IEEE Access
Multidisciplinary ⁝ Rapid Review ⁝ Open Access Journal

Received February 27, 2019, accepted March 19, 2019, date of publication March 28, 2019, date of current version April 23, 2019.

Digital Object Identifier 10.1109/ACCESS.2019.2908033

Evading Anti-Malware Engines With Deep Reinforcement Learning

ZHIYANG FANG, JUNFENG WANG, BOYA LI, SIQI WU, YINGJIE ZHOU, AND HAIYING HUANG
College of Computer Science, Sichuan University, Chengdu 610065, China

Corresponding author: Junfeng Wang (wangjf@scu.edu.cn)

This work was supported in part by the National Key Research and Development Program under Grant 2018YFB0804503 and Grant 2016YFB0800605, and in part by the National Natural Science Foundation of China under Grant U1836103, Grant 91338107, and Grant 61801315).

ABSTRACT To reduce the risks of malicious software, malware detection methods using machine learning have received tremendous attention in recent years. Most of the conventional methods are based on supervised learning, which relies on static features with definite labels. However, recent studies have shown the models based on supervised learning are vulnerable to deliberate attacks. This work tends to expose and demonstrate the weakness in these models. A DQEAF framework using reinforcement learning to evade anti-malware engines is presented. DQEAF trains an AI agent through a neural network by constantly interacting with malware samples. Actions are a set of reasonable modifications, which do not damage samples' structure and functions. The agent selects the optimal sequence of actions to modify the malware samples, thus they can bypass the detection engines. The training process depends on the characteristics of the raw binary stream features of samples. The experiments show that the proposed method has a success rate of 75%. The efficacy of the proposed DQEAF has also been evaluated by other families of malicious software, which shows good robustness.

INDEX TERMS Anti-malware engines evasion, deep machine learning, malware detection, reinforcement learning.

I. INTRODUCTION

Due to the massive exchange of messages on the network, variety of malware and their variants appear explosively [1]. New malicious software have the characteristics of rapid growth, powerful survivability, and strong anti reconnaissance [2], which make many security-sensitive applications more vulnerable to be attacked. To tackle this problem, various malware detection methods have been developed, which depend on signature [39], the sequence of program API calls [14], supervised machine learning [4], and so on.

Owing to the hardware advancement [3], machine learning plays a critical role in extracting meaningful information out of the huge amount of data and therefore vast of detection methods depend on that to discover unknown malware families and variations [7]. Most state-of-the-art classifiers used today are based on supervised learning, which are trained on dataset with labels to classify the given software.

The associate editor coordinating the review of this manuscript and approving it for publication was Ilsun You.

However, the robustness of these methods is not certain [28], [29], especially methods using supervised learning, which tends to extract static features and statistical characteristics, instead of doing dynamic or in-depth analysis. That may make supervised learning vulnerable to be attacked [20] or evaded.

As many malware detection methods rely on machine learning for automated decisions, several attacks have emerged on using potential vulnerabilities introduced by supervised learning algorithms to evade detections [8]. Sophisticated attackers have strong incentives to manipulate the results and models generated by supervised learning algorithms to achieve their objectives [19]. For instance, attackers can deliberately influence the training dataset to manipulate the results of a predictive model [9] (in poisoning attacks [10]–[12]), causing misclassification of new data in the testing phase [13] (in evasion attacks [21]–[23]). Potential adversaries can also modify the samples, while preserving malicious behaviors, to evade the detection. For example, genetic programming can be used to make evasive variants [31] and reinforcement learning can offer a sequence of actions to modify the samples to obfuscate detection [7]. These circumvention technology will cause serious consequences, including failure of the detection tools and risks to security of system and network.

The aforementioned references illustrate that classifiers using supervised learning may fail under certain scenarios. Thus, the malware detection based on supervised learning could lose defense when encountering complicated and powerful attackers. Our work aims to expose the weaknesses in their architectures by modifying the malicious samples and making them evade the detection without knowledge of the classifier. The result can be used to improve the efficacy of classifiers. Besides, due to the variety of formats and huge number of malware, our work focuses on PE file to prove the lack of robustness in the detection method using supervised learning.

A general framework using deep Q-network to evade anti-malware engines (DQEAF) is proposed in this work. A specific attacking model is established, and an AI agent is set up to confront the malware detector. The *agent* firstly inspects malware samples, and then selects a sequence of functionality-preserving *actions* deliberately to modify the samples. For any given malware sample, the framework can eventually determine the optimal sequence of *actions* to evade the supervised-learning-based detector. The newly modified sample may not be easily detected by the anti-malware engine. In the study, the static Windows portable executable (PE) malicious softwares are used as the training samples to validate the proposed DQEAF. Results show that 75% success rate of evasion on samples can be achieved. For extraction of features from the PE malicious files, raw binary byte stream is used. As the features are irrelevant to file structure, the proposed method can be extended to other platforms such as ELF, Android, etc.

The proposed DQEAF has the following characteristics:

- Fewer functionality-preserving actions (only 4 actions);
- Low-dimensional raw binary stream features to observe samples (only 513 dimensions);
- High evasion success rate;

The rest of this paper is organized as follows: Section II discusses the related work. Section III describes the proposed DQEAF in details. Section IV presents the experimental set-up and numerical simulation results. Section V concludes the paper.

II. RELATED WORK

There are several attempts to bypass or directly attack the machine-learning-based malware classifiers to achieve the purpose of confrontation. At present, machine learning technology is widely used in human-machine interaction [40], recommender systems [15], security protection [16] and other fields [17], [18]. However, models based on machine learning are always fronted with deliberate attacks. According to attacker's understanding of target classifier, methods can be divided into two cases:

A. WHITE BOX ATTACK

Adversaries can interact with detection systems in the process of generating adversarial attack data, for the algorithm used by the malware classifiers and the parameters used by the algorithm are available. Most of the previously reported successful attacks assume that the attacker has full knowledge of the learned model. It can, therefore, be argued that reducing the amount of knowledge leaked about the model, as well as a proactive response to potential exploitation of such knowledge would provide adequate protection against adversarial data manipulation. Šrndić and Laskov [22] performed on a real learning-based system, PDFrate, for detection of PDF malware. They defined an orthogonal set of evasion strategies reflecting various degrees of available knowledge and inserted dummy content into PDF files to confuse the classifier. Their study was limited to evading the initial binary classifier and limited to known detector. Grosse *et al.* [26] proposed to use the gradient-based approach to generate adversarial Android malware examples evading their own white box classifier. Biggio *et al.* [27] also presented a gradient-based approach to systematically assess different classification algorithms against evasion attacks. They exhibited different risk levels for the classifier by increasing the attacker's knowledge of the model and samples.

B. BLACK BOX ATTACK

An attacker does not know the algorithms and parameters used by the malware classifiers, but they can still interact with the system, and has the ability to retrieve a Boolean label. Researchers generate adversarial networks and some other methods to evade detection without knowledge of the models [25]. Xu *et al.* [31] modified malware samples iteratively and tested them against a detection method using genetic programming for guiding this process. In addition, Hu and Tan [30] proposed a model called MalGAN which generates adversarial examples to attack black-box malware detection algorithms. Instead of attacking the black box directly, the attacker creates a substitute model trained to reproduce outputs observed by probing the target model with corresponding inputs. Then, the substitute model is used for gradient computation in a modified GAN to produce evasive malware variants.

Among the black box attacks, Anderson's pioneering works [7], [34] use deep reinforcement learning to attack the malware engines. Without prior knowledge of the static PE malware classifier's structure, features and parameters, the authors design a series of actions to interact with the malware samples. Their main purpose is to come up with a framework to solve problems in security field using reinforcement learning. There are some comparisons with these pioneering works in Section IV.

This work focuses on the following points and has made some improvements: (1) Each action is guaranteed to be effective on all samples and will not lead to corruption. Thus, the training process would not be interrupted; (2) Low dimensions of the observation is used to reduce the instability raised

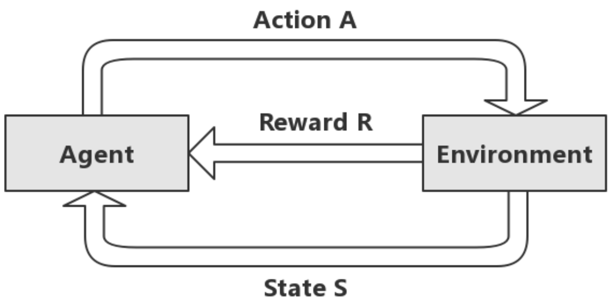


FIGURE 1. **Reinforcement learning general structure.**

by high dimensions; (3) Priority is taken into consideration during the replay of the past transitions.

