Scientific Research Publishing

Journal of Information Security, 2020, 11, 103-120
https://www.scirp.org/journal/jis
ISSN Online: 2153-1242
ISSN Print: 2153-1234

# Malware Detection for Forensic Memory Using Deep Recurrent Neural Networks

**Ioannis Karamitsos[1], Aishwarya Afzulpurkar[1], Theodore B. Trafalis[2]**

[1]Rochester Institute of Technology, Dubai, UAE
[2]University of Oklahoma, Norman, USA
Email: ixkcad1@rit.edu, aa3852@rit.edu, ttrafalis@ou.edu

**How to cite this paper:** Karamitsos, I., Afzulpurkar, A. and Trafalis, T.B. (2020) Malware Detection for Forensic Memory Using Deep Recurrent Neural Networks. *Journal of Information Security*, **11**, 103-120.
https://doi.org/10.4236/jis.2020.112007

**Received:** April 1, 2020
**Accepted:** April 23, 2020
**Published:** April 26, 2020



## Abstract

Memory forensics is a young but fast-growing area of research and a promising one for the field of computer forensics. The learned model is proposed to reside in an isolated core with strict communication restrictions to achieve incorruptibility as well as efficiency, therefore providing a probabilistic memory-level view of the system that is consistent with the user-level view. The lower level memory blocks are constructed using primary block sequences of varying sizes that are fed as input into Long-Short Term Memory (LSTM) models. Four configurations of the LSTM model are explored by adding bidirectionality as well as attention. Assembly level data from 50 Windows portable executable (PE) files are extracted, and basic blocks are constructed using the IDA Disassembler toolkit. The results show that longer primary block sequences result in richer LSTM hidden layer representations. The hidden states are fed as features into Max pooling layers or Attention layers, depending on the configuration being tested, and the final classification is performed using Logistic Regression with a single hidden layer. The bidirectional LSTM with Attention proved to be the best model, used on basic block sequences of size 29. The differences between the model's ROC curves indicate a strong reliance on the lower level, instructional features, as opposed to metadata or string features.

## Keywords

BiLSTM, Deep Learning, Forensic Memory, LSTM, RNN

## 1. Introduction

The recent explosion in Internet of Things (IoT), Big Data and social networking technologies has unintentionally led to an increasing rise in global cyber threats.

Isolated attacks that previously exploited common vulnerabilities across systems have now given way to "scan-based" attacks that identify and exploit system-specific vulnerabilities across networks. These networks of compromised nodes, commonly referred to as botnets, representing personal computers, cellphones, even fax machines [1], morph, divide and link malware components in a way that allows unhindered, undetected propagation at incredibly fast speeds. Not only is the process of identifying and stopping the malware complicated, but certain species such as spyware make it difficult even to suspect that there has been an attack. Zero-day attacks that target firmware or hardware level components of a system mostly make it impossible to remove the malware, without complete disassembly of the motherboard [2]. The scope of malware and its detection methods is prohibitively enormous so that this paper will be focusing on one crucial aspect: volatile memory analysis on personal computers.

Memory forensics refers to the extraction and analysis of reliable volatile and non-volatile memory "dumps" in order to infer the state of a machine at a given time interval. It is preferred over the injection of higher-level APIs as the latter is prone to interference by malware, whereas the latest memory acquisition approaches have successfully been able to extract uncorrupt views of the system [3]. Analyzing memory blocks to reveal higher-level information has so far been in the realm of a handful of security experts as it requires extensive knowledge and expertise in the area and is difficult to automate in a data-agnostic fashion.

Recent advances in machine learning and deep learning have generated a plethora of new probabilistic approaches for malware detection, without the need for extensive expert analysis [4] [5] [6] [7]. Deep learning has achieved tremendous results in several areas such as natural language processing, that were previously thought to need supplementary semantic or causal models [8] [9]. The significance of this research is to explore the application of deep learning to forensic memory analysis, such that the expertise required to analyze a system's memory will, to an extent, be acquired by the system itself.

The primary motivation for this work is to detect neural networks patterns in machine-level code that may directly link to the functionality of that code. Specifically, it was to find a direct mapping between lower-level machine/assembly code and the functional requirements of or expectations from an application, circumventing API calls and other higher-level functionality. The fact that the attention-based model outperformed the model using the max-pooling layer tells us that there are such patterns. The paper is organized as follows. In Section 2 and Section 3 discuss the host-based intrusion detection related work relevant to this paper. Section 4 describes the data mining methodology. In Section 5, we describe four different models based on Long Short-Term Model (LSTM) with the description of each block. Section 6 discusses the performance analysis and results. That is Section 6-1 compares the data mining results obtained from four different block sizes against the four selected models. Section 7 and Section 8 contain the conclusions and the future work.

## 2. Host-Based Intrusion Detection

The success of machine learning algorithms applied to the problem of malware detection depends heavily on the quality of the extracted features. Traditional machine learning approaches are simplistic and cannot extract contextual information from raw data, so current research is moving towards employing deep learning pipelines to be able to propagate different representations of features, each layer of the pipeline ingesting a more contextually complex set of features. Malware detection can be performed by either analysing and abstracting the functionality of benign programs or doing so for malware. Having generated a hidden representation of benign programs, deviations from this representation, or anomalies, are then labelled as malware. Solely analyzing malicious programs allows one to gain an understanding of the general concepts used by malware and can generate a more substantial categorization for different types of malware. This allows analysts to create a bounding box around the general concepts exploited by malware. This is insightful because malware behavior can mimic behavior of benign apps to a large extent. Hence deviations from "normal behavior" can be challenging to detect and understand the inner workings of specific categories of malware yields rich features that can be fed into a "benign program analysis" framework, in order to generate a combined pipeline of different inputs, different models and more accurate outputs. Using such combinations, throughout literature, has proven always to be more effective than focusing on a specific set of functionalities that span the system, but fail to branch out effectively.

