# ScriptNet: Neural Static Analysis for Malicious JavaScript Detection

Jack W. Stokes*, Rakshit Agrawal†, Geoff McDonald‡ and Matthew Hausknecht*
*Microsoft Research, Redmond, Washington, 98052, USA, Email: jstokes,mahauskn@microsoft.com
†University of California at Santa Cruz, Santa Cruz, California, 95064, USA, Email: ragrawa1@ucsc.edu
‡Microsoft Corp., #305 876 14th Ave. W, Vancouver, British Columbia, V5Z 1R1, Canada, Email: geofm@microsoft.com

***Abstract*—Malicious scripts are an important computer infection threat vector for computer users. For internet-scale processing, static analysis offers substantial computing efficiencies. We propose the ScriptNet system for neural malicious JavaScript detection which is based on static analysis. We also propose a novel deep learning model, Pre-Informant Learning (PIL), which processes Javascript files as byte sequences. Lower layers capture the sequential nature of these byte sequences while higher layers classify the resulting embedding as malicious or benign. Unlike previously proposed solutions, our model variants are trained in an end-to-end fashion allowing discriminative training even for the sequential processing layers. Evaluating this model on a large corpus of 212,408 JavaScript files indicates that the best performing PIL model offers a 98.10% true positive rate (TPR) for the first 60K byte subsequences and 81.66% for the full-length files, at a false positive rate (FPR) of 0.50%. Both models significantly outperform several baseline models. The best performing PIL model can successfully detect 92.02% of unknown malware samples in a hindsight experiment where the true labels of the malicious JavaScript files were not known when the model was trained.**

***Index Terms*—Malware detection, Neural models, LSTM**

## I. Introduction

The detection of malicious JavaScript (JS) is important for protecting users against modern malware attacks. Because of its richness and its ability to automatically run on most operating systems, malicious JavaScript is widely abused by malware authors to infect users' computers and mobile devices. JavaScript is an interpreted scripting language developed by Netscape that is often used by webpages to provide additional dynamic functionality [1]. In addition, JavaScript is sometimes included in malicious webpages, PDFs and email attachments. To combat this growing threat, we propose ScriptNet, a novel deep learning-based system for the detection of malicious JavaScript files.

There are numerous challenges posed by trying to detect malicious JavaScript. Malicious scripts often include obfuscation to hide the malicious content which unpacks or decrypts the underlying malicious script only upon execution. Complicating this is the fact that the obfuscators can be used by both benign and malware files. Curtsinger *et al.* [2] measured the distributions of malicious and benign JavaScript files containing obfuscation. The authors showed that these distributions are very similar if a file is obfuscated and concluded that the presence of obfuscation alone cannot be used to detect malicious JavaScript.

Another difficulty is that a large number of file encodings (*e.g.*, UTF-8, UTF-16, ASCII) are automatically supported by JavaScript interpreters. Thus, individual characters in the script may be encoded by two or more bytes. As a result, malware script authors can use the encoding itself to attempt to hide malicious JavaScript code [3].

Furthermore, these JavaScript files may be very long which can present computational difficulties. Complex deep learning models which use a graphic processing unit (GPU) may provide higher accuracies but must be able to fit into the onboard memory.

While a wide range of different systems have been proposed for detecting malicious executable files [4], there has been less work in investigating malicious JavaScript. Previous JavaScript solutions include those based purely on static analysis [5], [6]. To overcome the limitations imposed by obfuscation, other methods, [2], [3] and [7] include both static and some form of dynamic (*i.e.*, runtime) analysis to unroll multiple obfuscation layers. In some cases, the solution is focused on the detection of JavaScript embedded in PDF documents [7]–[9]. In addition, deep learning models have recently been proposed for detecting system API calls in PE files [10]–[12], JavaScript [13], and Powershell [14].

When latency or computational resources are problematic, we would like to have an effective, purely static analysis approach for predicting if an unknown JavaScript file is malicious. Three important applications of this work are large-scale, fast webpage, antimalware and email scanning services. Search engine companies often scan large numbers of webpages searching for drive-by downloads. Antimalware companies may scan hundreds of thousands or even millions of unknown files each day. Similarly, large-scale email hosting providers often scan email attachments to identify malicious content. To scan an individual webpage or file, a specially instrumented virtual machine (VM) must first be reset to a default configuration. The webpage or email attachment is then executed, and dynamic analysis is used to determine whether the unknown script makes any changes to the VM. This process is time consuming and can require vast amounts of computing resources for extremely large-scale email services. If a script classifier can be trained to accurately predict that a script attachment is benign based solely on fast static analysis, this could possibly allow search and email service providers to reduce the number of expensive dynamic analysis executions

performed using full instrumented VMs in the cloud. In this study, we focus on identifying malicious JavaScript for a large-scale, production antimalware service. If we can learn an effective model, it may have application on client computers as well.

To address these challenges, ScriptNet employs a sequential, deep learning model for the detection of malicious JavaScript files based solely on static analysis. The deep learning system allows high accuracy even in the presence of obfuscation. We present a novel variant of a sequence learning model which is capable of capturing malicious behavior in any kind of JavaScript file.

Since the system operates directly on the byte representation of characters instead of keywords, it is able to handle the extremely large vocabulary of the entire script instead of detecting only the key API calls [8], [9]. In ScriptNet, a Data Preprocessing module first translates the raw JavaScript files into a vector sequence representation. The Neural Sequential Learning module then applies deep learning methods on the vector sequence to derive a single vector representation of the entire file. In this module, we propose a novel deep learning model model called Pre-Informant Learning (PIL). This model can operate on extremely long sequences and can learn a single vector representation of the input. The next module of ScriptNet, the Sequence Classification Framework, then performs binary classification on the derived vector and generates a probability $p_{m,z}$ of the $z^{th}$ subsequence of the input file being malicious. To handle extremely long JavaScript files, the final probability $p_m$ is chosen as the maximum probability of all of the subsequences in the script. Our models are trained with end-to-end learning where all the model parameters are learned simultaneously taking the JavaScript file directly as the input.

