Robust Neural Malware Detection Models for Emulation Sequence Learning

Rakshit Agrawal[⋆], Jack W. Stokes[†], Mady Marinescu[‡], and Karthik Selvaraj[‡]
[⋆]University of California at Santa Cruz, Santa Cruz, CA 95064 USA
[†]Microsoft Research, One Microsoft Way, Redmond, WA 98052 USA
[‡]Microsoft Corp., One Microsoft Way, Redmond, WA 98052 USA

***Abstract*—Malicious software, or malware, presents a continuously evolving challenge in computer security. These embedded snippets of code in the form of malicious files or hidden within legitimate files cause a major risk to systems with their ability to run malicious command sequences. Malware authors even use polymorphism to reorder these commands and create several malicious variations. However, if executed in a secure environment, one can perform early malware detection on emulated command sequences.**

The models presented in this paper leverage this sequential data derived via emulation in order to perform Neural Malware Detection. These models target the core of the malicious operation by learning the presence and pattern of co-occurrence of malicious event actions from within these sequences. Our models can capture entire event sequences and be trained directly using the known target labels. These end-to-end learning models are powered by two commonly used structures - Long Short-Term Memory (LSTM) Networks and Convolutional Neural Networks (CNNs). Previously proposed sequential malware classification models process no more than 200 events. Attackers can evade detection by delaying any malicious activity beyond the beginning of the file. We present specialized models that can handle extremely long sequences while successfully performing malware detection in an efficient way. We present an implementation of the Convoluted Partitioning of Long Sequences approach in order to tackle this vulnerability and operate on long sequences. We present our results on a large dataset consisting of 634,249 file sequences, with extremely long file sequences.

***Index Terms*—Malware detection, Neural models, LSTM, Convoluted Partitioning of Long Sequences**

I. Introduction

Malicious software, or malware, is a persistent and continuously growing problem in computer security. Malware can cause severe issues for computer users. By embedding certain snippets of code within benign software, it can successfully run malicious commands without being detected by anti-virus software. This execution can be further modified by malware authors as they can use polymorphism to reorder malicious commands within different files. While ensuring coverage of commands relevant to malware, they can hide within large sets of legitimate commands and remain undetected. Before the file is allowed to run on the native operating system, the anti-malware engine first analyzes it using lightweight emulation. This task of detecting malware within the emulation, however, is extremely difficult. With continuously increasing number of subtle variations observed in malware, simply constructing sets of rules for detection can become obsolete very quickly. But an underlying fact for any variant is its dependence on the sequence of commands it needs to operate. While those commands can be spread out within the execution, they cannot be eliminated or invariably reordered. The nature of their co-occurrence in sequence or in some patterns is still essential for these variants to operate.

Machine learning models can be trained to learn both the presence and sequential occurrence of events leading to malicious actions. Neural networks for sequential learning like Long Short-Term Memory or LSTM [1], [2] recurrent neural networks excel at tasks of learning from sequences with a fixed vocabulary. They have shown exemplary results over the past few years in major sequential learning domains like speech [3] and natural language [4], [5]. Besides these, LSTMs have shown significant success in a much broader range of sequential problems as well. Pointer Networks [6], Neural Turing Machines and Differentiable Neural Computers [7], [8] have demonstrated the ability of LSTMs to be used for far more complex tasks when augmented with attention [9] and memory [10].

AI-based learning models for malware detection, therefore, can be constructed with the help of LSTMs when event sequences are available. The use of language models with Recurrent Neural Networks (RNNs) has been demonstrated by [11], [12] on malware detection tasks. The two primary objectives in such problems are *event presence detection* and *sequential occurrence binding*. LSTMs, in particular, are excellent operators for sequential occurrences, and hence lead to significantly improved results in the process. Presence detection can be performed efficiently using the Convolutional Neural Networks, or CNNs [13]. While extremely effective in computer vision [14], [15], CNNs have also shown success on sequential learning problems [16], [17]. For the closely related task of multi-class classification among malware families, the authors in [18] have shown the use of CNNs as they capture essential elements across the event sequences.

In this paper, we present robust models for Neural Malware Detection that can operate directly on emulated file event sequences in order to learn a probability of the file being malicious in nature. These models are probabilistic, robust and resilient against polymorphism observed in malicious files. We present results from these deep models on a large dataset consisting of 634,249 sequences of variable length. We believe that this is by far the largest study of malware behavioral models.

On performing structural evaluation of such models, we explored a vulnerability persistent even in highly accurate models that can potentially limit their long-term use if the attackers learn to construct resilient malicious files. All of these models are limited by the length of the sequences they can operate on, making them ineffective against malware hidden in extremely long sequences, or malicious files that execute long running loops of legitimate commands before reaching the malicious events. Potentially, the use of LSTMs allows these models to learn very long sequences, as the cells can continue to retain context. In language modeling, this ability helps retain the necessary word context. However in software event sequences, as observed in malware detection data, context can be reset at several places along the sequence and a distant hidden event can be harder to correlate with consecutive malicious events. Apart from that, capturing full-length sequences also results in extremely compute-intensive models that are harder to train and deploy.

