DeepMal: Deep Convolutional and Recurrent Neural Networks for Malware Classification

Ly VD, Trong Kha Nguyen, Seong Oun Hwang

***Abstract*—Malware classification tasks have traditionally been solved using hand-crafted features obtained through heuristic processes by experts and machine learning methods. It requires much more domain knowledge of malware domain and feature design. To address this problem, we proposed DeepMal, an automatic malware classification based on deep hybrid convolutional and recurrent neural network over raw assembly code inputs. In particular, DeepMal models complicated relationships between executable code blocks of malware disassembly, which perform individual functionalities, therefore perform efficiently mapping the inputs and predicted outputs. In each code block, it also captures temporal information about assembly instructions that are fundamental for discovering malware functionalities. Using a dataset of 3000 malicious and 3000 benign executables, we achieved the F1 score of 0.98.**

***Index Terms*—IEEE, IEEEtran, journal, LATEX, paper, template.**

I. Introduction

Malicious software or malware is any software that causes harm to a user, computer, or network. Currently, millions of malicious programs exist in the wild, and more are encountered every day. In the second quarter of 2015 alone, a Spanish security firm observed an average of 230,000 malware samples created each day [39]. Another security firm put this figure closer to one million [33]. The primary reason behind this phenomenon is that these new types of malware were mainly variants of previously developed malware. Malware authors tend to employ various evasion techniques on these variants to thwart detection by signature-based security systems, which are based on hand crafted rules constructed by analysts. They create new variants of existing malware that can bypass Antivirus vendors. To keep pace with the variety of malware emerging without manually writing detection rules by various machine learning approaches (e.g. [1], [2], [3], [5], [17], [28], [29], [34], [38], [39], [41], [44], [46], [47], [49], [50], [55], [61]). In particular, Machine learning algorithms like Association Rule, Support Vector Machine, Decision Tree, Random Forest, Naive Bayes and Clustering have been proposed for detecting unseen malware samples. These approaches rely on hand-drafted features such as portable executable (PE) headers, strings, byte sequences in [49], n-gram of byte sequences [28], function call graphs [29], and function length frequency [55] to automatically differentiate malware from benign executable.

Although machine learning is a very successful technology, it is limited in its ability to process natural data, for example raw hexadecimal bytes. For decades, constructing machine-learning systems have required careful engineering and considerable domain expertise to design a feature extractor that transformed the raw data into a suitable internal representation or feature vector from which the learning subsystem, often a classifier, could detect or classify patterns in the input. Humans do not know what feature representations are best for a given task. As a result, we might want to automatically find and represent optimal features and handle indirect relationships between features and goals. In other words, we tackle program representation problem.

Representation learning is a set of methods that allows a machine to be fed with raw data and to automatically discover the representations needed for detection or classification.In particular, deep learning methods ([31]) are making major advances in solving this problem, as they contain neural networks that take advantage of non-linear information processing in various layers for feature extraction and classification. The network is organized hierarchically, with each layer processing the outputs of the previous layer. The key aspect of deep learning is that these layers of features are not designed by human engineers. Rather they are learned from data using a general-purpose learning procedure. As a result, deep learning techniques have recently outperformed many conventional methods in image recognition ([16], [30], [53], [56]) and natural language processing ([8], [12], [52]).

Convolutional neural networks (ConvNets or CNNs) ([32], [33]) are essential tools for deep learning. They are designed as feature extractors for spatial features while RNNs are an essential tool for learning sequences. CNN can be used to learn spatial features while RNN is an essential tool for learning sequences. By stacking several convolutional operators into a network, we can create a hierarchy of progressively more abstract features. Such models are able to learn multiple layers of feature hierarchies automatically or perform representation learning. On the other hand, recurrent neural networks (RNNs) ([15], [24]) are such neural networks for modeling the dependencies in time sequences. They are inherently deep in time and able to retain all past inputs. As such, the network can discover correlations between input data at different states of the sequence.

Particularly, there are two important reasons for incorporating CNNs and RNNs into malware detection. First, when treating the disassembly of an executable (malicious or benign) as a kind of raw signal at a character level we can stack several convolutional operators to create a hierarchy of more abstract features based on their spatial structure. Second, if considering the disassembly as code blocks or sequences, we can utilize RNNs to identify spatial dependencies, thereby improving detection accuracy.

Applying deep learning to malware detection at first involves selecting raw features that the model can interact with. Example features such as header, tags, bytecode, API calls ([25]), and control flow graphs ([40]) are not close to raw forms of an executable. Using raw features is not only useful for deep learning processing, but also hard to obfuscate. Instead, we choose to represent an executable as sequences of code blocks. Each block has individual functionalities.

Our proposed method consists of the following steps in a supervised training phase. First, given an executable (benign or malicious), we disassemble it into sequences of assembly codes by using a recursive traversal disassembly algorithm and control flow analysis. Next, these sequences are fed into several CNN layers which pick out invariant features of an one-dimensional spatial structure. Then, the output of the CNN layer is fed through the RNN layers to reduce temporal variation. Finally, the output of the last RNN layer is input into the fully connected DNN layers, which transform the features into a space that allows the output to be classified as benign or malicious.

The contributions of this paper are as follows:

- We present DeepMal: a deep learning framework composed of convolutional and recurrent layers, capable of automatically learning feature representation and modeling temporal dependencies between their activations.
- We show that the system works directly on raw assembly data with minimal preprocessing overhead, thereby making it particularly general and minimizing engineering bias.
- We show that the proposed architecture outperforms the state-of-the-art alternative malware detection domains including n-gram and API call frequency approaches.

