A Method of Mutating Windows Malwares using Reinforcement Learning with Functionality Preservation

Do Thi Thu Hien
hiendtt@uit.edu.vn
Information Security Laboratory, University of Information Technology Vietnam National University, Ho Chi Minh city
Ho Chi Minh city, Vietnam

Phan The Duy
duypt@uit.edu.vn
Information Security Laboratory, University of Information Technology Vietnam National University, Ho Chi Minh city
Ho Chi Minh city, Vietnam

Le Anh Hao
18520047@gm.uit.edu.vn
Information Security Laboratory, University of Information Technology Vietnam National University, Ho Chi Minh city
Ho Chi Minh city, Vietnam

Nguyen Duy Lan
18520976@gm.uit.edu.vn
Information Security Laboratory, University of Information Technology Vietnam National University, Ho Chi Minh city
Ho Chi Minh city, Vietnam

Nghi Hoang Khoa
khoanh@uit.edu.vn
Information Security Laboratory, University of Information Technology Vietnam National University, Ho Chi Minh city
Ho Chi Minh city, Vietnam

Van-Hau Pham
haupv@uit.edu.vn
Information Security Laboratory, University of Information Technology Vietnam National University, Ho Chi Minh city
Ho Chi Minh city, Vietnam

ABSTRACT

Recently, the development in both the quantity and complication of malware has raised a need for powerful malware detection solutions. The outstanding characteristics of machine learning (ML) and deep learning (DL) techniques have been leveraged in the fight against malware. However, they are proven to be vulnerable to adversarial attacks, where intended modifications in malware can flip the detection result and then evade the detector's eyes. This research area is being focused on and deeply interested in many publications due to its significance in the evaluation of malware detection approaches. In such works, using Generative Adversarial Networks (GANs) or Reinforcement Learning (RL) can help malware authors craft metamorphic malware against antivirus. Unfortunately, the functionality of created malware is not mentioned and verified during the mutation phase, which can result in evasive but useless malware mutants. In this paper, we focus on Windows Portable Executable malware and propose an RL-based malware mutant creation approach to fool black-box static ML/DL-based detectors. Specifically, we introduce a validator to confirm functionality preservation, which is one of our requirements for a successfully created malware. The experiment results prove the effectiveness of our solution in crafting elusive and executable Windows malware mutants.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**;

KEYWORDS

Malware, malware mutant, reinforcement learning

ACM Reference Format:
Do Thi Thu Hien, Phan The Duy, Le Anh Hao, Nguyen Duy Lan, Nghi Hoang Khoa, and Van-Hau Pham. 2022. A Method of Mutating Windows Malwares using Reinforcement Learning with Functionality Preservation. In *The 11th International Symposium on Information and Communication Technology (SoICT 2022), December 1–3, 2022, Hanoi, Vietnam.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3568562.3568631

SoICT 2022, December 1–3, 2022, Hanoi, Vietnam

ACM ISBN 978-1-4503-9725-4/22/12…$15.00
https://doi.org/10.1145/3568562.3568631

1 INTRODUCTION

Malware has become one of the most popular security threats, which can be any **mal**icious soft**ware** or pieces of software generated based on the harmful intent of an attacker. This threat type can range from viruses, worms, and Trojans to spyware, rootkits, etc. Once disseminated, they can impact numerous computing devices and infiltrate the secrecy, integrity, and availability of a compromised system. This can lead to not only emotional affect but also an economic threat with real financial losses. Along with the development of technologies, malware has experienced exponential growth in number [19][21]. Moreover, they become more and more sophisticated to retain their functionality without being detected by security protectors or defenders [25]. Besides, as Windows is the most used operating system on devices compared to other counterparts, there is numerous malware targeting this OS and its Portable Executable (PE) files [2]. Hence, there is a need for effective malware detectors to enhance the security of systems against this threat.

In the fight against malware, many works, as well as industrial products, utilize the traditional signature-based detection approach. In such a method, each malware is identified by a unique sequence of bytes that is figured out through the malware analysis process. However, the shortage of those solutions is based on the knowledge of malware, which is not available in the case of unknown malware. Besides, those signatures have to be updated continuously to keep

up with the development of malware variants, or the detection will fail. Hence, more cutting-edge technologies have been applied in the field of malware detection to improve the protection capability, one of which is machine learning/deep learning. The effectiveness of ML and DL have been proved in many malware detection solutions [12][5][24]. Unfortunately, ML and deep learning models are recently discovered to be vulnerable to adversarial attacks [17][1][16], so do their applications. Hence, adversarial malware samples, which are created by adding unrecognizable perturbations into the original one and can be considered malware mutants, can also evade the ML or DL-based malware detector. When creating adversarial malware, the original structure cannot be destroyed and the original functions of malware need to be preserved for valid malware. Though several works have achieved the goal of fooling ML/DL-based malware detectors by various approaches, such as using GAN [13] or reinforcement learning (RL) [4][7][9], none of them consider whether the functionality of those samples remains.