III. METHOD

A. PRIMARY CONCEPT

Reinforcement learning is a branch of machine learning where an *agent* seeks to learn optimal *action* decision-making by trying to maximize cumulative *rewards*. The learning process requires interaction between *Environment* and *Agent*, as is shown in Figure 1.

The functions of *Environment*, which integrate with observation and rewards evaluation, are to provide state (the description of current stage) and *reward* (*action* evaluation value) for *Agent*. *Agent* is composed of parameters and algorithm to determine the *action* to take next. So, the input of reinforcement learning is *rewards* and states of *Environment* observation. The output is the strategy of *action* selection in the *Agent* for a certain problem.

B. MODEL STRUCTURE

Using reinforcement learning to tackle anti-malware evasion problem, appropriate definition of *environment*, *action*, *reward* and *agent* are necessary. In more details, Figure 2 exhibits the entire architecture of DQEAF. The *environment* represents a holistic view of the malware samples, which consists of a feature extractor observing the sample state and a black-box malware classifier to judge the samples. The judge result (malware or benign) is used to define the *reward* as detailed in section III-E. The *reward* and state are then fed back into the *agent*.

Then the *agent* takes the features of the samples and *reward* with two neural networks to compute action-value Q and target action-value $\hat{Q}$. During training process, networks are updated for global optimization of $\hat{Q}$. Then it can choose one reasonable *action* based on action choosing policy, which is exerted on the file sample to evaluate for the next turn. After several turns when this malicious sample is classified as a benign one, the training for this sample ends. When all trainings come to an end, the *agent* achieves optimal parameters of neural networks. Its network value with extracted features as input decides which *action* to adopt. The *action* space consists of a set of modifications exerted on the PE malware samples which do not break the format and functions of them.

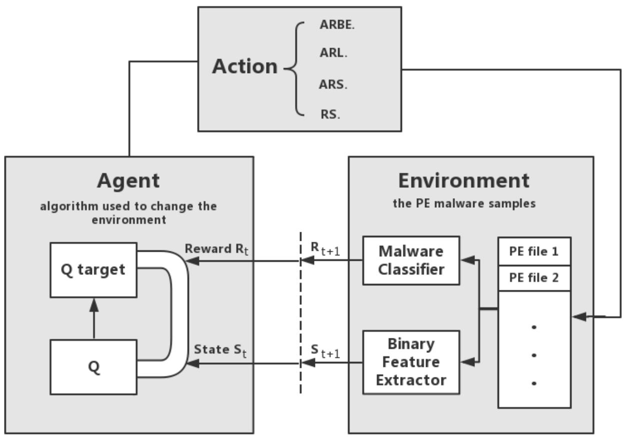


FIGURE 2. **DQEAF model structure.**

There exists a series of states transitions $(s_1, s_2, s_3, \ldots, s_t)$ to an end state under Markov hypothesis. The transition probability of state and the output probability of observation are only dependent on the current state. Due to its dependency, we record

$$(s_t, a_t, r_{t+1}, s_{t+1}) \tag{1}$$

as one piece of transition information in the training, each of which works out a Q with some weights θ to present its correctness in *action* selection.

The intelligent learning in our framework comes from MDP (Markov Decision Process) which consists of 4-tuple, $(S, A, \gamma, R(A, S))$.

- S: a set of states (as detailed in section III-C Environment) representing the current composition of the malware sample, offered by the *environment* and then transferred to *agent*, of which S_t indicates the state at the time of t.
- A: a set of actions (as detail in section III-D Action Space) to alter the sample's structure, of which A_t indicates the chosen *action* at the time of t.
- γ: a discount factor $\gamma \in [0, 1]$ referring to the relative significance of future rewards to the present.
- $R(A, S)$: reward value $R_a(s, s')$ (how to define R will be explained in the section of III-E Reward) offered by the *environment* after the state transiting from s to s' through *action a*.

Transitions will be used in training section III-F as past experience to replay. With several pieces, an *action* that brings out the maximum *reward* in the condition of s can be found. That is the final approximate action for state s in Equation (2).

$$a_t = \arg\max_a Q(s_t, a; \theta) \tag{2}$$

After this maximization process, our chief goal, to obtain the best DQEAF *agent* which will take the optimal *actions* to modify a malicious software to evade detection, is achieved.

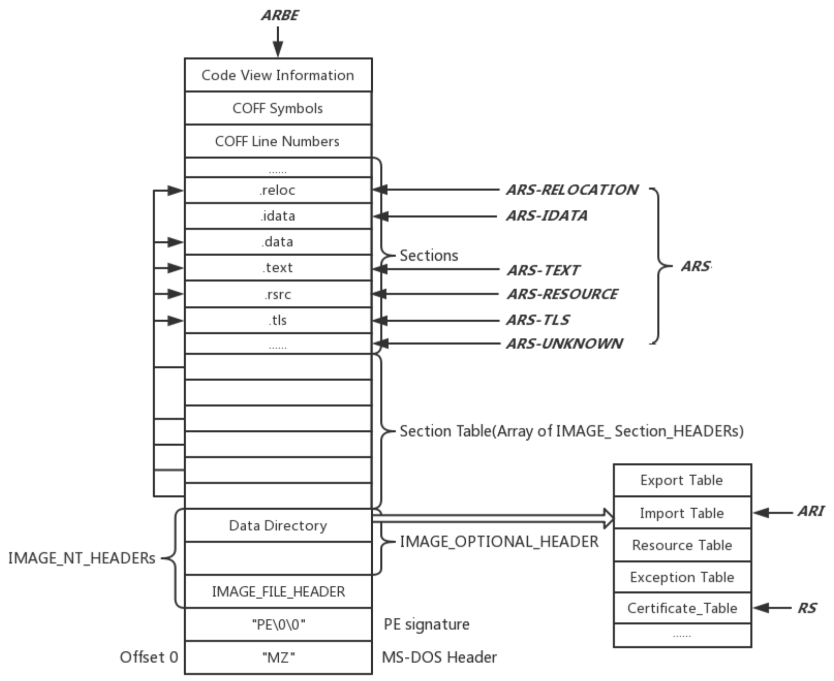


FIGURE 3. PE file format and actions to modify the content.

C. ENVIRONMENT

In DQEAF, the *environment* is used to observe the manifestation and composition of PE malware samples. The more comprehensive and reasonable the observation is, the easier the training results may reach the expected rate. Due to the characteristics of reinforcement learning, the pre-process of the samples is not needed.

The *environment* is defined by the raw binary feature as it can represent a holistic view of the malware samples and it is sensitive to each step of *action*. We count byte histogram normalized to sum to unity, which is done by counting number of occurrences of each feature value over the entire binary file. Besides, two-dimensional entropy histogram [4] is used to represent the distribution of bytes, which is implemented by sliding a window with constant bytes and computing the base-2 entropy of each window.

D. ACTION SPACE

After the *agent* observed the state of the *environment*, it must choose one *action* from the set of all available *actions*. Formally the set of all possible *actions* is defined as $A = \{\text{ARBE}, \text{ARI}, \text{ARS}, \text{RS}\}$. Therefore, at time t, the *agent* chooses an *action* a_t. All *actions* exerted on PE files are shown in Figure 3. In this paper, the *actions* are only a few possible choices. Reinforcement learning with extremely large *action* spaces will increase the difficulty of model training and the time elapse [33]. As for the *action* space designing, two aspects are taken into consideration.

It is necessary to ensure that the modification of the file is as simple as possible to guarantee the successful modification. Modification process should not result the failure, such as the collapse of malware due to the modification of ''rude action''. These failures will interrupt the training process. According to the training of this work, the simplest actions as follows are chosen.

1) **ARBE**. Append random bytes to the end of PE file.
2) **ARI**. Append a randomly named library with random function name to the import address table of PE file.
3) **ARS**. Append a randomly named section to the section table of PE file. The following section types are randomly set for additional data.
   ARS-BSS. Set the section type to BSS. The BSS section, which refers to ''Block Start with Symbol'', stores any uninitialized static and global variables.
   ARS-UNKNOWN. Set the section type to unknown.
   ARS-IDATA. Set the section type to IDATA. The IDATA section contains information about functions (and data) that the module imports from other DLLs.
   ARS-RELOCATION. Set the section type to RELOC. The RELOC section holds a table of base relocations.
   ARS-RESOURCE. Set the section type to RSRC. The RSRC section contains all the resources for the module.
   ARS-TEXT. Set the section type to TEXT. The TEXT section is the default section for code.
   ARS-TLS. Set the section type to TLS. The TLS section, which refers to ''thread local

storage", is related to the TlsAlloc family of Win32 functions.

4) **RS**. Remove signature from certificate table of the *DataDirectory*.