The rest of this paper is organized into several major sections. In Section V, we present relevant neural network layers that made up our model. In Section VI, we explain our motivation in applying deep neural networks, as well as the particular specifications we chose to use in each network layer. We give the results of our empirical evaluation and comparisons in Section VIII. In Section IX, we describe some related work in the areas of malware classification and neural networks. In Section X, we draw our conclusion.

II. Reducing Data Preprocessing

We disassemble the file and analyze assembly code instructions that make up the program. Examining the instructions requires more specialized knowledge, we therefore propose splitting an instruction into characters. For example, the instruction "mov eax, ebx" is spitted into three characters 'm', 'o' and 'v', 'a', 'b', 'x'. This implication of engineering could be crucial for a single system can work for different instruction set, since characters always constitute a necessary construct regardless of whether segmentation into words is possible. Working on only characters also has the advantage that the feature set is fixed even though our aim to capture both mnemonic and arguments of an instruction. Most approaches in this field always remove arguments of instructions or normalizing them, and capture only information of mnemonic. As a result, their approach is not sufficient for characterizing malware behavior as ours.

III. Automate Feature Engineering

In this section, we address the problem of representing feature by designing an multi layer model that are able to yield conceptual abstractions by each layer in a hierarchical way. Working with high dimension input data as malicious code results in a large feature space. Researchers always reduce the number of input dimension and smoothen out some non-convexity through space. We therefore provides a technique for reduction of input space dimension, retaining important features to be learned automatically. Specifically, each network layer in convolutional neural networks (CNN) acts as a detection filter for the presence of specific features or pattern in the original data. In addition to capture spatial features in input data, we tend to exploit relationship between these features or temporal features. Instead of representing them as a linear vector of individual and not correlated features, we add each feature to a cell state. Each cell state is able to connect previous information to the present task, such as using previous instruction might inform the understanding of the present instruction. For example, the recent instruction "push ebp, esp" (stores the previous base pointer (ebp)) suggests that the next instruction is probably the instruction "mov ebp, esp" (set the base pointer as it was the top of the stack). The two instructions are know as prolog assembly function. Clarifying relationship between instructions is crucial. The dependency between instruction can be represented as a graph but it is expensive. To fully capture the relationship of instructions in the entire binary, we design two recurrent operations, one which operates from the beginning of the binary to the end, and another which operates in the other direction.

IV. Optimizing model hyperparameter

Hyperparameter optimization helps choosing a set of optimal hyperparameters. Current approaches always ignore optimizing hyperparameter of machine learning algorithms. To get best hyperparameters for model, we apply Grid search technique.

We try experimenting with various hyper parameters to get the best ones. In particular, for selecting the number of epochs that we need to run, we try with a small number of epoch and increase the epoch if the performance is getting better. In case the performance is not increasing, we stop the increment.

A. *Optimizing Batch size and Number of Epochs*

We will evaluate a suite of different mini batch sizes from 10 to 100 in steps of 10. From Table I we can see that the

TABLE I
Experiments with a variety number of epochs

Number of epochs	10	20	30	40
F-measure	0.9813	0.9816	0.9866	0.9813

accuracy is increasing until stop point number 40.

B. *Optimizing Batch size and Number of hidden layers*

We will evaluate a suite of different number of layers from 1 to 6 . Table II shows that with only 1 hidden layer we achieved the best performance.

TABLE II
EXPERIMENTS WITH A VARIETY NUMBER OF HIDDEN LAYERS

Number of hidden layers	1	2	3	4
F-measure	0.9813	0.98	0.9783	0.5192

C. Tuning the optimization algorithm

We tune the optimization algorithm used to train the network, each with default parameters. A list of optimizers incude : SGD, RMSprop, Gdagrad, Adadelta, Adam, Adamax, Nadam

TABLE III
EXPERIMENTS WITH A VARIETY NUMBER OF OPTIMIZERS

Optimizer	Adam	Adadelta	SGD	RMSprop	Adamax
F-measure	0.9813	0.66	0.8961	0.9701	0.9799

Among many optimizations, Adam achieved the best performance as shown in Table III

D. Tuning Network Weight Initialization

A list of network weight initialization methods consist of : Uniform, lecun uniform, normal, zero, glorot normal, glorot uniform, he normal, he uniform.

TABLE IV
EXPERIMENTS WITH A VARIETY NUMBER OF INITIALIZATION METHODS

Initialization method	he_uniform	glorot_uniform	glorot_normal
F-measure	0.9793	0.9746	0.9813

Table IV shows that the Glorot normal initialization method performs the best.

E. Tuning the Neuron activation function

A list of activations: softmax, softplus, softsign, relu, tanh, sigmoid, hard sigmoid, linear

F. Tuning Dropout Regularization

We will try dropout percentages between 0.0 and 0.9 and maxnorm weight constraint values between 0 and 1.

TABLE V
EXPERIMENTS WITH A VARIETY NUMBER OF DROPOUT VALUES

Dropout value	0	0.25	0.5	0.75	1
Validation Fmeasure	0.9694	0.9813	0.9810	0.9745	0.9713

From Table V we choose dropout value of 0.5.

G. Tuning the number of training samples

We try increasing the number of training samples and monitor the change of the model's behavior.

As shown in Table VI we can see that performance of the system can improve as more samples have been adding.

V. BUILDING NEURAL NETWORK LAYERS

In this part, we provide background for four types of layers in a neural network that will be used later to construct a single network for malware classification.

A. Embedding Layer

Neural networks process their inputs by multiplying them with weight parameters, which only makes sense when the input values represent intensities. We convert each code block in an executable disassembly of a malware to a concatenated form of one-hot vectors. The one-hot vector of the i-th symbol in a vocabulary is simply a binary vector whose elements are all zeros except for the i-th element which is set to one. In our case, each assembly code block is a sequence of T one-hot vectors $(x_1, x_2, ..., x_T)$.