In this paper, we present neural models that are immune to the length of the input sequence. We implement a version of Convoluted Partitioning of Long Sequences (CPoLS) [19] which can capture sequences of any variable length. This allows our model to detect malware where events are hidden deep within the sequences. This model partitions the input sequence and distributes the learning process by using CNNs within a recurrent learning setting. We present models for efficiently learning complete event sequences with variable and extremely long lengths in order to detect malware hidden deep within the sequences. The CPoLS approach captures the entire length of a sequence by splitting it into chunks and distributes the learning process by using CNNs in a recurrent fashion. As shown by Stokes *et al.* [19], such models retain the sequential nature of input entirely without any loss of data. By preventing the loss of data within the sequence, we also make our models resilient against evasion.

The paper presents a description of all of our models with an emphasis on removing any vulnerability in order to create future resilient Neural Malware Detection systems. We start by discussing the motivation behind this problem and the need for neural models. We then discuss the data. This is followed by a detailed discussion of our models. Next we describe the experimental process and results obtained on our data. We then present a small discussion on emerging challenges in malware detection, as well as the potential use of our full-length models in other domains. This discussion is followed by a review of related literature. We conclude the paper with a discussion of the benefits of the proposed neural models, and the advantages of CPoLS inspired methodologies, for performing lossless learning on extremely long sequences.

II. Motivation

Advances in the space of malware detection using artificial intelligence and machine learning have significantly strengthened the success rates in detection. Moreover, the concept of using emulation sequences for file commands provides us with a peek into the internal operations of a malicious file, allowing for a learning model to understand its core. The primary objective of learning malicious actions is finding the presence of potential events within entire sequences and detecting their sequential occurrence. This seemingly trivial learning task, in reality, is much more challenging because of the nature of these underlying commands. Individually, all these commands refer to legitimate system operations and cannot be associated with a level of *badness*. It is the ordered co-occurrence of several such commands which triggers critical actions within the system processing to deviate from standard actions. Numerically, such models can still exhibit a very low error rate and achieve a high accuracy. However, due to the nature of such systems and their underlying data, error rate alone is not a strong representation of a model. Due to their real-world impact, false positive rates in malware detection are also a critical metric of concern since they can potentially prevent a computer from starting or working correctly [20]. A slight miss in the probability measurement from a malware detection model can turn into a huge cost for the underlying system.

Therefore, it is critical for a good detection model to judge the event sequences at a much finer granularity and still maintain a general enough weight association with events to prevent false positives. In order to facilitate such learning, the training methodology must put into consideration certain characteristics of the dataset according to the algorithms being used. We believe that neural models are a good choice here because of the sequential nature of the data and their ability to learn from a fixed vocabulary for the set of events observed in these sequences. An important aspect for a robust Neural Malware Detection system is to exclude as many assumptions and limitations from the training data as possible. We suggest using sequences with variation in lengths, with similar distribution over event vocabulary in both the label classes, and avoiding loss of information throughout the sequences in order to regularize the dataset sufficiently. We also propose that the model should not be associated with direct embeddings of these events throughout and must generate learned tensors in order to escape hard binding of individual events with malware. It is often possible among training batches to incur some bias and associate higher weights with certain events commonly found in sequences leading to a similar label. In such cases, more significance might be learned towards particular events rather than learning from entire sequences. A robust model should be able to maximize detection among sequences, while ensuring generalization over the entire length of an individual sequence. Such systems can then learn patterns and ordered co-occurrence of events causing malicious actions, instead of directly learning event-based activations.

This criticality of the problem and the requirement to retain learning within the respective general blocks motivates our methods for malware detection. Our models leverage the language model-based approach of LSTMs in order to join the event sequences, perform convolutions to derive significant occurrences within extremely long sequences, and ensure end-to-end training in the attempt to learn embeddings driven directly by the true label.

TABLE I
EXAMPLE OF THE FIRST BEHAVIOR EVENTS IN A FILE

File Event
createfile
virtualalloc
virtualalloc
getmodulehandle
getmodulefilename

III. DATA

The event sequences that we analyze in this paper are generated by a modified version of the Microsoft production anti-malware engine which logs the system API calls made by a portable executable (PE) file (*e.g.*, .exe, .dll). All of the PE files in our dataset are written for the Microsoft Windows operating system. To generate the data, we analyzed a large set of malicious and benign files by scanning them with the anti-malware engine and collected the emulation logs corresponding to each file. During the data collection, the malware did not have internet access to prevent the infection of other computers. The logs were collected in August 2017, and thus represent recent malware families. The length of the original sequences varies from file to file and is determined by heuristics employed by the anti-malware engine based on the earlier behavior of the file.

Adversaries sometimes use polymorphism which uses different API calls to achieve the same goal. For example to create a file, they may call ofstream.open in C++, fopen in C, or the Windows user mode CreateFile function. From kernel mode in Windows, they may call the ZwCreateFile or NtCreateFile to create a file. To handle polymorphism, multiple API calls can be mapped to the same high-level event, and the total number of high-level events in our data is 156. The first five events in a sample file are shown in Table I. In addition, each file is assigned a label $\tau \in \{0, 1\}$ where 1 indicates that file is malware, and a benign file has a label of 0.

The files were randomly selected from our incoming production streams of files to be analyzed. In order to be included in this dataset, files were selected from a larger collection according to the following criteria. First, we only included a single file instance for each distinct event sequence. Second we discarded any file sequences which had multiple labels: we did not include any files with the same event sequence, but labeled as both *malware* and *benign*. Third, we discarded any files with less than 50 events. The original dataset thus obtained includes the results of emulating 634,249 files which is then randomly split into separate training, validation, and test datasets including 443,974, 63,425, and 126,850, respectively. Distribution over labels in these datasets corresponds to 75% malicious samples, and 25% benign samples.