Evaluating the proposed models on a large corpus of 262,200 JavaScript files, we demonstrate that the best performing PIL model offers a true positive rate of 98.10% for the first 60K byte subsequences and 81.66% for the full-length files, at a false positive rate of 0.50%. We also show that the best PIL system was able to discover 3310 malicious JavaScript files which were not known to be malware at the time that the model was trained. We summarize the primary contributions of this paper as follows: 1) A comprehensive definition of a modular system is provided for detecting the malicious nature of JavaScript files using only the raw file content. 2) A novel deep learning model is proposed for learning from extremely long sequences. In addition, an efficient algorithm is proposed to overcome the limited memory of the current generation of GPUs for the processing of arbitrary length files. 3) Strong malware detection results are demonstrated using ScriptNet on a large corpus of JavaScript files collected by hundreds of millions of computers running a production antimalware product. The results show the robustness of ScriptNet on predicting the malicious nature of JavaScript files that were obtained in the future and were not known at the time of training.

| DataSet | Start Date | End Date | Total | Num Malware | % Malware | Num Benign | % Benign |
|---|---|---|---|---|---|---|---|
| Training Files | 09/14/2017 | 12/28/2017 | 151,840 | 126,505 | 83.31 | 25,335 | 16.69 |
| Training Subseq | 09/14/2017 | 12/28/2017 | 592,872 | 552,699 | 93.22 | 40,173 | 6.78 |
| Validation Files | 12/29/2017 | 02/01/2018 | 45,251 | 38,693 | 85.51 | 6,558 | 14.49 |
| Validation Subseq | 12/29/2017 | 02/01/2018 | 286,457 | 270,767 | 94.52 | 15,690 | 5.48 |
| Test Files | 02/02/2018 | 03/03/2018 | 65,109 | 57,037 | 87.60 | 8,072 | 12.40 |
| Test Subseq | 02/02/2018 | 03/03/2018 | 290,428 | 277,120 | 95.42 | 13,308 | 4.58 |
| Total Files | 09/14/2017 | 03/03/2018 | 262,200 | 222,235 | 84.76 | 39,965 | 15.24 |
| Total Subseq | 09/14/2017 | 03/03/2018 | 1,169,757 | 1,100,586 | 94.09 | 69,171 | 5.91 |

TABLE I
DATASET STATISTICS.

## II. DATA COLLECTION AND DATASET GENERATION

Large labeled datasets are required to sufficiently train deep learning systems, and constructing a dataset of malicious and benign scripts for training ScriptNet's models is a challenge. When unknown JavaScript is encountered by the user during normal activity, it is submitted to the antimalware engine for scanning. The datasets for this study were generated from JavaScript files encountered in the wild by Microsoft's Windows Defender antimalware engine which were submitted to its production file collection and processing pipeline.

**Methodology:** Entire JavaScript files are extracted from the incoming flow of files input to the production pipeline. The antimalware engine is the only source of these files in this study which are uploaded from hundreds of millions of end user computers. A user must provide consent (*i.e.*, opt-in) before their file is transmitted to the production cloud environment. In many cases, JavaScript files may be extracted from installer packages or archives which are also processed by the antimalware engine and input to the production pipeline.

**Labels:** Similar to the raw script content, we use labels which are provided by the Windows Defender analysts and other sources, and the labeling process which is used in production cannot be changed for our study.

**Datasets:** As described in Table I, our anti-virus partner provided the full content of 262,200 JavaScript files which contained 222,235 malicious and 39,965 benign scripts. For this research, JavaScript files were subsampled from the production pipeline from September 2017 through March 2018. These JavaScript files were partitioned into training, validation, and test sets containing 151,840, 45,251, and 65,109 samples, respectively, based on the non-overlapping time periods denoted in the table. In the next section, we propose to divide full-length scripts into subsequences of length $T$ (*e.g.*, 60,000) which can be processed on a modern GPU. The result is several subsequences per file, and the statistics for the full subsequence dataset are also provided in Table I.

## III. SCRIPTNET SYSTEM DESIGN AND MODEL

ScriptNet is motivated by the objective of building a system, which can predict the malicious nature of a script by analyzing the file in the absence of any additional information. In this section, we describe the architecture of ScriptNet in detail

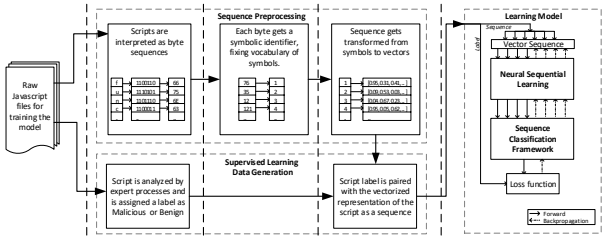


Fig. 1. ScriptNet system architecture for the training phase.

which is designed to achieve this objective. The ScriptNet system is comprised of multiple modules banded together in a specific order. The high-level illustration of the ScriptNet system is shown in Figure 1. In this paper, we present a new model for sequence learning called Pre-Informant Learning (PIL). We present the PIL model, and its simpler Convoluted Partitioning of Long Sequences (CPoLS) variant [15], in Figure 2. The figure includes descriptions which indicate the purpose of each of the individual layers.

**Data Preprocessing:** The first stage of ScriptNet is to process the raw file data and prepare it for utilization by a deep learning model. In their raw form, the script files are simply text files written using readable characters. The content inside script files is in the form of programming code. This means that the text includes operators, variable names, and other syntactical properties. In natural language, the semantic meaning of a certain word is limited to a small space. Whereas in programming code, words cannot be directly mapped to a limited semantic space. Moreover, the number of words and operators can grow infinitely as variable names do not need to follow any linguistic limitations. Therefore, we need to represent the scripts at a much finer level than using words. We achieve this by interpreting the script files as byte sequences. Using this method to read the files, we limit the space of possible options to the number of different bytes, i.e., $(2^8) = 256$.