The *embedding layer* maps each of the one hot vectors into a d-dimensional continuous vector space $\mathbb{R}^d$. This can be achieved by merely multiplying the one-hot vector starting from the left with a weight matrix W $\in \mathbb{R}^{d \times |V|}$, where $|V|$ is the number of unique symbols in a vocabulary:

$$e_t = Wx_t$$

After the embedding layer, the input sequence of one-hot vectors becomes a sequence of dense, real valued vectors $(e_1, e_2, ..., e_T)$ with the precise representation learned during the training of the neural network.

B. Convolutional Layer (CNN)

A CNN with a single layer extracts features from the input signal through a convolution operation of the signal with a set of d' filters (or kernels) of receptive filed size r. In the context of CNN, a kernel or filter is capable of filtering input data and removing outliers, or acting as a feature detector. It is defined to respond maximally to specific temporal sequences with the timespan of the kernel. On the other hand, the activation θ of a unit represents the result of the convolution of the kernel with the input signal. By computing the activation of a unit on different regions of the same input (using a convolutional operation), it is possible to detect patterns captured by the kernels, regardless of where the pattern occurs. Formally, given r, where $F \in \mathbb{R}^{d' \times r}$ is applied to the input sequence, a typical sequence is calculated by:

$$f_t = \theta(F[e_{t-(r/2)+1}; ...; e_t; ...; e_{t+(r/2)}])$$

, where θ is a nonlinear activation function such as the hyperbolic tangent function $tanh$ or rectifier linear unit ($ReLu$) ([37]). However, $ReLu$ can train six times faster than $tanh$ to reach the same training error ([37]) and has the following formulation:

$$RELU(x) = \begin{cases} 0 & \text{if } x < 0 \\ -(n+1)/2 & \text{if } x >= 0 \end{cases}$$

The activation function is applied to every time step of the input sequence, which generates a sequence $F = (f_1, f_2, ..., f_T)$. For each convolutional layer, we also insert a global max-pooling layer. In this way, the sequence F is *max-pooled* with a max pooling layer of size r':

$$f'_t = max(f_{(t-1) \times r' + 1}, ..., f_{t \times r'})$$

, where max function is applied for each item of the vectors, which result in the below sequence:

$$F' = (f'_1, f'_2, ..., f'_{T/r'})$$

TABLE VI
EXPERIMENTS WITH A VARIETY NUMBER OF TRAINING SAMPLES

Number of training sample	100	500	1000	2000	3000	4000	4800
Validation Fmeasure	0.8875	0.9378	0.9566	0.9683	0.9759	0.9786	0.9813

Additional layers of convolution and max-pooling make the neural network 'deeper' and enable the model to extract abstract features.

C. Recurrent Layer (RNN)

A recurrent layer maps sequences to sequences using a recursive function f which takes one input vector and the previous hidden state as inputs, and outputs the new hidden state:

$$h_t = f(x_t, h_{t-1})$$

, where $x_t \in \mathbb{R}^d$ is one time step from the input sequence $x_1, x_2, ..., x_T$. $h_0 \in \mathbb{R}^{d'}$ is the weight matrix. This naive recursive function, however, is known to suffer from the problem of vanishing gradient ([6], [21]).

In research literature, it is more common to use a complicated function which learns to control the flow of information for preventing the vanishing gradient and allows the recurrent layer to more easily capture long-term dependencies. Long short-term memory (LSTM) unit from ([22], [18]) and Gated Recurrent Unit (GRU) ([10]) are representative examples. The LSTM unit typically consists of four child-units: input, output, forget gates and a candidate memory cell. They are formalized by:

$$\begin{aligned} i_t &= \sigma(W_i x_t + U_i h_{t-1}) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1}) \\ o_t &= \sigma(W_o x_t + U_o h_{t-1}) \\ \tilde{c}_t &= tanh(W_c x_t + U_c h_{t-1}) \end{aligned}$$

Based on the above formulas, the LSTM unit first computes the memory cell as:

$$c_t = i_t \odot \tilde{c}_t + f_t \odot c_{t-1}$$

and then computes the output, or activation as:

$$h_t = o_t \odot tanh(c_t)$$

The sequence from the recurrent layer is then defined as:

$$(h_1, h_2, ..., h_T),$$

where T is the length of the input sequence to the layer.

GRU is a slightly more dramatic variation on the LSTM. While sharing the same goals of avoiding the long-range dependency problems that have plagued RNN, it combines the forget and input gates into a single update gate. It also merges the cell state and hidden state. The GRU model is therefore simpler than the standard LSTM model:

$$\begin{aligned} c_t &= tanh(W_c x_t + U_c h_{t-1}) \\ i_t &= tanh(W_i x_t + U_i(c_t \odot h_{t-1}) \\ f_t &= \sigma(W_f x_t + U_f h_{t-1}) \end{aligned}$$

Unlike LSTM, there exists no separate memory state c_t from the hidden state. The network exposes the entire hidden state at each time step h_t as follows:

$$h_t = f \odot h_{t-1} + (1 - f) \odot i$$

D. Classification Layer

The final layer of connections in the netwok is a fully-connected layer. That is, this layer connects every neuron form the previous layer to each one of the 2 outputs. More specifically, given a fixed-dimensional input from the previous layer, the classification layer essentially transforms it, followed by a *sigmoid* activation function. The output of a sigmoid neuron with inputs $x_1, x_2, ...$, weights $w_1, w_2, ...$, and bias b is:

$$\frac{1}{1 + \exp(-\sum_j w_j x_j - b)}$$

This classification layer takes a *fixed-dimensional* vector as input, while the recurrent layer or convolutional layer returns a variable-length sequence of vectors which is determined by the input sequence. This can be addressed by either simply max-pooling the vectors ([26]) over the time dimension (for both convolutional and recurrent layers), taking the last hidden state (for recurrent layers) or taking the last hidden states of forward and reverse recurrent networks (for bidirectional recurrent layers).