In order to use these sequences with our model, we first create a vocabulary of the 156 event IDs and then translate the event sequences into our vocabulary space. Each learning batch therefore consists of sequences over the 156 vocabulary space, resulting in a three-dimensional tensor for the batch input. The input dimensions specify the batch size, sequence length, and event embedding dimension, respectively.

IV. SYSTEM DESIGN

As discussed previously, our Neural Malware Detection models learn by using the emulation event sequences and are trained in an end-to-end fashion where the loss gradient flows directly from the final label. Our objective with these sequences is to detect the presence of potential events and their sequential correlation with other events causing the malicious action. These events can be both grouped or scattered throughout the sequence. They are not necessarily consecutive, but are ordered in most cases. Therefore, both presence and order play an important role in this detection. In this section, we discuss our learning models in detail. We also comment on certain limitations of our fixed-length models and present a model that can operate efficiently over sequences of any length.

A. Building Blocks

We begin with a short review of the basic units forming our models. In particular, we use LSTM recurrent neural networks to capture the sequential data. In our extension models, we also use CNNs to extract significant event occurrences within the entire length of the long sequences. Throughout the models, we also use the concept of one-dimensional max pooling. This property helps reduce the dimensionality within the model at different steps and helps extract significant embeddings and activation wherever required.

1) LSTM:: Long Short-Term Memory [1], [2] neural networks are a memory-based variant of Recurrent Neural Networks (RNNs) where each neuron is defined to be a gated cell with memory. These networks are less vulnerable to the vanishing or exploding gradient problems [21], [22] and operate by maintaining both a hidden state and memory at each time step. This capability of LSTMs has made them popular in language models and sequential architectures. Among the different variants of the algorithm that are commonly employed, we use the following equations for our implementation of the LSTM in this paper.

$$
\begin{aligned}
\mathbf{i}_t &= \sigma(\mathbf{W}_{hi} * \mathbf{h}_{t-1} + \mathbf{W}_{xi} * x_t + \mathbf{b}_i) \\
\mathbf{f}_t &= \sigma(\mathbf{W}_{hf} * \mathbf{h}_{t-1} + \mathbf{W}_{xf} * x_t + \mathbf{b}_f) \\
\mathbf{o}_t &= \sigma(\mathbf{W}_{ho} * \mathbf{h}_{t-1} + \mathbf{W}_{xo} * x_t + \mathbf{b}_o) \\
\mathbf{c}_t &= \mathbf{f}_t \odot \mathbf{c}_{t-1} + \mathbf{i}_t \odot \tanh(\mathbf{W}_{hc} * \mathbf{h}_{t-1} + \mathbf{W}_{xc} * x_t + \mathbf{b}_c) \\
\mathbf{h}_t &= \mathbf{o}_t \odot \tanh(\mathbf{c}_t)
\end{aligned}
\tag{1}
$$

where σ is the logistic sigmoid function, $\mathbf{i}_t, \mathbf{f}_t, \mathbf{o}_t, \mathbf{c}_t$ are the input gate, forget gate, output gate and cell activation, respectively. $\mathbf{W}_h(\cdot)$ are the recurrent weight matrices for each gate, $\mathbf{W}_x(\cdot)$ are the input weight matrices per gate, and $\mathbf{b}(\cdot)$ are the biases for each gate. At each timestep t, the network takes vector x_t as input, updates the cell memory $\mathbf{c}_t$, and provides a hidden state $\mathbf{h}_t$. The input vector x_t can be the representation of the input in any format such as a one-hot encoding or a dense embedding. Function $\odot$ represents the pairwise product between two vectors.

LSTMs are often used in deep models by stacking multiple layers on top of each other. Stacked-LSTMs [23] allow deeper learning of the structures where each layer gets an embedding from a lower layer while traversing across the timesteps of the sequence.

2) CNN:: Convolutional Neural Networks [13] are extremely powerful models often used in the space of computer vision [14], [15]. Compared to the LSTMs, CNNs are faster and more efficient architectures which use smaller kernels that are trained at different locations within the input data. In images, this refers to training smaller blocks within an entire image with the same set of weights instead of using a larger weight tensor for the complete image size. Similarly for one-dimensional data like sentences, CNNs traverse over smaller chunks of the input and perform convolutions across each chunk while updating the smaller weight kernel. Recently, CNNs have also shown success on sequential learning problems [16], [17] and continue to be explored in this new space.

3) Max Pooling:: Pooling operations are often used in CNNs [24], [25] to reduce dimensionality and extract significant features in deeper models. Max pooling can also be used with one-dimensional sequences for the extraction of higher activations. On a sequence, max pooling extracts a single vector with maximum activations across each dimension.

B. End-to-End Models

In order to learn from the emulation sequences directly using a target label of malicious behavior, we present our end-to-end models below. Given an input event sequence E of length T timesteps which consists of events e_t at each timestep t in the sequence and a known truth label τ, we need to predict the probability p_m of sequence E being malicious. In terms of the data under investigation, these models are required to perform two crucial tasks. The first task is the detection of potential events that can lead to a malicious action, and the second is the sequential linkage of events which combine to represent a malicious action. Since these individual events are standard system commands, the mere occurrence of an event within the sequence cannot be used to predict malware. Our initial model designs are inspired by the language model and classifier-based malware detection models presented in [11], [12].

