Generation of Adversarial Malware and Benign Examples Using Reinforcement Learning

Matouš Kozák, Martin Jureček, and Róbert Lórencz

Abstract Machine learning is becoming increasingly popular among antivirus developers as a key factor in defence against malware. While machine learning is achieving state-of-the-art results in many areas, it also has drawbacks exploited by many with white-box attacks. Although the white-box scenario is possible in malware detection, the detailed structure of antivirus is often unknown. Consequently, we focused on a pure black-box setup where no information apart from the predicted label is known to the attacker, not even the feature space or predicted score. We implemented our exploratory integrity attack using a reinforcement learning approach on a dataset of portable executable binaries. We tested multiple agent configurations while targeting LightGBM and MalConv classifiers. We achieved an evasion rate of 68.64% and 13.32% against LightGBM and MalConv classifiers, respectively. Besides traditional modelling of malware adversarial samples, we present a setup for creating benign files that can increase the targeted classifier's false positive rate. This problem was considerably more challenging for our reinforcement learning agents, with an evasion rate of 3.45% and 36.62% against LightGBM and MalConv classifier, respectively. To understand how these attacks transfer from classifiers based purely on machine learning to real-world anti-malware software, we tested the same modified files against seven well-known antiviruses. We achieved an evasion rate of up to 47.09% in malware and 14.29% in benign adversarial attacks.

1 Introduction

Malware detection is one of the most important problems in information security since the detection of malware in advance allows us to block it. Malware detection is a binary classification problem of distinguishing between malware and benign

M. Kozák · M. Jureček (✉) · R. Lórencz
Faculty of Information Technology, Czech Technical University in Prague, Prague, Czechia
e-mail: kozakmat@fit.cvut.cz; martin.jurecek@fit.cvut.cz

M. Stamp et al. (eds.), *Artificial Intelligence for Cybersecurity*, Advances in Information Security 54, https://doi.org/10.1007/978-3-030-97087-1_1

files [3]. One of the main problems of malware detection systems is insufficient accuracy while keeping the false positive rate at an acceptable level. There is a need to build a machine learning framework suited for real-life practical use that generically detects as many malware samples as possible, with a very low false positives rate. The significant problem to be solved is how to detect malware that has never been seen before.

To defend against malware, users typically rely on antivirus (AV) products to detect a threat before it can damage their systems. Antivirus vendors rely mainly on a database of sequences of bytes (signatures) that uniquely identify the suspect files and are unlikely to be found in benign programs [36]. The major weakness of signature detection is that malware writers can easily modify their code, thereby changing their program's signature and evading virus scanners. The signature detection technique is unable to detect obfuscated and zero-day malware. Encryption, polymorphism, metamorphism, and other code obfuscation techniques are widely used by malware authors to evade signature detection techniques. For this reason, malware researchers are investigating novel detection strategies.

Nowadays, antivirus vendors face several problems concerning malware detection. The concept of employing machine learning to malware detection provides promising solutions [31]. Moreover, since malware developers create more and more sophisticated techniques, it is necessary to use the latest techniques from machine learning to keep the error rate and false positive rate as low as possible. This game may someday converge to the point when artificial intelligence of attackers will fight against the artificial intelligence of malware researchers.

Machine learning models are vulnerable to adversarial attacks that can fool the models [9]. For instance, an adversary can craft malware that has a similar feature vector to some benign file's feature vector. As a result, the training set may have different statistical distribution than the distribution of the testing set. Therefore, it is necessary to create defence techniques in order for machine learning algorithms can resist such adversarial attacks.

The goal of this paper is to implement a black-box exploratory integrity attack using reinforcement learning. We implement executability preserving modifications and train reinforcement learning agents to alter Windows portable executable binaries with an aim to avoid detection by a targeted machine learning classifier. These evasion techniques are later tested on real-world antivirus software. In comparison with other works, we do not only focus on malware adversarial samples, but we also deal with an inverted scenario of benign adversarial examples.

This paper is organized in the following form. Section 2 gives the necessary background to our work with a brief description of adversarial machine learning, reinforcement learning, portable executable file format, etc. In Sect. 3, we present our implementation and dataset description. Next, Sect. 4 contains all information about experiments and achieved results. We summarize related work in Sect. 5. The conclusion to this paper and ideas for future work can be found in Sect. 6.

