Neural Malware Control with Deep Reinforcement Learning

Yu Wang
Microsoft Corporation
Bellevue, Washington, 98004, USA
wany@microsoft.com

Jack W. Stokes
Microsoft Research
Redmond, Washington, 98052, USA
jstokes@microsoft.com

Mady Marinescu
Microsoft Corporation
Redmond, Washington, 98052, USA
mady@microsoft.com

***Abstract*—Antimalware products are a key component in detecting malware attacks, and their engines typically execute unknown programs in a sandbox prior to running them on the native operating system. Files cannot be scanned indefinitely so the engine employs heuristics to determine when to halt execution. Previous research has investigated analyzing the sequence of system calls generated during this emulation process to predict if an unknown file is malicious, but these models often require the emulation to be stopped after executing a fixed number of events from the beginning of the file. Also, these classifiers are not accurate enough to halt emulation in the middle of the file on their own. In this paper, we propose a novel algorithm which overcomes this limitation and learns the best time to halt the file's execution based on deep reinforcement learning (DRL). Because the new DRL-based system continues to emulate the unknown file until it can make a confident decision to stop, it prevents attackers from avoiding detection by initiating malicious activity after a fixed number of system calls. Results show that the proposed malware execution control model automatically halts emulation for 91.3% of the files earlier than heuristics employed by the engine. Furthermore, classifying the files at that time significantly improves the classifier's accuracy. This new model improves the true positive rate by 61.5%, at a false positive rate of 1%, compared to the best baseline classifier.**

***Index Terms*—Malware detection, Deep reinforcement learning**

I. Introduction

Malicious software, or *malware*, continues to be a serious threat to computer users. As a first line of defense, users and organizations often rely on commercial antimalware (*i.e.*, antivirus) products to detect malware being installed on their computers, and the antimalware *engine* is a key component of these malware detection systems. Prior to allowing an unknown file to be executed on the native operating system, the antimalware engine often tries to detect malware using two main approaches. First, static analysis employs malware "signatures" (*i.e.*, rules) to scan the unknown file to search for malicious byte sequences in the file without execution. Next, the engine utilizes one form of dynamic analysis called emulation to execute the file in a lightweight sandbox. Lightweight emulation does not analyze the unknown file in a full virtual machine (VM). Instead, the emulator mimics the response of a typical operating system. If the engine can detect malicious behavior during emulation, the antimalware system blocks the file from being executed on the native operating system and alerts the user that the file they are trying to install is malicious. As a result, the user's computer is not infected.

Previous dynamic analysis research has focused on analyzing the *sequence* of system application programming interface (API) calls made by the unknown file during emulation. Typically, the authors propose a recurrent, deep learning model to discriminate between the behavior of malicious and benign files. Pascanu et al. [1] used a recurrent neural network (RNN), or an echo state network (ESN), in combination with either a logistic regression classifier or a multi-layer perceptron (MLP) to detect malware. Athiwaratkun et al. [2] replaced the RNN with a long short-term memory (LSTM) recurrent network or a gated recurrent unit (GRU), and they also proposed a character-level convolutional neural network (CNN) to predict if an unknown file is malicious. A CNN followed by an LSTM is proposed for this task by Kolosnjaji, et al. [3]. In these solutions, the authors consider a fixed-length input buffer containing the events executed from the beginning of the file. The length of this pre-defined window, ν, varies depending on the study with $\nu \in \{50, 100, 200, 65000\}$. More recently, Agrawal, et al. [4] proposed a model using a combination of a CNN and an LSTM which processes the entire file in order to classify its contents.

In this paper, we propose a novel algorithm, based on deep reinforcement learning (DRL), to learn the best time to halt the engine's emulation to predict whether the unknown file is malicious or benign. DRL has been used previously to create adversarial samples to attack a malware classifier [5]. Initially, we tried to train a classifier to halt emulation based on the previous t_f (*e.g.*, 200) behavioral events, but the accuracy of this classifier was not sufficient to effectively halt the emulation. To the best of our knowledge, this is the *first* paper to propose using deep reinforcement learning to *protect* users from malware. This DRL-based neural network, combined with an event classifier and a file classifier, learns whether to halt emulation after enough state information has been observed or to continue emulation if more events are needed to make a highly confident prediction. Unlike previously proposed solutions, the DRL algorithm allows the engine to decide when to stop emulation on a *per file* basis.

Results from analyzing a collection of malware and benign files demonstrates a significant improvement in the early stopping of the execution of the file. The DRL-based system halts execution of 91.3% of the files earlier than heuristics used by the production antimalware engine. When the execution is stopped by the DRL model, the true positive detection rate exhibits a relative increase of 61.5% at a false positive rate of 1.0% compared to the best performing baseline model proposed in [2]. This paper makes the following contributions:

- We propose a deep reinforcement learning-based system which predicts when to stop emulating an unknown file.
- We implement the system and evaluate its performance on a collection of 75 thousand files.
- We demonstrate that the proposed system significantly outperforms several recent neural malware classification systems.

