# NEURAL SEQUENTIAL MALWARE DETECTION WITH PARAMETERS

*Rakshit Agrawal*[*][⋆] *Jack W. Stokes*[†] *Mady Marinescu, Karthik Selvaraj*[±]

[⋆] University of California, 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

Sequential models which analyze system API calls have shown promise for detecting unknown malware. Athiwaratkun and Stokes recently proposed a two-stage model which uses a long short-term memory (LSTM) model for learning a set of features which are then input to a second classifier. Kolosnjaji *et al.*, first use a convolutional neural network followed by an LSTM to predict unknown malware. However, neither of these models consider the parameters which are input to the system API calls. These input parameters offer significant information regarding malicious intent. In this paper, we extend Athiwaratkun's model to include each system API call's two most input parameters. We then show that the proposed model dominates these previously proposed models in terms of the receiver operating characteristic (ROC) curve.

***Index Terms*—** Malware Classification, Deep Learning, Long Short-Term Memory, Char-Grams

## 1. INTRODUCTION

The detection of malware (*i.e.*, malicious software) is important to protect a user's computer from infections. Microsoft Corporation receives hundreds of thousands of previously unknown files each day. Malware classification helps to automatically create signatures for the client computer and detect malware in its cloud protection service.

**Related Work**: Malware classification has been extensively studied for over two decades [1]. Kephart first proposed to use a neural network for malware classification in [2]. Dahl *et al.*, [3] investigated deep learning for malware classification for features generated by dynamic analysis in 2013. Saxe and Berlin [4] next proposed a deep neural network for static classification-based features. Huang and Stokes [5] extended Dahl's work in a multi-task framework where the two tasks which were used to train the model included binary classification and family classification.

Pascanu *et al.*, [6] first proposed recurrent models for evaluating system API calls. The authors proposed a two-stage model (RNN-NN) which employs either a recurrent neural network or an echo state network for recurrent model in the first stage, followed by a neural network as the classifier. Kolosnjaji *et al.*, [7], proposed a convolutional neural network-based architecture for capturing nearby event representations followed by an LSTM [8, 9] and a classifier (CNN-LSTM). They presented results on classification among malware families in pre-detected malware. Athiwaratkun *et al.*, [10] improved Pascanu's model by replacing the first stage with either an LSTM or a Gated Recurrent Unit (GRU) [11]. They also investigated a character-level CNN [12] in their work. The LSTM-based system outperformed the other recurrent models proposed by Pascanu, *et al.*, [6]

**Data Source and Features**: Windows Defender, which has been included in all versions of client and server Windows operating system since Windows 8, provides antivirus signatures in addition to antispyware signatures. For Windows versions prior to Windows 8, Windows Defender only included signatures for antispyware. Whenever the user attempts to install a file on their Windows computer, the antivirus system is called before it is allowed to run on the native operating system. For example, Windows Defender is called by Outlook, Microsoft's email client, when the user clicks on an executable attachment, by Windows SmartScreen when the user attempts to download an executable from the internet, or by the operating system when the user begins to install a program from a DVD.

In this research, we only consider the detection of Windows portable executable (PE) files. Before the executable file is permitted to run on the native operating system, Windows Defender first attempts to emulate the file. In other words, the file is investigated using dynamic analysis (*i.e.*, behavioral analysis). In part, the Windows Defender engine detects system API calls and their parameters as the file is emulated.

Executable files may call multiple system APIs which accomplish the same task. For example, to create a new file, an executable file may call either a user mode or kernel mode API to accomplish this task. Similarly, the PE file may call the libc API fopen(). Polymorphic malware may attempt to call a different combination of APIs for each installation to change its behavior and possibly avoid detection. To handle this type of behavior, the Windows Defender antivirus engines maps a number of equivalent APIs to a single high-level event to reduce the challenges of detecting polymorphic files. Since similar APIs may have different parameters, we only consider two parameters for each high-level API system call event.

**Proposed Solution**: In this paper, we propose a novel architecture, using Long Short-Term Memory (LSTM) [8, 9] recurrent neural networks, which extends previously proposed sequential malware classification models to include the two most important system API call parameters. We provide efficient methods of exploring a wide vocabulary space of parameters and derive representations that can be learned along with the remaining system. Our model includes the parameter data along with event sequences, and aligns itself in order to leverage their presence without compromising sequential learning over the events.

