and Unwanted Software (MALWARE), 2015 10th International Conference on*, pages 11–20. IEEE, 2015.

[36] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. Data mining methods for detection of new malicious executables. In *Security and Privacy, 2001. S&P 2001. Proceedings. 2001 IEEE Symposium on*, pages 38–49. IEEE, 2001.

[37] Dinggang Shen, Guorong Wu, and Heung-Il Suk. Deep learning in medical image analysis. *Annual Review of Biomedical Engineering*, (0), 2017.

[38] Andrew H Sung, Jianyun Xu, Patrick Chavez, and Srinivas Mukkamala. Static analyzer of vicious executables (save). In *Computer Security Applications Conference, 2004. 20th Annual*, pages 326–334. IEEE, 2004.

[39] S Momina Tabish, M Zubair Shafiq, and Muddassar Farooq. Malware detection using statistical analysis of byte-level file content. In *Proceedings of the ACM SIGKDD Workshop on CyberSecurity and Intelligence Informatics*, pages 23–31. ACM, 2009.

[40] Gil Tahan, Lior Rokach, and Yuval Shahar. Mal-id: Automatic malware detection using common segment analysis and meta-features. *Journal of Machine Learning Research*, 13(Apr):949–979, 2012.

[41] Yellepeddi Vijayalakshmi, Neethu Natarajan, P Manimegalai, and Dr Sasidhar Babu. Study on emerging trends in malware variants. 116:479–489, 01 2017.

[42] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, 5(2), 2007.

[43] Heng Yin, Dawn Song, Manuel Egele, Christopher Kruegel, and Engin Kirda. Panorama: capturing system-wide information flow for malware detection and analysis. In *Proceedings of the 14th ACM conference on Computer and communications security*, pages 116–127. ACM, 2007.

[44] Wenzhi Zhao and Shihong Du. Spectral–spatial feature extraction for hyperspectral image classification: A dimension reduction and deep learning approach. *IEEE Transactions on Geoscience and Remote Sensing*, 54(8):4544–4554, 2016.