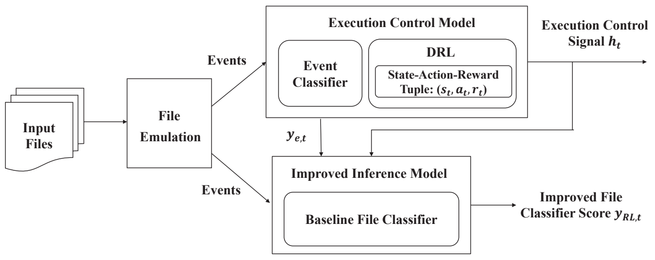


Fig. 1: Deep reinforcement learning system for halting the execution of an unknown file and improved malware classification.

II. Conventional Reinforcement Learning

Before considering the proposed neural malware control model, we first provide a brief overview of the standard definitions for conventional reinforcement learning (RL), as introduced by [6]. Conventional reinforcement learning is normally formulated as a stochastic Markov Decision Process (MDP). There are five main components in a standard reinforcement learning structure including agents, states, actions, rewards and policies. Each plays a different role in formulating the RL environment. A general interpretation is that reinforcement learning is a technique to help an agent learn what is the best action and policy to take such that its expected rewards/penalties can be maximized/minimized under a stochastic MDP environment [6].

We next provide definitions of the elements in conventional reinforcement learning.

1. **Agent and States (s_t).** An agent interacts with its environment by moving from the current state s_t at time t to another state s_{t+1} at time $t+1$. Each state is normally defined by the useful information from the interaction between an agent and its environment.

2. **Actions (a_t).** By taking an action a_t at state s_t, an agent can transfer from its current state to any of its connected neighbors at its next state s_{t+1} with different probabilities, since the agent can only arrive at one of its neighbors at $t+1$.

3. **Rewards (r_t).** The agent receives reward r_t at time t. The discounted reward R_t is defined as $R_t = \sum_{t=t_0}^{\infty} \gamma^{t-t_0} r_t$ where γ is the discount factor with $\{0 \le \gamma \le 1\}$, and t_0 indicates the starting time step. After reaching a state, an agent obtains the expected discounted reward ($\mathbb{E}[R_t|a_t, s_t]$) by considering the policies from the current state s_t to its neighbors s_{t+1} and so on. The expected discounted reward includes both the pre-defined reward at state s_t and the accumulated discounted rewards to be obtained in the future by taking a specific action a_t.

4. **Policy (π).** A policy π is a mapping from states to actions. There are three main types of reinforcement learning: value-based, policy gradient and actor-critic. In this study, we focus on a value-based algorithm called Q-learning given the small action space in our problem. The optimal action-value function in conventional Q-learning is defined as Q^*, which is the maximum expected reward obtained by selecting the best policy π at state s_t, $Q^*(s_t, a_t) = \max_\pi \mathbb{E}[R_t|a_t, s_t, \pi]$.

III. Neural Malware Control and Improved File Classification

In this paper, we propose a new deep reinforcement learning-based system to control the execution of an unknown file by an antimalware engine. The output of this DRL-based system can also be used to obtain an improved prediction of whether the file is malicious or benign. An overview of the proposed system is shown in Figure 1, and it has three main components: *file emulation*, the *execution control model* and the *improved inference model*.

The antimalware engine includes a sandbox which aims to instrument the runtime behavior of an unknown, executable file before it is allowed to run on the native operating system. The unknown file is first **emulated** by the sandbox, and this generates a sequence of behavioral file events, $e_t \in E$. These events typically include operations related to processes, files, networking, and registry behavior. Most of the events are associated with APIs invoked during execution, but other behavioral events, such as unexpected instructions or constructs, are also captured.

The **execution control model** processes E and is responsible for controlling the file's execution. If the execution control model can make a confident decision that the file is either malicious or benign, the execution is halted. As it is received, each individual event e_t is first processed by an *event classifier* which makes a prediction, $y_{e,t}$, indicating whether the most recent event history includes malicious activity or not. Initially, we tried halting the emulation based solely on the *event classifier's* output, but the classifier's accuracy was not sufficient to accomplish this task and motivated the need for *deep reinforcement learning*. Even though it is a weak signal, $y_{e,t}$ is used to construct a reward signal for the DRL model, which then produces the execution control signal, h_t, indicating if the file execution should be halted or allowed to continue.

The primary purpose of the DRL model is to better control the file's execution. However, we also found that it can be used to significantly improve the overall classification of an unknown file. This improved prediction, $y_{RL,t}$, which indicates whether the file is malicious or benign, is generated by the **improved inference model**. The improved inference model boosts the weak predictions from the *event classifier*, $y_{e,t}$, based on h_t and the output of a *baseline file classifier* which offers an initial estimate of the probability, y_f, that the file is malicious based on the initial t_f (*e.g.*, 200) events generated by the file. We next provide details on each of these three main system blocks.

IV. File Emulation and Events

As described previously, the file events are generated by emulation in the antimalware engine. Attackers often use polymorphic tactics to avoid detection. In polymorphism, they rearrange or rewrite their code in different ways which appear to be different but accomplish the same task. To deal with polymorphism, our antimalware engine maps multiple low-level API calls into a single high-level event. For example, the attacker may use a user mode API (CreateFile), a kernel mode API (ZwCreateFile), or the C++ API (ofstream::open) to create a file. All of these events are mapped into the same high-level FileCreate event. In our data, the antimalware engine produced 114 high-level behavior events representing many more individual low-level API calls. An example of the first five events includes CreateFile, VirtualAlloc, VirtualAlloc, GetModuleHandle, and GetModuleFilename.