To begin with, the agent reads the content of original binary PE file, append or remove content at the specified location, and amend the relative virtual address for sample with ensurance of the integrity of the file. Details are in Algorithm 1.

In addition to that, it is necessary to ensure that the original function of the file is not damaged after the *actions*. The essential thing is that *actions* will not have any impact on the structure and functions of PE files. Proof of this part is available in the experimental section IV-C.

Algorithm 1 Section-Table Related Manipulate Algorithm

```
1: Parse the of content original binary file $bin_t$
2: Create an unused new section $Section_{new}$
3: Fill $Section_{new}$ with content
4: Amend the RVA address
   $RVA_{Section_{new}} = \max RVA_{bin_t sections}$
5: $Section_{new} + bin_t \Rightarrow bin_{t+1}$
6: return $bin_{t+1}$
```

E. REWARD

The *reward* is an element that differentiates reinforcement learning from other types of machine learning. How to select the appropriate *reward* for a given task is an unanswered problem in traditional reinforcement learning. It would be desirable if the *agent* could choose its own *reward*, instead of requiring an expert to define it.

An independent classifier is used to evaluate the observation and the output is a label (malicious or benign) defined as *reward*. After an *action* a_t is executed, the classifier will return a label according to current *environment*. As our aim is to evade detection and take actions as few as possible, we define *reward* for each training "TURN" based on the label from the classifier and the amount of actions taken. The *reward* is 0 if the label is "malicious", and is calculated using Equation (3) when the label is "benign". The trend of this equation is shown in Figure 4, which means that the smaller the "TURN" is, the larger the *reward* will be. "MAXTURN" is defined, which means that the *agent* should claim failure if "MAXTURN" steps of modification have been taken and the reward is still 0. Whenever a positive reward is returned, which means the malicious file evades successfully, the process will end and the *agent* can learn from it.

$$r_t = 20^{-(TURN-1)/MAXTURN} * 100 \tag{3}$$

In this part, this work is inspired by CarRacing-v0 on OpenAI Gym [37]. Reward of it is −0.1 for every frame and +1000/N for every track tile visited, where N is the total number of tiles in track. Thus, the agent can be trained to finish all tiles as fast as possible.

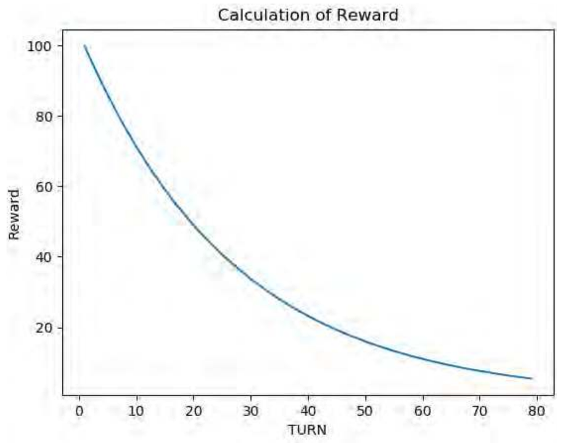


FIGURE 4. The calculation of reward.

In addition, due to *environment* is multivariable, it's uncertain whether the same *reward* can be gotten after the next same *action*. The more the times are, the more the differences are. Therefore, discount future incentives is necessary. The *reward* value is calculated by Equation (4) in step t. γ is a discount factor between 0 and 1, and the farther away from now, the less the reward is. It is easy to see that the value of the discounted future rewards in the step t can be expressed in the same way as in the step $t+1$.

$$\begin{aligned} R_t &= r_t + \gamma r_{t+1} + \gamma^2 r_{t+2} + \ldots + \gamma^{n-t} r_n \\ &= r_t + \gamma(r_{t+1} + \gamma(r_{t+2} + \ldots)) \\ &= r_t + \gamma R_{t+1} \end{aligned} \tag{4}$$

The discount factor is defined as $\gamma = 0$, then the strategy will be too easy, which is based on instant *reward*. The immediate and future *rewards* could be kept balanced when the discount factor equals to 0.9. If *environment* is determined and the same *actions* always lead to the same *reward*, then we can define the discount factor as $\gamma = 1$. In the context, we tried several values of γ, and finally we summarized that the best γ (see details in section IV-B.4).

F. AGENT

Agent may contain the algorithm itself or simply provide an integration between an algorithm and the *environments*. It describes the method to run a reinforcement learning algorithm against an *environment*. In this case, it decides which training path DQEAF to take. The value of different *actions* in a certain state is valuable. If we know the value of each action, we can choose the most valuable *action* to perform. We designed a deep convolutional Q-network as an *agent*, taking efficient actions to modify a malware to evade malware detection. A convolutional network is a class of deep, feed-forward artificial neural networks that use a variation of multilayer perceptrons designed to require minimal preprocessing. As the dimension of data goes up, the required resources for training and computation grow exponentially. However, artificial neural network can generalize from the input data and solve the problem of high-dimensional sensory input.

The deep Q-network in DQEAF is an extension of convolutional neural networks, by adding some new features with an action-value Q and a target action-value $\hat{Q}$ which map states to *action* utilities (which is the value of each *action* according to state or observation) for long-term *reward*.

Two networks used Q_{value} and Q_{target} separately in computing loss function. The loss function is defined in Equation (5), as shown at the bottom of the next page.

In the context of evading malware detection, network takes the features extracted from the raw binary stream of a malware sample as input, then produces the next malware manipulation the *agent* will take. As for the output layer of network, it's set to action-index in *actions* set A. In this context, the output dimension equals 4, which refers to the total dimension of *action* space.

In more details, the hidden layers, the values of which are not observed in the training set, are designed to 2 layers. The first consists of a convolutional layer with 256 filters using linear activation function, a batch-normalization layer [35] and a rectified linear unit function. The second consists of a convolutional layer with filter number of 64, a batch-normalization layer and a rectified linear unit function. Here, batch normalization is used to improve generalization with choosing Dropout of the network and initiate a large learning rate to speed up the training. What's more, the advantages of rectified linear unit functions are faster convergence in SGD, alleviating the problem of gradient disappearance and a simple implementation. These layers cooperate into a strong network to compute from the input malware features to the output action-index. The network diagram is showed as Figure 5.

Dropout has been demonstrated to be a very simple and efficient approach for preventing over-fitting in deep neural network. The dropout solution is potentially more resilient to imperfect or dirty data (which is common when extracting features from similar malware that was compiled or packed using different software) [4]. Rectified linear units (ReLU) have been shown to significantly speed up network training over traditional sigmoid activation functions.

G. TRAINING

The training process is illustrated in Training Algorithm (Algorithm 2) integrated with Testing Algorithm (Algorithm 3) to generate models with a higher evasion accuracy.

When it comes to the action-selection policy of the learning, exploration and exploitation policy is considered. Exploration refers to random choice of modifications to explore more possibilities. Exploitation is the choice of the best modification that has been executed to improve the model. Here $\epsilon - greedy$ policy is used, which practices a random *action* with a probability ϵ and selects the most valuable *action* with a probability $1 - \epsilon$ otherwise. The ϵ reflects how much importance is attached to exploratory *action* policy [5]. The value of ϵ decreases during training epochs progress according to Equation (6). Where n is the current training step and N is the total number of training steps.

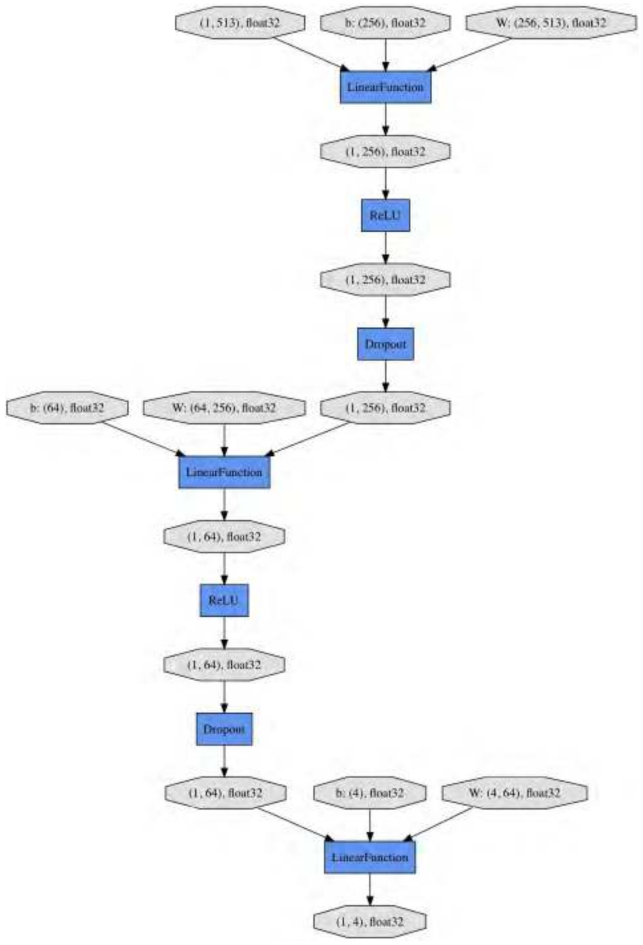


FIGURE 5. DQEAF training network diagram.

$$\epsilon_n = 1.0 - \frac{n}{N} \tag{6}$$

Experience replay is used to let reinforcement learning agents remember and reuse experiences from the past. In the process of replay, temporal correlations are broken and transitions are replayed with the same frequency. Our framework is inspired by prioritized experience replay [6], which takes the priority of transitions into consideration and transitions with high significance can be replayed more frequently. In this work, only the successful modifications of the malware can get high *reward* and these transitions should have higher priority to be replayed than others.