Features used in machine learning-based malware detection can be divided into three subcategories: 1) using a lower-level machine or assembly code and 2) using higher-level API calls, system logs, and events generated by applications and 3) combination of the two categories.

### 2.1. Low-Level Machine or Assembly Code

Low-level code used to be off-limits simply because of its nearly unreadable semantics—the recent surge in lower level malware has pushed specialists to develop tools that help with the readability and analysis of assembly code.

An industry company Sophos [10] introduced Invincea which, it uses raw data without unpacking or filtering binaries to train and deploy a low resource model quickly. It uses PE import features, metadata information on PE files, and binary values of a two-dimensional byte entropy histogram that models a file's distribution of bytes. The data is fed into multiple deep neural networks (DNN) models, using ReLU as activation function and high dropout rates. The insight found was that string values corresponding to metadata and aligning with higher-level semantics override lower-level details.

Raff *et al.* [11] introduce an approach of Byte n-grams, and it is a commonly used feature type in static analysis. It condenses information within files such every n-gram is a unique combination of n consecutive bytes. This feature type is easy to construct as it requires almost no knowledge of context. It was observed

that n-grams rely on and better generalize ASCII string information, or predefined data, rather than execution information contained in program code. N-grams can be constructed for program-level, language-specific bytes, opcode, or machine language bytes, or DLL and PE import information bytes. Similar accuracies and results were observed across multiple values of n and types of n-grams used along with regularized Elastic-Net models [11]. This reflects the fact that n-grams extract little abstract, contextual information beyond string values, compared to models run on hand-crafted features.

Raff *et al.* [12] use the concept of KiloGrams, and it has reported positive results by increasing the size of n-thousand fold. The difference in accuracy and feature retention becomes significant at n = 64. The theory behind this was that at large n sizes, the top k most frequent features would start getting redundant, allowing for a higher number of features to be included—the concept of approaching a redundancy limit on features by increasing the value of n before adding more features and repeating the process is referred to as hashing stride. This shows the lack of effectiveness of machine learning for narrowly scoped tasks—training machine learning algorithms on chunks on data that appear within the same spatial and temporal space results in less generalizability and worse results.

Qiao *et al.* [13] attempts to detect function boundaries using raw binary file bytes as data. In general, it was assumed that analyzing return and call pointers as the only "features" would give one a reasonably good idea about function boundaries, but the paper shows that the accuracy of such manual analysis is about 70%. In contrast, a machine learning-based approach, which uses features that a manual analyst may deem irrelevant, increases it to 98% accuracy

## 2.2. High-Level API Calls

API calls cover a variety of functional spaces: registry, network sockets, and memory management. Combinations of inputs and corresponding outputs of the same API calls but in different parts of a sequence of execution are fed into several kinds of machine learning models to extract insights or predictive capacity.

Temporal API [14] call information is harder to capture and is susceptible to more attacks—these features are usually modelled using Markov Chains with a prior understanding of the general movement of calls. What makes temporal information challenging to process and utilize the fact that most systems allow asynchronous processing of variable length API and system calls. This makes it necessary to preprocess temporal API call data using statistical analysis and information-theoretic techniques before being able to analyze it and gain insights. Usually, a combination of local and global calls is used for features. Local calls provide a large number of features within context. However, garbage collection and timing-based attacks can successfully evade local call analysis, by injecting code at the system call level. Global calls make more robust feature sets as they

take into account entire call streams across scopes, instead of individual calls that just locally branch out.

Behavior API [15] call graphs have become a popular trend in the recent past. Heterogeneous information networks classify API calls using scope; for example, two calls in the same file would have a "meta-path" connecting them. Different meta-paths are used for different scopes, including function scope, package scope, code block, and invoke method. While theoretically sound, HINs fail to capture patterns outside of those that are already visible using standard, non-HIN techniques [16]. This is because API calls are only as specific as the symbol table allows them to be, and any changes made dynamically in the symbol table are never reflected in the higher-level calls during program execution.

A right amount of machine learning effort has been put in with Android API calls, both in the context of dynamic and static analyses. DroidDolphin used Support Vector Machines (SVM) to build a model using thirteen API call features and deployed the checking engine dynamically. CopperDroid used system calls, which are a level below application-based calls, and process communications (intra as well as inter) as features that were sandboxed and dynamically monitored by a virtual machine inspection framework. DroidMat used applications permissions and intention messages as additional input along with API calls as features for a k-NN classifier. DroidMiner [17] used an associative classifier using simple API calls, and many others have acquired similar but different results using the same features on different models such as Decision Trees and Regression-based algorithms.

The general problem with using higher-level semantics is that it captures a higher level, abstract relationships that can remain intact even on a compromised system. While this allows one to retain general contextual information over more extended periods, it does not keep track of lower-level changes that have a structure of their own. Malware that operates at the firmware level will almost always be able to evade systems that use API calls as features, hence the need for processing assembly code.

## 3. Combination High-Level and Low-Level Data

The danger in combining lower level and higher-level data is that lower-level data may be perceived by an abstraction engine to be too noisy and chaotic and might be inclined to ignore that data. It is essential to systematically ensure that higher-level features add value to the lower level ones, instead of overriding and diminishing them.

Microsoft [18] using sequential models on the lower level, opcode sequences along with higher-level system logs. The two models tested were: RNNs which have proven their strength with handwriting and speech recognition, and ESNs, which are used extensively for chaotic systems. They show that RNNs failed to capture salient features and that Echo state network (ESN) models utilized all features more effectively and resulted in much better accuracy. The hidden states

of the recurrent models were fed into a temporal max-pooling layer, to detected reordered temporal patterns, and logistic regression was used afterwards for performing the final classification. Only half the hidden states were fed into the classifier (referred to as half-frame), to avoid overfitting and leaky units were used to increase long-term memory.

Athiwaratkun *et al.* [19] present several different training models and configurations, including Character level CNNs, RNNs, LSTM/GRU models and Attention models. In this study, a number of 75,000 Windows PE files were used, which were equally split between malicious and benign files were trained, validated and tested. The LSTM model with temporal max-pooling outperformed the Char-CNN, GRU, Attention-based ones as well as all previously tested models, *i.e.* RNN/ESN.