Those above issues have become the motivation of our work, in which we attempt to propose an RL-based malware mutant generation framework equipped with the capability of functionality validation. In the scope of this paper, we focus on malware targeting Windows devices, which are in the form of PE files and whose created mutant aims to evade ML/DL-based detectors which use the static analysis approach.

The remaining portions of this paper are organized as follows. In **Section 2**, we mention some previous research works on malware and RL. The following **Section 3** describes the workflow of our framework in detail. Later, this solution is brought to the real-life in implementation mentioned in **Section 4**. The experimental settings and performance evaluation outcomes are shown in **Section 5**. Finally, **Section 6** includes our main conclusion and future work.

2 BACKGROUND AND RELATED WORKS

2.1 Background

2.1.1 Portable Executable format. Portable Executable (PE) [2] is the generic format of executable files on both 32-bit and 64-bit Windows devices. This format consists of data chunks, which are divided into two primary parts of Header and Section. **Figure 1** describes the content of a PE file.

The Header part, which indeed contains multiple headers, provides overview information for the OS to map the file into memory for execution. The first header is *DOS header* used for backward compatibility with the Microsoft Disk Operating System. The following header is *PE header* with useful information describing the rest of the file. Besides a signature for the file to be recognized as a PE file, this header provides information such as the number of sections, creation time, the size of the following optional header, etc. An *PE Optional Header* contains most of the meaningful information about the file, including the size of the file or code, address of the entry point, initial stack size, section alignment information, and so on. Then, a *Section table* immediately follows the optional header to provide the name, size, offset, and other information of all sections present in PE files.

The main content of the PE file is placed in sections to provide a logical and physical separation of the different parts of a program. Each section holds a specific piece of information, such as code, data, or resources. The number of sections in a PE is flexible and mentioned in the PE header, but some common sections are usually found in PE files. For example, *.text* section contains the executable code, while *.data* or *.rdata* are sections for data to be used in operations of the file.

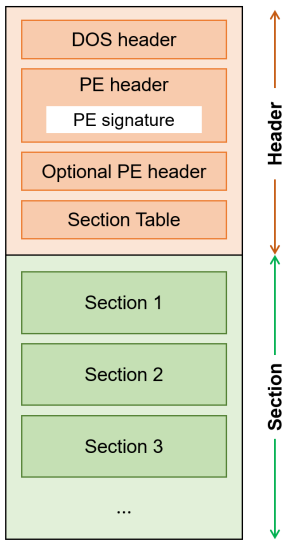

Figure 1: PE format for executable files on Windows.

2.1.2 Malware detection. Malware detection is a process of analyzing PE files to recognize the sign of malicious properties. The sooner the malware is figured out, the less harm the system may suffer. There are 3 main approaches to detecting malware based on the interaction with it [8]. First is static analysis techniques, in which we gather useful information from the sample without running it [8]. This can be achieved by interpreting the file with tools to view its general information such as sections, strings, byte sequence, used functions, etc. Meanwhile, dynamic analysis approaches try to execute the file to figure out maliciousness via its behaviors [1][8]. In this case, a dedicated environment is in need to run and monitor events during the execution. Moreover, a hybrid malware detection approach that analyzes in both static and dynamic manner is also promising [8].

Recently, the rapid development of ML/DL has led to its application in various fields, including information security. In the aspect of malware detection, these cutting-edge techniques can become promising solutions to deal with zero-day attacks/malware, which may be the weakness of the above malware detection approaches. Many researchers have made steps in the journey of adopting ML/DL to their solutions to prevent malware [12][24].

2.1.3 Reinforcement Learning. Reinforcement Learning is a branch of machine learning, where *agents* perform *actions* to interact with an *environment*. *Agents* then get the corresponding *reward* based on the *state* of that environment. After the training stage of many *episodes*, the *agent* should be able to choose the best actions to retrieve the highest reward.

RL model is defined based on Markov Decision Process as a tuple of four elements (S, A, P, R).

- S: A set of states, each of which is the state of the environment at a time.
- A: Action space, which consists of available actions for the agent to take on the environment.
- $P(s_t, a_t, s_{t+1})$: State transition function, describing how the state change when performing an action in the environment with a given state. This indeed provides the probability of getting state s_{t+1} after taking action a_t in state s_t based on a probability distribution over a set of possible transitions.
- $R(s_t, a_t)$: the function to create reward for agent after transition from state s to state s_{t+1} with action a_t.

Besides that, many RL models also define an additional element called *discount factor* γ used to control the importance of future rewards.

In each time step (*turn*) t, *Agent* receives a state s_t from the environment and use its policy $\pi(a_t|s_{t+1})$ to determine the *action* $a_t \in A$ to take. Then, the environment transit to state s_{t+1} after performing action a_t. A corresponding *reward* r_t is also calculated and returned to the *agent*. *Reward* r_t and *state* s_{t+1} then play as the input of *agent* in the next time step. The goal of this learning model is to maximize *reward* by deriving an optimal policy π for *action* selection.