In each step, transition $Experience_t = (s_t, a_t, r_{t+1}, s_{t+1})$ (previously mentioned in Equation (1)) will be stored at maximal priority in memory to make sure that each experience will be replayed at least once. As is shown in Training Algorithm 2, after B transitions are stored, the experience replay begins. K transitions from M will be sampled and the

Algorithm 2 Training Algorithm

```
 1: Initialize Memory $M$ to capacity $N$
 2: Initialize two identical networks, action-value $Q$ and target-value $\hat{Q}$,
    with random weights $\theta$ and weights $\hat{\theta} = \theta$ respectively
 3: for $episode = 1$ to $D$ do
 4:     Initialize binary stream $bin_{init}$ of a file selected from the malware files list $E$
 5:     Extract binary features
        $s_{init} = ExtractBinaryFeature(bin_{init})$
 6:     for $t = 1$ to $T$ do
 7:         With probability $\epsilon$ using equation (6) select a random action $a_t$ or
            choose $a_t = \arg\max_a Q(s_t, a; \theta)$ using equation (2)

 8:         Modify $bin_t$ by action $a_t$ to $bin_{t+1}$ and
            $s_{t+1} = ExtractBinaryFeature(bin_{t+1})$,
            observe reward $r_{t+1}$
 9:         Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$ using (1)
            with priority $p = \max p_i$
10:         if $sizeofM > B$ then
11:             for $i = 1$ to $K$ do
12:                 Sample transition $X_i \sim P(X_i) = p_{X_i} / \sum_j p_j$
                    using equation (7)
                    (j represents each transition in $M$)
13:                 Compute TD-error $\delta_{X_i}$ using equation (8)
14:                 Update $p_{X_i} = |\delta_{X_i}|$
15:                 Set loss function $Loss_{X_i}$ using equation (5)
16:             end for
17:             Exert Adam optimizer to optimize parameter $\theta$
18:         end if
19:         Every $U$ step reset $\hat{\theta} = \theta$
20:         if done then
21:             break
22:         end if
23:     end for
24:     if episode mod TEST_INTERVAL == 0 then
25:         Run Testing Algorithm and get success ratio $SR$
26:         Store Q and $\hat{Q}$ to a new model $m$
27:         if $SR >$ MAX_RATIO then
28:             break
29:         end if
30:     end if
31: end for
```

probability of sampling transition X_i is

$$P(X_i) = p_{X_i} / \sum_j p_j \tag{7}$$

Algorithm 3 Testing Algorithm

```
 1: Initialize total reward $R = 0$
 2: The amount of test files $F$
 3: for $i = 1$ to $F$ do
 4:     Analyze binary stream $bin_{init}$ of $F$
        and get $s_{init} = ExtractBinaryFeature(bin_{init})$
 5:     for $t = 1$ to $T$ do
 6:         Compute $\hat{Q}$ and choose $a_t = \arg\max_a \hat{Q}(s_t, a; \hat{\theta})$
 7:         Alter $bin_t$ by action $a_t$ to get $bin_{t+1}$,
            $s_{t+1} = ExtractBinaryFeature(bin_{t+1})$,
            observe reward $r_{t+1}$
 8:         Add up $r_{t+1}$ to $R$
 9:         if done then
10:             break
11:         end if
12:     end for
13: end for
14: Compute the value of $SR$
15: return $SR$
```

The priority of each transitions will be recalculated after replay. TD error is used to measure the significance of transitions, and it is calculated by Equation (8). The priority is updated according to TD error, and valuable transitions will get higher priority and are more possible to be replayed.

$$\delta_{X_i} = r_{X_i} + \gamma \hat{Q}(s_{t+1 X_i}, \arg\max_a Q(s_{t X_i}, a_{X_i}; \theta); \hat{\theta}) - Q(s_{X_i}, a_{X_i}; \theta) \tag{8}$$

Optimizer can reduce the loss with discount factor γ to approach the optimum. The unique feature of DQN lies in the future reward from target action-value $\hat{Q}$ instead of action-value Q. After U turns, $\hat{Q}$ will be updated to Q. While training, the *agent* progressively gets close to the optimal action-values through the *reward* offered by the *environment*.

IV. PERFORMANCE EVALUATION

The DQEAF's architecture for evading static PE anti-malware engines is implemented based on reinforcement learning, and the experimental results are shown in this section.

A. THE IMPLEMENTATION

DQEAF's *environment* is defined on the gym toolset, a toolkit for developing and comparing reinforcement learning algorithms, provided by OpenAI. The basic class of *environment* which named *MalwareEnv* is extended to fit into the malware detection evasion problem. The *environment* must define its observation space and action space and have at least two methods: *reset* and *step*.

$$Loss_i = \begin{cases} (r_i - Q_{value})^2 & \textit{episode terminates at step } i{+}1 \\ (r_i + \gamma Q_{target} - Q_{value})^2 & \textit{otherwise} \end{cases} \tag{5}$$

TABLE 1. Number of samples in training set and testing set.

Type	Training set	Testing set
Backdoor	2.1K	200
Trojan	3.2K	200
Worm	1.6K	200
Email	1.9K	200

- *env.reset* function will reset the *environment* to initial state and return the initial observation s_0.
- *env.step* will execute a given *action*, move to the next state, then calculate the detection result of the malware sample in the next state.

DQEAF is implemented in Python 3.6. Experiments use the Chainer deep learning library to implement the neural network model. Modification of the PE file uses LIEF [36] toolkit version 0.7.0, which provides a cross platform library to parse, modify and abstract PE formats. The latest version cannot support our framework, because byte stream of PE file cannot be used as input. Each sample's features are extracted by a single thread process. Plotting experimental data uses Visdom library.

B. EXPERIMENT SETUP

1) SAMPLES

a: SAMPLE SELECTION

Samples used in this work comes from VirusTotal, which is open source. In the classifier used in this paper, all samples are classified as malicious files. These samples are divided into training set and testing set. In order to reduce the disk IO operation to speed up the training process, all the samples are cached before training, read all the binary data of the sample into memory, and read the observation space directly from the memory during the training process. The number of samples are shown in Table 1. Four agents are trained based on four families of malicious PE files under the Win32 platform, including "Backdoor", "Trojan", "Worm" and "Email".

b: FEATURES NORMALIZATION

In addition to use batch normalization for each layer of the network in agent training, the 513-dimensions observation of each step is normalized to [0.5, −0.5].

c: SAMPLE EFFECTIVENESS

In order to prevent training process interrupted by the failure of the "rude action", mentioned in section III-D, each sample is executed in turn to ensure that all actions are effective on all samples. Besides, all samples are classified as malicious software by the target classifier.

2) ACTIONS

a: SUCCESSFUL MODIFICATION

To prove that the *actions* are executed successfully, some tests are done. For each sample, a random sequence of *actions* is generated and all *actions* in action space are ensured to be covered. After each step of modification, the binary streams and some features before and after the modification are compared to determine whether the *action* is executed successfully. The features considered in this work include: length of sections, section name, existence of signature, existence of debug section and imported function.