2 Background

In this section, we provide the minimal necessary background to understand our paper. We firstly introduce adversarial and reinforcement machine learning. Then we briefly describe portable executable file format and finish with the definition of used evaluation metrics.

2.1 Adversarial Machine Learning

Machine learning outperforms human capabilities in many ways, yet we are reserved in trusting its decisions in areas such as self-less car driving or disease diagnostics. One of the reasons is the insufficient interpretability of the decisions and the resulting possible weakness or bias of the system [15]. *Adversarial machine learning* is a research area specializing in strengthening machine learning (ML) systems to be resistant against attacks both from the inside (data poisoning) and outside (evasion attacks).

In common terminology, an action to bypass or mislead a ML system is an adversarial attack. An attacker is called an adversary but both are acceptable and used interchangeably. In this section, we summarize the taxonomy of adversarial attacks and describe some prevalent adversarial attack strategies, focusing mainly on malware detection domain.

2.1.1 Taxonomy

In this part, we will closely follow the taxonomy laid down by Huang et al. in [18] as it is one of the most complete overviews of this topic we have found. They identify three main ways how to break down adversarial attacks based on these three properties: *influence*, *security violation*, *specificity*.

Influence The first property is the way we can look at how adversary influences targeted ML model. There are two main categories. The first is *exploratory* attacks that do not alter the model itself but try to circumvent the model to achieve attacker's goal—usually, misclassification of a group of malicious files.
The second group is called *causative* attacks where the attacker impacts the model itself, in particular the training phase. It can be in form of wrongly labelled malware samples in training dataset.

Security violation The second property is characterized by the objective of adversary. There are three groups. *Integrity* attacks cause an increase in false-negative rate. In the domain of malware detection, a false negative is malware sample classified as benign.

The second group is *availability* attacks where the adversary does not focus on a single class to be misclassified but targets the model's accuracy as a whole resulting in the model becoming completely unusable.

When an adversary tries to steal information from the model itself, e.g., what training data the model was learned on, it is called a *privacy* attack.

Specificity The last property describes how large the attacker's target set is. In *targeted* attacks, there is a small set of samples that are supposed to be misclassified, i.e., an author of malicious software wants his particular program to be installed on the victim's device.

Whereas in *indiscriminate* attacks adversary does not specify which but rather how many samples should be mislabelled. This attack can be used as a one of the proves that given AV is not secure.

Individual categories can be combined together, e.g., a causative targeted attack can be when an adversary inserts malware binaries labelled as benign into a classifiers training dataset to prevent the classifier from correctly predicting for a particular malicious file.

Attacks can also be classified based on the available knowledge of the targeted model to attackers. If the attacker does not have any information about the model and is left with only the model's output, it is called a *black-box* attack. An opposite attack is called a *white-box* where all information about the model is known. In between these two is a *grey-box* attack where partial knowledge is accessible, e.g., feature space of targeted classifier.

2.2 Reinforcement Learning

The key idea of *reinforcement learning* (*RL*) is a simple one. An *agent* (a learning system) wants to achieve some goal. To achieve its objective, the agent must adapt its behaviour. To learn which actions are good and bad, the agent needs a stimulus in the form of signals from an *environment* where the agent works. This section is based on [29] from which we adopted the following notions.

Reinforcement learning is a triplet of an agent policy, a reward function and a value function. In some cases a model of environment is used as well.

The *agent policy* represents the agent's behaviour at a given time and environment state. It is a function which maps pairs of states and action to the probability of taking individual action at given state. The policy can be in the form of a simple automaton, lookup table, but also a sophisticated algorithm.

The *reward function* is an immediate response from the environment to the agent's action. The return value is a number that grades the action taken by the agent based on the goal. Formally, this number defines the agent's purpose, and under no circumstances can the agent change this function, i.e., change the goal it is facing.

The *value function* is a look to the future on what is an expected cumulative reward from the current environment state. This function can be understood as a heuristic function, and it is a critical part of any RL model. While reward function can be usually calculated easily, the value function must be recalculated over and over based on past observations. It is clear that agent looks for states with the highest value since these will maximize future rewards.

The *model of environment* simulates the environment and allows the agent to predict future states and rewards. This part is optional, and not all RL systems contain it.