In general, the malware tends to contain more events than the benign files. For this data, more than 50% of malware files in the test set contain more than 1000 events, but less than 20% of the benign files are longer than 1000 events. Furthermore, malware tends to contain long loops after executing the first several events. Benign files, on the otherhand, tend to exhibit more random behavior. While they do contain loops, these files tend to be shorter and less repetitive. Ideally, our execution control model can leverage these patterns to decide when to stop the emulator and perform the evaluation.

V. Deep Execution Control

The details of the execution control model in Figure 1 are depicted in Figure 2. The input is e_t, and the outputs are h_t and $y_{e,t}$ in both figures. As each event is received, it is inserted into the *event queue*, a first in, first out (FIFO) queue. The event classifier then predicts $y_{e,t}$ for the most recent subsequence stored in the event queue. Since the output layer of the event classifier is a sigmoid function, $y_{e,t}$ is the probability that the most recent subsequence of behavioral events corresponds to malicious activity.

The DRL model depends upon its states, actions, and rewards. The state, s_t, includes information related to all the events received up to and including the most recent event. For each e_t, the event classifier's prediction $y_{e,t}$ is used as part of the DRL model's reward function, r_t. The actions for the DRL model include continuing and halting file execution. Based in part on $y_{e,t}$ and s_t, the DRL model generates separate Q values which are the estimated expected discounted rewards associated with these actions. The Q value signals are noisy

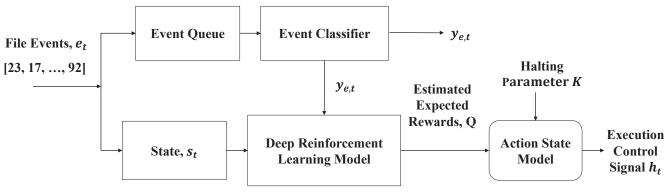


Fig. 2: Details of the execution control model in Figure 1. In both figures, the input is e_t, and the outputs are h_t and $y_{e,t}$.

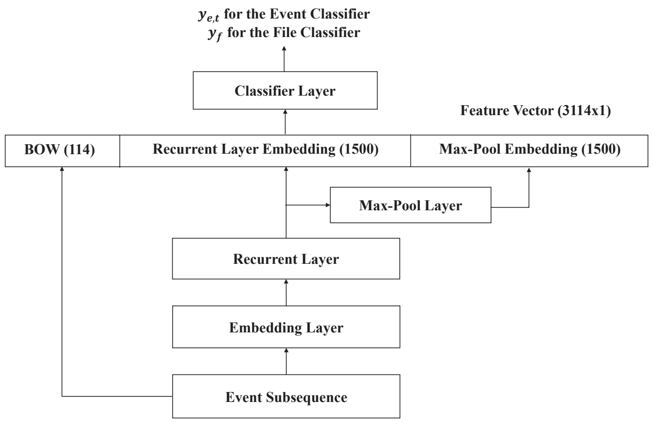
and cannot be used directly. The *action state model* filters the Q values to generate the halting signal h_t, which is then used by the antimalware engine to stop the file's execution.

Event Classification. The recurrent model structure which is used for the event classifier, and later for the baseline file classifier, is shown in Figure 3. For each new event, the event classifier makes a prediction, $y_{e,t}$, that the behavior associated with the most recent t_f behavioral events is malicious. This event subsequence is stored in the event queue.

Each event in the event queue is input to an embedding layer, and the result is then input to a recurrent layer. We use a recurrent neural network (RNN) for the recurrent layer. As proposed in [1], the recurrent layer's hidden state is input to a max-pool layer which is able to better detect malicious activity within the subsequence. We next construct a sparse binary, feature vector consisting of a bag of words (BOW) representation of the event subsequence (114), the final hidden state of the recurrent layer which is the recurrent layer's embedding (1500), and the output of the max-pool layer which is the max-pool embedding (1500). This feature vector is then input to a shallow neural network in the classifier layer. The output layer of the neural network is a sigmoid function. Thus, $y_{e,t}$ is the probability that this most recent event history contains malicious activity.

Deep Reinforcement Learning. We next present a deep reinforcement learning model to control the antimalware engine's execution of an unknown file. The first task is to choose the type of reinforcement learning model for our problem. The main feature in our problem is a very large state space together with a small action space consisting of two actions $A \in \{continue, halt\}$. Considering the small action space, we prefer a value-based reinforcement learning technique which compares the value functions of the actions directly, instead of learning another policy estimator to find the best policy as in policy gradient [7], [8] or actor-critic-based approaches [9]–[11].

The action-value function, $Q^*(s_t, a_t) = \max_\pi \mathbb{E}[R_t|a_t, s_t, \pi]$, is the expected reward of taking action a_t at state s_t following the policy π. To calculate this value using a conventional value-based approach, it is necessary to store all the Q values in a table for all the state-action pairs encountered during training, which is not feasible if the state space or action space is large.