Apart from HIN models, the above literature presented machine learning models based on file-based data, *i.e.* performed classification on files rather than smaller or larger chunks and allowed a human analyst to narrow down the problem using patterns in infected files. This strategy may not always work because a) malware can be disseminated across entire suites of files affecting critical and wide-ranging functionality or b) changes to single files can be too subtle for even a rule-based checking engine to detect them. This is evidently in practice already as malware is increasingly exploiting techniques such as return-oriented programming (ROP), Just-in-time compilation (JIT) and concept drift.

DeepCheck [20] uses CFI checking using a deep learning model. CFI retains lower-level branches, so it is fundamentally stronger but can be inefficient: updates in programs can completely change CFGs, which requires modifying runtime libraries or binary files themselves—often not recommended because of security loopholes in doing so and the possibility of unexpected program execution. DeepCheck [20] requires no such modifications because it uses an external framework based on predictions generated by the machine learning engine. It uses lower-level details, *i.e.* the Intel Processor Trace, logging branches taken and not taken, as well as application binary code. This study provided good accuracy scores, and it showed that a multi-layered deep NN architecture is performed better than using a single and large NN for malware detection.

Kernel Object Detection using Deep RNNs [21] presents the graph approach. A graph is constructed from linked and adjacent lower level memory blocks and fed into a graph neural network as features, from which kernel objects were identified. The high classification accuracy of the model was dependent on the fact that kernel objects are represented in memory in a probabilistically deterministic way. Although subject to high amounts of feature preprocessing, the model in this study presents a good benchmark for further tests of similar nature.

## 4. Methodology

Before presenting the different proposed configurations in the next section, we

first describe the dataset used in this study.

### 4.1. Data Collection

The raw data used for this paper came from memory dumps of single processes running on a Windows OS—this includes loaded modules, including both statically and dynamically linked libraries. Assembly code for portable executable (PE) 50 files on Windows format was gathered using IDA Dissasembler [22] including code for application files such as Google Chrome, Notepad, Command Prompt, and kernel files such as NTOSkrnl and NSLOOKUP. Our dataset is further randomly splitted into three datasets a ratio of 60:20:20 for the training, validation and testing sets.

The need to train for several epochs and also using a larger dataset would mean possibly training on a more massive cluster of GPUs, like those available on the cloud. The models for this study were trained and tested locally, on a Core i7 laptop to test for resource consumption patterns. It took over 48 hours to run each model configuration without "hot start" configured. Training models on the cloud would have resulted in superior performance, and access to more binaries may have resulted in better accuracies, comparable to previous literature.

Ten types of malware published online, including Trojans, Rootkits, zero-day attacks, that operate both on the host as well as a network were downloaded. All malware samples affected the kernel and RPC call functionality. So out of the 50 portable executables (PE) files used, 38 of them were affected by running the malware samples. Both the benign version as well as malicious versions of the files were used. For this study, a total of 88 sample PE files were used for data.

In Figure 1 within each block, a single call and a single jump exist, which

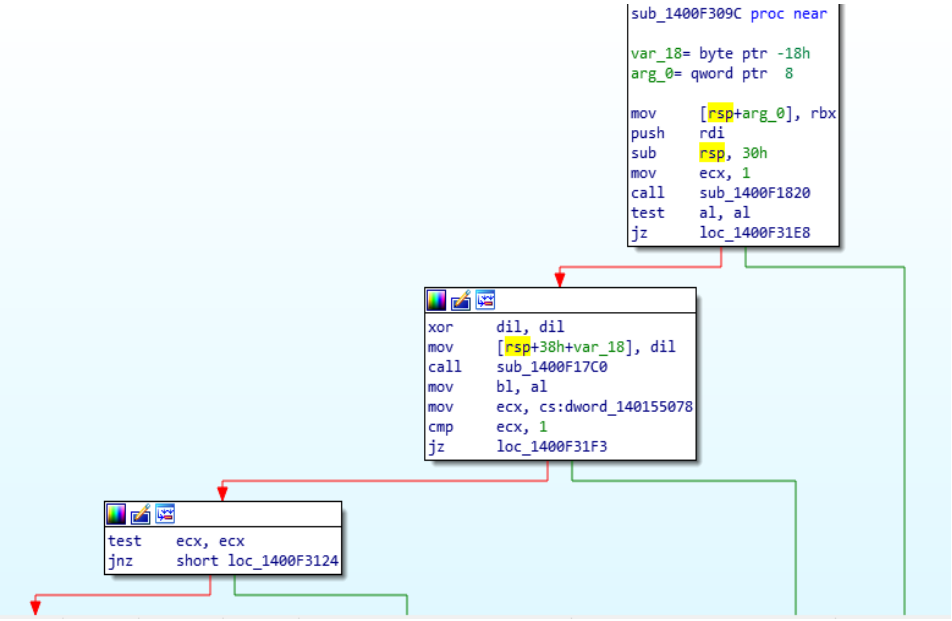


Figure 1. IDA's graphical representation f basic blocks in chrome.exe.

are the two indirect blocks, which point to other direct blocks (the green arrow for the call path, the red for jumps). The opcode is the string on the left, *i.e.* xor, mov, and function boundaries are conservatively assumed to be at call locations that do not revert to the block they stemmed from.

A script was written to read IDA's Basic block Extraction utility and modify the structure of the data, so it resembled a 3-D matrix. Keras TF APIs [23] was used to build the model, train, validate and test the data on a single laptop with Core i7. The reason for not porting the model to the cloud was because of the small number of files analyzed, as the creation of long short-term memory (LSTM)-consumable input was not purely automated. Differences in IDA's interpretation of file headers and data variables required some amount of manual analysis and crafting of features, though not as much as required by a purely manual feature base.