E. Bidirectional Recurrent Layer

A bidirectional recurrent layer comes into play when there is an imbalance in the amount of information seen by the hidden states at different time steps in the recurrent layer. The previous hidden states only capture a few vectors from the low layer, while the later ones are mainly computed based on the lower-layer vectors. This can be easily reduced by constructing a bidirectional recurrent layer which is composed of two recurrent layers working in opposite directions. The composed layer will return two sequences of hidden states from the forward and reverse recurrent layers, respectively. The vector formula should thus be adjusted as follows:

$$\begin{aligned} h_{f_t} &= H(W_{xh_f} x_t + W_{h_f h_f} h_{f_{t-1}} + b_{h_f}) \\ h_{b_t} &= H(W_{xh_b} x_t + W_{h_b h_b} h_{b_{t-1}} + b_{h_b}) \end{aligned}$$

, where $h_f \in R^d$ and $h_b \in R^d$ denote the output vector of the forward layer and backward layer, respectively. Different from former research, the final output in our study $y_t = [h_{f_t}, h_{b_t}]$ is the concatenation of these parts, which means $y_t \in R^{2d}$.

TABLE VII
NEURAL NETWORK LAYERS USED IN OUR EXPERIMENTS

Model	Embedding Layer		Convolutional Layer				Recurrent Layer	Classification Layer	
	$[V]$	d	d'	r	r'	$\emptyset$	d'	d	$\emptyset$
CNN-RNN CNN-BiRNN	41	10	D	5,3,3	2	relu	D	300,300,1	relu,relu,sigmoid

VI. Hybrid Character-Level Convolutional-Recurrent Network for Malware Detection

In this section, we propose a hybrid of convolutional and recurrent networks for character-level malware detection.

A. Motivation

One of the main motivations for using a convolutional layer is that it specializes in learning to extract high level global and invariant features for local translation. By stacking multiple convolutional layers to form a full ConvNet architecture, the network can extract higher-level, abstract (local) translation invariant features from the input code sequences. In spite of this advantage, we observed that it usually requires many layers of convolution to capture long-term dependencies, due to the locality of the convolution and pooling ([20]) (Section V-B). This becomes more severe as the length of the input sequence grows. In the case of character-level modeling, it is usual for an assembly code block to be a sequence of hundreds or thousands of characters. Ultimately, this leads to the need for a very deep network having a large number of convolutonal layers. In contrast, the recurrent layer presented in Section V-C is able to capture long-term dependencies even when there is only a single layer. This is especially true in the case of the bidirectional recurrent layer (Section V-E) which allows access to both the past and the future in making a prediction for the present. In this layer, one operates from the beginning of the sequence to the end, and another operates in the order direction. As a result, each hidden state relates to the hidden states of both the previous and next time steps covering the whole input sequence. The recurrent layer, however, has the side effect of complexity. Normally it grows linearly with respect to the length of the input sequence and most of the computations need to be done sequentially. This is in contrast to the convolutional layer where computations can be efficiently done in parallel.

Based on these observations, we propose a method of combining the convolutional and recurrent layers into a single model. This network can capture the long term dependencies in code sequences of the disassembly more efficiently for the task of malware detection.

B. Model Description

The proposed model, which we refer to as a convolution-recurrent network (CRNN), starts with a one-hot sequence input:

$$X = (x_1, x_2, ..., x_T)$$

This input sequence is turned into a sequence of dense, real-valued vectors:

$$E = (e_1, e_2, ..., e_T)$$

using the *embedding layer* from Section V-A . We apply multiple *convolutonal layers* (Section V-B) to E to obtain a shorter sequence of feature vectors:

$$F = (f_1, f_2, ..., f_{T'})$$

This feature vector is then fed into a *bidirectional recurrent layer* (Section V-E), resulting in two sequences:

$$H_{f_t} = (\overrightarrow{h_1}, \overrightarrow{h_2}, ..., \overrightarrow{h_{T'}})$$
$$H_{b_t} = (\overleftarrow{h_1}, \overleftarrow{h_2}, ..., \overleftarrow{h_{T'}})$$

We take the last hidden states of both directions and concatenate them to form a fixed-dimensional vector:

$$h = [\overrightarrow{h_{T'}}; \overleftarrow{h_1}]$$

The vector h is then fed to a fully-connected time distributed dense layer to constrain the same function for all outputs at consecutive time steps.

$$F = TimeDistributed(h)(E)$$

The second bidirectional recurrent layer decodes h' to form the final vector h':

$$H'_{f_t} = (\overrightarrow{h'_T}, \overrightarrow{h'_{T-1}}, ..., \overrightarrow{h'_1})$$
$$H'_{b_t} = (\overleftarrow{h'_1}, \overleftarrow{h'_2}, ..., \overleftarrow{h_{T'}})$$
$$h' = [\overrightarrow{h'_1}; \overleftarrow{h'_T}]$$

Finally, the fixed-dimensional vector h' is fed into the *classification layer* (Section V-D) to compute the predictive probabilities of the categories being malicious or benign given the input sequence *X*.

Figure 1 depicts the high-level overview of training and evaluating for *DeepMal*. The top left row provides the steps required for extracting code blocks from an executable. On the right, *DeepMal* accepts a varying length code block sequence to predict whether the executable is malicious or benign.