For the system to clearly identify the different bytes, we need to provide them unique identifiers. At this stage, therefore, we create an index to map each byte with a symbolic identifier. Following the convention in deep learning models, we refer to this index as the vocabulary $V$ for the model. Using this vocabulary, we can now transform the input file into a sequence usable by the learning model. At first, by reading the text as bytes, we get a byte sequence. Next, we perform a lookup through the vocabulary index and represent each byte with its symbolic representation. We refer to the derived sequence as $B$ in Figure 2, where $B = [b_1, b_2, b_3, \ldots] \ \forall b_i \in V$ denotes a sequence of symbols $b_i$ each of which is identified in our vocabulary $V$. To increase accuracy and speed up processing, we next split the sequence $B$ into chunks labeled $C$ in Figure 2.

For learning purposes, it is possible to directly use this symbolic representation. However, symbols serve information at a very low level of dimensionality. When represented as symbols, any similarity between two kinds of bytes cannot be directly identified. In neural networks, the concept of representations, or 'embeddings' is extensively used for this purpose. By representing symbols with embedding vectors labeled as $E$ in Figure 2, we can increase the dimensionality of the information associated with each element. These vectors can be learned from the data itself. The distance between these vectors also serves as a measure of semantic similarity. In our case, we use this concept of representations and transform the symbolic sequence into a sequence of vectors. The initial value of these vectors is randomly selected using initialization methods by Glorot and Bengio [16]. During the training phase, the vectors are updated along with the model.

**Neural Sequential Learning:** In our preprocessing phase, we converted the input files into vector sequences. Since the lengths of these files can vary, the derived sequences are also of different length. General learning methods based on feature vectors read fixed-length vectors as input and operate on them. For our case with more complex-shaped data, we need to use modules that can process two-dimensional input data. Therefore, to learn from sequences, we use advanced neural network architectures like Recurrent Neural Networks (RNNs) and Convolutional Neural Networks (CNNs) [17].

In our models, we use a memory-based variant of RNNs known as Long Short-Term Memory (LSTM) [18], [19] neural networks. CNNs have also recently shown success in sequential learning [20], [21] and are much more efficient than RNNs. In the RECURRENTCONVOLUTIONS block in Figure 2, a multi-layer CNN performs a convolution over a sliding window of smaller partitions within the input of embedding vectors and produces sequence $E'$.

The Pre-Informant Learning PIL model is an extension of the CPoLS model [15] where we try to make use of key information available within the processing of CPoLS by adding a pre-informant layer in Figure 2. While operating over individual subsequences using RECURRENTCONVOLUTIONS in step 4, we derive a vector representation of the subsequence. Since the malicious nature of the JavaScript file can be hidden in any of the subsequences, we can use this vector representation as an early indicator of information. Therefore, after step 5 of the PIL model, we perform some additional steps before sending the sequence $E'$ to the LSTM-based learning. For each vector $e'_i \in E'$, we use a neural sigmoid layer. A *sigmoid* layer refers to a fully connected neural layer which takes an input vector $X$ and produces a scalar output $y \in (0, 1)$, using $y = \sigma(\mathbf{W} * X + \mathbf{b})$ where $\mathbf{W}$ is the weight matrix, and $\mathbf{b}$ is the bias in the layer. $\sigma$ is a non-linear activation function. In this case, we use the logistic sigmoid function, $\sigma(x) = 1/(1 + e^{-x})$.

**Step 5P-1**: For each vector $e'_i$, we pass it through the sigmoid layer and derive a pre-informant scalar $y_i$.

**Step 5P-2**: In the overall end-to-end model, we use this output $y_i$ as an auxiliary output. Therefore, the model now generates auxiliary outputs $y_i \forall e'_i \in E'$. By using these auxiliary outputs, we can train the pre-informant by two

sources (final output and auxiliary output) and can add implicit regularization to the model.

**Step 5P-3**: This pre-informant scalar $y_i$ can now be used further in the model as an additional feature holding information on each subsequence. We next concatenate this scalar $y_i$ to the subsequence vector $e'_i$ to derive $e''_i \forall (e'_i, y_i) \in E'$.

**Step 5P-C**: These vectors $e''_i$ are finally combined again in order to generate a vector sequence $E''$ which can be processed through the standard sequence learning.

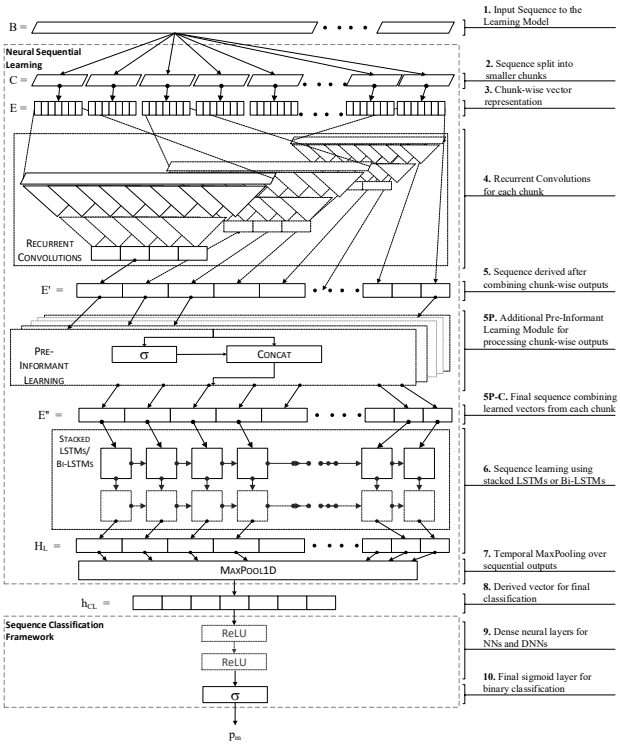


Fig. 2. Pre-Informant Learning Model Overview.

We now obtain a reduced-length sequence of vectors $E''$. This sequence can now be processed using a standard sequence learning approach. We, therefore, next pass this sequence through an LSTM. For an input sequence $E'$ of length $n$, this layer produces a learned sequence $H_L$ of length $n$ but with a different fixed dimensionality.