### 4.2. Data Processing

The input type for long short-term memory (LSTM) is a three-dimensional matrix, the axes representing the sequence length, timesteps, and batch size. Four basic block sequence configurations (*i.e.* direct-indirect-direct is a configuration of size 3) were tested:

1) a chain of direct and indirect blocks of size 5.
2) a chain of direct and indirect blocks of size 9.
3) a chain of direct and indirect blocks of size 19.
4) a chain of direct and indirect blocks of size 29.

The reason for choosing basic blocks instead of whole functions as datum was that there are almost always more basic blocks than functions. Indirect calls are prime sites for exploitation and considering whole functions with multiple indirect calls as one chunk risked leaving out important contextual information about the general nature of indirect calls, *i.e.* a pattern of appearance. Conditionals, loops, jumps, calls are all indirect edges connecting basic blocks, which consist of a determined sequence of executions. This also made feature extraction easier as extrapolating function boundaries would have required another machine learning model to be run as a preprocessing step.

The purpose here is to assign an execution order to an incoming stream of execution events. Execution events can be defined in terms of basic blocks, which are transformed into block embeddings. Our goal is to identify an original short-term order of execution and possibly a general, app-wide, broad longer-term order, without a need for order in intermediate block sequence orderings. This is because intermediate orderings often contain exponential state spaces that are hard to manage, as opposed to short-term and long-term orderings.

## 5. Models under Study

The base model chosen for this paper is the long short-term memory (LSTM) model as it is excellent at retaining more extended periods of information and

has been probably more accurate than other sequence-based models previously tested in the literature. The LSTM model has also been used in natural language processing (NLP), for sentence parsing and document classification. The hierarchy of words to sentences and paragraphs to documents is similar to the hierarchy of basic blocks to short, intermediate and longer-term sequences. The regular recurrent neural networks (RNN) has a limited capacity to remember word probabilities beyond the phrase level and the LSTM enhances its capacity by remembering bursts of short-term sequences on a longer-term. This makes the LSTM better able to predict the next words in sentences and paragraphs, but not in entire documents. In this study, four different model configurations are tested:

Model 1) Pure LSTM with temporal max pooling and Logistic regression for classification

Model 2) Pure LSTM with attention layer and logistic regression for classification

Model 3) Bidirectional LSTM with temporal max pooling and Logistic regression for classification

Model 4) Bidirectional LSTM with attention layer and logistic regression for classification

In the following Figure 2 the four models are depicted.

In all four configurations, the LSTM predicts the next-in-sequence basic block. Instructions within the basic block are features of the x-axis, basic sequential blocks populate the y-axis, each row ending at a maximum of N basic blocks (one of 5, 9, 19, and 29).

### 5.1. Long Short Term Memory (LSTM)

Long Short Term Memory (LSTM) concept are proposed firstly by Hochreiter

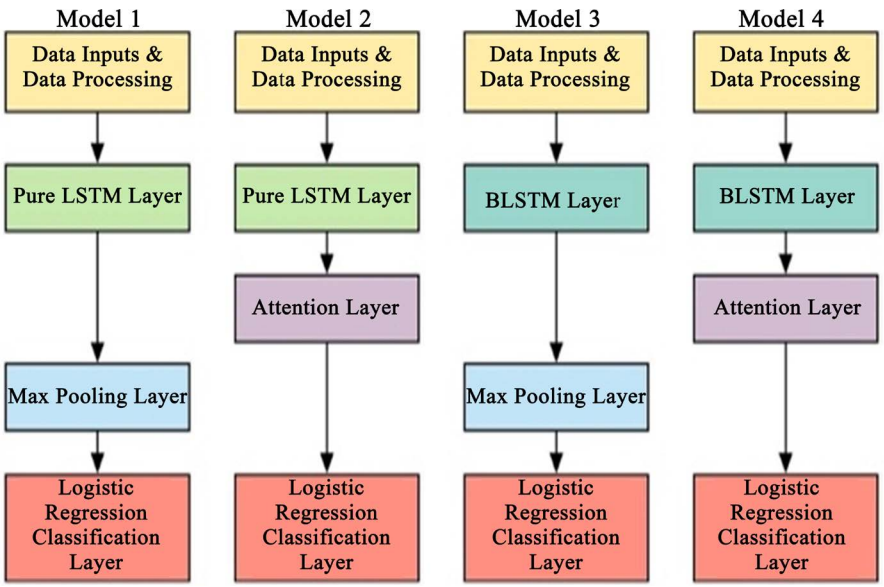


Figure 2. Four different models under study.

and Schmidhuber [25] to overcome gradient vanish. LSTM is a variant of the Recurrent Neural Network (RNN) with two important differences. LSTM uses a cell state $c_t$ which serves as explicit memory and for the hidden states computed four interactions that give the network the ability to remember or forget specific information about the preceding element of sequence. The main idea is to introduce an adaptive gating mechanism, which decides the degree to which LSTM units keep the previous state and memorize the extracted features of the current data input. Typically, a LSTM-based recurrent neural network unit consists of four components: one input gate $i_t$ with corresponding weight matrix $W_{xi}, W_{hi}, W_{ci}, b_i$; one forget gate $f_t$ with corresponding weight matrix $W_{xf}, W_{hf}, W_{cf}, b_f$; one output gate $o_t$ with corresponding weight matrix $W_{xo}, W_{ho}, W_{co}, b_o$, all of those gates are set to generate some degrees, using the current input $x_t$, the state $h_{i-1}$ that was generated by the previous step, and current state of this cell $c_{i-1}$ for the decisions whether to take the inputs, that forget the memory stored before, and output the state generated latter. The four layers are presented in the following equations:

$$i_t = \sigma\left(W_{xi}x_t + W_{hi}h_{t-1} + W_{ci}c_{t-1} + b_i\right) \tag{1}$$

$$f_t = \sigma\left(W_{xf}x_t + W_{hf}h_{t-1} + W_{cf}c_{t-1} + b_f\right) \tag{2}$$

$$g_t = \tanh\left(W_{xc}x_t + W_{hc}h_{t-1} + W_{cc}c_{t-1} + b_c\right) \tag{3}$$

$$c_t = i_t g_t + f_t c_{t-1} \tag{4}$$

$$o_t = \sigma\left(W_{xo}x_t + W_{ho}h_{t-1} + W_{co}c_{t-1} + b_o\right) \tag{5}$$

$$h_t = o_t \tanh\left(c_t\right) \tag{6}$$

Hence, the current cell state $c_t$ will be generated by calculating the weighted sum using the previous cell state and current information generated by the cell [27].