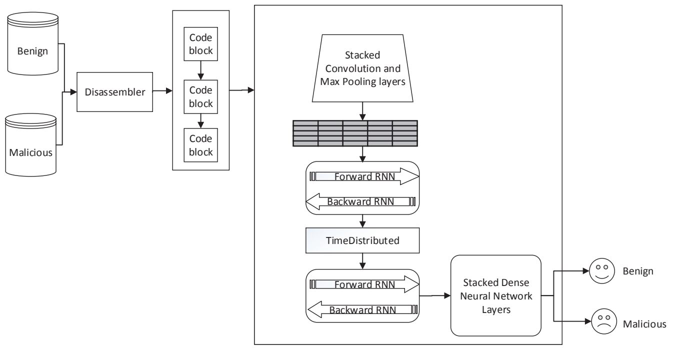

Fig. 1. Graphical illustration of the convoluton-recurrent network for malware classification.

VII. System Model and Experimentations

A. Dataset

Before discussing implementation, we briefly describe the dataset that we used for malware detection experimentations. We collected malware samples from Virusshare [58] and Windows executable software as benign files. We relied on the Microsoft Software Removal Tool [35] to label the samples. In total, our dataset contains 3000 malicious and 3000 benign selected samples. We published the dataset at [59] so that other researchers could conduct their own research.

B. Model Settings

Referring to Section V-A, the alphabet vocabulary V used in all of our models consists of 41 characters, including 24 English characters, 10 digits, and 7 other characters. The 41 non-space characters could already capture all possible assembly instructions of interest :

$$abcdefghijklmnoprstuwxyzv0123456789+-*,:[]$$

Character embedding size d is set to 200. As described in Section VI-A, we believe by adding recurrent layers, one can effectively reduce the number of convolutional layers needed in order to capture long-term dependencies. Thus for the data set, we consider models with three convolutional layers. Following notations in Section V-B, each of three layers has $d' = 196, 196, 256$ filters and a receptive field size $r = 5, 3, 3$ in order. Max pooling size r' is set to 2. Rectified linear units (ReLUs, [19]) are used as activation functions in the convolutional layers. The recurrent layer (Section V-C) is either a single layer of RNN, LSTM, GRU or their bidirectional variant for comparison. The hidden state of dimension d' is set to 128. More detailed setups are described in Table VII.

Dropout ([51]) is a regularization technique for reducing overfitting in neural networks. In this study, we apply dropout after the last convolutional layer and the recurrent layer. In the absence of dropout, the inputs to the recurrent layer x_t's are:

$$x_t = f'_t$$

where f'_t is the t-th output from the last convolutional layer defined in Section V-B. After adding dropout, we have

$$r^i_t \sim Bernoulli(p)$$
$$x_t = r_t \odot f'_t$$

, where p is the dropout probability which we set to 0.25 and r^i_t is the i-th component of the binary vector $\mathbf{r}_t \in \mathbb{R}^{d'}$. We insert 2 dropout modules in between the 3 fully-connected layers to regularize.

C. Environment

We base our model on open source tools. To extract code sequences of an executable, we use the Capstone disassembler ([43]). We build deep neural networks from the Keras deep learning library ([11]). These open source frameworks run on Ubuntu 16.04.2 LTS (Xenial Xerus) 128GB RAM.

D. Performance Metrics

To evaluate the performance of the proposed model, we use the precision, recall, and F1 metrics. They have the following definitions:

$$Precision = \frac{TP}{TP + FP}$$

$$Recall = \frac{TP}{TP + FN}$$

$$F1 = \frac{2 \cdot Precision \cdot Recall}{Precision + Recall}$$

TP represents the number of true positive predictions. FP is the number of false negative predictions. The $F1$ score is the harmonic mean of precision and recall, which allows us to conveniently compare them using one number.

E. Training and Validation

We partition the dataset into three parts: the training set (60%), the validation set (20%) and the test set (20%). Details of using these sets are depicted in Figure 2.

The models are trained by minimizing the following regularized log-likelihood or cross entropy loss. Given a list of correct input-output pairs $(x_1, y_1), ..., (x_n, y_n)$ for the purpose of training the network, the loss is defined as

$$\frac{1}{n}\sum_{i=1}^{n} d(y_i, f(x_i)).$$

We train our models using an Adam optimizer with ([27]) with a learning rate $\rho = 1$, $\epsilon = 1e^{-8}$ and a batch size of 16. Examples are padded to the longest sequence in each batch and masks are generated to help identify the padded region. The corresponding masks of the outputs from the convolutional layers can be analytically computed layers and are used by the recurrent layer to properly ignore padded inputs. At each epoch, we calculate and record the validation loss. We report the test error rate using the model with the lowest validation error.

```
Data: A dataset: $d$
Result: a trained model, and its performance metrics
Initialization: train, validation, test set
Validation error threshold: $t$
// Training and Validation phase
foreach epoch do
    foreach training data instance do
        propagate error through the network
        adjust the weights
        calculate the accuracy over the training data
    end
    foreach validation data instance do
        calculate the accuracy over the validation data
        tune model parameters
        calculate the accuracy over validation data

        if the threshold validation $t$ is met then
            return the model
            exit validating
        else
            continue validating
        end if
    end
end
// Testing phase
foreach test data instance do
    calculate the accuracy over the test data
end
```