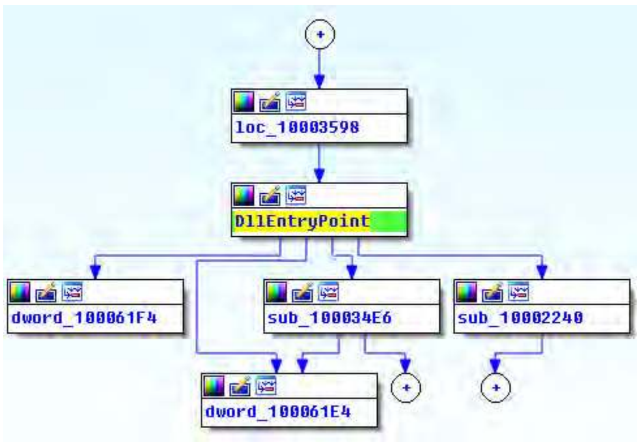


FIGURE 6. The flow chart of Backdoor.Win32.PcClient.sx and the flow chart of it after executes ARBE action.

b: NO CORRUPTION

In order to verify action manipulations will not corrupt the samples' execution, the Cuckoo sandbox [38] is used. Cuckoo runs a submitted sample and analyzes it in a virtual machine. It also reports comprehensive analysis results that outline what the malware does while running inside an isolated operating system, the behavior of a sample including network APIs called, their parameters and so on. The experimental results show that the modified samples can be successfully executed without corruption.

c: FUNCTION PRESERVATION

It is necessary to determine whether variants preserve malicious behavior. IDA Pro is used to disassemble the files to generate function call graphs, flow charts and binary files, and display the import tables and function tables of files. By comparing the function call graphs of all the samples before and after modification respectively, the flow charts have not changed, which can prove that *actions* do not affect the structure and functions of the files. The flow chart of a sample is showed in Figure 6.

3) INDEPENDENT CLASSIFIER

This classifier is trained on 50k malicious and 50k benign samples, and extracts features including byte-level data, header, section and import/exports. The model is based on gradient boosted decision tree and trained by Sklearn. To evaluate this classifier, Receiver Operating Characteristic curve (ROC curve) and Area Under the Curve (AUC) are used. ROC is widely used in machine learning and data mining, describing the compromise between Detection Rate (DR) and False Positive Rate (FPR). Besides, ROC curve can

TABLE 2. List of major parameter and their values in the training algorithm.

Parameter settings	Value	Description
MAXTURN	80	The *agent* is allowed to perform up to "MAXTURN" steps modification before declaring failure, corresponds to the parameter T from Algorithm 2.
replay start size	1000	A uniform random policy is run for this number of steps before learning starts, corresponds to the parameter B from Algorithm 2.
replay buffer max size	$5 * 10^5$	The capacity of replay buffer memory, corresponds to the parameter N from Algorithm 2.
minibatch size	32	The number of training cases over which each stochastic gradient descent (SGD) update is computed, corresponds to the parameter K from Algorithm 2.
target update interval	100	The frequency with which the target network is updated, corresponds to the parameter U from Algorithm 2.
discount factor	0.99	Discount factor γ used in the Q-learning update.
initial exploration	1	Initial value of ϵ in ϵ-greedy exploration.
final exploration	0.1	Final value of ϵ in ϵ-greedy exploration.
final exploration steps	1000	The number of steps over which the initial value of ϵ is linearly annealed to its final value.
TEST_INTERVAL	1000	The number of steps we carried out the evaluation to the agent.
MAX_RATIO	7	Finish training if the mean score is greater or equal to this value.
training files count	2163	The number of training sample list, corresponds to the parameter E from Algorithm 2.
testing files count	200	The number of testing sample list, corresponds to the parameter F from Algorithm 3.

keep constant while the distribution of positive and negative samples changes. AUC is a method to calculate DR based on ROC, and the larger the value is, the better the model is. The AUC score of the model is 0.96.

4) PARAMETERS SETTING

In experiments, the independent machine learning classifier to be evaded is a gradient boosted decision tree model trained in 50,000 malicious and 50,000 benign samples. As is shown in training algorithm of section III-G, training rounds terminate early when the agent bypass the detector prior to the *T=80* rounds. Each model can be trained for up to *D=30,000* rounds, of which the *agent* is evaluated each *TEST_INTERVAL = 1000* steps, and each evaluation is tested on *F=200* test samples one by one. After training, if the evaluation mean score is more than *MAX_RATIO =7*, the training is stopped ahead of time (by this time the success rate has exceeded *70%*, and it is proved that the continued training cannot be higher in this work).

What's more, for the DQEAF parameters setup, discount factor γ uses *0.99*. Adam algorithm with minibatche of size *32* is used, replay buffer start-size *B* uses *1000*. *Agent* target-update-interval uses *100* and uses *"hard"* update method. *Agent* also utilizes a *"LinearDecayEpsilonGreedy"* exploration/exploitation strategy, in which mutations are drawn proportionally to their expected Q-value. The behavior policy during training was $\epsilon-$*greedy* with ϵ annealed linearly from *1* to *0.1* over the first *1000* steps, and fixed at *0.1* thereafter. The major parameter settings in the training algorithm are shown in Table 2.

C. RESULT

In supervised learning, one can easily track the performance of a model during training by evaluating it on the training and validation sets. In reinforcement learning, however, accurately evaluating the progress of an *agent* during training can be challenging [24].

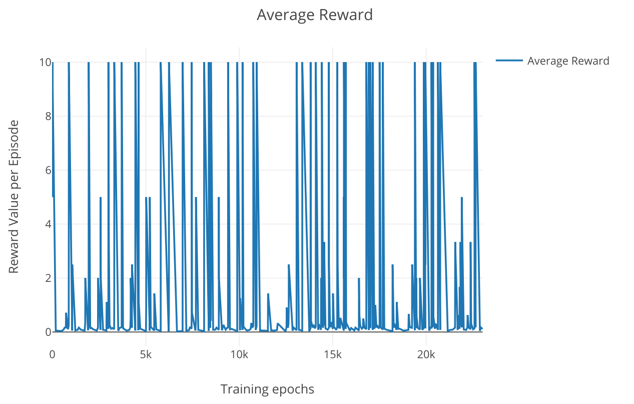


FIGURE 7. The average reward per episode during training.

The result of the model is trained on "Backdoor" files. Other models are trained and evaluated in the same way.

Statistical metric of model evaluation steps in the training process are recorded periodically. In Table 3, the indicators include metric of the average total reward value, the policy's estimating action-value function *Q*, which provides an estimate of how much discounted reward the *agent* can obtain by following its policy from any given state, and the average value of the loss. As shown in the table, the model goes into a stage when training for about *23,019* steps, the *SR* is up to 7.5.

The *agent*'s improvement after training can be seen below. The average total reward plot in Figure 7 tends to be very noisy because small changes to the weights of a policy can lead to large changes in the training path. The plot in Figure 9 shows loss value dropped to a very small value after *20,000* steps of training. The trend of the change of average predicted Q in Figure 8 is much more smoothly than the average total reward in Figure 7.

TABLE 3. Evaluation metric during training steps.

Steps	Episodes	Elapsed	Mean	Stedv	Average Q	Average Loss
1003	26	954.08	4.50	5.01	0.5171	0.0041
2008	53	1624.32	5.00	5.03	1.0056	0.2713
3000	82	2082.60	5.62	4.99	1.5118	0.2466
4027	112	2758.14	5.50	5.01	2.0725	0.2280
5005	139	3530.84	5.25	5.03	2.4343	0.2055
6040	164	4398.04	4.38	4.99	2.4120	0.1744
7032	186	4999.66	5.38	5.02	2.8273	0.1638
8049	211	5496.33	5.75	4.97	2.7582	0.1451
9034	239	6407.98	5.00	5.03	2.5164	0.1439
10013	262	6988.33	5.50	5.01	2.3886	0.1264
11050	289	7584.60	5.75	4.97	2.1938	0.1262
12009	310	8777.15	4.50	5.01	2.3688	0.1047
13035	335	9918.68	5.38	5.02	2.1504	0.0927
14031	362	11410.55	0.00	0.00	1.9472	0.0874
15007	393	12607.03	5.38	5.02	2.0214	0.0854
16045	426	13707.65	5.00	5.03	1.8591	0.0801
17003	454	14685.71	6.38	4.84	1.8792	0.0760
18059	485	15461.82	5.38	5.02	1.6266	0.0714
19043	509	16562.02	5.50	5.01	1.7199	0.0607
20012	544	17332.88	5.12	5.03	1.5969	0.0718
21050	573	18308.19	6.25	4.87	1.6261	0.0584
22008	601	19246.25	5.12	5.03	1.6409	0.0577
23019	**628**	**20104.99**	**7.50**	**4.49**	**1.5424**	**0.0528**

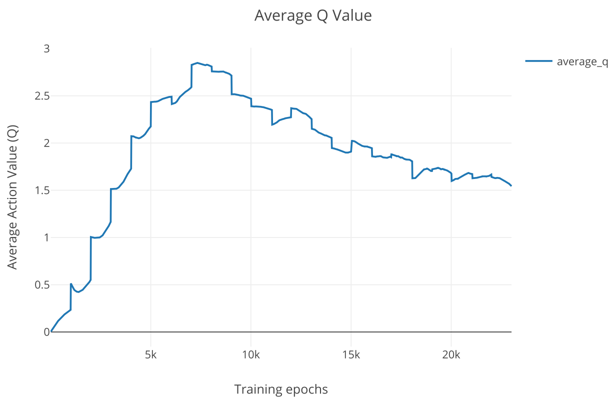


FIGURE 8. The average value of action-value function Q during training.