$y_{e,t}$ for the Event Classifier
y_f for the File Classifier
Classifier Layer
Feature Vector (3114x1)
BOW (114)
Recurrent Layer Embedding (1500)
Max-Pool Embedding (1500)
Max-Pool Layer
Recurrent Layer
Embedding Layer
Event Subsequence

Fig. 3: Structure of the event classifier and the baseline file classifier.

One approach to overcome this difficulty is by training a nonlinear approximator $Q(s_t, a_t|\theta_t)$, such as a deep neural network, to estimate $Q^*(s_t, a_t)$ at time step t. However, these types of nonlinear estimators tend to be unstable in practical applications since convergence is not guaranteed. To address this issue, [12], [13] recently proposed using a replay buffer in the deep Q network (DQN) which demonstrates better convergence properties. Since our problem has a very large state space, we also use a DQN as our DRL model structure in this paper.

In a DQN-based DRL structure, the deep neural network action-value function estimator $Q(s_t, a_t|\theta_t)$ is normally defined at state s_t as $Q(s_t, a_t|\theta_t) \sim Q^*(s_t, a_t)$ by taking the state s_t as the input of the neural network, where θ_t represents the neural network parameters. We next describe the state design, action design, reward design, and training for the DRL model using experience replay.

Design of States. The DRL model's state contains three parts: the position (*i.e.*, index) of the current event in the file ρ_t, the current event ID, and the histogram of all the previous events. The current event position in the file will be used later to define the reward r_t for the deep reinforcement learning model. We initially tried to use a one-hot encoding of the event ID, but found that using the event ID directly provides slightly better performance while reducing the size of the state. The event ID histogram captures the history of the events which have been observed so far.

Design of Actions. The agent (*i.e.*, antimalware engine) can perform two types of actions a_t: **continue**, which is labeled as C, and **halt**, labeled as H. The action indicates whether the agent should continue or halt the execution of the file. In our deep reinforcement learning model, the selected action a_t for state s_t is inferred from the output of the neural network. As shown in the Figure 2, the outputs of the deep neural network are the estimated action value function $Q_C = Q^*(s_t, a_t = C) = \max_\pi \mathbb{E}[R_t|a_t = C, s_t, \pi]$ for action C at state s_t and $Q_H = Q^*(s_t, a_t = H) = \max_\pi \mathbb{E}[R_t|a_t = H, s_t, \pi]$ for action H at state s_t. By comparing the two $Q^*(s_t, a_t)$ values for actions H and C, the action with the larger Q value is selected and performed.

Design of Rewards. The reward r_t at each state s_t is designed based on two criteria:
1. We prefer for the DRL network to learn to halt emulation as quickly as possible. Therefore, shorter emulation sequence lengths are assigned a higher reward and longer sequence lengths are given a smaller reward.
2. The closer an event prediction is to the true label of the file, the larger the reward should be given at that state.

Based on the above two criteria, the reward is defined as

$$r_t = (0.5 - |y_{e,t} - L|)e^{-\beta\rho_t} \tag{1}$$

where $y_{e,t}$ is the event-based prediction generated by the most recent $t_f = 200$ events, and $L \in \{0, 1\}$ is defined as the true label of the training file. The decay factor β is chosen experimentally, and ρ_t is the position of current event in the file.

DRL Training. To train this neural network-based estimator, we use an l_2 loss function defined as $\mathcal{L}(\theta_t) = \mathbb{E}_{s_t}[(\hat{Q}(s_t, a_t|\theta_t) - Q(s_t, a_t|\theta_t))^2]$ where $\hat{Q}(s_t, a_t|\theta_t)$ is an estimate of $Q(s_t, a_t|\theta_t)$. $\hat{Q}(s_t, a_t|\theta_t)$ is computed using the current state reward r_t together with its neighbors' estimations from the neural network in an iterative manner, *i.e.*, $\hat{Q}(s_t, a_t|\theta_t) = r_t + \gamma \max_{a_{t+1}} Q(s_{t+1}, a_{t+1}|\theta_t)$ where s_{t+1} are the neighbors of s_t, and a_{t+1} are the corresponding actions generated by the neural network.

In [12], experience replay is used to train the DRL model. Experience replay helps to alleviate the potential issues of non-stationary distributions and correlated data and is performed by randomly sampling the state pairs. Whenever one stochastic step is taken by the agent, the current state s_t, obtained reward r_t, action taken a_t and next state s_{t+1} are combined as one agent experience set (s_t,r_t,a_t,s_{t+1}), and pushed into the replay memory queue M. Throughout the learning process, the reinforcement learning updates are performed in minibatches of size B_{RL}, drawing from the replay memory randomly. The algorithm for training the DRL model using experience replay is provided in Algorithm 1.

We tested several different stochastic gradient descent optimization methods for training the DRL model and found that adadelta [14] performed best. Furthermore, the convergence of DRL is not always guaranteed. To help with the convergence, the sum of $r_t + \gamma$ should be within the range of [0,1]. It is important to note that we first train the *event classifier*, as well as the *baseline file classifier* in the *improved inference model*, in isolation prior to training the DRL model – the system is not trained in an end-to-end fashion. Thus, $y_{e,t}$ in the reward r_t is generated by the pre-trained event classifier, and the reward function has the same value for the same event sequence. Otherwise, the DRL's reward function can become non-stationary.