### 5.2. Bidirectional Long Short Term Memory (BLSTM)

Bidirectional LSTM networks extend the unidirectional LSTM networks by introducing a second layer, where the hidden to hidden communication flow is in opposite order. The model is able to exploit information both from the past and the future. In the following **Figure 3** a bidirectional LSTM configuration is depicted.

### 5.3. Attention

Attentive neural networks have recently demonstrated success in a wide range of tasks ranging from questions answering, machine translations, speech recognition, to image captioning [24] [26]. Bahdanau *et al.* [24] proposed a new attention mechanism to align the input and output sentences in the context of neural machine translation. In this work, we adapt their attention mechanism for the models 2 and 4 as an alternative to the temporal max pooling proposed by Pascanu *et al.* [18]. While temporal max pooling chooses the maximum hidden

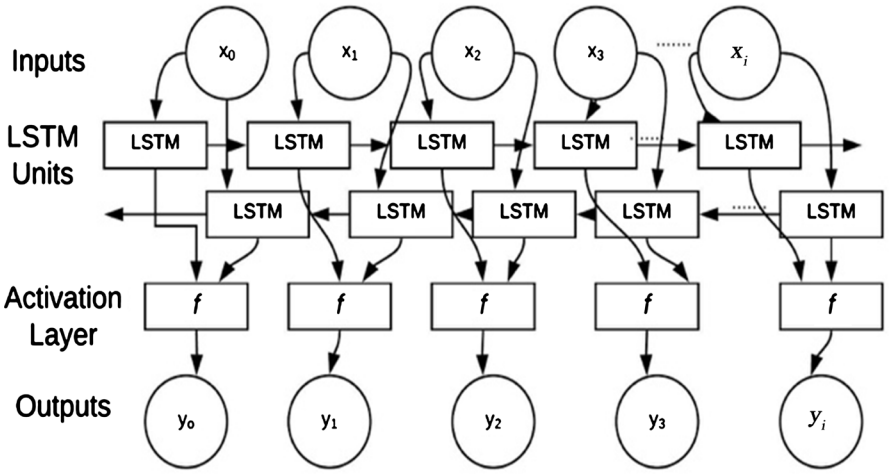


**Figure 3.** BiLSTM model with activation function.

units across all of the hidden vectors in the sequence, the attention mechanism instead first learns the attention score for each time step and then compute the temporal average of all hidden vectors. The attention mechanism generates a vector $\boldsymbol{h}^{att}$ of length D where:

$$\boldsymbol{h}^{att} = \sum_{t=0}^{T-1} a^t \boldsymbol{h}^t \tag{7}$$

and the attention scores $a^t$ are calculated from a dense network with $\boldsymbol{h}^t$ as an input. The final vector representation is used as the input to the logistic regression classifier.

### 5.4. Max Polling

Once the LSTM has run, its hidden states are passed in as input to the temporal max-pooling layer. This layer acts as an aggregator and makes sure that blocks belonging to the same row entries in the data matrix can be reordered (taking into account execution optimizations and lower-level resource handling). The temporal max-pooling layer outputs single feature vectors corresponding to the basic blocks within individual modules of individual files. The reason for dividing files into modules was because the malware used for this study it targets only a subsection of functionality within entire files, and we get labels for the specific functionality being targeted by each malware specimen. Let $\boldsymbol{h}^t$ be hidden vectors generated from a recurrent neural network for time steps $t = 0, \cdots, T-1$ where each file event sequence is divided into subsequences of length T. Temporal max pooling generates a vector $\boldsymbol{h}^{\max}$, where $h_i^{\max} = \max_{t \in 0, T-1} h_i^t$ for index $i = 0, \cdots, D-1$, and $D$ is the dimension of $\boldsymbol{h}^t$.

### 5.5. Logistic Regression Classification

In this study, we use a logistic regression classifier. Once the max-pooling layer outputs embeddings for the modules, a supervised logistic regression classification is performed to obtain the final classification. For this study, the logistic regression classification is done module-wise and not file-wise.

## 6. Performance Analysis

In this section, we evaluate the proposed configurations. In the following Table 1, we show the statistics of the row lengths, in terms of instruction size. The number of sequences denotes the total number of vertical entries in the LSTM input matrix or the number of rows. Each row contains instruction sequences of varying sizes, corresponding to the fixed number of basic blocks the instructions.

Table 1. Data statistics.

| Block Size | 5 | 9 | 19 | 29 |
|---|---|---|---|---|
| No of sequences | 51,768 | 28,760 | 13,623 | 8925 |
| Maximum instruction size | 78 | 124 | 254 | 306 |
| Minimum Instruction size | 11 | 57 | 97 | 110 |
| Mean instruction size | 33 | 88 | 152 | 201 |
| Median instruction size | 34 | 88 | 159 | 204 |

For block sizes 5 and 9, the block sequence ordering was retained the way it appeared within programs as the block size is small enough to generate enough randomness. For block sizes 19 and 29, the block sequence was randomized for half the data and retained the way it was for the other half. This configuration generated the best and most consistent results.