In fact, there are two approaches to achieving the optimal policy π in a given state, including state-value and action-value functions. Among them, the action-value function, also known as Q-function, is considered to be simpler [7]. It indicates how good it is to take action a_t in state s_t under policy π, as in **Eq. (1)**.

$$Q_\pi(s_t, a_t) = \mathbb{E}_\pi[\sum_{i=0}^{\infty} \gamma^i R_{t+i+1} | s_t, a_t] \tag{1}$$

The optimal policy has an optimal Q-function, which is defined as Q_π^* as in **Eq. (2)**.

$$Q_\pi^*(s_t, a_t) = [R_t + \gamma \max_{a' \in A} Q(s_{t+1}, a_{t+1})] \tag{2}$$

Instead of the traditional way of tracking Q-values along with state-action pair in a 2D table, a deep neural network with the weight of θ has been introduced [18], which is trained to approximate the values and minimize its lost function $L(\theta)$.

2.2 Related works

Various researchers have made their force in proposing methods to generate PE malware mutants that can evade malware detectors.

Research by Hu and Tan introduced a GAN-based system to create adversarial malware samples, called MalGAN [13]. They use a list of API calls to add to the original malware sample to create its mutant that can bypass black-box malware detectors. Later, another work by Kawai et al. [14] tries to improve the proposed MalGAN to achieve better performance. For instance, multiple API call lists are used during the training of Improved-MalGAN and black-box detectors rather than a single list as in the original MalGAN.

Meanwhile, a research group by Anderson has made the first steps in applying Reinforcement Learning (RL) to the malware mutant creation field. For instance, they introduce Gym-malware [4] to provide a malware manipulation model. The defined set of 10 actions is aimed to result in no effect on the functionality of the original malware. The learning process finishes either when all 10 actions are added or created malware successfully evades detectors. They conclude that 24/100 generated malware can bypass malware detectors, while the undetected rate of 200 created malware is 16.25%. Besides, other works can utilize the library as means to enforce and motivate RL-based solutions for malware detection. Although the authors claimed that the functionality of the malware is unaffected by actions in their publications, there is a lack of validation methods to confirm this conclusion.

In the same approach, Fang et al. also suggested DQEAF [9], which is a system to create adversarial malware samples using RL. The idea of the author is to reduce the input size to achieve better performance in sample creation. In fact, their action space only consists of 4 functionality-preserving actions, and the dimension of features is low. The agent selects the optimal sequence of actions to modify the malware samples. Thus, they can bypass the detection engines. The proposed solution is proven to be effective with an evasion success rate of more than 70% for backdoor samples and higher better performance in evading detectors compared to Gym-malware. However, a functionality validation for created malware is also not mentioned. Moreover, a few actions repeatedly added 80 times into malware can become a signature for detectors to recognize malware created by DQEAF.

Another work called AIMED-RL [7] inherits and improves the model of Gym-malware by making more consideration in malware functionality preserving. In specific, AIMED-RL evaluates the similarity of adversarial malware samples and original ones and uses this metric as a sign that the functions have remained. The higher similarity that adversarial samples can achieve, the higher reward the agent will receive. Hence, agents in their models are encouraged to select suitable actions without significant effects on functionality.

There is a lack of functionality validation methods for generated malware samples in the abovementioned publications, especially in RL-based works. Hence, it is the goal of our paper to propose an RL-based model for malware mutant creation with functionality preservation and validation.

3 METHODOLOGY

In this section, our proposed RL model for malware mutant creation is described in detail. The goal of this model is to generate malware samples that can evade malware detectors.

3.1 Assumption

Depending on the level of model knowledge that the attacker has, the attacks against ML-based detectors can be categorized as black-box, gray-box, or white-box approaches. Attacks where attackers have full access to the information of the model, including algorithms, hyperparameters, and training data, are referred to as white-box attacks. A gray-box attack, on the other hand, just gets in hand a small amount of this information, such as feature sets or labels. In a black-box attack, the attackers try to attack the target model without any prior knowledge.

The RL-based model in our work is created intending to combat black-box malware detectors. In the context of this attack, some presumptions must be noted. Firstly, the attackers are unaware of the algorithm, hyperparameters, and training data of the target detector. Second, the attack is carried out in the prediction phase

Table 1: Actions in Action space of proposed RL-based model

Action	Description
imports_append	Add functions into Import table
section_rename	Change the name of sections
section_add	Create new sections
section_append	Add junk bytes into the end of sections
create_new_entry	Create an entry point to change the execution flow
remove_signature	Remove file signature
remove_dedug	Modify debug information
upx_pack/ upx_unpack	Pack or unpack file (using upx tool)
break_optional _header_checksum	Modify checksum information
overlay_append	Add junk bytes into the end of file

after model training. Next, a produced malware mutant must be both executable and able to trick the detector to be considered acceptable.