1) Direct Sequence Learning: The Direct Sequence Learning model (DSL) is our most basic end-to-end architecture. This model relies on the capability of the LSTMs to learn sequences and capture relevant information within the last activation. This model uses LSTMs as the sequence learning mechanism, which helps convert an entire sequence into an embedding by using only the last activation from the LSTM. This activation is then passed through a regular dense layer for learning the derived representation. We produce the final activation using a logistic sigmoid layer on top which provides the probability of the input sequence being malicious. Formally, the DSL model is defined as:

$$\begin{aligned} h_L &= \mathrm{LSTM}(E)[T] \\ h_{CL} &= \mathrm{ReLU}(W_L * h_L) \\ \mathbf{p_m} &= \sigma(W_D * h_{CL}) \end{aligned} \tag{2}$$

where h_L is the hidden state activation from the LSTM at the last timestep T, and h_{CL} is the activation derived from a fully connected ReLU neural network layer. $\mathbf{p_m}$ is the final probability of sequence E being malicious, derived through a final logistic sigmoid layer σ. W_L is the weight matrix for the ReLU layer, and W_D is the weight matrix for the final sigmoid layer which generates the output probability.

2) LSTM and Max Pooling: While DSL is able to translate the sequence into a single vector embedding, it is optimized to predict the last activation. The structure of our event sequence is similar to a sentence in a language model, where the objective of the RNN is to predict the next word in the sequence. Therefore, the last activation from the LSTM is best trained for predicting the representation of the next event and might not be of strong assistance in our objective of finding malicious event sequences. In [12], the authors have used an LSTM and Max Pooling for capturing events from the sequence. Using a similar LSTM and Max Pooling (LaMP) concept in our models, we perform a temporal max pooling over the resulting LSTM sequence. While in [12], the language model is independently trained using a recurrent neural network, we tie the training of the LSTM with the complete model similar to [19]. The LaMP model, therefore, first learns the representation for the input sequence events, next performs a temporal max pooling operation on the sequential hidden states to create a final sequence embedding, and then proceeds with one or more neural network layers to learn the output probability. Formally, the LaMP implementation is defined as:

$$\begin{aligned} H_L &= \mathrm{LSTM}(E) \\ h_L &= \mathrm{MaxPool1D}(H_L) \\ h_{CL} &= \mathrm{ReLU}(W_L * h_L) \\ \mathbf{p_m} &= \sigma(W_D * h_{CL}) \end{aligned} \tag{3}$$

where H_L is the complete sequential output returned by the LSTM, and MaxPool1D is the temporal max pooling layer that extracts the final embedding h_L from H_L.

3) Auxiliary-Output Language Learning: The Auxiliary-Output Language Learning model (AoLL) is a regularized extension of LaMP, built to assist gradient flow in the complete model. Both LaMP and DSL train the LSTM layers directly from the loss gradient using the final target label. We believe, that while this objective helps direct a specific gradient flow throughout lower layers of the model, it can be assisted by an additional loss in order to address the sequential nature of the data. Models presented in [11], [12] train a language model over the input sequences where the RNN, at each timestep, is trained to predict the next event in the sequence. In order to incorporate such a goal within the end-to-end learning model, we use two objectives in our AoLL model. We learn the probability p_m in the same way as presented in LaMP. In addition to this, the AoLL model also obtains an auxiliary output from the LSTM layer in the form of its final activation. We use this output to predict the next event within the sequence, leveraging the sequential learning properties of LSTMs. The complete model is now trained with two objectives, and two loss functions, each of which generates a gradient flow within the model that is used to update the weights. Formally, AoLL is defined as:

$$\begin{aligned}
&H_L = \text{LSTM}(E) \\
&h_L = \text{MaxPool1D}(H_L) \\
&h_{CL} = \text{ReLU}(W_L * h_L) \qquad (4)\\
&\kappa_{\mathbf{L}}[t] = \text{softmax}(W_\kappa * H_L[t]) \qquad \forall t \in [0, T-1] \\
&\mathbf{p_m} = \sigma(W_D * h_{CL})
\end{aligned}$$

where $\kappa_{\mathbf{L}}$ is a softmax output for each timestep of the sequence providing the probability over the entire vocabulary V. At each timestep t, $\kappa_{\mathbf{L}}[t] \in \mathbb{R}^V$ is defined as a vector of size V representing an output probability distribution for each possible event. This measure is aligned with the objective of the language model to predict the next word (event) in the input sequence.