Intermediate results are omitted in this paper since it took more than 30 runs per model to achieve optimal accuracy. The process involved readjustment of weights, fine-tuning of the model and addition of optimizations and regularizations, as well as dropouts. The final LSTM configuration used for all 16 combinations of data and models was: Epochs = 50, Num_steps = 30, Batch_size = 20, Hidden_size = 500, Dropout = 0.3. The softmax activation function was used for the LSTM final layer, and LeakyReLU was used for intermediate layers, including the recurrent layers. The advantage of LeakyReLU over ReLU is that it maintains state in a stickier manner by allowing a small gradient when an LSTM unit is not active. Bias weight regularization was not helpful, but input weight elastic regularization (L1L2 = 0.01) showed significant improvement for the models. The logistic regression model included a single hidden unit, which performed better than multiple hidden units and used the ReLU activation function.

### Results

In this section, we compare the receiver operating characteristic (ROC) curves for the different configurations. We first evaluate different block sizes for the configuration of pure LSTM with temporal max pooling and logistic regression for classification. In Figure 4, block size of 29 and 19 are the two best performing configurations. For the false positive rate (FPRs) greater than 0.70%, the block size of 29 performs best. Block size 29 and 19 offers similar true positive rates (TPRs).

In Figure 5, the true positive rate (TPR) block size of 19 has slightly 1% lower

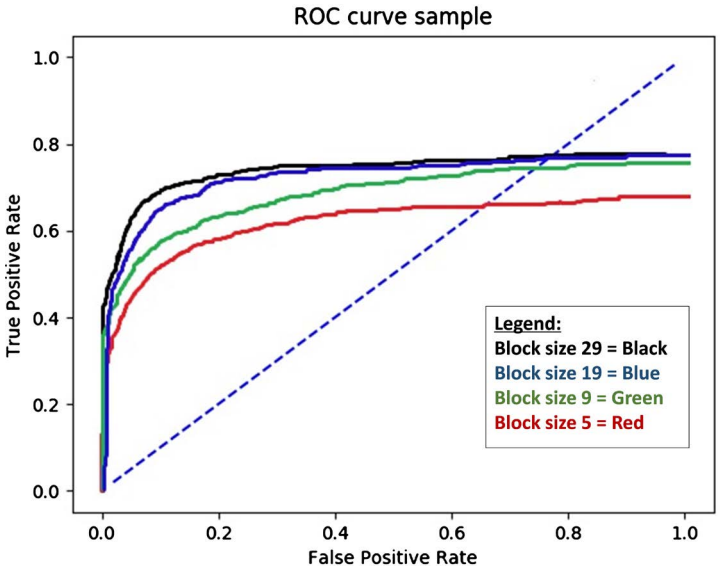


Figure 4. ROC curves for the Pure LSTM with temporal max pooling and logistic regression for classification. The largest block size 29 is performed best, and the smallest block size = 5 is performed worst.

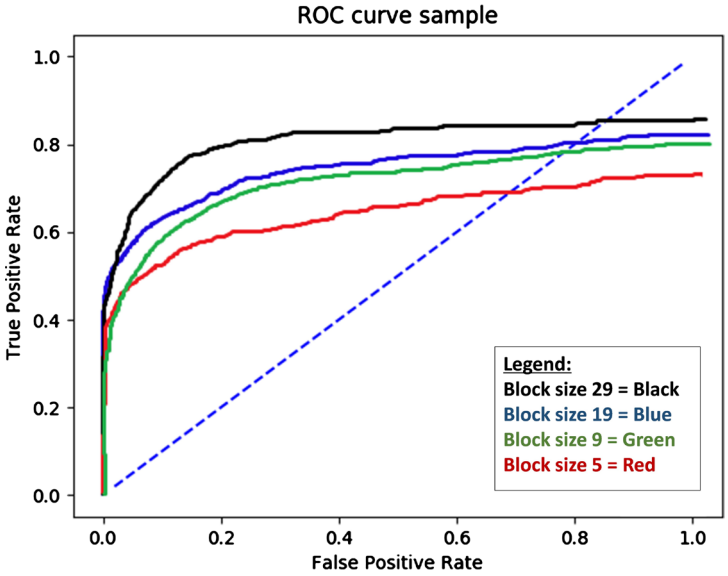


Figure 5. ROC curves for the Pure LSTM with attention and logistic regression for classification. The largest block size 29 is performed best, and the smallest block size = 5 is performed worst.

performance than the block size of 29. An alternate configuration was tried by feeding a combination of LSTM outputs generated by data of block sizes 29 and 19 into a standard temporal max pooling layer.

The performance of the bidirectional LSTM with temporal max pooling and logistic regression classification is depicted in Figure 6.

The ROC curves are more tightly clustered for FPRs greater than 0.80% for a block size of 29, 19 and 9.

The result was almost identical to the result produced by running just the data

with block size 29. Again, this was probably a result of greater randomness in smaller basic block sequences, which the model considered to be noise. So, the features of data with basic block size 19 consistently vanished, and the bi-directionality possibly enforced this phenomenon. Simply put, the smaller the block size, the greater randomness there is in the data, and the worse the performance.

Figure 7 depicts the performance for the bidirectional LSTM with attention

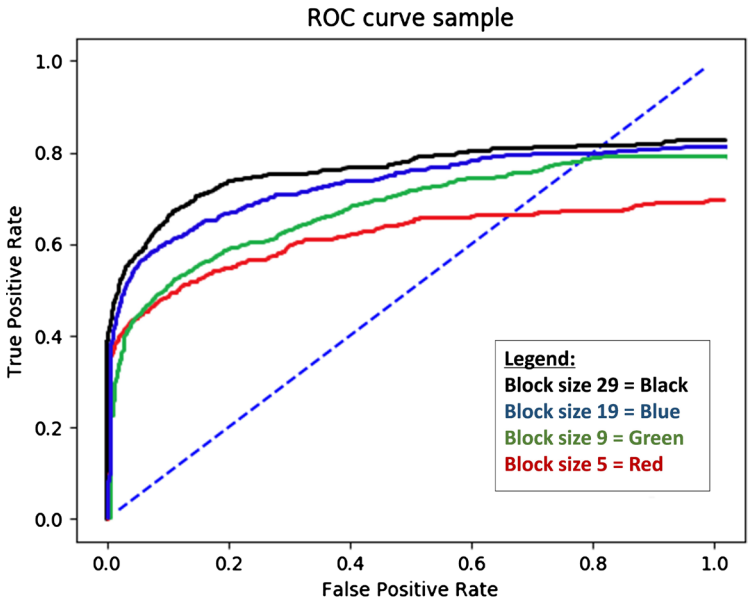


Figure 6. ROC curves for the bidirectional LSTM with temporal max pooling and logistic regression for classification. The largest block size 29 is performed best, and the smallest block size = 5 is performed worst.