Formally reinforcement learning is defined as repeated interactions between agent and environment. In our space of modifications of binary files these interaction take place at discrete time steps $t = 0, 1, 2, \dots$ At time step t the environment is at state $S_t \in S$ where S is the set of all observable states. The agent receives state S_t and, based on its policy π, chooses action $A_t \in A(S_t)$ where $A(S_t)$ is a set of all possible actions at state S_t. When the environment receives response from agent in the form of action A_t it computes and sends back a reward $R_{t+1} \in R \subset \mathbb{R}$ and changes its state: $S_t \xrightarrow{A_t} S_{t+1}$. In this notation, R_{t+1} represents the reward for the state S_{t+1}.

As stated above, the agent looks for states that maximize future rewards. To formalize future rewards, we denote $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$ a sum of all rewards after time step t, where T denotes the final time step. If $T \neq \infty$ we can call the S_T a *terminal state* and the entire process of states, actions and rewards from time step $t = 0$ to $t = T$ an *episode*. After the end of each episode the environment is reset to initial state and new independent episode begins. This repeats until terminal condition is met. Many problems will not have any terminal state. These problems are called *continuous*, and they are not covered in this work.

In real situations, the agent does not know the exact value of G_t at time step t. The value of G_t is approximated by the value function. In the computation, it is common to use *discounted* future rewards,

$$G_t \approx R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = \sum_{i=0}^{T-t-1} = \gamma^i R_{t+i+1}$$

where $0 \leq \gamma \leq 1$ is called *discount rate*. This helps regulate importance of looking far into the future. Further we outline an *action-value function for policy* π, $q_\pi(s, a)$, that defines value of action a when in state s if following policy π. The value is in form of expected future reward G_t, formally,

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t | S_t = s, A_t = a]$$

where $\mathbb{E}_\pi[\cdot]$ is expected value of random variable for policy π.

The rest of this section describes two algorithms which we later used in Sect. 3. Note that there are numerous other algorithms for reinforcement learning.

Q-learning is an algorithm introduced by Watkins [34] to learn the action-value function. It works by iteratively updating the learned action-value function, Q, in the following manner,

$$Q^{new}(S_t, A_t) = Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a\{Q(S_{t+1}, a)\} - Q(S_t, A_t)]$$

where α is learning rate, $a \in A(S_{t+1})$. Therefore, $\max_a\{Q(S_{t+1}, a)\}$ is an estimation of the best future value. Learned values of function Q are stored in so-called Q-table. In our work we used extension of this algorithm called *deep Q-learning* or *deep Q network* (DQN) where the Q-table is replaced with neural network [24].

Second algorithm we describe is *proximal policy optimization* (*PPO*) [27]. It is based on *policy gradient* methods. Informally, these methods use *gradient ascent* algorithm to approximate policy weight vector $\theta \in \mathbb{R}^n$. The policy of taking action a is then conditioned not only by the current state S_t but also weight vector θ. PPO improves the approximation by vanilla policy gradient method with multiple epochs of the gradient ascent before updating the policy vector.

2.3 Portable Executable File Format

Portable executable (PE) format is a file format commonly found on Windows operating system. Executable files (EXE), object codes and dynamically link libraries used on 32-bit and 64-bit systems adhere to this format. The structure of the PE file is as follows. The program starts with *MS-DOS header* and *stub* which are nowadays almost unused. Following is the signature and *file header* which contains information such as a target machine or size of section table. Next is *optional header* that contains, among other things, the necessary data directories. Finishing the program is a section table with corresponding section data. Precise specification is available in [19].

3 Implementation

Based on the taxonomy introduced in Sect. 2.1.1 our main approach belongs to the category of exploratory integrity attack. In other words, we are trying to mislead the classifier (antivirus) to predict malicious files as benign incorrectly. We also experimented with an idea to make an exploratory integrity attack where we interject classes. Therefore our goal would be to mislead the classifier to incorrectly predict benign files as malicious.

We utilized the existing `gym-malware` [2] framework. This framework provides a setup for deploying a custom RL agent to generate adversarial samples against a malware classifier. Both the agent and the classifier can be easily changed with even remote classifier supported. The environment is in `OpenAI Gym` [5] format and binary manipulations are implemented in `LIEF` [30] library. However, we have found that their implementation is not ideal. In particular, modifications of

PE files using LIEF do not preserve functionality for most of their implemented modifications according to our testing procedure and make unnecessary changes to original files. We also disagree with their approach to using the target classifier's feature space as an observation space for the RL agent as it can give an unfair advantage to the RL agent as opposed to a pure black-box setup where no information apart from the result label is known. For the reasons mentioned above, we have modified the existing implementation [2] and our implementation contain the following differences: minimizing unneeded changes to PE files, deploying different RL agents, implementing pure black-box setup.