As mentioned above, the AoLL model uses two objective functions and therefore requires two loss functions. For the three models (DSL, LAMP, and AoLL) defined above, we use Log Loss as our loss function. For determining the probability p_m, we use a loss in the form of binary cross-entropy. For the stepwise outputs $\kappa_{\mathbf{L}}$ in AoLL, we use categorical cross-entropy as the loss function when predicting the next event in the sequence. For all three models, with prediction p_m and known target label τ, we measure the final loss $\mathcal{L}$ as

$$\mathcal{L} = \text{LogLoss}(\mathbf{p_m}, \tau). \qquad (5)$$

For AoLL, with predictions $\kappa_{\mathbf{L}}$ and events e_t in the event sequence E for timestep t, we measure an auxiliary loss $\mathcal{L}_{aux}$ as

$$\mathcal{L}_{aux} = \text{LogLoss}(\kappa_{\mathbf{L}}[t], e_{t+1}) \qquad \forall t \in [0, T-1]. \qquad (6)$$

C. Model Limitations

The models defined above are all trained end-to-end using a known target label and input event sequences only. A significant strength of these models is their ability to learn the sequence embeddings, that are critical informants of the malicious behavior in the sequence caused by the consecutive or sequential occurrence of certain events. While the use of the LSTM provides us with the ability to use variable-length sequences, it often becomes computationally very expensive to train as the lengths of the sequences increase. Generally in sequence learning, it is often common to cut the sequences up to a certain prescribed length and then use those fixed-length subsequences for training. In non end-to-end models, it is even possible to train the sequences on a certain sequence length and then use a different length for further representation when used with deeper models. However these solutions are still limited by the computation and memory capacities.

In language model-based applications, capturing a limited length of the input sequence can often yield a sufficient representation of the complete sequence. However in the case of detection within a longer sequence, our objective is to find all the events that can lead to a malicious action, and they can be separated by very long distances within the entire sequence. For instance, consider a malicious activity that requires events v_a, v_b, and $v_c \in V$ to happen sequentially but does not need them to occur consecutively. This event sequence, therefore, can incorporate a large number of random events $v_{r1} \ldots v_{rn}$ between v_a and v_b, and v_b and v_c, respectively. An optimal system needs to detect the presence of these malicious events and generate an appropriate activation on detecting their sequential occurrence. If the number of random events n increases drastically, the model can lose the context of the presence of v_a by the time it reaches v_b. One approach to retain distant event occurrences is by using Bidirectional LSTMs [23], [26], but in very long sequences this becomes even more computationally expensive for the end-to-end model to train.

Another problem with limited-length sequences in this case is that malware can often be written as a long series of legitimate benign events followed by malicious events much later in the sequence (*i.e.*, file). Limiting the length of such sequences provides potentially benign sequences to the system with a malicious label. Therefore, for training on limited length, we need to filter data for prevention against such cases, leading to the loss of data. When given a very large dataset with malicious events occurring within the smaller-length sequences, a model can be trained well. However, limiting the length of sequences adds a vulnerability to the model for the detection of malware. It is therefore essential for a model to capture the entire length of the sequences in order to find the events located at larger distances and still maintain their sequential order.

D. Convoluted Partitioning of Long Sequences

In order to capture both, the presence of events, and their sequential relationship in very long sequences, we use the Convoluted Partitioning of Long Sequences (CPoLS) approach introduced in [19] for detecting malicious JavaScript and VBScript. A similar approach is presented in [27] where the authors performed a static analysis on the chunked PE file byte sequence. The implementation adopted in [27] uses the last hidden output from the LSTM. However based on our experimental results with RNNs and LSTMs, temporal max pooling often performs significantly better than using the last hidden state when performing event detection within a sequence. We believe that this model approach applies well to our problem of malware detection since the length of PE event sequences can be extremely long. In our CPoLS models, we are able to learn end-to-end models from the entire length of very long sequences while maintaining their order. We formalize the adaptation of the CPoLS model in Algorithm 1.

For an input event sequence E, we first split it into a chunked sequence $C = [c_1, c_2, c_3, \ldots]$, where $E = c_1 | c_2 | c_3 | \ldots$ ($'|'$ denotes the concatenation operation), using Algorithm 2. Since E itself is a sequence, subsequences $c_i \in C$ are ordered by index. This operation increases the dimensionality of the batch input tensor from three to four dimensions. We now treat

Algorithm 1 CONVOLUTED PARTITIONING OF LONG SEQUENCES

```
Input: Sequence $E$, Chunk Size $s$
$C$ = CHUNKIFY($E$, $s$)
$H_{RC}$ = RECURRENTCONVOLUTIONS($C$)
$E' = [h_{RC1}, h_{RC2}, h_{RC3} \ldots]$
$\mathbf{p_m}$ = LAMP($E'$)
return $\mathbf{p_m}$
```

Algorithm 3 RECURRENTCONVOLUTIONS

```
Input: Chunks $C$
for $c_k$ in $C$ do
    $h_{Ck}$ = CONV1D($c_k$)
    $h_{MPk}$ = MAXPOOL1D($h_{Ck}$)
end for
$H_{MP} = [h_{MP1}, h_{MP2}, h_{MP3}, \ldots]$
return $H_{MP}$
```

each chunk c_i as an element of our top-level sequence which is passed through a time distributed layer that sequentially processes each chunk. Within each of these recurrent processes, we perform convolutional learning on the subsequence within that timestep as shown in Algorithm 3. By performing convolutions, our goal is to extract the presence of significant events from each subsequence and to reduce the overall dimensionality of the problem. Therefore, each chunk c_i, in the recurrent processing, is passed through a one-dimensional, convolutional neural network (CONV1D), and then through a time distributed max pooling layer (MAXPOOL1D). This allows for us to reduce the size of each chunk, and therefore, of the entire sequence. We recombine the outputs of each chunk to form a new sequence representing activations from the recurrent convolutions. We then pass this derived sequence through the LSTM of our end-to-end models. Due to the usage of the derived sequences, we cannot use the AoLL model with CPoLS.