**Threat Model**: The threat model makes the following assumptions about the attackers. The file must be submitted to the Windows Defender antivirus engine for analysis. Files dropped by other means such as drive-by downloads may not be submitted by the operating system or any of its components for scanning. Next, the file must be

[*] The first author performed the work while at Microsoft Research

successfully emulated by the antivirus engine. In some cases, malware may attempt to detect that it is being emulated and either halt execution or only execute benign activity. Finally, the proposed detection method relies on the execution of system API calls. If the attacker decides to re-implement portions of the underlying operating system in assembly or C++, the model may fail to detect the malicious behavior.
Contributions of this work include:

1. We propose and implement a novel neural sequential malware classification model which processes system API calls and includes the two most important input parameters.
2. We conduct a study to evaluate the model which analyzes recent malware collected by a production antimalware engine.
3. We demonstrate that the proposed model significantly outperforms other neural sequential malware classification models, particularly at low false positive rates, which do not include the input parameters.

## 2. METHODS

The data for detecting malware, as discussed earlier in the paper, provides us with rich parameter information associated with each API call within sequences. This added parameter information builds a relation horizontally with the event and vertically with following events and parameters in the sequence. While learning on the events only, each identical event holds the same meaning for the model whenever it occurs in the sequence, but when parameters are included with each event, their presence can help detect the difference in the overall meaning of the event in a particular context. For instance, a system call to remove a file can generate an event 'Remove File' whose parameters can include the name of the file to be removed. This event, therefore, can vary significantly in its impact on the system depending on the importance of file to be removed.

While it can be established that the presence of parameters can potentially provide significantly more information about the actions, their structure over a large dataset can be extremely complex for conventional learning methods. Parameters, unlike events, are not members of a limited vocabulary space, and instead span a huge space with semantically highly uncorrelated entries. Parameters can include specific names, API-specific arguments, numerical values, and many such varying formats.

With our methods for processing parameters within the neural framework, we present neural models that can leverage additional parameter information along with the events in a sequence and use them to improve the learning capabilities for detecting malware. This section describes these methods in detail and presents two models - *"Events Only"* and *"Events + Parameters"*. Both models are trained completely via backpropagation through time [13] using the loss derived from the target label associated with each file.

### 2.1. Parameter Preprocessing

The parameters in our large dataset, with their high variability, make it difficult to learn embeddings individually for each parameter term in the corpus. Since these parameters can be of any format and are not specific numerical values, they cannot be used as normalized scalars that can be concatenated with event embeddings while learning them sequentially.

We observe that parameters hold both semantic and structural significance within a sequence. While some parameters hold relevance with their semantic meaning in the dataset, others might be defined by their patterns of use. These parameters can also hold different meaning depending on their position within the sequence, as well as whether they are the first or second parameter associated with an event. Parameters like file paths, relevant file extensions, and command attributes can update the meaning of events in the overall processing of a file.

**Algorithm 1** CharGram

```
Input: Parameter Corpus $C_p$, N-gram size $\nu$, Threshold $k$
Initialize $grams = [\,]$.
for $i = 0$ to $len(C_p) - 1$ do
    $A_p$ = Characterize($C_p[i]$)
    for $j = 0$ to $(len(A_p) - \nu)$ do
        Append($A_p[j : j + \nu - 1], grams$)
    end for
end for
$FD_{Cp}$ = FreqDist($grams$)
$grams$ = Sorted($FD_{Cp}$)[:k]
return $grams$
```

**Algorithm 2** Parameterize

```
Input: File Sequence $F$, Parameter Count $m$, Grams $g$
Initialize $E = [\,], P_1, \ldots P_m = [\,], \ldots [\,]$
for $i = 0$ to $len(F) - 1$ do
    $event$ = $F[i][0]$
    $params$ = $[\,]$
    for $j = 1$ to $m$ do
        $params[j]$ = $F[i][j]$
        $params[j]$ = Vectorize($params[j], g$)
        Append($params[j]$, $P_j$)
    end for
    Append($event$, $E$)
end for
return $E, P_1, \ldots P_m$
```

Due to the variation in the structure and size of the vocabulary, we use these parameters in a limited character space. Each parameter, once translated into characters, can be restricted within a reasonably-sized space for learning. We use these parameters by learning character grams from them. The process of constructing character grams, or Char-Grams, is explained in Algorithm 1.