3.1 Overview

We propose a complete setup consisting of custom gym class in `OpenAI Gym` format, manipulator of PE files preserving their executability and reinforcement learning agent which learns to maximize evasion rate against targeted classifier while minimizing the number of PE file modifications. *Evasion rate* is a key metric that is used in adversarial machine learning on malware detection domain. It represents the proportion of files that were misclassified by the target classifier,

$$evasion\ rate = \frac{misclassified}{total}$$

3.2 Dataset

We use two separate datasets of PE executables. The first, malware dataset consists of 5000 malware files from the VirusShare [32] repository. The second, benign dataset was gathered from fresh Windows 10 installation and Windows university computers and contains 1592 files. Both datasets contain only executables. Dynamically linked libraries and object files were not included in the datasets.

3.3 PE File Modifications

We implemented most of the modifications in `pefile` [6] library by extending existing implementation of PE file modifications by `MAB-Malware` [28] and `PEsidious` [7]. `MAB-Malware` implementation is described in Related Work under paper [28] and `PEsidious` is an adversarial malware generator built on the top of `gym-malware` framework. We opted to rather create more than fewer modifications and let the RL agents choose which are the most valuable. Be aware that we focus on black-box attacks where we do not have any information about what features the targeted model might use so we cannot design modifications tailored against specific malware detector.

All modifications were tested on a fresh Windows 10 virtual box environment. We selected randomly 50 benign binaries from our dataset as a test set and considered modification successful if the given program executed and showed itself in the `tasklist` command. This protocol has its limitation which we discuss in Sect. 6. We accepted the modification if at least 45 out of 50 binaries pass the test. We ensured that a set of 50 benign binaries passed the test in the first place. In total we have 9 working modifications:

Rename Section:	Chooses a section at random and renames it to one of common benign section name.
Add Section:	Adds new section with benign content (if enough space for a new entry in Section table).
Remove Certificate:	Removes certificate table.
Remove Debug:	Removes debug data.
Break Checksum:	Zeros out CheckSum in Optional header.
Append Overlay:	Adds benign content at the end of file.
Increase TimeDateStamp:	Increases TimeDateStamp by 500 days.[1]
Decrease TimeDateStamp:	Decreases TimeDateStamp by 500 days.
Append Imports:	Selects library from a list of common imported libraries. Adds a new section with library name and its typical functions.

We had experimented with other modifications as well, e.g., shuffling section headers. However, we did not achieve an acceptable execution ratio. Comparison of number of executing files after modifications can be seen in Table 1. We compared `gym-malware`, `PEsidious` and our modifications extending `MAB-Malware` and using `pefile` library. We can see that our set of PE file modifications (pefile column) achieves higher execution rates than other implementations on most of the tested modifications. All nine operations from the pefile column were later used as the RL agent action space.

3.4 *Target Classifier*

We studied two primary scenarios. In the first one, the target classifier is the LightGBM model, and in the second one, it is MalConv, both trained by authors of EMBER dataset [1]. LightGBM model is a gradient boosted method and is trained on PE files transformed to feature space of 2,381 float numbers. On the other hand, the MalConv classifier is deep convolutional neural network and represents binary files with their first 200,000 raw bytes.

[1] 500 days were chosen to represent a substantial period of time and not a multiple of one year.

Table 1 Numbers of files executed successfully after modification from total of 50 binaries. The symbol $\times$ denotes that given operation was not implemented

Action	Gym-malware	PEsidious	pefile
Break checksum	4	$\times$	47
Create new entry point	14	$\times$	$\times$
Append new import	42	48	48
Overlay append	50	47	46
Remove debug	5	$\times$	50
Remove certificate	22	$\times$	49
Add new section	4	48	46
Append to section	8	$\times$	$\times$
Rename section	5	4	49
upx pack	46	$\times$	$\times$
upx unpack	49	$\times$	$\times$
Increase TimeDateStamp	$\times$	$\times$	49
Decrease TimeDataStamp	$\times$	$\times$	49