Algorithm 2 CHUNKIFY

```
Input: Sequence $E$, Chunk Size $s$
Initialize $chunks$ = [ ]
for $i = 0$ to $len(E)/s$ do
    APPEND($E[s * i : (s + 1) * i - 1]$, $chunks$)
end for
return $chunks$
```

V. EXPERIMENTS AND RESULTS

We performed an extensive evaluation of the models presented above with our emulation sequence dataset. We implemented these models on the Tensorflow [28] platform using the Keras [29] library. Both models and data access operations were written using the Python programming language. Our models were trained on a cluster of Nvidia Tesla K40m GPUs.

For each model, we ran several iterations to identify the best hyper-parameter settings. We use the LSTM as our recurrent neural network module in each model. The hidden dimension of 1500 was used for the LSTM in each model. For our vocabulary of size 156, we used an embedding dimension of size 114 throughout the models. The RELU layer used in our models had a hidden dimension of 64. The CNN layer used in the CPoLS model performed one-dimensional convolutions on a window of 10 items, with a stride size of 5. The CNNs used a channel size of 114, which is equal to the embedding dimension, in order to consider each dimension within the computation. Each model was run for 15 epochs on 443,974 training samples, and validated on 63,425 samples after each epoch. For the limited-length models, we used sequences of length 256. We used the ADAM optimizer [30] with a learning rate of 0.001. For the limited-length models, we used a mini-batch size of 64. For full-length models, we used a mini-batch size of 32. The results presented in this section use 126,850 test samples on our trained models. For each model, we present the average results over multiple iterations using the best settings.

We present results across three significant metrics. We first discuss the average result accuracies derived by our models in predicting the probability p_m of a file being malicious. Accuracies provide us with an overall strength of the Neural Malware Detection systems in detecting malicious actions within a sequence. As we discussed earlier, this problem is more critical and sensitive to false positives due to the nature of the underlying systems. Therefore, we also compare our models using the Receiver Operating Characteristic (ROC) plots, in order to get a finer sense of each model's performance. Along with the ROC plot, we also measure the True Positive Rate (TPR) at fine scale in order to learn the performance of our system. We therefore, measure the TPR at a False Positive Rate (FPR) of 1%. Table II summarizes the accuracies and TPRs across models and configurations. Figure 1 presents the ROC plots for the best performing version of each model. We have limited the x-axis scale on the ROC plots to a finer scale of 2% on the FPR. By visualizing at an FPR of 2%, we evaluate each model's performance at a much finer scale than possible through accuracies. Even with minor variation in accuracies, it can be observed that the CPoLS model operating on full-length sequences performs much better at a very low FPRs as well, maximizing the effective area under the curve.

As can be seen, CPoLS not only builds resilience against long sequences, but also performs significantly better than the other models on our dataset. These neural models respond aptly to the criticality of our problem. The model which combines a CNN followed by an LSTM proposed by Kolosnjaji, et al. [18] performs better than DSL in terms of average accuracy, but it has a significantly lower performance in terms of the TPR@1% and the ROC curve. Our experiments with AoLL display comparable performance to LAMP [11], [12], highlighting the ability of end-to-end models to identify optimal gradient flow even when using multiple loss functions, but also signifying the effectiveness of a single loss function when optimizing the entire model end-to-end.

In our experiments with full-length models, we also tested different chunk sizes and observed that the selection of chunk size did not significantly affect the results but did increase training speeds. We experimented with different sizes between 32 and 256, but could not obtain significant accuracy variation.

TABLE II
TEST ACCURACIES (%) AND TPRS (%) AT AN FPR = 1% FOR ALL THE MODELS

MODEL NAME	ACCURACY	TPR@1%
DSL n_{LSTM}=1	0.881	68.44
Kolosnjaji et al.	0.932	59.96
AoLL n_{LSTM}=1	0.922	68.14
AoLL n_{LSTM}=2	0.932	67.40
LaMP n_{LSTM}=1	0.947	76.50
LaMP n_{LSTM}=2	0.951	76.27
CPoLS n_{LSTM}=1	0.956	83.50

This might be due to the fact that each of these chunk sizes were sufficient to capture important event occurrence and can be subject to further evaluation in the future. The speed of training, however, improved with larger chunks going under convolutions, hence lowering the sequence lengths for the LSTMs at the following stages.

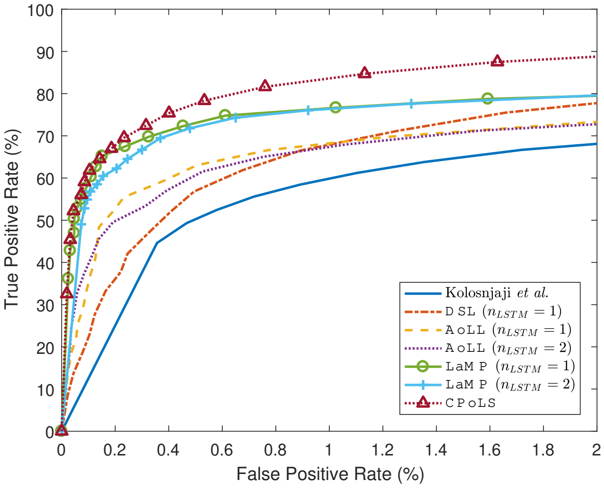

Fig. 1. Receiver operating characteristic plots for the models described above

VI. DISCUSSION

It is important to consider evasion when considering machine learning models for security applications. Papernot, et al. [31] recently proposed an adversarial attack for recurrent neural networks. Recurrent models are more challenging to attack than deep neural networks due their recurrent nature. To attack a recurrent structure, the authors first unfold (*i.e.*, unroll) the recurrent model into essentially a deep neural network with many layers. Next they compute the unfolded Jacobian and perturbations to craft adversarial sequences which can be used to trick the RNN into predicting the incorrect class.