3.2 Proposed malware mutating model

The overall architecture of the proposed RL-based malware mutant creation model is depicted in **Figure 2**.

3.2.1 Reinforcement Learning model. Reinforcement Learning in the scope of this paper is defined as follows.

Action space. Being inherited from research of Anderson et al. [4], 10 actions mentioned in **Table 1** make up the action space in our RL model.

Environment. Each malware file is analyzed to extract useful information by *Feature Extractor* in the form of a feature vector. Those feature vectors are considered as *state* of the environment.

Table 2: Features of sample in feature vectors

Feature groups	Features
PE file-related features	Metadata, Section metadata, Import table, Export table, Strings, General information
Byte distribution features	Byte histogram, 2D byte-entropy histogram

All features used in our models are summarized in **Table 2**, make up vectors of 2351 dimensions. The state of the environment is changed upon the action taken on a PE malware.

The *environment* also contains the functionality test and malware detection, which results in a corresponding *reward* for *agents* based on their actions and state changes using **Eq. (3)**. The reward depends on the result of the malware detector via R_{det}, the execution capability of the modified malware with R_{func}, and the number of performed actions reflected in R_{dis}.

$$R = R_{det} * \omega_{det} + R_{dis} * \omega_{dis} + R_{func} * \omega_{func} \quad (3)$$

For more specifics, based on the output labels of the black-box detector for created malware, R_{det} is set to 10 for benign results due to its capability to evade the detector; otherwise, it takes the value of 0. The same assignment is also performed in R_{func} with 10 for successful functionality test on the virtual machine and 0 in case of broken malware. Besides, R_{dis} is calculated by **Eq. (4)** with the number of performed actions t and the expected maximum action t_{max}. Note that, this action threshold can be customized. Moreover, each element making up the reward has the corresponding weight ω, which can also be flexibly modified due to various requirements.

$$R_{dis} = \frac{R_{max}}{t_{max}} * t \quad (4)$$

In addition to being rewarded, *agents* with bad behaviors should also be punished. In particular, repeatedly choosing a same action

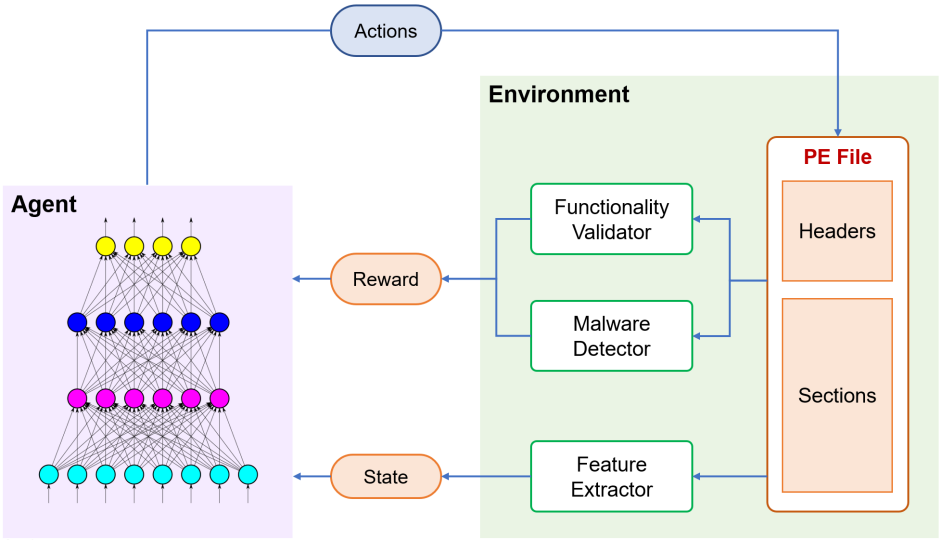

Figure 2: Proposed RL-based malware mutant creation model.

for p times to take on *environment* should cause some decrease in the returned reward [7] according to **Eq. (5)**.

$$R = \begin{cases} R & \text{if } p = 0 \\ R * 0.8 & \text{if } p = 1 \\ R * 0.6 & \text{if } p \geq 1 \end{cases} \quad (5)$$

Agent. In our RL model, *agents* are built with DiDQN [6] algorithm, Adam optimizer and Noisy Nets explorer [10]. While DiDQN helps the model focus on reward distribution rather than reward value, Noisy Nets is proven to have high performance in action space exploring and diversifying modifications on feature vectors.

3.2.2 Black-box Malware Detector. The black-box malware detector in our framework represents DL/ML-based ones that can be deployed within a system in the real world. Specifics, they are DL/ML algorithms trained with datasets for malware detection and have a certain capability to protect the system against malware. In attacks targeting those black-box detectors, only their prediction results on a malware sample can be obtained without any knowledge of the underlying architecture or parameters.