3.5 Agent and Its Environment

We implemented multiple environment setups, all adhering to the `OpenAI Gym` structure. Key methods which must be implemented are `reset` and `step` methods. The `reset` method resets the environment to the initial state to get ready for the next episode. The return value of `reset` method is an *observation*. The **Observation** is a representation of the environment state which is presented to the agent. We implemented different observations based on targeted classifier. We either used raw bytes from the beginning of the binary or extracted features from PE files such as bytes histogram, imports and sections info or printable string. The `step` method performs the given agent's action on the environment. It is responsible for changing the environment state, tracking episode length and calculating reward for the agent's action. We limited the length of the episode to 10 calls of `step` method. It returns quadruplet of observation, reward, done (flag if an episode has ended), info (debugging information). The *reward* is either 0 if the action does not cause misclassification or (maximum episode length - number of taken actions) $\times$ 10 + 100. By taking the number of taken actions into account, we tried to force the agent to prioritize minimal modifications to the binaries.

4 Evaluation

In this section, we describe all the experiments we performed and present our results. In total, we have two main experiments with multiple evaluation phases and initial setups.

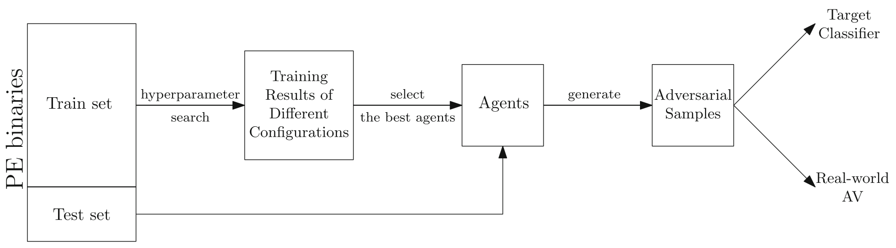

Fig. 1 Overview of our experiment workflow

All experiments follow this protocol. We start with a dataset of PE binaries and split it to train set and test set in 80:20 ratio. We create a training gym environment and use `Ray Tune` [23] to find optimal hyperparameters. `Ray Tune` is a Python library providing easy to use hyperparameter tuning interface. We explore the space of several hyperparameters. We perform a grid search on: agent type (DQN vs PPO), learning rate (lr) and discount rate (gamma γ) with other parameters left to default values as set by the authors of `Ray Tune`.

After identifying suitable hyperparameters values, we train the best performing (the one with the highest mean reward in a single training iteration, in the subsequent tables marked as episode reward) agents again up to 15,000 episodes. In the end, we test the agents on the test set. In the test results, we highlighted the highest evasion rate values in bold. In both the training and testing procedure, we discard any files that are already misclassified before adversarial modification. After evaluation on our test set against target classifier, we test the same modifications made by the best performing agent on real-world AVs from cybersecurity companies Avast, Cylance, Symantec (NortonLifeLock), ESET, Kaspersky, McAfee and Microsoft using VirusTotal [33] website. In our presentation of results, we anonymize the names of AVs to minimize the potential misuse of this work. Overview of our experiment workflow is pictured in Fig. 1.

4.1 Adversarial Malware Examples

In the first experiment, we focused on generating adversarial malware samples, i.e., we modified malware binaries to evade detection by the target classifier. This procedure is called an exploratory integrity attack. For this task, we defined multiple environment setups. In the first one, we use the LightGBM classifier as a target model, and we use the first 1024 (4096, 8192) bytes of PE binary as observation space for the agent. We labelled this setup *M-1*.

Table 2 presents a comparison of agents with different hyperparameters values with regard to their mean reward and episode length. It is clear that DQN agents significantly outperform PPO agents in all environment and hyperparameters setups.