For detecting malware, we want to obtain the important malicious signal information within the sequence $H_L$. An effective method for such cases is the use of temporal max pooling, MaxPool1D, as proposed by Pascanu *et al.* [12]. Given an input vector sequence $S = [s_0, s_1, \dots s_{M-1}] \in S$ of length $M$, where each vector $s_i \in \mathbb{R}^K$ is a $K$-dimensional vector, MaxPool1D computes an output vector $s_{MP} \in \mathbb{R}^K$ as $s_{MP}(k) = \max(s_0(k), s_1(k), \cdots s_{M-1}(k)) \forall k \in K$. The vector $s_{MP}$, therefore, for each dimension, contains the maximum value observed in the sequence for that dimension. At this stage, we pass the sequence $H_L$ through MaxPool1D to obtain the final vector $h_{CL}$.

**Sequence Classification Framework:** The output vector $h_{CL}$ from the Neural Sequential Learning module, like PIL, can be used by any classifier for performing binary classification. The simplest such model can be a logistic regression model that uses $h_{CL}$ and derives a probability of maliciousness $p_m$. We can even use complex models, such as feed-forward neural networks, or deeper neural networks (DNNs) with multiple layers for the same purpose. For our experiments, we use DNNs with the Rectified Linear Unit (ReLU) activation function, which is defined as $f(x) = max(0, x)$ where $f$ represents the ReLU function on an input $x$.

**Learning Phase:** The modules described above create the complete ScriptNet system. We use gradient descent-based methods to train our models. In such methods, a loss $\mathcal{L}$ is measured by comparing the prediction $p_m$ generated by the learning model and the available ground truth label $\tau$. This loss is then used to update the coefficients (*i.e.*, weights) of the model. For our objective of binary classification, we use the *binary cross-entropy* loss function, which is defined as $\mathcal{L} = -(\tau log(p_m) + (1-\tau) log(1-p_m))$ where $\tau$ is the known ground truth, $p_m$ is the predicted probability of maliciousness, and $\log$ is the natural logarithmic function.

**End-to-End Learning:** Due to the modular nature of our system, we have the freedom to train it in different ways. While we present neural models in this paper, the system can also use different components from machine learning. For instance, in the Sequence Classification Framework, we can ideally use any classifier like an Support Vector Machine or Naive Bayes, which may or may not support gradient-based updates.

By keeping our models in the realm of neural networks, we are also able to utilize the concept of end-to-end learning. This means that our system can train itself completely by just using the input files and labels. We do not need to train different modules individually in such a setting. The results presented in this paper were trained using end-to-end models.

**Processing Full File Content:** Up to this point, we have only processed the first $S$ bytes of the JavaScript file where $S$ is chosen (*e.g.* 60 kilobytes) such that the model fits in the on-board memory of the GPU. If the length of the JavaScript file is shorter than $S$, the file is processed normally. However, in some cases, the JavaScript file is longer than $S$ bytes, and we must further improve our model to prevent attackers from simply appending the malicious content to the end of the script to avoid detection. To handle longer files, we further divide the entire JavaScript file into subsequences of $S$ bytes, and each of these subsequences is evaluated by the model separately. The predicted score for the entire file is then found by taking the maximum score for each of these $Z$ subsequences $p_m = max_{z \in Z}(p_{m,z})$.

## IV. Experimental Results

In this section, we evaluate the performance of the proposed ScriptNet classifier models on the JavaScript files described in Section II.

| Model | Parameter | Description | Value |
|---|---|---|---|
| PIT | $R_{PIL}$ | Chunk Length | 200 |
| PIL | $T_{PIL}$ | Subsequence Length | 60,000 |
| PIL | $B_{PIL}$ | Minibatch Size | 50 |
| PIL | $H_{PIL}$ | LSTM Hidden Layer Size | 250 |
| PIL | $E_{PIL}$ | Embedding Layer Size | 100 |
| PIL | $W_{PIL}$ | CNN Window Size | 10 |
| PIL | $S_{PIL}$ | CNN Window Stride | 5 |
| PIL | $F_{PIL}$ | Number of CNN Filters | 100 |
| PIL | $D_{PIL}$ | Dropout Ratio | 0.5 |
| LaMP | $T_{LaMP}$ | Maximum Sequence Length | 200 |
| LaMP | $B_{LaMP}$ | Minibatch Size | 200 |
| LaMP | $H_{LaMP}$ | LSTM Hidden Layer Size | 1500 |
| LaMP | $E_{LaMP}$ | Embedding Layer Size | 50 |
| LaMP | $D_{LaMP}$ | Dropout Ratio | 0.5 |

TABLE II
HYPERPARAMETER SETTINGS FOR THE VARIOUS MODELS. THE HYPERPARAMETER SETTINGS FOR THE CPOLS VARIANT ARE IDENTICAL TO THE PIL MODEL.

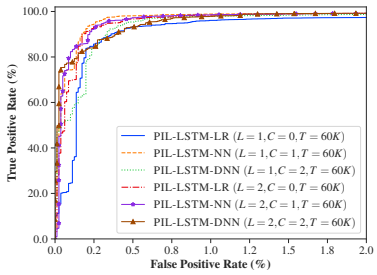


Fig. 3. ROC curves for different JavaScript PIL models zoomed into a maximum FPR = 2%.

**Experimental Setup:** All the experiments are written in the Python programming language using the Keras [22] deep learning library with TensorFlow [23] as the backend deep learning framework. The models are trained and evaluated on a cluster of NVIDIA P100 graphical processing unit (GPU) cards. The input vocabulary size is set to 257 since the sequential input consumed by each model is a byte stream, and an additional symbol is used for padding shorter sequences within each minibatch. All models are trained using a maximum of 15 epochs, but early stopping is employed if the model fully converges before reaching the maximum number of epochs. The Adam optimizer [24] is used to train all models.

We did hyperparameter tuning of the various input parameters for the JavaScript models, and the final settings are summarized in Table II. With these settings, we evaluate the classification error rate on the test set for the JavaScript dataset.