3.2.3 Functionality Validator. To validate the functionality of a Windows malware sample, we utilize a Windows-based virtual machine as an environment to load and run it. Note that in the scope of this paper, only the functionality which means executability is examined, not specific malicious functions. A malware sample that can start in our virtual machine is considered valid.

3.3 Workflow of the proposed model

The training process of our RL model is presented in **Algorithm 1** which contains the following steps.

(1) The environment randomly selects a malware sample and set a limitation on the number of actions to use.
(2) Features are extracted from the malware sample to make up a feature vector.
(3) The agent then receives this feature vector to decide an action to perform.
(4) Once the action is taken on the sample, the environment converts the modified feature vector into the form of a PE file. The purpose of this step is to ensure that later analysis can be performed on the actual customized malware file, not its represented features.
(5) The created binary file is forwarded to the malware detector.
(6) The detection result of a benign sample may prove that the created malware can evade the malware detector and then is sent to the functionality validator. Otherwise, it is considered as not good enough, and it comes to step (8).
(7) The functionality validator receives evasive malware samples and tries to execute them on the virtual machine. If the malware sample can run properly, it is confirmed to be a successful adversarial malware or malware mutant. Or else, malware is marked as a failure.
(8) The corresponding reward is calculated based on the action and the state, which is the feature vector.
(9) The reward, as well as the new feature vector, are sent back to the agent to continue the training from Step (3).

Algorithm 1 The training process of RL-based malware mutant creation model.

Input: M_{max}: The number of experience categories;
$EPISODES$: The total number of malware to load;
MAX_TURN: The maximum number of actions to use on a sample;
A: Action space;
ϵ: noise parameter used in Adam optimizer;
$batchSize$: batch size
Output: Trained RL model.

Initialize Memory M with size M_{max};
Initialize Network Q with weight θ;
for $episode = 1$ to $EPISODES$ **do**
  Select a binary $bin_{original}$ randomly from malware folder;
  Extract binary features $s_{original} = ExtractFeatures(bin_{original})$;
  for $t = 1$ to MAX_TURNS **do**
    With noise parameter ϵ, select an action a_t;
    Modify bin_t by action a_t to make bin_{t+1} and $s_{t+1} = ExtractFeatures(bin_{t+1})$, observe reward r_{t+1};
    Store transition $(s_t, a_t, r_{t+1}, s_{t+1})$;
    if $sizeof(M) > batchSize$ **then**
      for $i = 1$ to K **do**
        Sample transition $X_i P(X_i) = p_{x_i} / \sum jp_j$;
        Calculate $CategoricalDQN(X_i)$ by **Algorithm 2**;
      end for
      Exert Adam optimizer to optimize parameter θ;
    end if
  end for
end for

Algorithm 2 CategoricalDQN in DiDQN.

Input: A transition $X_t = (x_t, a_t, r_t, x_{t+1})$;
$\gamma \in [0, 1]$: discount factor;
N: the number of output neuron;
Output:

$V_{max} = -V_{min} = 10$;
$Q(x_{t+1}, a) := \sum z_i p_i(x_{t+1}, a)$;
$a^* \leftarrow argmax_a Q(x_{t+1}, a)$;
$m_i = 0, i \in 0, ..., N-1$;
for $j \in 0, ..., N-1$ **do**
  $\hat{T}z_j \leftarrow [r_t + \gamma_t z_j]_{V_{min}}^{V_{max}}$;
  $b_j \leftarrow (\hat{T}z_j - V_{min})/\delta_z \ \{b_j \in [0, N-1]\}$;
  $l \leftarrow \lfloor b_j \rfloor, u \leftarrow \lceil b_j \rceil$;
  $m_l \leftarrow m_l + p_i(x_{t+1}, a^*)(u - b_j)$;
  $m_u \leftarrow m_u + p_j(x_{t+1}, a^*)(b_j - l)$;
end for
return $-\sum_i m_i \log p_i(x_t, a_t)$

4 IMPLEMENTATION

In this section, the implementation of our proposed RL-based malware mutant creation model is described. An Ubuntu 20.04 virtual machine with 16G RAM and 100 GB storage with Python 3.7.13 installed is used to deploy it.

4.1 Dataset

To train and evaluate our model, we use the EMBER [3] and Virus Totals datasets.

Firstly, our malware detectors are trained with the EMBER dataset, which includes malware features extracted by the LIEF library of Quarkslab [22]. There are 1.1 million binaries, with 400,000 malware, 400,000 benign files, and other unlabeled samples. Each binary is represented in a set of 189 features.

Secondly, in training the RL model, we use a malware dataset consisting of Windows 32-bit 16,068 binaries. While benign samples are gathered from Windows 7, 8, and 10 machines, malware ones are retrieved from VirusTotal where they are labeled as malicious by at least half of the detectors. Testing on a different dataset from the one used in detector training can ensure the black-box attack, where attackers are also unaware of the training dataset.