To the best of our knowledge, no adversarial defenses have been proposed for recurrent neural networks. Thus, defenses against adversarial attacks directed at recurrent models, such as those proposed in this paper, are an open research topic.

Besides malware detection, the models presented in this paper also provide potential approaches for handling extremely large sequences in general. Neural network-based models are applied to a large number of applications across several domains. Depending on the nature of the respective data, the length of sequences in these domains can play a significant role in the model's learning capability. While we presented our models for malware detection, our architecture, in particular is not tied to the underlying problem and can be generalized for a broader set of problems. Learning on sequences involving event detection, event prediction or any related objective can benefit from our models in efficiently using the entire length of the sequences. We believe, with certain domain-specific modifications, and with more generalization within the overall architecture, our models can help perform lossless sequential learning irrespective of data lengths.

VII. RELATED WORK

The persistent increase in the growth and distribution of malware has called for the alarming need of detection and prevention. Anti-virus and anti-malware systems developed over time have performed in-depth analysis of malicious software families, have developed signatures, tagging methods and several other mechanisms to tackle this ever growing problem. Exploration of using machine learning in this space has witnessed the use of both traditional and deep learning models. Support Vector Machines (SVM) have been incorporated into this task by [32]. Hidden Markov Models have been explored by [33]. Assistance in signature generation using deep learning has been demonstrated in [34] where they used Deep Belief Networks to generate malware signatures that could then be passed through classification models. Entering more into the space of neural network-based models, language structure within emulated sequences of a malicious executable have been utilized by [11], [12]. They have built RNN-based language models to learn the relationship between the event structure and malware.

The CPoLS model used in this paper is based on [19] and uses CNNs over sequential structures. While popularly used in image and graphical data, CNNs have also been used by [16], [17], where they have shown exceptional results in the space of language models by constructing Sequence-to-Sequence models using CNNs. In malware classification, [18] and [35] also demonstrated the use of CNNs along with sequential learning. In their models for malware classification, they demonstrated the ability to classify among malicious families from a dataset of malicious files. Among other applications, an interesting mix of RNN and CNNs has been presented by [36] by first using RNNs for feature extraction and generating image representations from files which are classified further by CNNs.

VIII. CONCLUSIONS

Neural Malware Detection models, as presented in this paper, help combat the challenge of detecting malware from extremely long API call sequences generated through an anti-malware engine. While helping to resolve a major concern in computer security, our models also address the critical and sensitive nature of this problem.

Through this paper, we presented several end-to-end models that use combinations of LSTM recurrent neural networks and CNNs. Our models learn entirely from event sequences while learning event embeddings within the deep models themselves. Through our full-length based models, we presented efficient methods to perform lossless sequence learning on extremely long sequences in detection tasks. Through our training and results on the largest malware dataset of 634,249 sequences, we demonstrated the significance of using entire sequence lengths when performing malware detection.

In our AoLL model, we have also demonstrated the use of multiple objectives and losses in an end-to-end setting that utilize additional gradient flows within the model while maximizing the primary objective of predicting the probability of maliciousness. Through CPoLS, we demonstrated the advantages of using Convolutional Neural Networks for event detection in association with LSTMs for sequential binding. Through our adaptation of the CHUNKIFY approach, we presented an effective method for the partioning of extremely long sequences in learning tasks without losing the sequential nature at any stage of the model.

In summary, this paper targets the core of the malware operation style and learns a Neural Malware Detection model from it, which can be used directly with emulators, can be embedded within anti-malware systems to serve as a detection operator, can efficiently handle long sequences, and is resilient to variations and loops emerging in malware over time.

References

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

[2] F. A. Gers Jj, U. Schmidhuber, and F. Cummins, “Learning to Forget: Continual Prediction with LSTM,” 1999. [Online]. Available: www.idsia.ch

[3] A. Graves, A. Mohamed, and G. Hinton, “Speech recognition with deep recurrent neural networks,” pp. 6645–6649, May 2013.

[4] D. Bahdanau, K. Cho, and Y. Bengio, “Neural Machine Translation by Jointly Learning to Align and Translate,” sep 2014. [Online]. Available: http://arxiv.org/abs/1409.0473

[5] I. Sutskever, O. Vinyals, and Q. V. Le, “Sequence to Sequence Learning with Neural Networks.” [Online]. Available: https://papers.nips.cc/paper/5346-sequence-to-sequence-learning-with-neural-networks.pdf

[6] O. Vinyals, M. Fortunato, and N. Jaitly, “Pointer networks,” pp. 2692–2700, 2015. [Online]. Available: http://papers.nips.cc/paper/5866-pointer-networks.pdf

[7] A. Graves, G. Wayne, and I. Danihelka, “Neural Turing Machines,” oct 2014. [Online]. Available: http://arxiv.org/abs/1410.5401

[8] A. Graves, G. Wayne, and E. al., “Hybrid computing using a neural network with dynamic external memory,” *Nature (in Press)*, vol. 538, no. 1, 2016.