The PIL model and its CPoLS variant are designed to operate on the full JavaScript sequences. However, training on the full-length sequences exhausts the memory capacity of the NVIDIA P100s in our cluster, depending on the particular variant and parameter settings of the model. To overcome this limitation, we truncated the sequence length to $T$ = 60,000 bytes for all the PIL and CPoLS experiments with the exception of the full-length experiments. Similarly, we truncated the sequences to lengths of $T \in \{200, 1000\}$ bytes for the LaMP and Kolosnjaji CNN [11] baseline models.

**Pre-Informant Learning and CPoLS Models:** We first evaluate the performance of the PIL and CPoLS models. Their common performance metrics, along with the metrics of all the other models, are summarized in Table III. These performance metrics include the accuracy, precision, recall, F1 score, and the area under the receiver operating characteristic (ROC) curve (AUC). The table indicates that, in general, most of the models perform reasonably well, although some models clearly outperform others.

The ROC curves for the Pre-Informant Learning models with several different combinations of LSTM stacked layers $L_{PIL}$ and classifier hidden layers $C_{PIL}$, are depicted in Figure 3. The results are reported for a single forward LSTM layer ($L_{PIL} = 1$) and two stacked LSTM layers ($L_{PIL}$=2). Similarly, we investigate models with $C_{PIL} = 0$ (Logistic Regression (LR)), $C_{PIL} = 1$ (Shallow Neural Network (NN)), and $C_{PIL} = 2$ (Deep Neural Network (DNN)) classifiers. The variants with $C_{PIL} = 1$ are denoted as PIL-LSTM-NN where NN refers to a shallow neural network. Similarly, the PIL-LSTM-DNN models include $C_{PIL} = 2$ hidden layers in the deep neural network (DNN) stage. If the classifier stage does not use any hidden layers ($C_{PIL} = 0$), the classifier is simple logistic regression and the model is denoted as PIL-LSTM-LR. Even with the truncated JavaScript file sequences, all of the models approximate an ideal classifier.

Since it is difficult to evaluate the performance of the different models from the full ROC curves, we provide zoomed-in versions with a maximum FPR of 2% in Figure 3. All subsequent ROC curves are also zoomed-in and have a maximum FPR = 2%. Above a false positive rate (FPR) of 0.15%, the best performing PIL model utilizes a single LSTM layer and single classifier hidden layer, $L_{PIL} = 1, C_{PIL} = 1$. This result has several benefits. Since the model has a fixed size, increasing the number of layers can often lead to overfitting the learned parameters in the model, leading to performance degradation on model evaluation. Single layer models also help limit the number of parameters of the PIL model and make it faster and more compact for deployment at scale.

**Baselines:** We now compare the performance results of the best performing PIL and CPoLS models to a number of baseline systems summarized in Tables III. The LaMP model originally proposed in [10] for Windows PE files is evaluated for this new task of detecting malicious JavaScript. Table III indicates that we evaluated six variants of the LaMP architecture in ScriptNet. Similarly, we implemented the sequential CNN model proposed in [11], and denoted as KOL-CNN, which is adapted for the new task of detecting malicious JavaScript. Like [10], this sequential KOL-CNN model was proposed to detect Windows PE files. We also re-implemented the logistic regresion model (SDA-LR) [25] which uses autoencoders to detect malicious JavaScript. We also compare against trigrams of byte using logistic regression (LR-Trigram) and a support vector machine (SVM-Trigram) as proposed in [6]. Naive Bayes with trigrams is also considered since Zozzle [2] used Naive Bayes for its classifier.