The corpus in Algorithm 1 is built using each parameter associated with any event API within the entire dataset. We convert each of these parameters into sequences tokenized over characters instead of words. We then build n-grams over these characters in the corpus, followed by the construction of a frequency distribution over the set of n-grams. Due to the extremely large number of total n-grams, we extract only the top-K, most frequent n-grams from this distribution and use them as our bag of words for the parameter dataset. Using Algorithms 2 and 3, each parameter can now be represented as a $K$-sized sparse vector with each position specifying the presence or absence of a specific n-gram. Algorithm 2 helps generate a new sequence from these sparse vectors that can be used as an updated input for the model, where the events are vectorized on a fixed vocabulary, and the parameters are vectorized using the process described above.

Attaining the sparse vector representations of the parameters allows us to use them as a part of a neural model along with the system API calls. However, using them directly in this form with the API call sequences would drastically increase the parameter space for sequence learning. We, therefore, propose the use of a shared dense $\tanh$ layer through which each parameter representation passes be-

**Algorithm 3** VECTORIZE

```
  Input: Parameter $\rho$, Grams $g$
  Initialize $vec = Zeros(len(g))$
  for $i = 0$ to $len(g) - 1$ do
      if $g[i] \in \rho$ then
          $vec[i] = 1$
      end if
  end for
  return $vec$
```

**Algorithm 4** API-PARAMETER SEQUENCE GENERATOR

```
  Input: File sequence $F$, Parameter Count $m$, Grams $g$
  $E, P_1, \ldots P_m$ = PARAMETERIZE($F, m, g$)
  Initialize $E' = [\,]$
  for $i = 0$ to $len(F) - 1$ do
      for $j = 1$ to $m$ do
          $R_{Pj} = \tanh(W_P * P_j[i])$
      end for
      $E'[i]$ = CONCATENATE($[E[i], R_{P1}, \ldots R_{Pm}]$)
  end for
  return $E'$
```

fore being used with the remaining model. Learning of the weights, $W_P$, for this layer is directed by the loss from the final label associated with the entire sequence only and is done when backpropagating the loss gradient through the complete model. Algorithm 4 explains the process of obtaining updated sequences containing the combined event and parameter information in the form of embeddings.

### 2.2. Sequential Processing

Once we have obtained the updated sequence consisting of events and parameter embeddings, we need to use sequential learning mechanisms that can help us capture the vertical relationship in the sequence along with the horizontal relationships derived using parameter embeddings. While parameter embeddings provide a *more complete* representation of each event, their co-occurrence with other events can only be learned through sequence learning. Similar to the neural models discussed in [10], we pass the updated sequence through LSTM layers for sequence learning. Our *Events Only* model is based on [10] and can be used with a variable number of LSTM layers, while being trained directly from the final label. We use $N_{LSTM}$ to denote the number of LSTM layers used over the sequences.

Like [6], we perform temporal max pooling over the LSTMs in order to derive a vectorized representation of the sequence, that can be used with classifiers. Since the detection of malware relies on signals observed throughout the sequence, performing max pooling over the sequence, instead of using the final activation from the LSTM, helps retain any relevant activation learned throughout the sequence, and not necessarily the predicted activation after a certain point within the sequence.

### 2.3. Classification

The primary objective of the neural models performing malware detection is to classify a file sequence input as being *malicious* or *clean*. Neural classification layers include the use of linear models like logistic regression, and denser models like multi-layer perceptrons. We built our system to use $H$ variable hidden dense ReLU layers followed by a *sigmoid* layer for predicting the probability of input file being malicious.