Algorithm 1 Deep Reinforcement Learning Training

```
1: Epochs: $N \leftarrow 2000$
2: Minibatch Size: $B_{RL} \leftarrow 50$
3: Decay Factor: $\beta \leftarrow 0.01$
4: Initialize a replay memory $M$ with size $\mu \leftarrow 50000$, DRL
   model with 3 layers
5: for n=1 $\rightarrow$ N do
6:     Time step in state space: $t \leftarrow 0$
7:     Randomly select an initial state $s_t$
8:     while !End of File do
9:         $Q(s_t, a_t|\theta_t) \leftarrow$ DRL($s_t$)
10:        $a_t^*$=argmax$_{a_t}$ $Q(s_t, a_t|\theta_t)$
11:        Perform action $a_t^*$, generating next state $s_{t+1}$
12:        Push tuple $(s_t, r_t, a_t^*, s_{t+1})$ into replay memory $M$
13:        for b=1 $\rightarrow$ $B_{RL}$ do
14:            Randomly select a tuple $m$ from $M$
15:            $s_t \leftarrow m(0)$, $r_t \leftarrow m(1)$, $s_{t+1} \leftarrow m(3)$
16:            $Q(s_t, a_t|\theta_t) \leftarrow$ DRL($s_t$)
17:            $Q(s_{t+1}, a_{t+1}|\theta_t) \leftarrow$ DRL($s_{t+1}$)
18:            Input $y_{e,t}$ from Event Classifier
19:            $r_t \leftarrow (0.5 - |y_{e,t} - L|) \times e^{-\beta \rho_t}$
20:            Update $\hat{Q}(s_t, a_t|\theta_t)$
21:            Update the network by minimizing loss $\mathcal{L}(\theta_t)$
22:        end for
23:        $t \leftarrow t + 1$
24:    end while
25: end for
```

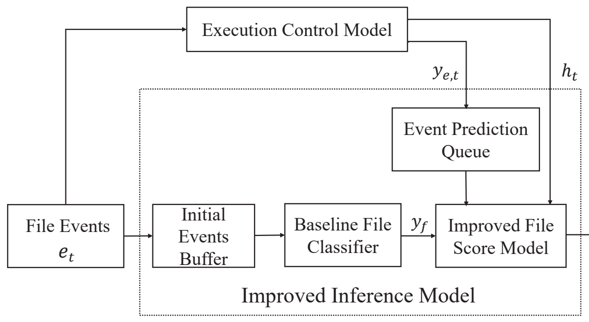


Fig. 4: Details of the *improved file inference model* in Figure 1. The inputs are e_t, $y_{e,t}$, and h_t, and the output is $y_{RL,t}$ in both figures.

Action State Model. The final block in Figure 2 is the action state model which generates the halting signal h_t. The Q value signal is noisy and cannot be used directly to compute h_t. The halting signal is a binary signal which is initialized to have a value of 0 and remains 0 for each e_t until $Q_H > Q_C$ for K consecutive events. At that point, the value of h_t transitions to 1 and continues to maintain that value if any additional events are processed.

VI. Improved Inference

The purpose of the *improved inference model* in Figure 1 is used to generate a better DRL-based prediction, $y_{RL,t}$, of whether or not an unknown file is malicious. The details of *improved inference model* are depicted in Figure 4. This model has three inputs, namely the events generated by the file (e_t), the most recent event predictions ($y_{e,t}$), and the execution control signal (h_t). The most recent K values of $y_{e,t}$ are stored in the *event prediction queue*, which is another FIFO queue. After h_t signals that the file execution has been halted, the *improved file score model* evaluates the event predictions in the queue to generate $y_{RL,t}$.

The individual $y_{e,t}$ values, stored in the *event prediction queue*, are noisy and can be difficult to analyze. In some cases, only setting $y_{RL,t}$ to be the most recent $y_{e,t}$ value can lead to an incorrect prediction that the file is malicious. To overcome this issue, the *improved inference model* also employs an additional *baseline file classifier* to improve the accuracy of $y_{RL,t}$. To accomplish this task, the initial t_f (*e.g.*, 200) events generated by the file are stored in the *initial events buffer*, and these are processed by the *baseline file classifier* to produce an initial prediction that the file is malicious, y_f.

Baseline File Classifier. The baseline file classifier also utilizes the structure shown in Figure 3 and follows [2]. The input *event subsequence* in Figure 3 corresponds to the first t_f events for each file stored in the *initial events buffer* (Figure 4). Here, t_f is the same value which denotes the length of the event classifier's input *event queue*. An LSTM is used for the recurrent layer, and the classifier layer uses logistic regression for the file's prediction y_f. Similar to the event classifier, y_f is the initial estimate of the probability that the file is malicious based on the initial behavior of the file.