| Model | Accuracy (%) | Precision (%) | Recall (%) | F1 | AUC |
|---|---|---|---|---|---|
| PIL-LSTM-LR ($L = 1, C = 0, T = 60K$) | 98.3395 | 98.4515 | 99.6721 | 0.9906 | 0.9968 |
| PIL-LSTM-NN ($L = 1, C = 1, T = 60K$) | 98.6236 | 98.5739 | 99.8737 | 0.9922 | 0.9988 |
| PIL-LSTM-DNN ($L = 1, C = 2, T = 60K$) | 99.0092 | 99.1269 | 99.7475 | 0.9944 | 0.9980 |
| PIL-LSTM-LR ($L = 2, C = 0, T = 60K$) | 98.8372 | 98.9373 | 99.7440 | 0.9934 | 0.9984 |
| PIL-LSTM-NN ($L = 2, C = 1, T = 60K$) | 98.8541 | 98.8899 | 99.8124 | 0.9935 | 0.9986 |
| PIL-LSTM-DNN ($L = 2, C = 2, T = 60K$) | 99.0799 | 99.2993 | 99.6528 | 0.9948 | 0.9983 |
| PIL-LSTM-NN ($L = 1, C = 1, R = 200, T = Full$) | 97.4288 | 97.5455 | 99.5704 | 0.9855 | 0.9916 |
| PIL-LSTM-NN ($L = 1, C = 1, R = 50, T = Full$) | 96.7315 | 96.6968 | 99.6739 | 0.9816 | 0.9922 |
| PIL-BiLSTM-LR ($L = 1, C = 0, T = 60K$) | 99.0399 | 99.1649 | 99.7440 | 0.9945 | 0.9979 |
| PIL-BiLSTM-NN ($L = 1, C = 1, T = 60K$) | 99.4577 | 99.6896 | 99.6914 | 0.9969 | 0.9984 |
| PIL-BiLSTM-DNN ($L = 1, C = 2, T = 60K$) | 97.5914 | 97.3677 | 99.9527 | 0.9864 | 0.9977 |
| PIL-BiLSTM-LR ($L = 2, C = 0, T = 60K$) | 98.6774 | 98.6556 | 99.8510 | 0.9925 | 0.9979 |
| PIL-BiLSTM-NN ($L = 2, C = 1, T = 60K$) | 99.3241 | 99.5187 | 99.7107 | 0.9961 | 0.9989 |
| PIL-BiLSTM-DNN ($L = 2, C = 2, T = 60K$) | 99.0937 | 99.1500 | 99.8211 | 0.9948 | 0.9985 |
| CPoLS-LSTM-LR ($L = 1, C = 0, T = 60K$) | 98.8725 | 98.9990 | 99.7212 | 0.9936 | 0.9985 |
| CPoLS-LSTM-NN ($L = 1, C = 1, T = 60K$) | 98.1997 | 98.2232 | 99.7492 | 0.9898 | 0.9937 |
| CPoLS-LSTM-DNN ($L = 1, C = 2, T = 60K$) | 98.1966 | 98.1135 | 99.8615 | 0.9898 | 0.9964 |
| CPoLS-LSTM-LR ($L = 2, C = 0, T = 60K$) | 99.0399 | 99.3888 | 99.5160 | 0.9945 | 0.9975 |
| CPoLS-LSTM-NN ($L = 2, C = 1, T = 60K$) | 98.9708 | 99.1626 | 99.6668 | 0.9941 | 0.9984 |
| CPoLS-LSTM-DNN ($L = 2, C = 2, T = 60K$) | 98.8771 | 99.1909 | 99.5301 | 0.9936 | 0.9981 |
| CPoLS-BiLSTM-LR ($L = 1, C = 0, T = 60K$) | 98.5683 | 98.5394 | 99.8457 | 0.9919 | 0.9967 |
| CPoLS-BiLSTM-NN ($L = 1, C = 1, T = 60K$) | 98.6928 | 98.7318 | 99.7896 | 0.9926 | 0.9956 |
| CPoLS-BiLSTM-DNN ($L = 1, C = 2, T = 60K$) | 98.7035 | 98.8149 | 99.7159 | 0.9926 | 0.9969 |
| CPoLS-BiLSTM-LR ($L = 2, C = 0, T = 60K$) | 98.7988 | 98.8773 | 99.7615 | 0.9932 | 0.9982 |
| CPoLS-BiLSTM-NN ($L = 2, C = 1, T = 60K$) | 99.1398 | 99.2981 | 99.7229 | 0.9951 | 0.9986 |
| CPoLS-BiLSTM-DNN ($L = 2, C = 2, T = 60K$) | 97.5530 | 97.4753 | 99.7913 | 0.9862 | 0.9969 |
| LaMP-LSTM-LR ($L = 1, C = 0, T = 200$) | 95.9861 | 96.6608 | 98.8321 | 0.9773 | 0.9766 |
| LaMP-LSTM-NN ($L = 1, C = 1, T = 200$) | 97.0138 | 96.9490 | 99.7295 | 0.9832 | 0.9892 |
| LaMP-LSTM-DNN ($L = 1, C = 2, T = 200$) | 96.3953 | 96.4409 | 99.5592 | 0.9798 | 0.9873 |
| LaMP-LSTM-LR ($L = 2, C = 0, T = 200$) | 87.5983 | 87.5983 | 100.0000 | 0.9339 | 0.5000 |
| LaMP-LSTM-NN ($L = 2, C = 1, T = 200$) | 94.1814 | 96.0273 | 97.3866 | 0.9670 | 0.9500 |
| LaMP-LSTM-DNN ($L = 2, C = 2, T = 200$) | 96.1169 | 97.8491 | 97.7151 | 0.9778 | 0.9748 |
| SdA-LR ($T = 2000$) | 87.6020 | 87.6020 | 100.0000 | 0.9339 | 0.5012 |
| KOL-CNN ($T = 200$) | 97.0753 | 97.4956 | 99.2097 | 0.9835 | 0.9853 |
| KOL-CNN ($T = 1000$) | 96.7446 | 96.8356 | 99.5363 | 0.9817 | 0.9851 |
| LR - Trigram ($T = 60K$) | 97.5975 | 97.9394 | 99.3477 | 0.9864 | 0.9237 |
| SVM - Trigram ($T = 60K$) | 97.5560 | 97.7683 | 99.4810 | 0.9862 | 0.9185 |
| NB - Trigram ($T = 60K$) | 97.5560 | 97.7683 | 99.4810 | 0.9862 | 0.9185 |

TABLE III
PERFORMANCE OF THE VARIOUS MODELS WHICH WERE EVALUATED FOR THIS STUDY.

None of these models are designed to process very long sequences. In fact, we tried to implement the LaMP models with length $T$ = 1000 JavaScript bytes, but all those experiments generated out of memory exceptions. We were able to process KOL-CNN with length $T$ = 1000 sequences. We were also able to process length $T$ = 2000 sequences with SDA-LR.

The ROC curves for the best performing PIL and CPoLS models and some of the baseline models are presented in Figure 4. We did not plot the results of the SDA-LR model because it predicted that all the JavaScript files in the test set were malicious for a number of variants that we explored. Overall, the PIL-LSTM-NN model offers the best performance. However, the CPoLS-LSTM-NN variant does provide a slightly better TPR for extremely low values of FPR. The PIL-BiLSTM-NN is also competitive, but does not offer significantly better detection capabilities and requires more computational resources to evaluate the reverse direction LSTM during inference.

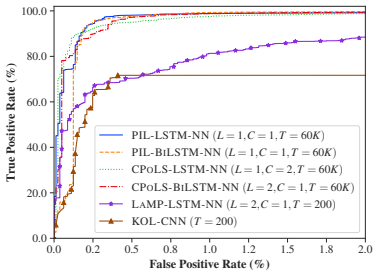


Fig. 4. ROC curves for different JavaScript models zoomed into a maximum FPR = 2%.

**Evaluation on Full Script Content:** The previous performance results are based on the evaluating only the first 60 kilobyte subsequence of the JS file. We compare the results of the best, 60K PIL model to the those of two variants of the proposed full script model which returns the maximum prediction for all of the 60K subsequences in the file in Figure 5. The first variant uses $R = 200$ length chunks in each 60K subsequence which matches this parameter's setting for the 60K PIL model and has 494,230 model parameters in total. We also plot the results for full-length model with $R = 50$ length chunks consisting of 70,105 parameters. While the results for the full-length models are not as good as the first subsequence classifier, the full-length models do prevent the attacker from simply embedding the malicious JavaScript at the end of the file. Comparing the two full-length models, $R = 200$ variant offers better results at very low FPRs but a slightly higher TPRs for FPRs greater that 0.6%.