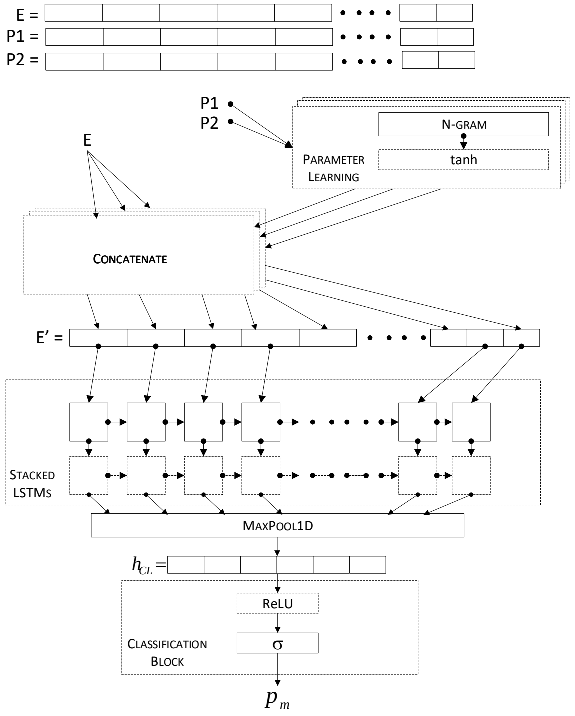

**Fig. 1**. System Architecture using Events and Parameters

This classification module completes our model, which takes a file sequence as input, and a malware label as the model target. The *Events Only* model uses the sequence of just the events, whereas the *Events + Parameters* model uses events along with the parameters associated with each event. The classification module for both models is the same and operates on vectorized representations obtained from the LSTMs.

The classification layer is supervised with the label associated with each file, which is used to measure a loss while training the model. This loss is used to update the weights throughout the model using backpropagation and is connected completely from input sequences to the final prediction. The *Events + Parameters* model using two parameters with each event is illustrated in Figure 1.

## 3. EXPERIMENTAL RESULTS

In this section, we discuss the datasets which are used to train and test our model, describe the experimental setup which was used to conduct the experiments, and present the results which compare our model with previously proposed models.

**Datasets**: We collected the system API calls and their parameters from 75,000 files, which were evenly split between malware and benign files, in September 2017. The data from all of the files is unique, and there is a total of 158 event types in our data. This collection was next randomly split into training, validation, and test sets of 50,000, 10,000, and 15,000, respectively, while maintaining the even split between malware and benign files.

**Experimental Setup**: All models are implemented and evaluated using Keras [14] with the Tensorflow [15] backend. We follow the model settings for the CNN-LSTM as suggested by Kolosnjaji *et al.*, in [7].

We next describe the parameter settings for all of the recurrent models. The maximum sequence length is 200, and the hidden layer dimension is set to 1500. The minibatch size is equal to 128, and the embedding size is 114. The number of epochs is set to 5 for training the recurrent model with parameters and 15 for the remainder of the models. All recurrent models include bag-of-words features in addition to the events and possibly their parameters. Parameter pre-processing uses character n-grams of size 3 (tri-grams). The maximum length of the parameters is set to 24. We use the top 10,000 tri-grams to vectorize each parameter. The output dimension of the $tanh$ processing layer for the parameters is set to 256.

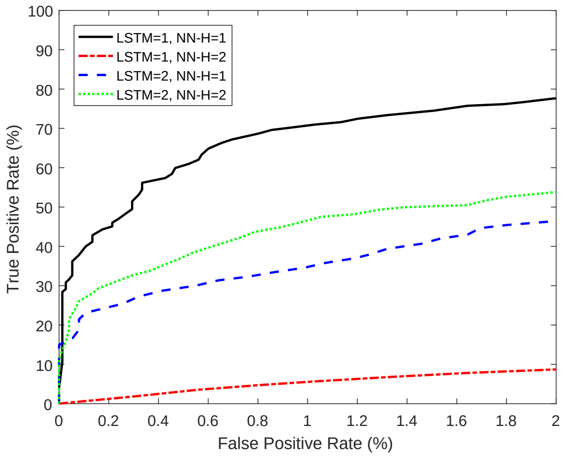

**Fig. 2**. ROC curves for the *Events Only* LSTM+NN models. The model with a single layer LSTM and a single hidden layer neural network offers the best performance among these models.

**Model Evaluation**: Athiwaratkun and Stokes originally proposed an *Events Only* LSTM-NN model with a two-layer LSTM, $N_{LSTM} = 2$, and a shallow neural network classifier with a single hidden layer, $H = 1$. We evaluate the performance of the *Events Only* LSTM-NN architecture for our data with all configurations of $N_{LSTM} \in \{1, 2\}$ and $H \in \{1, 2\}$ in Figure 2. The figure clearly indicates that the best performing *Events Only* LSTM-NN model for our data is $N_{LSTM} = 1$ and $H = 1$.

With $N_{LSTM} = 1$ and $H = 1$, we next evaluate the performance of the proposed *Events* + *Parameters* model in Figure 3. We also include the results of three *Events Only* models, including the CNN-LSTM and two versions of the LSTM-NN models for comparison. Our *Events* + *Parameters* model clearly dominates all of the *Events Only* models, and offers significant improvement over all models, particularly at lower false positive rates (FPRs). This performance is notable because malware classifiers must be tuned to have low false positive rate so that they do not remove benign files including those installed by the operating system (OS). Quarantining or removing operating system critical files could prevent the OS from booting rendering the computer useless.