[9] K. Xu, J. Ba, R. Kiros, K. Cho, A. Courville, R. Salakhudinov, R. Zemel, and Y. Bengio, “Show, attend and tell: Neural image caption generation with visual attention,” vol. 37, pp. 2048–2057, 07–09 Jul 2015. [Online]. Available: http://proceedings.mlr.press/v37/xuc15.html

[10] J. Weston, S. Chopra, and A. Bordes, “Memory Networks,” oct 2014. [Online]. Available: http://arxiv.org/abs/1410.3916

[11] 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.

[12] 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.

[13] Y. LeCun and Y. Bengio, “Convolutional networks for images speech and time series,” 1995.

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

[15] O. Russakovsky, J. Deng, H. Su, J. Krause, S. Satheesh, S. Ma, Z. Huang, A. Karpathy, A. Khosla, M. Bernstein *et al.*, “Imagenet large scale visual recognition challenge,” *International Journal of Computer Vision*, vol. 115, no. 3, pp. 211–252, 2015.

[16] J. Gehring, M. Auli, D. Grangier, and Y. N. Dauphin, “A Convolutional Encoder Model for Neural Machine Translation,” nov 2016. [Online]. Available: http://arxiv.org/abs/1611.02344

[17] J. Gehring, M. Auli, D. Grangier, D. Yarats, and Y. N. Dauphin, “Convolutional Sequence to Sequence Learning,” may 2017. [Online]. Available: http://arxiv.org/abs/1705.03122

[18] 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.

[19] J. W. Stokes, R. Agrawal, and G. McDonald, “Neural classification of malicious scripts: A study with javascript and vbscript,” *CoRR*, 2018.

[20] E. Bott, “Defective mcafee update causes worldwide meltdown of xp pcs,” Apr 2010. [Online]. Available: http://www.zdnet.com/article/defective-mcafee-update-causes-worldwide-meltdown-of-xp-pcs/

[21] S. Hochreiter, “The vanishing gradient problem during learning recurrent neural nets and problem solutions,” *Int. J. Uncertain. Fuzziness Knowl.-Based Syst.*, vol. 6, no. 2, pp. 107–116, Apr. 1998. [Online]. Available: http://dx.doi.org/10.1142/S0218488598000094

[22] Y. Bengio, P. Simard, and P. Frasconi, “Learning long-term dependencies with gradient descent is difficult,” *IEEE Transactions on Neural Networks*, vol. 5, no. 2, pp. 157–166, Mar 1994.

[23] A. Graves, N. Jaitly, and A.-r. Mohamed, “Hybrid speech recognition with Deep Bidirectional LSTM,” in *2013 IEEE Workshop on Automatic Speech Recognition and Understanding*. IEEE, dec 2013, pp. 273–278. [Online]. Available: http://ieeexplore.ieee.org/document/6707742/

[24] D. Scherer, A. Müller, and S. Behnke, “Evaluation of pooling operations in convolutional architectures for object recognition,” *Artificial Neural Networks–ICANN 2010*, pp. 92–101, 2010.

[25] D. C. Cireşan, U. Meier, J. Masci, L. M. Gambardella, and J. Schmidhuber, “Flexible, high performance convolutional neural networks for image classification,” in *Proceedings of the Twenty-Second International Joint Conference on Artificial Intelligence - Volume Volume Two*, ser. IJCAI’11. AAAI Press, 2011, pp. 1237–1242.

[26] R. Pascanu, C. Gulcehre, K. Cho, and Y. Bengio, “How to Construct Deep Recurrent Neural Networks,” dec 2013. [Online]. Available: http://arxiv.org/abs/1312.6026

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

[28] M. Abadi, A. Agarwal, P. Barham *et al.*, “TensorFlow: Large-scale machine learning on heterogeneous systems,” 2015, software available from tensorflow.org. [Online]. Available: http://tensorflow.org/

[29] F. Chollet *et al.*, “Keras,” https://github.com/fchollet/keras, 2015.

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

[31] N. Papernot, P. McDaniel, A. Swami, and R. Harang, “Crafting adversarial input sequences for recurrent neural networks,” 2016.

[32] J. Pfoh, C. Schneider, and C. Eckert, *Leveraging String Kernels for Malware Detection*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 206–219.

[33] S. Attaluri, S. McGhee, and M. Stamp, “Profile hidden markov models and metamorphic virus detection,” *Journal in Computer Virology*, vol. 5, no. 2, pp. 151–169, May 2009. [Online]. Available: https://doi.org/10.1007/s11416-008-0105-1

[34] O. E. David and N. S. Netanyahu, “DeepSign: Deep Learning for Automatic Malware Signature Generation and Classification *.”

[35] B. Kolosnjaji, G. Eraisha, G. Webster, A. Zarras, and C. Eckert, “Empowering convolutional networks for malware classification and analysis,” in *2017 International Joint Conference on Neural Networks (IJCNN)*. IEEE, may 2017, pp. 3838–3845.

[36] S. Tobiyama, Y. Yamaguchi, H. Shimada, T. Ikuse, and T. Yagi, “Malware detection with deep neural network using process behavior,” in *2016 IEEE 40th Annual Computer Software and Applications Conference (COMPSAC)*, vol. 2, June 2016, pp. 577–582.