Table 2 Training results of the search for hyperparameters for the M-1 setup. Each table represents different observation space (1024B, 4096B, 8192B). Tables are sorted by episode reward in descending order

Agent	Gamma	lr	Episode reward	Episode length
DQN	0.5	0.001	125.8	5.0
DQN	0.75	0.001	119.47	5.4
DQN	0.999	0.001	118.17	5.4
DQN	0.999	0.01	115.46	5.74
DQN	0.75	0.01	110.97	5.84
DQN	0.5	0.0001	110.4	5.65
DQN	0.5	0.01	109.51	5.88
DQN	0.75	0.0001	107.44	5.86
DQN	0.999	0.0001	105.91	5.96
PPO	0.5	0.001	92.09	6.82
PPO	0.999	0.001	86.86	7.1
PPO	0.75	0.001	86.47	7.1
PPO	0.5	0.0001	83.83	7.09
PPO	0.5	0.01	83.04	7.26
PPO	0.75	0.0001	82.22	7.19
PPO	0.999	0.01	81.62	7.35
PPO	0.999	0.0001	80.3	7.38
PPO	0.75	0.01	74.95	7.64

(a) 1024B

Agent	Gamma	lr	Episode reward	Episode length
DQN	0.75	0.01	110.99	5.85
DQN	0.5	0.0001	110.49	5.92
DQN	0.5	0.01	110.04	5.93
DQN	0.999	0.001	109.96	6.02
DQN	0.75	0.0001	108.25	5.9
DQN	0.999	0.0001	107.62	6.09
DQN	0.5	0.001	106.08	6.15
DQN	0.75	0.001	105.45	6.17
DQN	0.999	0.01	100.91	6.34
PPO	0.75	0.001	90.97	6.9
PPO	0.5	0.001	90.07	6.89
PPO	0.75	0.01	86.97	7.08
PPO	0.75	0.0001	86.37	7.04
PPO	0.5	0.0001	86.16	7.09
PPO	0.999	0.0001	84.34	7.22
PPO	0.999	0.001	82.1	7.34
PPO	0.999	0.01	66.63	8.09
PPO	0.5	0.01	65.72	7.93

(b) 4096B

Agent	Gamma	lr	Episode reward	Episode length
DQN	0.5	0.0001	110.64	5.91
DQN	0.999	0.0001	110.48	5.95
DQN	0.75	0.0001	109.88	5.97
DQN	0.75	0.01	109.23	6.02
DQN	0.5	0.01	107.96	6.14
DQN	0.75	0.001	105.94	6.15
DQN	0.5	0.001	105.94	6.24
DQN	0.999	0.01	105.91	6.14
DQN	0.999	0.001	104.41	6.24
PPO	0.5	0.0001	91.32	6.86
PPO	0.5	0.001	89.74	6.92
PPO	0.999	0.0001	88.32	7.06
PPO	0.75	0.001	87.14	7.06
PPO	0.75	0.0001	86.83	7.11
PPO	0.75	0.01	86.45	7.21
PPO	0.999	0.001	80.48	7.47
PPO	0.5	0.01	61.25	8.32
PPO	0.999	0.01	60.79	8.32

(c) 8192B

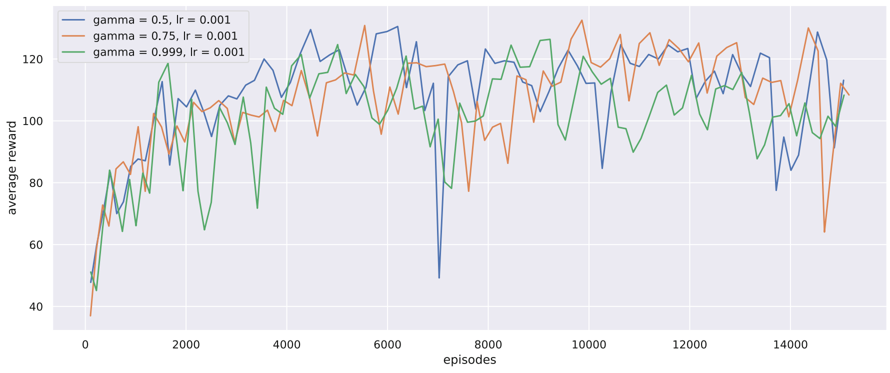

Fig. 2 Training runs of the three most promising agents from the M-1 setup

Table 3 Test results of the three most promising agents from the M-1 setup

Agent	Gamma	lr	Evasion rate [%]
DQN	0.5	0.001	**68.64**
DQN	0.75	0.001	65.21
DQN	0.999	0.001	60.22

Based on the results from Table 2 the environment with an observation space of 1024 bytes performed the best.

We took the first three best configurations (based on the mean episode reward) from Table 2 and tested them up to 15,000 episodes. The training runs are shown in Fig. 2 below.