4.2 Malware detector

To demonstrate the black-box attack context, we deploy different types of DL/ML-based malware detectors. This allows us to evaluate the effectiveness of our proposed model on various detectors rather than a specific one. Three static malware detectors are considered in our works, including LGBM [15], MalConv [23] and Random Forest. The outputs of those models are the possibility to be malware of a given sample, in the form of a confidence score. The higher this score is, the more malicious the sample is. We can use this ratio to map the sample to the corresponding label of attack or benign. For each above model, we define a threshold for its confidence score to distinguish malware and benign samples, as in **Table 3**. There is a difference in threshold between MalConv and two other detectors due to its low false positive rate in previous work [23].

4.3 Functionality Validator

To implement a virtual machine for functionality validator, a Windows 7 machine is deployed, most malware used in our work has been confirmed to be executable in this environment. This machine is created and managed by our proposed model using Virtual Box and its VBoxManage [20], which allows restarting and running samples via commands.

4.4 Malware mutant generator

Our RL model is built with the support of chainerrl library [11], which provides various RL algorithms and related deep learning mechanisms. In this work, DiDQN is used in our *agents*. **Table 4** describes parameters of our model in implementation. Besides some parameters inherited from previous research, such as γ, the weight ω is set to 0.33 for all elements for the assumption of their equal effect on the reward.

Moreover, besides supporting feature extracting, the LIEF library of Quarkslab [22] is also utilized to modify malware samples to create mutants. Its provided APIs allow customizing a given binary, and hence can be used to perform *actions* determined by *agents* during the training process.

Table 3: Threshold of Confidence score for 3 ML-based detectors

Detector	Threshold
LGBM	0.9
MalConv	0.7
RandomForest	0.9

Table 4: Parameters of RL-based model

Parameter	Value	Description
EPISODES	1,000	The total number of loaded malware files into environment.
MAX_TURNS	10	Maximum number of actions to perform on a sample
ϵ	1e-2	Noise parameter for Adam optimizer
γ	0.95	Discount factor for reward
$\omega_{det}, \omega_{dis}, \omega_{func}$	0.33	Weights of elements of reward
M_{max}	1,000	Number of experience categories in DQN model
Training dataset size	3,700	Number of samples in training dataset
Testing dataset size	300	Number of samples in testing dataset

5 EXPERIMENTS AND EVALUATION

5.1 Metrics

Before mutating malware samples against ML-based detectors, the metrics of Accuracy and AUC (Area Under the ROC Curve) to ensure the high performance of anti-malware on malware detection. To analyze the evasive capability of malware mutants created by the RL-based approach, the Evasion Rate (ER) and Evasive Rate with Functionality Preservation (ERFP). Given M malware samples for evasive capability testing, they are computed by **Eq. (6)** and **Eq. (7)**.

$$ER = \frac{e}{M} \tag{6}$$

where e is the number of evasive malware samples.

$$ERFP = \frac{ep}{M} \tag{7}$$

where ep is the number of evasive malware with functionality preservation on the virtual machine.

5.2 Experiment results

5.2.1 Robustness of black-box malware detectors. Before applying our proposed model, black-box malware detectors are trained on EMBER dataset and then tested its detection capability on VirusTotal dataset. As we can see in the results summarized in **Table 5**, all DL/ML-based malware detectors achieve good performance in recognizing malware, with accuracies of more than 93%. The results in the AUC metric also show the high True positive (TP) rate and low false positive (FP) rate of those models. Clearly, LGBM is the best model among those when it can label correctly 96.5% of test samples and achieve the AUC of 98.5%.

Table 5: Performance of 3 malware detectors on original VirusTotal dataset

Detector	Accuracy	AUC
LGBM	96.51%	98.5%
MalConv	93.74%	97.92%
RandomForest	95.81%	92.86%

5.2.2 Evasion performance of RL-based model without functionality validation. In this scenario, malware mutants are created by our RL-based model without functionality validation. The ER values in **Table 6** are observed evasion rates of these mutants on 3 malware detectors. As we can see, significant evasion rates of approximately 40% can be noticed in all cases of detectors, which means that nearly half of created mutants can fool those detectors. Besides, MalConv is the worst candidate in the fight with malware samples generated from our model, while Random Forest shows the most stable detection capability.

Moreover, a case of a random agent is also implemented to prove the effectiveness of our DiDQN agent. Instead of being trained, this random agent selects one of the 10 possible actions at random with equal probability for all of them. Clearly, the evasion rate of randomly created malware is much lower than using trained agents.

5.2.3 Evasion performance of RL-based model with functionality validation. When applying functionality validation on created malware samples, only executable ones are considered in the evasion attack against malware detectors. **Table 6** presents the results of this case in ERFP in two last columns. Obviously, the evasion rate of this scenario experience drops compared to results in the previous case due to some unworkable malware mutants. The highest performance is still in the case of fooling MalConv detector with an evasion rate of 18.6% and Random Forest also remains its stability. Moreover, the capability of the random agent in creating evasive malware samples appears to be unaffected by the functionality validation.