Testing method is the same as that evaluation approach during the training process. Two methods are used to test the 200 testing samples. In particular, for each of the 200 malware samples, the trained *agent* by DQEAF to manipulate the samples (up to MAXTURN = 80 times), and test whether any sequence of *actions* can result in a successful evasion.

This agent achieves general expectation of robustness. This work also evaluates the effectiveness of proposed methods on other quantities of "Backdoor" files, and this method achieves general expectation of robustness. The evasion success rate of 2000 samples can still reach 70% in the test process. Evasion rate on other quantities of test samples is shown in Table 4.

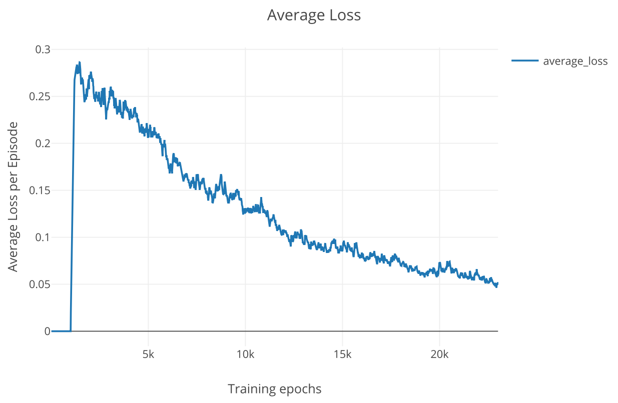


FIGURE 9. The average value of the loss during training.

TABLE 4. Evasion Success Rate on "Backdoor" Test Samples.

No. of test samples	No. of success samples	Evasion success rate
200	150	**75%**
1000	714	**71.40%**
2000	1403	**70.15%**

TABLE 5. Evasion Success Rate of models trained on different families.

Family	Evasion success rate
Backdoor	**75%**
Trojan	**17.5%**
Worm	**30%**
Email	**63.75%**

TABLE 6. Evasion Success Rate of cross validation.

Success Rate \ Test / Model	Backdoor	Trojan	Worm	Email
Backdoor	**75%**	**12.5%**	**43.75%**	**40%**
Trojan	**52.5%**	**17.5%**	**32.5%**	**27.5%**
Worm	**51.25%**	**8.75%**	**30%**	**46.25%**
Email	**53.75%**	**12.5%**	**27.5%**	**63.75%**

TABLE 7. Evasion Success Rates of models when DQEAF and Gym-Malware are trained on the same training set.

Family	DQEAF	Gym-Malware
Backdoor	**75%**	**35%**
Trojan	**17.5%**	**3%**
Worm	**30%**	**17%**
Email	**63.75%**	**21%**
Total	**46.56%**	**19%**

The efficacy of models trained on other families of PE malware are shown in Table 5.

Cross-validation is used to evaluate the robustness of each model. The models trained on one family samples are used to test other family samples. The result is in Table 6.

The work of Anderson *et al.* [7], [34] in adversarial machine learning has shown that deep learning models are susceptible to attacks and evasions. They present a general algorithm named Gym-Malware to attack the PE malware classifier by generating adversarial malware examples.

TABLE 8. 20 examples of successful modification sequence.

No.	Sample Name	Actions Sequence	Evasion Sample
1	Backdoor.Win32.PcClient.iuw	['ARS', 'RS', 'ARI', 'ARS', 'ARBE', 'ARI', 'RS', 'ARBE', 'ARBE', 'ARBE', 'RS', 'RS', 'ARI', 'ARS', 'RS', 'ARBE', 'ARBE', 'ARBE', 'ARBE', 'RS', 'ARI', 'ARS', 'RS', 'RS', 'ARI', 'ARBE', 'ARI', 'ARI', 'ARI', 'ARS', 'ARS', 'ARI', 'ARI', 'ARS', 'ARI', 'RS', 'ARS', 'ARS', 'RS', 'RS', 'ARS', 'ARI', 'ARI', 'ARI', 'RS', 'ARBE', 'ARS', 'ARBE', 'ARBE', 'ARI']	success
2	Backdoor.Win32.PcClient.ejl	['RS', 'ARI', 'ARBE', 'ARS', 'ARI', 'RS', 'ARI', 'ARI', 'ARBE', 'ARI', 'ARI', 'ARS', 'ARI', 'ARS']	success
3	Backdoor.Win32.PcClient.ejp	['RS', 'ARBE', 'RS', 'ARI', 'ARS', 'RS', 'ARBE', 'ARS', 'ARI', 'RS', 'ARBE', 'RS', 'ARI', 'RS', 'ARI', 'ARBE', 'ARBE', 'ARI', 'ARBE', 'RS', 'RS', 'ARBE', 'ARS', 'ARI', 'RS', 'ARBE', 'RS', 'ARS', 'ARBE', 'ARBE', 'ARI', 'ARBE', 'ARI', 'ARBE', 'RS', 'ARS']	success
4	Backdoor.Win32.PcClient.ctx	['ARS', 'ARBE', 'RS', 'ARI', 'ARS', 'ARS', 'ARBE', 'RS', 'ARBE', 'RS', 'ARBE', 'ARS', 'RS', 'RS', 'ARBE', 'ARBE', 'RS', 'ARI', 'ARS', 'ARI', 'RS', 'ARI', 'ARI', 'ARBE', 'RS', 'ARI', 'ARI', 'ARS', 'ARS', 'ARI', 'ARS', 'ARI', 'ARS', 'RS', 'ARBE', 'RS', 'ARI', 'ARI', 'ARI', 'ARI', 'RS', 'ARBE', 'ARI']	success
5	Backdoor.Win32.PcClient.ofy	['ARS', 'ARI', 'ARS', 'ARBE', 'ARBE', 'ARI', 'ARBE', 'ARBE', 'ARI', 'ARS', 'ARBE', 'RS', 'ARBE', 'ARBE', 'ARI', 'ARI', 'ARI', 'ARI', 'ARBE', 'RS', 'ARS', 'ARS', 'RS', 'RS', 'ARI', 'ARBE', 'ARBE', 'ARBE', 'ARS', 'RS', 'ARS', 'ARS', 'RS', 'ARS', 'ARI', 'RS', 'RS', 'RS', 'RS', 'ARBE', 'ARS', 'ARS']	success
6	Trojan-Spy.Win32.WinSpy.us	['RS', 'ARS', 'RS', 'RS', 'RS', 'RS', 'ARS', 'RS', 'ARI', 'ARBE', 'ARI', 'ARBE', 'ARS', 'ARS', 'RS', 'ARS', 'ARBE', 'ARS', 'ARI', 'ARBE', 'ARS', 'ARS', 'ARS', 'ARBE', 'RS', 'ARS', 'ARI', 'ARS', 'ARI', 'ARI', 'RS', 'ARS', 'ARS', 'RS', 'ARI', 'ARI', 'RS', 'ARBE', 'ARBE', 'ARS', 'ARI', 'ARBE', 'ARBE', 'ARS', 'ARI', 'RS', 'ARI', 'ARS', 'ARI', 'ARS', 'ARS', 'ARBE', 'ARS']	success
7	Trojan-Spy.Win32.WinSpy.rl	['RS', 'ARBE', 'ARI', 'ARS', 'ARS', 'ARBE', 'ARI', 'ARI', 'RS', 'ARS', 'ARS', 'ARBE', 'ARI', 'ARBE', 'ARI']	success
8	Trojan-Spy.Win32.WinSpy.pm	['RS', 'ARBE', 'ARBE', 'RS', 'ARS', 'ARBE', 'ARI', 'ARBE', 'ARBE', 'ARI', 'RS', 'RS', 'RS', 'RS', 'ARBE', 'ARI', 'ARI', 'ARS', 'ARI', 'ARI', 'ARI', 'RS', 'ARBE', 'ARBE', 'ARBE', 'ARS']	success
9	Trojan-Spy.Win32.Tofger.j	['ARI', 'ARBE', 'ARBE', 'ARBE', 'ARBE', 'ARS', 'ARBE', 'ARS', 'ARS', 'ARI', 'ARBE', 'ARS', 'ARI', 'RS', 'ARS']	success
10	Trojan-Spy.Win32.Sters.bk	['ARI', 'ARI', 'ARS', 'RS', 'RS', 'ARS', 'RS', 'ARI', 'RS', 'RS', 'RS', 'ARI', 'RS', 'RS', 'ARI', 'RS', 'ARBE', 'RS', 'ARI', 'ARBE', 'ARI', 'ARI', 'ARI', 'RS', 'ARS', 'ARBE', 'RS', 'RS', 'ARBE', 'ARBE', 'ARBE', 'ARS', 'ARS', 'RS', 'ARBE', 'ARI', 'ARBE', 'RS', 'RS', 'ARS', 'ARI', 'ARI', 'ARS']	success
11	Worm.Win32.AutoRun.bhy	['ARI', 'ARBE', 'RS', 'RS', 'ARS', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI']	success
12	Worm.Win32.AutoRun.rrv	['ARI', 'RS', 'ARI', 'ARS', 'ARI', 'RS', 'ARBE', 'ARI', 'ARI', 'RS', 'RS', 'ARI', 'ARI', 'RS', 'ARS', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI', 'RS', 'RS', 'RS', 'ARI', 'RS', 'ARBE', 'ARBE', 'ARI', 'ARS', 'RS', 'ARS', 'ARI', 'RS', 'ARBE', 'ARI', 'ARI', 'RS', 'ARI', 'ARI']	success
13	Worm.Win32.AutoRun.nap	['ARI', 'ARI', 'ARI', 'ARS', 'RS', 'ARS', 'ARS', 'ARBE', 'RS', 'ARS', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI', 'ARI', 'ARS', 'ARBE', 'ARI', 'ARBE', 'ARI', 'ARI', 'ARS', 'ARBE', 'ARI', 'ARBE', 'ARI', 'ARI']	success
14	Worm.Win32.AutoRun.mbd	['RS', 'ARBE', 'ARBE', 'ARI', 'ARBE', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARI', 'ARS', 'ARI']	success
15	Worm.Win32.AutoRun.ja	['ARI', 'ARI', 'ARS', 'ARS', 'ARBE', 'ARS', 'ARS', 'RS', 'ARS', 'RS', 'ARBE', 'ARS', 'ARI', 'ARI', 'ARBE', 'ARI']	success
16	Email-Flooder.Win32.Delf.o	['ARI', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'RS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARI', 'ARS', 'ARS', 'ARS', 'RS', 'ARS', 'ARS', 'ARS', 'ARS', 'RS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'RS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARI', 'ARS', 'ARBE', 'ARS', 'ARS', 'ARS', 'ARI', 'ARS', 'ARS', 'ARI']	success
17	Email-Flooder.Win32.Expout	['ARS', 'ARS', 'ARS', 'ARS', 'RS', 'ARS']	success
18	Email-Worm.Win32.Zhelatin.uu	['RS', 'ARS', 'ARS', 'ARI', 'ARS', 'ARI', 'ARI', 'RS', 'RS', 'ARI', 'ARS', 'RS', 'ARS', 'ARS', 'RS', 'ARBE', 'ARS', 'ARI', 'ARI', 'ARS', 'ARS', 'RS', 'RS', 'RS', 'ARI', 'ARS', 'ARBE', 'ARBE', 'RS', 'ARS', 'RS', 'RS', 'RS', 'RS', 'ARBE', 'ARBE', 'ARS', 'RS', 'ARS', 'ARBE', 'ARBE', 'RS', 'ARBE', 'ARBE', 'ARI', 'ARI', 'ARBE', 'RS', 'ARBE', 'ARS', 'ARS', 'RS', 'ARS', 'ARS', 'RS', 'RS', 'RS', 'ARS', 'ARS', 'ARBE']	success
19	Email-Worm.Win32.Zhelatin.vs	['ARI', 'ARI', 'ARS', 'RS', 'RS', 'ARS', 'RS', 'ARI', 'RS', 'RS', 'RS', 'ARI', 'RS', 'RS', 'ARI', 'RS', 'ARBE', 'RS', 'ARI', 'ARBE', 'ARI', 'ARI', 'ARI', 'RS', 'ARS', 'ARBE', 'RS', 'RS', 'ARBE', 'ARBE', 'ARBE', 'ARS', 'ARS', 'RS', 'ARBE', 'ARI', 'ARBE', 'RS', 'RS', 'ARS', 'ARI', 'ARI', 'ARS']	success
20	Email-Worm.Win32.Zhelatin.yr	['RS', 'ARI', 'RS', 'ARBE', 'ARBE', 'RS', 'ARI', 'ARS', 'ARBE', 'RS', 'ARBE', 'ARBE', 'ARBE', 'ARS', 'ARBE', 'ARS', 'ARS', 'ARBE', 'ARS', 'ARS', 'ARS', 'ARS', 'ARS', 'ARBE', 'ARBE', 'ARI', 'ARBE', 'ARS', 'ARBE', 'ARBE', 'ARS', 'ARS', 'ARI', 'RS', 'ARBE', 'ARS', 'RS', 'RS', 'ARS', 'ARI', 'RS', 'ARI', 'ARS', 'ARI', 'RS', 'RS', 'RS', 'ARI', 'ARS', 'ARI', 'RS', 'ARS', 'ARS', 'ARI', 'ARS', 'ARS', 'RS', 'ARI', 'ARS', 'ARS']	success