Improved File Score. We can now combine y_f with the $y_{e,t}$ history stored in the *event prediction queue* to compute the final improved file classifier score, $y_{RL,t}$. Since the initial estimates of the $y_{e,t}$ are noisy, we process the most recent K event predictions from the event prediction queue. Formally if h_t is equal to 1,

$$\begin{aligned} y_{RL,t} &= \max\{y_{e,t-K+1}, \cdots, y_{e,t}\} \quad \text{if } y_f > 0.5 \\ y_{RL,t} &= \min\{y_{e,t-K+1}, \cdots, y_{e,t}\} \quad \text{if } y_f \leq 0.5 \end{aligned} \tag{2}$$

where $y_{e,i}$ is the event classifier's prediction at step i, and $y_{RL,t}$ is the improved inference model's output, *i.e.*, the prediction probability that the unknown file is malicious.

VII. Data

The original data for our research was collected by scanning a large collection of Windows Portable Executable (PE) files with the production Microsoft antimalware engine. The behavioral events are logged using a special version of the antimalware engine. As noted earlier, the engine records 114 event types, $e_t \in \{0, \cdots, 113\}$, ranging from file IO, registry APIs, networking APIs, thread or process creation and control, inter-process communication, timing, and debugging APIs.

The labels utilized in this study correspond to production labels used by our antimalware product partners to train malware classifiers that identify malware targeting Windows computers. The files are labeled with $L \in \{0, 1\}$ where 1 corresponds to a malware file and 0 indicates that the file is benign. Benign files are identified as those which are known to be safe. For example, these might be files belonging to software products which are purchased by users (*e.g.* Microsoft Office) or downloaded from the internet from sites which are known to be legitimate (*e.g.* Adobe Acrobat Reader, Google Chrome). Other benign files are determined to be safe by professional analysts. Labels for malware files are also generated by manual inspection by professional analysts. In addition, all unknown files received by the company are scanned by over 20 additional production antimalware engines. If eight or more of these anti-malware engines detect that a file is malicious, these files will also be determined to be malicious.

We collected a dataset of 75 thousand files which had been evaluated by the antimalware engine. These files were equally split between the malware and benign classes. First, we discarded any files whose event sequences were shared between these two classes and which contained less than 50 events. Furthermore, we ensured that all the event sequences in the datasets were distinct. This requirement ensured that we did not overfit to one particular set of events. We then split the overall dataset into separate training, validation, and test sets with 50, 10, and 15 thousand files, respectively. Again, we maintained an equal split between the two classes for each of the individual datasets. The event and file classifiers were trained with the training and validation sets, while the DRL-model was trained with 2000 files from the training set. The results presented below are based on evaluating the model on the hold out test set.

VIII. Experimental Results

We next present the results for the proposed neural malware control model. We first describe the experimental setup. We then evaluate how quickly the DRL-based model halts the execution of a file. Finally, we compare the final prediction that the file is malicious or benign to the results from several baseline file classifiers.

Experimental Setup. For reproducibility, we provide the setup that was used to train the DRL-based neural malware control system and the baseline file classifiers described below. We implemented the proposed neural malware control model using Keras [15] with Theano [16] as the backend deep learning framework. Several hyperparameters were tuned on smaller datasets. The decay factor in the DRL model's reward function is $\beta = 0.01$. The DRL model uses 3 hidden layers with a size of $h_{hidden} = 128$ and is trained with a minibatch size $B_{RL} = 50$. The replay memory is initialized with a size $\mu = 50000$. The experiments were performed on an Intel Xeon CPU E5-1620, 3.50 GHZ with 16 GB of RAM. The GPU is an NVIDIA GeForce GTX 980 Ti.

How does the DRL-based model's stopping performance compare to the antimalware engine's heuristics? The files in our dataset were collected by a production antimalware engine, and the number of events recorded for each file represents the performance of the heuristics employed by the engine to halt emulation. Thus, by measuring how often the DRL-based model halts execution prior to reaching the end of the file, we can compare the performance between our model and the engine's heuristics. In cases where the DRL model reaches the end of the file without halting execution, we can infer that the proposed model was not confident enough to make a decision, and the DRL based-model would have continued to execute the file.

The results of this evaluation are presented in Table I and depend on two values: the number of training files (*i.e.*, epochs) N and the number of consecutive events where $Q_H > Q_C$ denoted by K. The fraction of files where the DRL-model halts execution before the end of the file, α, is computed as:

$$\alpha = \frac{\text{(Total number of early halted files)}}{\text{(Total number of files)}}. \qquad (3)$$

We make two observations from the results presented in the table. First, the percentage of files whose execution is halted by the DRL model earlier than engine's heuristics continues to increase as the number of training file N increases. Better training allows the engine to halt execution earlier. Second, the percentage of files which are halted early decreases with K. The value of K is a proxy for the DRL model's confidence in the decision to halt the file's execution. It is not surprising that the execution of fewer files is halted early as we require more confidence (*i.e.*, higher value of K) in the decision. Even so, the results show that over 91% of the files in the test set are halted early compared to the engine's heuristics after training with only 2000 files. This indicates that the engine's heuristics may be overly cautious when emulating a file. In addition, requiring less time for scanning an individual file leads to better performance when scanning all the files on the hard drive.