A key measurement is the amount of time required to train these models. The time required to train and test the proposed *Events* + *Parameters* model is 6 days, 7 hours and 51 minutes. Similarly, the training and test time of the best LSTM-NN baseline is 3 hours 23 minutes.

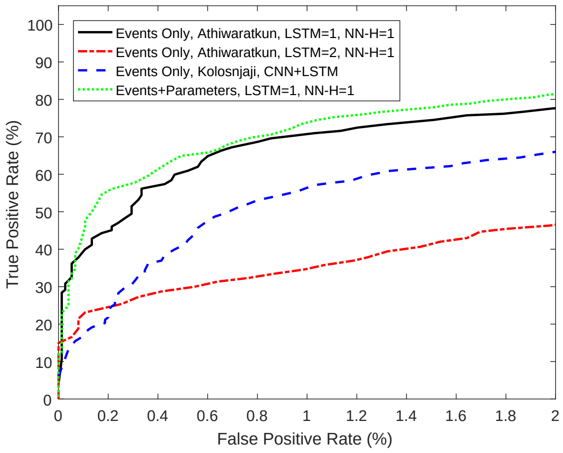

**Fig. 3**. Comparison of the ROC curves for two versions of the *Events Only* LSTM+NN model, the *Events Only* CNN+LSTM model, and the proposed *Events* + *Parameters* model.

## 4. CONCLUSIONS

The parameters associated with API calls are an important aspect of programmatic behavior. Sequential models which fail to include the parameters are ignoring important information. The results from Section 3 demonstrate that including the parameters significantly improves the performance compared to previously proposed models which do not include parameters as input.

This paper opens a new space of exploration by enabling the use of parameters within sequential models trained entirely via back-propagation. Methods described in Section 2 confirm the possibility of capturing extensively variable parameter information in a vector space for learning. Independent model design of parameter pre-processing also ensures the use of these parameters in sequential models as the trainable vectorized form can be attached with any sequence at the input level.

## 5. REFERENCES

[1] N. Idika and A.P. Mathur, "A survey of malware detection techniques," Tech. Rep., Purdue Univ., February 2007.

[2] Jeffrey O. Kephart, "A biologically inspired immune system for computers," in *In Artificial Life IV: Proceedings of the Fourth International Workshop on the Synthesis and Simulation of Living Systems*. 1994, pp. 130–139, MIT Press.

[3] George E. Dahl, Jack W. Stokes, Li Deng, and Dong 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.

[4] Joshua Saxe and Konstantin Berlin, "Deep neural network based malware detection using two-dimensional binary program features," *Malware Conference (MALCON)*, 2015.

[5] Wenyi Huang and Jack 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.

[6] Razvan Pascanu, Jack W. Stokes, Hermineh Sanossian, Mady Marinescu, and Anil Thomas, "Malware classification with recurrent networks," in *Proceedings of the IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP)*. IEEE, 2015, pp. 1916–1920.

[7] Bojan Kolosnjaji, Apostolis Zarras, George Webster, and Claudia Eckert, "Deep learning for classification of malware system call sequences," in *Australasian Joint Conference on Artificial Intelligence*. Springer International Publishing, 2016, pp. 137–149.

[8] Sepp Hochreiter and Jurgen Schmidhuber, "Long short-term memory," in *Proceedings of Neural Computation*, 1997, pp. 1735–1780.

[9] Felix A Gers Jj, Urgen Schmidhuber, and Fred Cummins, "Learning to Forget: Continual Prediction with LSTM," 1999.

[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] Kyunghyun Cho, Bart van Merrienboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio, "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation," jun 2014.

[12] Xiang Zhang, Junbo Zhao, and Yann LeCun, "Character-level convolutional networks for text classification," in *Proceedings of the 28th International Conference on Neural Information Processing Systems - Volume 1*, Cambridge, MA, USA, 2015, NIPS'15, pp. 649–657, MIT Press.

[13] P.J. Werbos, "Backpropagation through time: what it does and how to do it," *Proceedings of the IEEE*, vol. 78, no. 10, pp. 1550–1560, 1990.

[14] François Chollet et al., "Keras," `https://github.com/fchollet/keras`, 2015.

[15] Martín Abadi et al., "TensorFlow: Large-scale machine learning on heterogeneous systems," 2015, Software available from tensorflow.org.