As their open source implementations are available and their method only requires black-box access to the classifier, we made a comparison between the proposed DQEAF and their algorithm.

Their framework is used to train these four families samples aforementioned in Table 1. The comparative experimental results are shown in Table 7. These two models trained on the same training set against the same classifier.

Our framework shows more advantages and the evasion success rate of adversarial examples in each families from DQEAF are higher than that from Gym-Malware. Priority is taken into consideration during the replay of the past transitions, thus transitions with higher value can be replayed more frequently and the reinforcement learning network is optimized. Low dimensions observation is used to reduce the instability raised by high dimensions, and the convergence speed of network training is increased.

Successful evasion samples of "Backdoor" produced by DQEAF are uploaded to Virustotal, and found that the average detection ratio was 28 / 66, down from 54 / 66.

The focus of this paper is static Windows PE malware, and furthermore it can be extended to other sorts of malware. The training process does not extract any feature value based on the PE file structure features, but is based on the characteristics of binary byte stream. So this evasion method can also be applied to other platforms such as ELF, Android.

V. CONCLUSION

This paper proposed a framework named DQEAF using reinforcement learning to evade anti-malware engines. The core component of DQEAF is an AI agent, which is constantly interacting with malware samples. The agent could choose optimal sequences of functionality-preserving actions deliberately by deep reinforcement learning, which aims to evade the supervised detector. Experiments show that the proposed DQEAF has a high success rate in PE samples. In the future work, we will consider other platforms with different file structures and try to explore the defense of the attacks.

APPENDIX A LOG OF TRAINING ACTIONS SELECTION

We logged the selection of actions in training process for each sample. Details about the sequence of actions for samples that have been modified successfully can be seen in Table 8. As listed in the table, 4 families that include 20 examples are shown.

REFERENCES

[1] L. Xu, C. Jiang, J. Wang, J. Yuan, and Y. Ren, "Information security in big data: Privacy and data mining," *IEEE Access*, vol. 2, pp. 1149–1176, 2017.

[2] *Mobile Threat Intelligence Report*, Symantec, Mountain View, CA, USA, 2017.

[3] V. Sze, Y.-H. Chen, J. Emer, A. Suleiman, and Z. Zhang, "Hardware for machine learning: Challenges and opportunities," in *Proc. IEEE Custom Integr. Circuits Conf.*, Apr./May 2017, pp. 1–8.

[4] J. Saxe and K. Berlin, "Deep neural network based malware detection using two dimensional binary program features," in *Proc. 10th Int. Conf. Malicious Unwanted Softw. (MALWARE)*, Fajardo, Puerto Rico, Oct. 2015, pp. 11–20.

[5] E. R. Gomes and R. Kowalczyk, "Dynamic analysis of multiagent Q-learning with ϵ-greedy exploration," in *Proc. Int. Conf. Mach. Learn.*, 2009, pp. 369–376.

[6] T. Schaul, J. Quan, I. Antonoglou, and D. Silver. (2016). "Prioritized experience replay." [Online]. Available: https://arxiv.org/abs/1511.05952

[7] H. S. Anderson, A. Kharkar, B. Filar, and P. Roth, "Evading machine learning malware detection," presented at BlackHat, Jul. 2017.

[8] M. Jagielski, A. Oprea, B. Biggio, C. Liu, C. Nita-Rotaru, and B. Li. (2018). "Manipulating machine learning: Poisoning attacks and countermeasures for regression learning." [Online]. Available: https://arxiv.org/abs/1804.00308

[9] B. Biggio, B. Nelson, and P. Laskov, "Poisoning attacks against support vector machines," in *Proc. ICML*, 2012, pp. 1–8.