	K=10	K=15	K=20
N=30	71.5%	64.1%	58.3%
N=200	82.9%	75.2%	69.2%
N=2000	98.2%	95.1%	91.3%

TABLE I: The fraction of files, α, in % where emulation is halted earlier by the proposed deep reinforcement learning model compared to heuristics used by the antimalware engine.

In Table I, we report that with $K = 20$ and $N = 2000$, the execution control model halts the execution of 91.3% of the files earlier than the heuristics used by the antimalware engine. In Figure 5, we provide the histogram indicating the distribution of the percentage of events which were executed before the execution control model halted the file's emulation compared to the number of the events in the file which was determined by the engine's heuristics. This figure indicates that a file's emulation is halted much earlier than the heuristics for the majority of the files.

Can DRL improve file classification? While the results in Table I indicate that emulation of the majority of the files

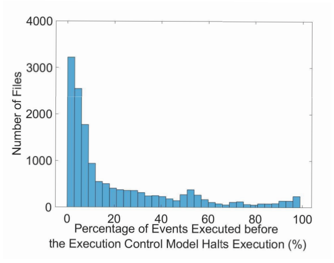


Fig. 5: Histogram of the percentage of behavioral events which are executed before the execution control model halts emulation compared to the number of executed events determined by the engine's heuristics.

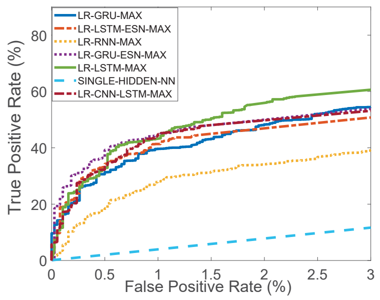


Fig. 6: Performance of several recently proposed recurrent models for the baseline file classifier.

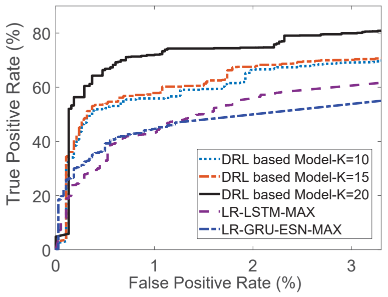


Fig. 7: Comparison between the proposed DRL-based model for $K \in \{10, 15, 20\}$ and the best two baseline file classifiers.

can be stopped earlier than the heuristics employed in the engine, it is important to understand how early halting affects the detection performance. To measure this, we first compute the receiver operating characteristic (ROC) curves for a range of models in Figure 6 for the baseline file classifier depicted in Figure 3. This analysis allows us to evaluate the best baseline file classifiers for our data based on recently published research. The models include LSTM [2], RNN [1], gated recurrent unit (GRU) [2], convolutional neural network (CNN) [3] and a simple, single hidden layer, feedforward neural network. We also include the echo state network (ESN) counterparts for the LSTM and GRU. All models use logistic regression (LR) for the classifier, because these slightly outperformed a shallow neural network for this dataset. All models use max-pooling as shown in Figure 3. These results indicate that the performance of none of the models we investigated dominated all of the other models. In particular, the ESN version of the GRU offers better performance and low false positive rates (FPRs) while the LSTM outperforms all other models above an FPR $\geq 1.2\%$.

We next compare the two best performing baseline file classifiers to the proposed DRL-based models for $K \in \{10, 15, 20\}$ in Figure 7. The figure clearly indicates that all the DRL-based models offer significantly better performance compared to the baseline file classifiers. In particular, the DRL-based model with $K = 20$ offers a relative improvement of 61.5% for the true positive rate (TPR) at an FPR of 1% compared to the GRU-ESN-based baseline file classifier. The relative improvement of the TPR is 65.7% at an FPR of 1% for the LSTM.

IX. Related Work

Deep Reinforcement Learning. Conventional reinforcement learning has been widely studied in the fields of machine learning and system control for over three decades [6], [17]–[20]. Recently, Mnih, et al. [12], [13] successfully applied deep neural network-based Q-learning (DQN) to playing a series of Atari games, by using a replay buffer to improve the system's convergence. Also, Silver, et al. [21], [22] developed novel algorithms, by applying reinforcement learning to Monte Carlo tree search, to play the Go game and beat human Go masters. Progress has also been made in improving value-based [23], [24], policy gradient [25]–[28], and actor-critic [11], [29] deep reinforcement learning algorithms, in order to find better policies more efficiently and to deal with a continuous action space.

More recently, there are many deep neural network-based reinforcement learning approaches proposed either using value-based, policy-based or actor-critic-based structures as in [21], [30]–[32].

Deep Learning for Malware Classification. A number of authors have proposed using DNNs for malware classification tasks. Dahl, et al. [33] first investigated the use of a DNN for malware classification for dynamic analysis. Huang and Stokes [34] proposed a DNN with multitask learning for dynamic analysis where the first task was binary (*i.e.*, malware versus benign) and the second task was malware family classification. A separate line of research has investigated using recurrent models for malware classification. Pascanu, et al. [1] first proposed using recurrent neural networks and echo state networks for classifying malware sequences. Athiwaratkun and Stokes [2] instead proposed an LSTM, a GRU, and a character-level CNN for the sequence classification. Kolosnjaji, et al. [3] used a CNN followed by an LSTM for this task.