**Evaluation on Unknown Malware:** We next evaluate how

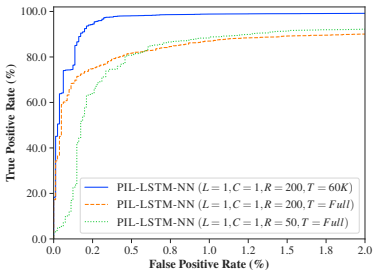


Fig. 5. Comparison of ROC curves, zoomed into a maximum FPR = 2%, for the best single subsequence PIL classifier model and the same model evaluated on the full file content.

well the best performing PIL model can predict unknown malware in the future. To do so, our anti-virus partners provided us with 3,597 JavaScript files which were not detected by their product at the start of the testing phase on February 2, 2018, when the models were trained. All of these files were later confirmed to be malicious. In order to be correctly predicted as malware, the inference score $S$ on the JavaScript files must be higher than the detection threshold ($S \geq T_c$). After tuning $T_c$, 3310 (92.02%) of these files had an inference score which would have allowed the model to correctly predict that these files are indeed malicious.

**Analysis of FPs and FNs:** After training, we manually investigated the false positives (FPs) and false negatives (FNs) that were predicted by the model. We next summarize an investigation of the top 20 FPs and FNs by a professional virus researcher.

Of the FPs, seven files were outputs of a specific commercial packer. One file appeared to be a corrupted browser cache file with some JavaScript but nothing was clearly malicious. Three examples were related to adversiting code where two were identical and one had light obfuscation. Several files were mistakenly labeled by the antimalware engine as being JavaScript, but they did not include JavaScript. Two of these were game binaries with another embedded script, and two were Perl files. Two appeared to be benign JavaScript files where one was very long and the other implemented calendar functionality. Three JavaScript files were incorrectly labeled and were malcious (*i.e.*, true positives).

Like the FPs, the FNs also fell into several different groups. Two of the files were identical auto-likers which were trying to get the user to automatically like a Facebook post. Several of the samples consisted mostly of other languages but with a small amount of malicious embedded JavaScript code. For example, three samples were identical, but random PDFs which included the same small snippet of embedded JavaScript. Several files were mostly HTML or PHP source with either one JavaScript line added to embed a malicious iFrame or a small amount of packed JavaScript appended to the end of the file. Other files were either C++ or C sharp source files which appeared to be used to generate exploits that included that included a small amount of JavaScript. Four samples were packed malicious JavaScript. One file was minified with a small malicious snippet. One of the examples appeared to be a labeling issue and was a benign JavaScript (*i.e.*, true negative).

**Computation Times:** The time required to complete the training and test phases are reasonable on a GPU card. For example, the best performing PIL-LSTM-NN model was trained on the first 60,000 bytes in 17 hours, 44 minutes, and 45 seconds on an NVIDIA P100 in our cluster. For the full-length scripts, training on all of the subsequences required 4 days, 22 hours, and 9 minutes. Similarly, the test phase completed in 45 minutes and 55 seconds, with a minibatch size of 50 for an average of 33.09 milliseconds per file on the same GPU and cluster configuration.

## V. Related Work

**JavaScript:** Maiorca *et al.* [9] propose a static analysis-based system to detect malicious PDF files which use features constructed from both the content of the PDF, including JavaScript, as well as its structure. Cova *et al.* [26] use the approach of anomaly detection for detecting malicious JavaScript code. In [5], Likarish *et al.* classify obfuscated malicious JavaScript using several different types of classifiers including Naive Bayes, an Alternating Decision Tree (ADTree), a Support Vector Machine (SVM) with using the Radial Basis Function (RBF) kernel, and the rule-based Ripper algorithm. A PDF classifier proposed by Laskov and Šrndić [8] uses a one-class SVM to detect malicious PDFs which contain JavaScript code. Zozzle [2] proposes a mostly static approach extracting contexts from the original JavaScript file. The system parses these contexts to recover the abstract syntax trees (ASTs). A Naive Bayes classifier is then trained on the features extracted from the variables and keyword found in the ASTs. Corona *et al.* [7], propose Lux0R, a system to select API references for the detection of malicious JavaScript in PDF documents. The features are then classified with an SVM, a Decision Tree and a Random Forest model. Wang *et al.* [13] use deep learning models in combination with sparse random projections and logistic regression to detect malicious JavaScript. They also present feature extraction from JavaScript code using auto-encoders. While they use deep learning models, the feature extraction and model architectures limit the information extractability from JavaScript code. Shah [6] propose using a statistical n-gram language model to detect malicious JavaScript. Our proposed system uses an LSTM neural model for the language model instead of the n-gram model proposed by Shah [6].

**Other File Types:** While more research has been devoted to detecting malicious JavaScript, partly because of its inclusion in malicious PDFs, only a few previous studies have considered malicious VBScript. In [27], a conceptual graph is first computed for VBScript files, and new malware is detected by identifying graphs which are similar to those of known malicious VBScript files. The method is based on static analysis of the VBScripts.

A number of deep learning models have been proposed for detecting malicious PE files including [10]–[12], [28], [29]. Raff *et al.* [30] discuss a model which is similar to CPoLS but noted it did not work for PE files. They did not provide any results for their model.

## VI. Conclusions

Malicious JavaScript detection is an important problem facing anti-virus companies. Neural language models have shown promising results in the detection of malicious executable files. Similarly, we show that these types of models can also detect malicious JavaScript files, in the proposed ScriptNet system, with very high true positive rates at extremely low false positive rates.

The performance results confirm that the PIL model using CNN, pre-informant, and LSTM neural layers is able to learn and generate representations of byte sequences in the JavaScript files. In particular, the PIL JavaScript malware script classification model using a single LSTM layer and a shallow neural network layer offers the best results. Therefore, the vector representations generated by these models capture important sequential information from the JavaScript files. ScriptNet extracts and uses this information to predict the malicious intent of these files.