Fig. 2. Details of using training, validation and test sets in our models

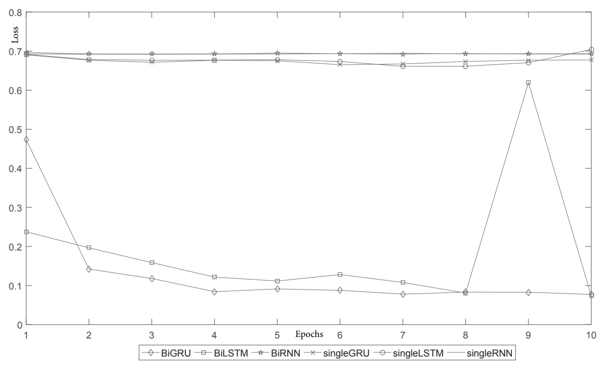

Fig. 3. Observed validation loss over model building

VIII. Summary of Results

A. Evaluations of Different Deep Networks

Table VIII and Figure 3 summarize our main experimental results. Table VIII compares how GRU and LSTM fare against conventional RNNs. As we might expect, GRU and LSTM perform better than RNN in both single and bidirectional versions. It turns out that stacking a number of bidirectional RNN layers provides a slightly greater benefit. At first glance, BiGRU and BiLSTM achieve the same F1 validation score on test benchmarks while single LSTM or GRU are defeated by the single RNN. However, their bidirectional variants beat RNNs on the same set.

TABLE VIII
Reported F1 scores for all architectures.

	Validation loss	Train	Validation	Test
CNN-RNN	0.69	0.46	0.67	0.64
CNN-LSTM	0.66	0.22	0.18	0.20
CNN-GRU	0.67	0.27	0.14	0.16
CNN-BiRNN	0.69	0.49	0.67	0.64
CNN-BiLSTM	0.07	0.97	0.98	0.97
CNN-BiGRU	0.08	0.99	0.98	0.97

We ran these models on the test set and achieved the results shown in Table VIII. The results show that it is more efficient to use a bidirectional recurrent network than a single network. As such, BiGRU and BiLSTM are very competitive.

B. Comparisons between Deep Learning and Other Shallow Learning based Classification Methods

In this section, using the same dataset described in Section VII-A, we conduct a comparison between our proposed deep learning framework (DeepMal) and other shallow learning based classification methods (i.e., Support Vector Machine, Instance Base Learning, and Decision Tree).

Kolter et al [28] used n-gram (instead of non-overlapping byte sequences) and a data mining method to detect malicious executables. They used different classifiers including Naive-Bayes, Support Vector Machine, Decision Tree and their boosted versions.

TABLE IX
Performance of the n-gram approach

Classifier	Precision	Recall	F1
SMO	0.75	0.66	0.61
IBk	0.76	0.66	0.61
RF	0.76	0.66	0.62
J48	0.74	0.65	0.61

The n-gram has a smaller F1 score at 0.35. This result shows that our work can achieve higher performance without feature engineering.

In the next comparison, we challenge the behavioral approach which requires more effort to conduct feature extraction but is able to capture a real executable's behavior. Rieck et al. [44] proposed a framework for automatic analysis of malware behavior using machine learning. Given a trace x in all possible call traces X and a feature s in a predefined set of features F, they obtained the frequency $f(x, s)$ by recording the number of occurrences of s. Then, they defined a threshold value T to measure the importance of strings. If a string frequency $f(x, s)$ exceeds the threshold T, it is considered to be dominant; otherwise, it is not important.

We constructed the API call frequency approach by counting the number of API calls in the system call trace. There are 274 API calls captured in total. Table X shows the performance of the API call frequency approach.

TABLE X
PERFORMANCE OF THE API CALL FREQUENCY APPROACH

Classifier	Precision	Recall	F1
SMO	0.769	0.764	0.765
IBk	0.931	0.931	0.931
RF	0.946	0.945	0.945
J48	0.950	0.950	0.950

We can see the proposed approach has a 0.2 greater F1 score compared to [44] which involves running the malware and observing its behaviors (system calls) which resut in feature extraction overhead. By contrast, our approach does not.

IX. Related Work

Security researchers has been take advantage of machine learning for detect unseen malware samples with numerous approaches in static and dynamic [14] approaches. Machine learning for malware detection and classification consist of two steps, the feature sets obtained from binary inputs by static or dynamic analysis, and a learning algorithm, which builds a classifier using these features. This research broadly falls into two main areas machine learning and deep learning for malware classification.

A. Machine Learning Methods for Malware Detection

In despite of static analysis is vulnerable to obfuscation [36], it does not require expensive setup for feature collection, and very large datasets with accurate labels can be created by simply aggregating the binaries files on public site anti-virus aggregator like VirusTotal [57].

Several approaches have been proposed for extracting static features from Portable Executable (PE) binaries. Schultz et al. [49] extracted strings, byte sequence and header information files such as a list of Dynamically Linked Libraries (DLLs), function calls from DLL and the number of different systems calls from within each DLL. Weber et al. [60] considered import tables, informational entropy and entropy of opcodes. The opcodes are also examined in term of frequency distribution [7] and a sequence [45] since they specify the operation of machine to be performed.

Machine learning method from text classification also employed in this domain such as n-gram method. Kolter et al. [28] encoded n-gram of consecutive byte sequences as features to detect malicious executables. There was an issue of high computational overhead since an enormous number of n-grams were present. They required an expensive feature selection to reduce the feature space.

Some authors have looked into information flow features in static, dynamic and hybrid approaches. Anderson et al. [1] presented a data representation from assembly level. They dynamically traced instructions and transformed it to a Markov chain graphs, then fed it to classifiers. They also extended with static features in a hybrid approach [2] to get a better performance. Cesare et al. [9] proposed a malware classification system using approximate matching of control flow graphs. They used a technique to extract q-grams and k-subgraphs of sets of control flow graphs and created feature vectors. Overall, those approaches require a lot of efforts in features extraction and feature selection to avoid the overhead of classifiers.

B. Neural Networks for Malware Detection and Classification

Neural network approaches have been developed over the last decade by IBM research for detecting the virus on boot sector in floppy devices [54]. These days, numerous malware research proposed various neural network model since it tends to perform better since data size increases [4].