X. Conclusion

We present a novel, neural malware control model which learns when to halt the execution of an unknown file based on deep reinforcement learning. This model is the first to use deep reinforcement learning to protect customers from malware. Fast scanning is an important feature for users when installing software, reading emails with attachments, and searching a hard drive for files which were maliciously dropped during a drive-by download. Our results indicate that the proposed model halts execution earlier than a production antimalware engine for more than 91% of the files in the test set. More importantly, we show a relative improvement of over 61% in the true positive detection rate of malware at a false positive rate of 1% compared to a number of baseline malware classifiers reported in the literature. Our model allows the detection to be delayed until later in the processing of the file compared to these existing solutions. Thus, the proposed model offers significantly better protection with less delay.

References

[1] R. Pascanu, J. W. Stokes, H. Sanossian, M. Marinescu, and A. 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.

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

[3] 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.

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

[5] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth, "Learning to evade static pe machine learning malware models via reinforcement learning," *arXiv preprint arXiv:1801.08917*, 2018.

[6] R. S. Sutton and A. G. Barto, *Introduction to reinforcement learning*. MIT Press Cambridge, 1998, vol. 135.

[7] D. Silver, G. Lever, N. Heess, T. Degris, D. Wierstra, and M. Riedmiller, "Deterministic policy gradient algorithms," in *International Conference on Machine Learning (ICML)*, 2014, pp. 387–395.

[8] S. Levine, C. Finn, T. Darrell, and P. Abbeel, "End-to-end training of deep visuomotor policies," *The Journal of Machine Learning Research (JMLR)*, vol. 17, no. 1, pp. 1334–1373, 2016.

[9] P. Wawrzyński, "Real-time reinforcement learning by sequential actor–critics and experience replay," *Neural Networks*, vol. 22, no. 10, pp. 1484–1497, 2009.

[10] P. Wawrzyński and A. K. Tanwani, "Autonomous reinforcement learning with experience replay," *Neural Networks*, vol. 41, pp. 156–167, 2013.

[11] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," *arXiv preprint arXiv:1509.02971*, 2015.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[14] M. D. Zeiler, "Adadelta: an adaptive learning rate method," *arXiv preprint arXiv:1212.5701*, 2012.

[15] F. Chollet *et al.*, "Keras," https://github.com/keras-team/keras, 2015.

[16] R. Al-Rfou, G. Alain, A. Almahairi, C. Angermueller, D. Bahdanau, N. Ballas, F. Bastien, J. Bayer, A. Belikov, A. Belopolsky *et al.*, "Theano: A python framework for fast computation of mathematical expressions," *arXiv preprint arXiv:1605.02688*, vol. 472, p. 473, 2016.

[17] R. S. Sutton, "Temporal credit assignment in reinforcement learning," *PhD thesis, University of Massachusetts*, 1984.

[18] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Machine learning*, vol. 8, no. 3-4, pp. 229–256, 1992.

[19] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *International Conference on Machine Learning (ICML)*, vol. 157, 1994, pp. 157–163.

[20] L. P. Kaelbling, M. L. Littman, and A. W. Moore, "Reinforcement learning: A survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[21] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *Nature*, vol. 529, no. 7587, pp. 484–489, 2016.

[22] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton *et al.*, "Mastering the game of go without human knowledge," *Nature*, vol. 550, no. 7676, p. 354, 2017.

[23] H. Van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning." in *AAAI*, vol. 16, 2016, pp. 2094–2100.

[24] Z. Wang, T. Schaul, M. Hessel, H. Hasselt, M. Lanctot, and N. Freitas, "Dueling network architectures for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1995–2003.

[25] J. Schulman, S. Levine, P. Abbeel, M. Jordan, and P. Moritz, "Trust region policy optimization," in *International Conference on Machine Learning (ICML)*, 2015, pp. 1889–1897.

[26] Y. Duan, X. Chen, R. Houthooft, J. Schulman, and P. Abbeel, "Benchmarking deep reinforcement learning for continuous control," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1329–1338.

[27] S. Gu, T. Lillicrap, Z. Ghahramani, R. E. Turner, and S. Levine, "Q-prop: Sample-efficient policy gradient with an off-policy critic," in *5th International Conference on Learning Representations (ICLR)*, 2017.

[28] B. O'Donoghue, R. Munos, K. Kavukcuoglu, and V. Mnih, "Pgq: Combining policy gradient and q-learning," in *5th International Conference on Learning Representations (ICLR)*, 2017.

[29] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *International Conference on Machine Learning (ICML)*, 2016, pp. 1928–1937.

[30] M. Hausknecht and P. Stone, "Deep reinforcement learning in parameterized action space," *arXiv preprint arXiv:1511.04143*, 2015.

[31] N. Heess, G. Wayne, D. Silver, T. Lillicrap, T. Erez, and Y. Tassa, "Learning continuous control policies by stochastic value gradients," in *Advances in Neural Information Processing Systems (NIPS)*, 2015, pp. 2944–2952.

[32] J. Schulman, P. Moritz, S. Levine, M. Jordan, and P. Abbeel, "High-dimensional continuous control using generalized advantage estimation," *arXiv preprint arXiv:1506.02438*, 2015.

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

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