[10] A. Newell, R. Potharaju, L. Xiang, and C. Nita-Rotaru, "On the practicality of integrity attacks on document-level sentiment analysis," in *Proc. Workshop Artif. Intell. Secur.* Montreal, QC, Canada: AIESEC, 2014, pp. 83–93.

[11] H. Xiao, B. Biggio, G. Brown, G. Fumera, C. Eckert, and F. Roli, "Is feature selection secure against training data poisoning?" in *Proc. 32nd Int. Conf. Mach. Learn.*, vol. 37, 2015, pp. 1689–1698.

[12] B. I. Rubinstein *et al.*, "Antidote: Understanding and defending against poisoning of anomaly detectors," in *Proc. 9th Internet Meas. Conf. (IMC)*, 2009, pp. 1–14.

[13] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *Proc. IEEE Secur. Privacy Symp. (SP)*, May 2017, pp. 39–57.

[14] A. Sami, B. Yadegari, H. Rahimi, N. Peiravian, S. Hashemi, and A. Hamze, "Malware detection based on mining API calls," in *Proc. ACM Symp. Appl. Comput.*, 2010, pp. 1020–1025.

[15] H.-T. Cheng *et al.*, "Wide & deep learning for recommender systems," in *Proc. 1st Workshop Deep Learn. Recommender Syst.*, 2016, pp. 7–10.

[16] H. Li *et al.*, "A machine learning approach to prevent malicious calls over telephony networks," in *Proc. IEEE Symp. Secur. Privacy*, San Francisco, CA, USA, May 2018, pp. 53–69.

[17] Y. Xue, P. Zhou, S. Mao, D. Wu, and Y. Zhou, "Pure-exploration bandits for channel selection in mission-critical wireless communications," *IEEE Trans. Veh. Technol.*, vol. 67, no. 11, pp. 10995–11007, Nov. 2018.

[18] Z. Qin, Y. Wang, H. Cheng, Y. Zhou, Z. Sheng, and V. C. M. Leung, "Demographic information prediction: A portrait of smartphone application users," *IEEE Trans. Emerg. Topics Comput.*, vol. 6, no. 3, pp. 432–444, Jul./Sep. 2018.

[19] I. J. Goodfellow, J. Shlens, and C. Szegedy. (2015). "Explaining and harnessing adversarial examples." [Online]. Available: https://arxiv.org/abs/1412.6572

[20] N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami, "The limitations of deep learning in adversarial settings," in *Proc. IEEE Eur. Secur. Privacy Symp. (Euro S&P)*, Mar. 2016, pp. 372–387.

[21] N. Papernot, P. McDaniel, X. Wu, S. Jha, and A. Swami, "Distillation as a defense to adversarial perturbations against deep neural networks," in *Proc. IEEE Secur. Privacy Symp. (SP)*, May 2016, pp. 582–597.

[22] N. Šrndić and P. Laskov, "Practical evasion of a learning-based classifier: A case study," in *Proc. IEEE Secur. Privacy Symp. (SP)*, May 2014, pp. 197–211.

[23] C. Szegedy *et al.* (2014). "Intriguing properties of neural networks." [Online]. Available: https://arxiv.org/abs/1312.6199

[24] V. Mnih *et al.*, "Playing Atari with deep reinforcement learning," in *Proc. Mach. Learn. Princ. Pract. Knowl. Discovery Databases*, 2013, pp. 1–9.

[25] H. S. Anderson, J. Woodbridge, and B. Filar, "DeepDGA: Adversarially-tuned domain generation and detection," in *Proc. ACM Workshop Artif. Intell. Secur.*, 2016, pp. 13–21.

[26] K. Grosse, N. Papernot, P. Manoharan, M. Backes, and P. McDaniel. (2013). "Adversarial perturbations against deep neural networks for malware classification." [Online]. Available: https://arxiv.org/abs/1606.04435

[27] B. Biggio *et al.*, "Evasion attacks against machine learning at test time," in *Proc. Eur. Conf. Deep Neural Netw. Malware Classification*, 2013, pp. 387–402.

[28] Q. Liu, P. Li, W. Zhao, W. Cai, S. Yu, and V. C. M. Leung, "A survey on security threats and defensive techniques of machine learning: A data driven view," *IEEE Access*, vol. 6, pp. 12103–12117, 2018.

[29] Z. Guan, L. Bian, T. Shang, and J. Liu, "When machine learning meets security issues: A survey," in *Proc. IEEE Int. Conf. Intell. Saf. Robot. (ISR)*, Aug. 2018, pp. 158–165.

[30] W. Hu and Y. Tan. (2017). "Generating adversarial malware examples for black-box attacks based on GAN." [Online]. Available: https://arxiv.org/abs/1702.05983

[31] W. Xu, Y. Qi, and D. Evans, "Automatically evading classifiers," in *Proc. Netw. Distrib. Syst. Symp.*, 2016, pp. 21–24.

[32] H. Hasselt, A. Guez, and D. Silver. (2015). "Deep reinforcement learning with double Q-learning." [Online]. Available: https://arxiv.org/abs/1509.06461

[33] G. Dulac-Arnold *et al.* (2016). "Deep reinforcement learning in large discrete action spaces." [Online]. Available: https://arxiv.org/abs/1512.07679

[34] H. S. Anderson, A. Kharkar, B. Filar, D. Evans, and P. Roth. (2018). "Learning to evade static PE machine learning malware models via reinforcement learning." [Online]. Available: https://arxiv.org/abs/1801.08917

[35] S. Ioffe and C. Szegedy. (2015). "Batch normalization: Accelerating deep network training by reducing internal covariate shift." [Online]. Available: https://arxiv.org/abs/1502.03167

[36] Quarkslab. (2018). *LIEF: Library for Instrumenting Executable Files*. [Online]. Available: https://lief.quarkslab.com

[37] *OpenAI Environment CarRacing-v0*. Accessed: Aug. 31, 2018. [Online]. Available: https://gym.openai.com/envs/CarRacing-v0/

[38] C. Guarnieri, A. Tanasi, J. Bremer, and M. Schloesser. *Cuckoo Sand-Box: A Malware Analysis System*. Accessed: Sep. 15, 2018. [Online]. Available: http://www.cuckoosandbox.org/

[39] J. Bai and J. Wang, "Improving malware detection using multi-view ensemble learning," *Secur. Commun. Netw.*, vol. 9, no. 17, pp. 4227–4241, 2016.

[40] E. Levin, R. Pieraccini, and W. Eckert, "A stochastic model of human-machine interaction for learning dialog strategies," *IEEE Trans. Speech Audio Process.*, vol. 8, no. 1, pp. 11–23, Jan. 2000.

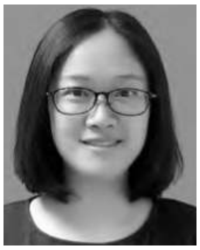
ZHIYANG FANG received the M.S. degree in computer science and technology from Sichuan University, Chengdu, Sichuan, China, in 2009, where he is currently pursuing the Ph.D. degree in computer science and technology. He is currently involved in research work on information security. His research interest includes software security.

JUNFENG WANG received the M.S. degree in computer application technology from the Chongqing University of Posts and Telecommunications, Chongqing, in 2001, and the Ph.D. degree in computer science from the University of Electronic Science and Technology of China, Chengdu, in 2004. From 2004 to 2006, he held a postdoctoral position with the Institute of Software, Chinese Academy of Sciences. Since 2006, he has been with the College of Computer Science and the School of Aeronautics and Astronautics, Sichuan University, as a Professor. His recent research interests include network and information security, spatial information networks, and data mining. He is currently serving as an Associate Editor of the IEEE ACCESS, the IEEE INTERNET OF THINGS, and *Security and Communication Networks*.

BOYA LI received the B.E. degree in computer science and technology from Sichuan University, China. Her research interest is software security.

SIQI WU received the bachelor's degree in computer science and technology from Sichuan University, Chengdu, Sichuan, China, in 2017, where she is currently pursuing the M.S. degree in computer science and technology. She is currently involved in research work on malware detection and attacking malware detection. Her research interests include software security and network traffic analysis.

YINGJIE ZHOU received the Ph.D. degree from the School of Communication and Information Engineering, University of Electronic Science and Technology of China (UESTC), China, in 2013. He was a Visiting Scholar with the Department of Electrical Engineering, Columbia University, New York. He is currently an Assistant Professor with the College of Computer Science, Sichuan University (SCU), China. His current research interests include network management, behavioral data analysis, and resource allocation.



HAIYING HUANG received the master's degree in biomedical engineering from the University of Electronic Science and Technology of China, in 2008. She is currently pursuing the Ph.D. degree with College of Computer Science, Sichuan University, Chengdu, China. She joined the Information Management Department of West China Second University Hospital, Sichuan University. Her research interests include software security and neural network in medical big data.