The most recent summary of the field of malware classification takes advantage of the conventional neural network such as a feed-forward network (FNN). It also requires an appropriate architecture to avoid the potential of over-fitting and reduce computation. That is related to selecting a number of independent input, hidden layer and output since the complexity of model can tend to over-fit data. Building neural network model for malware classification usually combine the FNN with a feature extraction method such as random projection to reduce the dimension of input layers. Static Dahl et al. [13] modified detection engine to extract API calls while Saxe et al. [48] extracted PE import features (hash of DLL name, import function), metadata features in which come from static analysis process. Huang et al. [23], on the other hand, achieved dynamic feature from a light-weight Instrumented Anti-Malware Engine with null-terminated tokens, API event plus parameter value, and API trigrams contain millions of potential features. Those approaches deal with the sparse input matrix by random projection method in order to select appropriate and smaller features space. Compare to our method, the model automatically extracts static feature at raw level of binary as assembly code and reduce spatial feature dimension without any feature selection or extraction method.

Another approach considered input variables as a sequence instead of independent variables like FNN network. Pascanu et al. [42] constructed a model of malware through sequences of executed instructions with recurrent neural networks. They represented a sequence by a bag of events and max-pooled in a temporal to extract relevant features. Our deep model represented the input at character-level in which it extracts character in a spatially and temporally.

X. Conclusions

There have been some previous work which apply machine learning techniques to problems of malware detection with high feature engineering overhead. To address this feature engineering issue, we proposed a deep neural network architecture for malware detection. By combining convolutional and recurrent layers, it is capable of automatically learning feature representation and modeling temporal dependencies between their activations. We also showed that the proposed architecture outperforms the state-of-the-art alternative malware detection approaches such as n-gram and API call frequency.

References

[1] B. Anderson, D. Quist, J. Neil, C. Storlie, and T. Lane. Graph-based malware detection using dynamic analysis. *Journal in computer Virology*, 7(4):247–258, 2011.

[2] B. Anderson, C. Storlie, and T. Lane. Improving malware classification: bridging the static/dynamic gap. In *Proceedings of the 5th ACM workshop on Security and artificial intelligence*, pages 3–14. ACM, 2012.

[3] M. Bailey, J. Oberheide, J. Andersen, Z. M. Mao, F. Jahanian, and J. Nazario. Automated classification and analysis of internet malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 178–197. Springer, 2007.

[4] M. Banko and E. Brill. Scaling to very very large corpora for natural language disambiguation. In *Proceedings of the 39th annual meeting on association for computational linguistics*, pages 26–33. Association for Computational Linguistics, 2001.

[5] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *NDSS*, volume 9, pages 8–11. Citeseer, 2009.

[6] Y. Bengio, P. Simard, and P. Frasconi. Learning long-term dependencies with gradient descent is difficult. *IEEE transactions on neural networks*, 5(2):157–166, 1994.

[7] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[8] A. Bordes, S. Chopra, and J. Weston. Question answering with subgraph embeddings. *arXiv preprint arXiv:1406.3676*, 2014.

[9] S. Cesare and Y. Xiang. Malware variant detection using similarity search over sets of control flow graphs. In *Trust, Security and Privacy in Computing and Communications (TrustCom), 2011 IEEE 10th International Conference on*, pages 181–189. IEEE, 2011.

[10] K. Cho, B. Van Merriënboer, C. Gulcehre, D. Bahdanau, F. Bougares, H. Schwenk, and Y. Bengio. Learning phrase representations using rnn encoder-decoder for statistical machine translation. *arXiv preprint arXiv:1406.1078*, 2014.

[11] F. Chollet. Keras, 2015.

[12] R. Collobert, J. Weston, L. Bottou, M. Karlen, K. Kavukcuoglu, and P. Kuksa. Natural language processing (almost) from scratch. *Journal of Machine Learning Research*, 12(Aug):2493–2537, 2011.

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

[14] M. Egele, T. Scholte, E. Kirda, and C. Kruegel. A survey on automated dynamic malware-analysis techniques and tools. *ACM computing surveys (CSUR)*, 44(2):6, 2012.

[15] J. L. Elman. Finding structure in time. *Cognitive science*, 14(2):179–211, 1990.

[16] C. Farabet, C. Couprie, L. Najman, and Y. LeCun. Learning hierarchical features for scene labeling. *IEEE transactions on pattern analysis and machine intelligence*, 35(8):1915–1929, 2013.

[17] I. Firdausi, A. Erwin, A. S. 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] F. A. Gers, J. Schmidhuber, and F. Cummins. Learning to forget: Continual prediction with lstm. *Neural computation*, 12(10):2451–2471, 2000.

[19] X. Glorot, A. Bordes, and Y. Bengio. Deep sparse rectifier neural networks. In *Aistats*, volume 15, page 275, 2011.

[20] G. E. Hinton, N. Srivastava, A. Krizhevsky, I. Sutskever, and R. R. Salakhutdinov. Improving neural networks by preventing co-adaptation of feature detectors. *arXiv preprint arXiv:1207.0580*, 2012.

[21] S. Hochreiter, Y. Bengio, P. Frasconi, and J. Schmidhuber. Gradient flow in recurrent nets: the difficulty of learning long-term dependencies, 2001.

[22] S. Hochreiter and J. Schmidhuber. Long short-term memory. *Neural computation*, 9(8):1735–1780, 1997.

[23] W. Huang and J. 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.

[24] H. Jaeger. *Tutorial on training recurrent neural networks, covering BPPT, RTRL, EKF and the" echo state network" approach*, volume 5. GMD-Forschungszentrum Informationstechnik, 2002.