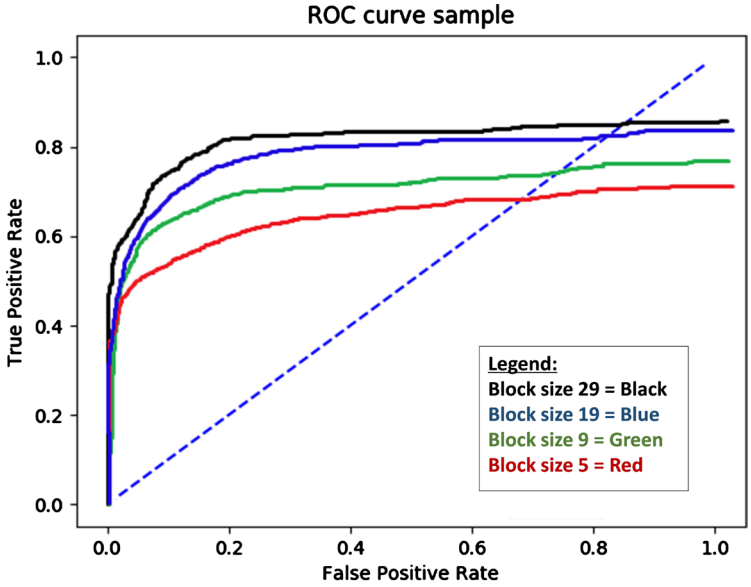


Figure 7. ROC curves for the bidirectional LSTM with attention and logistic regression for classification. The largest block size 29 is performed best and the smallest block size = 5 is performed worst.

and logistic regression classification.

Adding attention to the model affected results more positively than bi-directionality performed. It is possible that specific instruction sequences across basic blocks inherently reflected bi-directionality, by literally showing up in reverse order within the blocks. Specifically, the addition of attention increased the true positive rate faster than the false positive rate, and the flat-lined past the 0.3 false-positive rate mark. The same flat-lining can be observed for configuration depicted in the previous Figure.4, pure LSTM model. For the other two models, there is a steady increase in the false positive rate, as the true positives increase.

We can also observe that the changes in performance between models more obviously affected the model runs using larger block sizes. The difference in performance for data using a basic block of size 5 was almost negligible.

The ROC curves generated for the test data are slightly worse than those generated for the training data. Without LeakyReLU, dropout and regularization, the variance between the two was even higher. While these configuration changes lessened variance, they did decrease the overall result. It is expected that using a larger dataset of PE files can resolve this issue in the future.

From the figures, we see that using basic blocks of size 29 yielded the best results, and as block size shortened, model performance deteriorated. This is true across all four model configurations, and while the relative accuracies changed slightly, the general trends remained the same. The best model configuration was the bidirectional LSTM with Attention, and this model also took the longest time to run. The bidirectional LSTM concatenates output from running the time steps in both forward and backward order. The attention mechanism, with an in-built temporal max-pooling layer, is a better aggregator than the pure temporal max-pooling layer that we layered onto the model externally. If the model were picking metadata or string features for each of the PE modules, the attention-based models and the temporal max pooling-based models would have yielded similar results. This would have meant that lower-level data was again being reduced down to noise. However, the fact that attention improved the model performance tells us that lower level features are being used, which shows an inherent pattern exists even within lower-level machine code. This may be due to the significant variance in PE files chosen to be part of the dataset—any similarity between basic block sequences within widely differing files is picked up, and especially so for the longer primary block sequences.

## 7. Conclusion

Overall, the findings of this paper are positive and reflect a general trend that by using larger sequences of basic blocks as inputs to the sequential models results in a more robust hidden representation. The addition of attention and bi-directionality to the core LSTM model significantly enhanced accuracy and results, even when using a relatively small dataset. The results achieved are in synchronization with results shown by previous work on similar topics, and while accu-

racies for this study could have been better. Overall, using neighboring basic blocks as data to a sequential predictive model such as LSTM works well for generating relevant context for program executions. Complicated pointer arithmetic and vague rule-based checks can be traded for efficient, external, probabilistic checks that supplemented static and dynamic analysis methods.

## 8. Future Work

In the future, we plan to utilize the same concept of neighboring basic blocks, but for longer sequences, *i.e.* using document classification techniques. We would use results from this study, such as hidden LSTM states and max-pooling embeddings, as features for this next step. Hopefully, we will be able to test the same concepts on a larger sample of files, both benign and malicious.

## Conflicts of Interest

The authors declare no conflicts of interest regarding the publication of this paper.

## References

[1] Baldwin, R. (2018) If You're Still Using a Fax Machine for "Security" Think Again. Engadget. http://www.engadget.com/2018/08/20/fax-machine-hack

[2] Stüttgen, J. (2015) Acquisition and Analysis of Compromised Firmware Using Memory Forensics. *Digital Investigation*, **12**, S50-S60. https://doi.org/10.1016/j.diin.2015.01.010

[3] Stüttgen, J. and Cohen, M. (2013) Anti-Forensic Resilient Memory Acquisition. *Digital Investigation*, **10**, 105-115. https://doi.org/10.1016/j.diin.2013.06.012

[4] Sukhbaatar, S., Weston, J. and Fergus, R. (2015) End-to-End Memory Networks. In: *Advances in Neural Information Processing Systems*, The MIT Press, Cambridge, 2431-2439.