## Acknowledgment

The authors thank Marc Marino, Jugal Parikh, Daewoo Chong, Mikael Figueroa and Arun Gururajan for providing the data and helpful discussions.

## References

[1] Mozilla, "JavaScript." [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/JavaScript

[2] C. Curtsinger, B. Livshits, B. Zorn, and C. Seifert, "Zozzle: Fast and precise in-browser javascript malware detection," in *Proceedings of Usenix Security*, 2011.

[3] W. Xu, F. Zhang, and S. Zhu, "Jstill: Mostly static detection of obfuscated malicious javascript code," in *Proceedings of the Third ACM Conference on Data and Application Security and Privacy*, ser. CODASPY '13. New York, NY, USA: ACM, 2013, pp. 117–128.

[4] E. Gandotra, D. Bansal, and S. Sofat, "Malware analysis and classification: A survey," pp. 55–64, 2014.

[5] P. Likarish, E. Jung, and I. Jo, "Obfuscated malicious javascript detection using classification techniques," in *2009 4th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, oct 2009, pp. 47–54. [Online]. Available: http://ieeexplore.ieee.org/document/5403020/

[6] A. Shah, "Malicious JavaScript Detection using Statistical Language Model," *Master's Projects*, p. 70, 2016. [Online]. Available: http://scholarworks.sjsu.edu/etd_projects/476

[7] I. Corona, D. Maiorca, D. Ariu, and G. Giacinto, "Lux0r: Detection of malicious pdf-embedded javascript code through discriminant analysis of api references," in *Proceedings of the 2014 Workshop on Artificial Intelligent and Security Workshop*, ser. AISec '14. New York, NY, USA: ACM, 2014, pp. 47–57.

[8] P. Laskov and N. Šrndić, "Static detection of malicious javascript-bearing pdf documents," in *Proceedings of the 27th Annual Computer Security Applications Conference*, ser. ACSAC '11. New York, NY, USA: ACM, 2011, pp. 373–382.

[9] D. Maiorca, D. Ariu, I. Corona, and G. Giacinto, "A structural and content-based approach for a precise and robust detection of malicious pdf files," in *Proceedings of the International Conference on Information Systems Security and Privacy (ICISSP)*, 2015.

[10] B. Athiwaratkun and J. W. Stokes, "Malware classification with lstm and gru language models and a character-level cnn," in *2017 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, March 2017, pp. 2482–2486.

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

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

[13] Y. Wang, W. dong Cai, and P. cheng Wei, "A deep learning approach for detecting malicious javascript code," *Proceedings of Security and Communication Networks*, vol. 11, no. 9, pp. 1520–1534, 2016.

[14] D. Hendler, S. Kels, and A. Rubin, "Detecting Malicious PowerShell Commands using Deep Neural Networks," *Proceedings of the Asia Conference on Computer and Communications Security*, 2018.

[15] R. Agrawal, J. W. Stokes, M. Marinescu, and K. Selvaraj, "Robust neural malware detection models for emulation sequence learning," in *Proceedings of the Military Communications Conference (MILCOM)*, 2018.

[16] X. Glorot and Y. Bengio, "Understanding the difficulty of training deep feedforward neural networks," in *Proceedings of the Thirteenth International Conference on Artificial Intelligence and Statistics*, ser. Proceedings of Machine Learning Research, Y. W. Teh and M. Titterington, Eds., vol. 9. Chia Laguna Resort, Sardinia, Italy: PMLR, 13–15 May 2010, pp. 249–256. [Online]. Available: http://proceedings.mlr.press/v9/glorot10a.html

[17] Y. LeCun and Y. Bengio, "Convolutional networks for images speech and time series," 1995.

[18] S. Hochreiter and J. Schmidhuber, "Long short-term memory," *Neural Computation*, vol. 9, no. 8, pp. 1–32, 1997.

[19] F. A. Gers, J. Schmidhuber, and F. A. Cummins, "Learning to forget: Continual prediction with LSTM," *Neural Computation*, vol. 12, no. 10, pp. 2451–2471, 2000.

[20] J. Gehring, M. Auli, D. Grangier, and Y. N. Dauphin, "A convolutional encoder model for neural machine translation," *CoRR*, vol. abs/1611.02344, 2016. [Online]. Available: http://arxiv.org/abs/1611.02344

[21] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, "Convolutional sequence to sequence learning," *CoRR*, vol. abs/1705.03122, 2017. [Online]. Available: http://arxiv.org/abs/1705.03122

[22] F. Chollet *et al.*, "Keras," https://github.com/fchollet/keras, 2015.

[23] M. Abadi and et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[24] D. P. Kingma and J. Ba, "Adam: A Method for Stochastic Optimization," dec 2014. [Online]. Available: http://arxiv.org/abs/1412.6980

[25] Y. Wang, W.-d. Cai, and P.-c. Wei, "A deep learning approach for detecting malicious javascript code," *Security and Communication Networks*, vol. 9, no. 11, pp. 1520–1534. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/sec.1441

[26] M. Cova, C. Kruegel, and G. Vigna, "Detection and analysis of drive-by-download attacks and malicious javascript code," in *Proceedings of the 19th International Conference on World Wide Web*, ser. WWW '10. New York, NY, USA: ACM, 2010, pp. 281–290.

[27] S. Kim, C. Choi, J. Choi, P. Kim, and H. Kim, "A method for efficient malicious code detection based on conceptual similarity," in *International Conference on Computational Science and Its Applications (ICCSA)*, vol. 3983, 2006, pp. 567–576.

[28] G. E. Dahl, J. W. Stokes, L. Deng, and D. Yu, "Large-scale malware classification using random projections and neural networks," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*, 2013.

[29] W. Huang and J. W. Stokes, "Mtnet: A multi-task neural network for dynamic malware classfication," in *Proceedings of Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA)*, 2016, pp. 399–418.

[30] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro, and C. Nicholas, "Malware Detection by Eating a Whole EXE," *ArXiv e-prints*, 2017.