5.2.4 Variety of used actions on malware samples. A statistic summary of used actions is also given in **Figure 3**. *upx_pack* is the most used action in our model to create malware mutants, which meet the same usage trend in previous works [4][7]. In fact, packing malware samples causes significant changes in structure as well as its feature vector, this technique has been applied in malware creation to prevent being detected. This action appears in most of created evasive and executable mutants.

Table 6: Evasion rate (ER) and Evasion rate with functionality preservation (ERFP) of created malware mutants on 3 malware detectors

Detector	Without functionality validation (ER)		With functionality validation (ERFP)	
	DiDQN Agent	Random Agent	DiDQN Agent	Random Agent
LGBM	42.7%	6.8%	14.6%	5.6%
MalConv	43.8%	7.8%	18.6%	7.8%
RandomForest	33.4%	5.4%	12.3%	6.7%

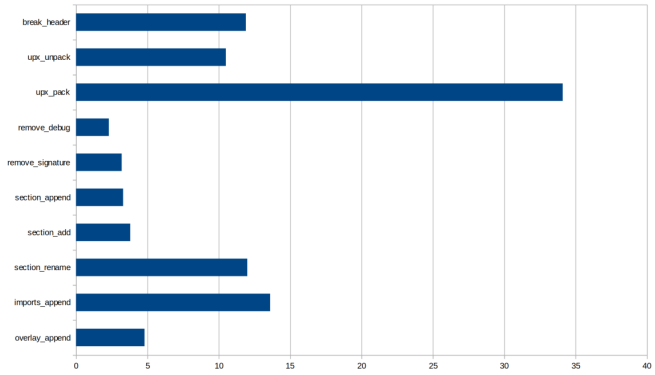

Figure 3: Usage rate (%) of actions in mutant creation.

Besides, we consider the effect of actions on the functionality of malware samples. Through our experiments of 1000 training episodes with LGBM and MalConv, 98% of created mutants become unworkable after *section_add* action performed by the agent. As a result, the trained agent attempts to minimize the usage of this action and use *upx_pack* instead.

5.2.5 Comparison of our proposed model and other works on evasion performance. In this scenario, we have a comparison between related works and our model on their evasion capability against malware detectors. The evasion rate is measured in the case of using LGBM detector with a threshold of 0.9 to recognize malware mutants, which is shown in **Table 7**.

As we can see, according to those results, our model is not the best to produce the most evasive malware mutants compared to its counterparts. However, its result can be considered reasonable while remaining a certain variety in available action space and a suitable number of turns to perform actions on samples. More specifically, although the work of Fang et al. [9] achieved the best evasion rate, small action space reduces malware diversity. Moreover, their large *MAX_TURN* causes repeated actions on malware samples, which may result in a recognizable signature for mutants from their approach. Besides the evasion rate, one of our main contributions is functionality preservation, presented in the executable evasion rate. Though this value is much lower than the evasion rate, it reflects mutants that are actually executable and not useless.

Moreover, to have a better comparison of our model and DQEAF of Fang, we implement experiments with various *MAX_TURN* values. The ER and ERFP on LGBM detector are observed and summarized in **Table 8**. Clearly, with the same max turn of 80, our proposed model has beaten DQEAF with an evasion rate of 47.57%. However, the increasing max turn causes a significant climb up in training time. Furthermore, though the evasion performance gets better through more turns, the ERFP experiences a drop in cases of more than 20 turns. This means too many actions can destroy the sample and its functionality, which does not meet our requirements. Hence, it is noticeable that performing more actions on malware samples is not always more effective.

Table 7: Evasion performance approaches comparison

Approach	Action space	Max turn	Evasion rate (ER)	Executable Evasion rate (ERFP)
Anderson et al. [3]	11	10	16.25%	-
Fang et al. [9]	4	80	46.56%	-
Castro et al. [7]	10	5	42.13%	-
Our approach	10	10	42.70%	14.65%

Table 8: The effect of Max turn on the evasion performance of created mutants on LGBM detector.

Action space	Max turn	Evasion rate (ER)	Executable Evasion rate (ERFP)	Training time (min)
10	10	42.70%	14.65%	330
10	20	42.84%	14.72%	612
10	40	44.23%	14.59%	1187
10	80	47.57%	13.18%	2136

6 CONCLUSION

In the presence of adversaries, the robustness of ML-based malware detectors is one of the major concerns due to the evolution of evasive malware. This problem encourages us to investigate the method of generating real malware variants to escape the eyes of antivirus. Our paper conducts experimental research on leveraging deep reinforcement learning (RL) for mutating an original malware to be classified as the normal one by various ML-based malware detectors. Through experimental results on modifying real malware samples collected from VirusTotal source, we conclude that the leverage of functionality preservation of malware samples during the RL agent training can craft new executable mutants of malware efficiently. Finally, this work encourages future research projects to focus on the ML-based malware evolution in the era of Artificial Intelligence.