[25] W. Jung, S. Kim, and S. Choi. Poster: Deep learning for zero-day flash malware detection. In *36th IEEE Symposium on Security and Privacy*, 2015.

[26] Y. Kim. Convolutional neural networks for sentence classification. *arXiv preprint arXiv:1408.5882*, 2014.

[27] D. Kingma and J. Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

[28] J. Z. Kolter and M. 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.

[29] D. Kong and G. Yan. Discriminant malware distance learning on structural information for automated malware classification. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1357–1365. ACM, 2013.

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

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

[32] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel. Backpropagation applied to handwritten zip code recognition. *Neural computation*, 1(4):541–551, 1989.

[33] Y. LeCun, L. Bottou, Y. Bengio, and P. Haffner. Gradient-based learning applied to document recognition. *Proceedings of the IEEE*, 86(11):2278–2324, 1998.

[34] T. Lee and J. J. Mody. Behavioral classification. In *EICAR Conference*, pages 1–17, 2006.

[35] Microsoft. Microsoft windows malicious software removal tool, 2015.

[36] A. Moser, C. Kruegel, and E. Kirda. Limits of static analysis for malware detection. In *Computer security applications conference, 2007. ACSAC 2007. Twenty-third annual*, pages 421–430. IEEE, 2007.

[37] V. Nair and G. E. Hinton. Rectified linear units improve restricted boltzmann machines. In *Proceedings of the 27th international conference on machine learning (ICML-10)*, pages 807–814, 2010.

[38] S. Nari and A. A. Ghorbani. Automated malware classification based on network behavior. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 642–647. IEEE, 2013.

[39] L. Nataraj, S. Karthikeyan, G. Jacob, and B. Manjunath. Malware images: visualization and automatic classification. In *Proceedings of the 8th international symposium on visualization for cyber security*, page 4. ACM, 2011.

[40] J. Ouellette, A. Pfeffer, and A. Lakhotia. Countering malware evolution using cloud-based learning. In *Malicious and Unwanted Software:" The Americas"(MALWARE), 2013 8th International Conference on*, pages 85–94. IEEE, 2013.

[41] Y. Park, D. Reeves, V. Mulukutla, and B. Sundaravel. Fast malware classification by automated behavioral graph matching. In *Proceedings of the Sixth Annual Workshop on Cyber Security and Information Intelligence Research*, page 45. ACM, 2010.

[42] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. Thomas. Malware classification with recurrent networks. In *Acoustics, Speech and Signal Processing (ICASSP), 2015 IEEE International Conference on*, pages 1916–1920. IEEE, 2015.

[43] N. A. Quynh. Capstone: Next-gen disassembly framework. *Black Hat USA*, 2014.

[44] K. Rieck, T. Holz, C. Willems, P. Düssel, and P. Laskov. Learning and classification of malware behavior. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 108–125. Springer, 2008.

[45] I. Santos, F. Brezo, X. Ugarte-Pedrero, and P. G. Bringas. Opcode sequences as representation of executables for data-mining-based unknown malware detection. *Information Sciences*, 231:64–82, 2013.

[46] I. Santos, C. Laorden, and P. G. Bringas. Collective classification for unknown malware detection. In *Security and Cryptography (SECRYPT), 2011 Proceedings of the International Conference on*, pages 251–256. IEEE, 2011.

[47] I. Santos, J. Nieves, and P. G. Bringas. Semi-supervised learning for unknown malware detection. In *International Symposium on Distributed Computing and Artificial Intelligence*, pages 415–422. Springer, 2011.

[48] J. Saxe and K. 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.

[49] M. G. Schultz, E. Eskin, F. Zadok, and S. 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.

[50] M. Siddiqui, M. C. Wang, and J. Lee. Detecting internet worms using data mining techniques. *Journal of Systemics, Cybernetics and Informatics*, 6(6):48–53, 2008.

[51] N. Srivastava, G. E. Hinton, A. Krizhevsky, I. Sutskever, and R. Salakhutdinov. Dropout: a simple way to prevent neural networks

from overfitting. *Journal of Machine Learning Research*, 15(1):1929–1958, 2014.

[52] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *Advances in neural information processing systems*, pages 3104–3112, 2014.

[53] C. Szegedy, W. Liu, Y. Jia, P. Sermanet, S. Reed, D. Anguelov, D. Erhan, V. Vanhoucke, and A. Rabinovich. Going deeper with convolutions. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 1–9, 2015.

[54] G. J. Tesauro, J. O. Kephart, and G. B. Sorkin. Neural networks for computer virus recognition. *IEEE expert*, 11(4):5–6, 1996.

[55] R. Tian, L. M. Batten, and S. Versteeg. Function length as a tool for malware classification. In *Malicious and Unwanted Software, 2008. MALWARE 2008. 3rd International Conference on*, pages 69–76. IEEE, 2008.

[56] J. J. Tompson, A. Jain, Y. LeCun, and C. Bregler. Joint training of a convolutional network and a graphical model for human pose estimation. In *Advances in neural information processing systems*, pages 1799–1807, 2014.

[57] V. Total. Virusshare.com, 2015.

[58] virusshare. Virusshare.com, 2015.

[59] L. Vu. Our dataset is accessible at https://goo.gl/dcauhp, 2017.

[60] M. Weber, M. Schmid, M. Schatz, and D. Geyer. A toolkit for detecting and analyzing malicious software. In *Computer Security Applications Conference, 2002. Proceedings. 18th Annual*, pages 423–431. IEEE, 2002.

[61] M. F. Zolkipli and A. Jantan. An approach for malware behavior identification and classification. In *Computer Research and Development (ICCRD), 2011 3rd International Conference on*, volume 1, pages 191–194. IEEE, 2011.