[5] Yang, X., Lo, D., Xia, X., Zhang, Y. and Sun, J. (2015) Deep Learning for Just-in-Time Defect Prediction. *IEEE International Conference on Software Quality, Reliability and Security*, Vancouver, 3-5 August 2015, 17-26. https://doi.org/10.1109/QRS.2015.14

[6] Shin, E., Song, D. and Moazzezi, R. (2015) Recognizing Functions in Binaries with Neural Networks. *Usenix Conference on Security Symposium*, 611-626.

[7] Zhang, Z. (2001) HIDE: A Hierarchical Network Intrusion Detection System Using Statistical Preprocessing and Neural Network Classification. *Proc. IEEE Workshop on Information Assurance and Security*, West Point, 5-6 June 2001, 85-90.

[8] Krueger, D., Maharaj, T., Kramar, J., Pezeshki, M., Ballas, N., Ke, N.R., Goyal, A., Bengio, Y., Larochelle, H. and Courville, A. (2016) Zoneout: Regularizing RNNS by Randomly Preserving Hidden Activations.

[9] Sutskever, I., Martens, J., Dahl, G. and Hinton, G. (2013) On the Importance of Initialization and Momentum in Deep Learning. *International Conference on Machine Learning*, 1139-1147.

[10] "Invincea." Invincea: A Sophos Company. http://www.sophos.com/en-us/lp/invincea.aspx

[11] Raff, E., Zak, R., Cox, R., Sylvester, J., Yacci, P., Ward, R., Tracy, A., McLean, M.

and Nicholas, C. (2016) An Investigation of Byte n-Gram Features for Malware Classification. *Journal of Computer Virology and Hacking Techniques*, **14**, 1-20. https://doi.org/10.1007/s11416-016-0283-1

[12] Raff, E., Fleming, W., Zak, R., Anderson, H., Finlayson, B. and Nicholas, C. (2019) KiloGrams: Very Large N-Grams for Malware Classification.

[13] Qiao, R. and Sekar, R. (2017) Function Interface Analysis: A Principled Approach for Function Recognition in COTS Binaries. 47*th Annual IEEE/IFIP International Conference on Dependable Systems and Networks* (*DSN*), Denver, 26-29 June 2017. https://doi.org/10.1109/DSN.2017.29

[14] Hou, S., Saas, A., Chen, L., Ye, Y. and Bourlai, T. (2017) Deep Neural Networks for Automatic Android Malware Detection. *Proceedings of the* 2017 *IEEE/ACM International Conference on Advances in Social Networks Analysis and Mining* 2017 *ASONAM* 17, Sydney, 31 July-3 August 2017, 803-810. https://doi.org/10.1145/3110025.3116211

[15] Xiao, F., Lin, Z., Sun, Y. and Ma, Y. (2019) Malware Detection Based on Deep Learning of Behavior Graphs. *Mathematical Problems in Engineering*, **2019**, Article ID: 819539. https://doi.org/10.1155/2019/8195395

[16] Zhang, D.K., *et al.* (2018) MetaGraph2Vec: Complex Semantic Path Augmented Heterogeneous Network Embedding. In: *Advances in Knowledge Discovery and Data Mining*, Springer, Berlin, 196-208. https://doi.org/10.1007/978-3-319-93037-4_16

[17] Faraz, A., Haider, H., Shafiq, M.Z. and Farooq, M. (2009) Using Spatio-Temporal Information in API Calls with Machine Learning Algorithms for Malware Detection. *Proceedings of the* 2*nd ACM Workshop on Security and Artificial Intelligence*, Chicago, 9 November 2009, 55-62. https://doi.org/10.1145/1654988.1655003

[18] Pascanu, R., Stokes, J.W., Sanossian, H., Marinescu, M. and Thomas, A. (2015) Malware Classification with Recurrent Networks. *Proceedings of the IEEE International Conference on Acoustics*, *Speech and Signal Processing*, Brisbane, 19-24 April 2015, 1916-1920. https://doi.org/10.1109/ICASSP.2015.7178304

[19] Athiwaratkun, B. and Stokes, J.W. (2017) Malware Classification with LSTM and GRU Language Models and a Character-Level CNN. *IEEE International Conference on Acoustics*, *Speech and Signal Processing* (*ICASSP*), New Orleans, 5-9 March 2017. https://doi.org/10.1109/ICASSP.2017.7952603

[20] Zhang, J. and Chen, W. (2019) DeepCheck, a Non-Intrusive Control-Flow Integrity Checking Based on Deep Learning. https://arxiv.org/pdf/1905.01858.pdf

[21] Song, W., Yin, H., Liu, C. and Song, D. (2018) DeepMem: Learning Graph Neural Network Models for Fast and Robust Memory Forensic Analysis. *ACM SIGSAC Conference on Computer and Communications Security*, Toronto, January 2018, 606-618. https://doi.org/10.1145/3243734.3243813

[22] "Hex Rays." Hex Rays. http://www.hex-rays.com/products/ida

[23] Keras Development Team (2016) Keras: Deep Learning Library for Theano and Tensorflow. https://keras.io/

[24] Bahdanau, D., Cho, K.H. and Bengio, Y. (2014) Neural Machine Translation by Jointly Learning to Align and Translate. *ICLR* 2015. https://arxiv.org/pdf/1409.0473.pdf

[25] Hochreither, S. and Schmidhuber, J. (1997) Long Short-Term Memory. *Neural Computation*, **9**, 1735-1780. https://doi.org/10.1162/neco.1997.9.8.1735

[26] Xu, Y., Mou, L., Li, G., Chen, Y., Peng, H. and Jin, Z. (2016) Classifying Relations

via Long Short Term Memory Networks along Shortest Dependency Path. https://doi.org/10.18653/v1/D15-1206 https://arxiv.org/pdf/1508.03720

[27] Graves, A. (2013) Generating with Recurrent Neural Networks. https://arxiv.org/pdf/1308.0850.pdf