ACKNOWLEDGMENTS

This research is funded by University of Information Technology - Vietnam National University HoChiMinh City under grant number of D1-2022-49.

REFERENCES

[1] Amir Afianian, Salman Niksefat, Babak Sadeghiyan, and David Baptiste. 2019. Malware Dynamic Analysis Evasion Techniques: A Survey. *ACM Comput. Surv.* 52 (2019).
[2] Ange Albertini. 2012. PE 101 - a windows executable walkthrough. https://github.com/corkami/pics/blob/master/binary/pe101/README.md.
[3] Hyrum Anderson and Phil Roth. 2018. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. (04 2018).
[4] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. *arXiv preprint arXiv:1801.08917* (2018).
[5] Ömer Aslan Aslan and Refik Samet. 2020. A Comprehensive Review on Malware Detection Approaches. *IEEE Access* 8 (2020), 6249–6271. https://doi.org/10.1109/ACCESS.2019.2963724
[6] Marc Bellemare, Will Dabney, and Remi Munos. 2017. A Distributional Perspective on Reinforcement Learning. (2017).
[7] Raphael Labaca Castro, Sebastian Franz, and Gabi Dreo Rodosek. 2021. AIMED-RL: Exploring Adversarial Malware Examples with Reinforcement Learning. In *Machine Learning and Knowledge Discovery in Databases. Applied Data Science Track.*
[8] Anusha Damodaran, Fabio Di Troia, Corrado Aaron Visaggio, Thomas H. Austin, and Mark Stamp. 2017. A comparison of static, dynamic, and hybrid analysis for malware detection. *Journal of Computer Virology and Hacking Techniques* (2017).
[9] Zhiyang Fang, Junfeng Wang, Boya Li, Siqi Wu, Yingjie Zhou, and Haiying Huang. 2019. Evading anti-malware engines with deep reinforcement learning. *IEEE Access* (2019).
[10] Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Matteo Hessel, Ian Osband, Alex Graves, Volodymyr Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. 2018. Noisy Networks For Exploration. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rywHCPkAW
[11] Yasuhiro Fujita, Toshiki Kataoka, Prabhat Nagarajan, and Takahiro Ishikawa. 2019. ChainerRL: A Deep Reinforcement Learning Library. *Journal of Machine Learning Research* (2019).
[12] Daniel Gibert, Carles Mateu, and Jordi Planes. 2020. The rise of machine learning for detection and classification of malware: Research developments, trends and challenges. *Journal of Network and Computer Applications* (2020).
[13] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on GAN. arXiv:1702.05983.
[14] Masataka Kawai, Kaoru Ota, and Mianxing Dong. 2019. Improved MalGAN: Avoiding Malware Detector by Leaning Cleanware Features. In *International Conference on Artificial Intelligence in Information and Communication (ICAIIC).*
[15] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In *Advances in Neural Information Processing Systems.*
[16] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. In *26th European Signal Processing Conference (EUSIPCO).*
[17] Deqiang Li, Qianmu Li, Yanfang (Fanny) Ye, and Shouhuai Xu. 2021. Arms Race in Adversarial Malware Detection: A Survey. *ACM Comput. Surv.* 55 (2021).
[18] Sajad Mousavi, Michael Schukat, and Enda Howley. 2018. Deep Reinforcement Learning: An Overview. *Lecture Notes in Networks and Systems.*
[19] Ori Or-Meir, Nir Nissim, Yuval Elovici, and Lior Rokach. 2019. Dynamic Malware Analysis in the Modern Era—A State of the Art Survey. *ACM Comput. Surv.* 52 (2019).
[20] Oracle. 2020. VBoxManage. https://docs.oracle.com/en/virtualization/virtualbox/6.0/user/vboxmanage-intro.html.
[21] Harun Oz, Ahmet Aris, Albert Levi, and A. Selcuk Uluagac. 2022. A Survey on Ransomware: Evolution, Taxonomy, and Defense Solutions. *Comput. Surveys* (2022).
[22] Quarkslab. 2017. LIEF - Library to instrument executable formats. https://github.com/lief-project.
[23] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. 2017. Malware Detection by Eating a Whole EXE. https://doi.org/10.48550/arXiv.1710.09435
[24] Jagsir Singh and Jaswinder Singh. 2021. A survey on machine learning-based malware detection in executable files. *Journal of Systems Architecture* 112 (2021).
[25] Ilsun You and Kangbin Yim. 2010. Malware Obfuscation Techniques: A Brief Survey. *Proceedings - 2010 International Conference on Broadband, Wireless Computing Communication and Applications, BWCCA 2010.*