From Fig. 2 it is not particularly clear what configuration will perform the best on the test set. However DQN agents with gamma 0.5 and 0.75 scored a bit higher than the 3rd configuration.

After the training had ended we showed agents the testing set and performed evaluation. The highest evasion ratio of 68.64% was achieved by DQN agent with $\gamma = 0.5$ and $lr = 0.001$. Full results are shown in Table 3.

In the second setup, we essentially switched the targeted classifiers feature space and agent's observation space. Here we used the LightGBM classifier's feature space as PE file representation for the agent and as targeted classifier we chose MalConv model. This setting is marked as *M-2*. In this experiment, we validated that DQN is better for our problem and we use it for all future experiments. Complete results are shown in Table 4.

We trained the three top agents up to 15,000 episodes, the training progress is shown in Fig. 3. The highest peak in terms of mean episode reward was recorded by configuration with $\gamma = 0.75$ and $lr = 0.01$. Compared to Fig. 2 the mean episode reward varied dramatically between training iterations and was significantly lower.

After we trained the agents, we evaluated their performance on the test set. The results are shown in Table 5. The results are significantly worse than in our M-

Table 4 Training results of the search for hyperparameters for M-2 setup. Table is sorted by episode reward in descending order

Agent	Gamma	lr	Episode reward	Episode length
DQN	0.999	0.01	37.43	9.39
DQN	0.5	0.01	35.62	9.46
DQN	0.75	0.01	35.59	9.46
DQN	0.75	0.001	35.27	9.56
DQN	0.75	0.0001	33.84	9.63
DQN	0.5	0.0001	33.29	9.58
DQN	0.999	0.0001	32.07	9.48
DQN	0.999	0.001	31.95	9.59
DQN	0.5	0.001	31.77	9.66
PPO	0.5	0.0001	21.19	10.1
PPO	0.5	0.001	20.14	10.18
PPO	0.999	0.0001	19.64	10.17
PPO	0.75	0.0001	19.63	10.16
PPO	0.75	0.001	17.6	10.29
PPO	0.999	0.001	16.84	10.32
PPO	0.75	0.01	15.57	10.35
PPO	0.5	0.01	14.1	10.39
PPO	0.999	0.01	11.2	10.53

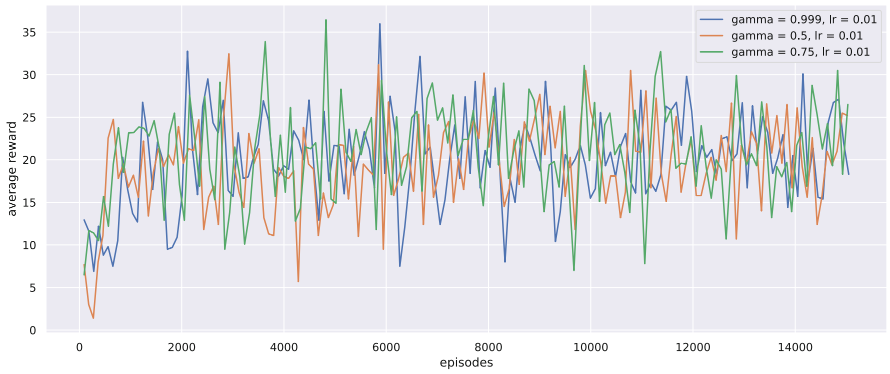

Fig. 3 Training runs of the three most promising agents from the M-2 setup

Table 5 Test results of the 3 most promising agents from the M-2 setup

Agent	Gamma	lr	Evasion rate [%]
DQN	0.999	0.01	12.56
DQN	0.5	0.01	11.94
DQN	0.75	0.01	**13.32**

1 setting and that might indicate that the MalConv classifier is more resilient to adversarial attacks than LightGBM.

In the third setting, we simulated the approach of `gym-malware` framework authors using EMBER feature space for agent's observation and EMBER classifier

Table 6 Training results of the search for hyperparameters for the M-3 setup. Table is sorted by episode reward in descending order

Agent	Gamma	lr	Episode reward	Episode length
DQN	0.75	0.0001	115.41	5.66
DQN	0.75	0.001	114.38	5.53
DQN	0.75	0.01	113.13	5.67
DQN	0.5	0.01	112.62	5.61
DQN	0.999	0.01	110.96	5.85
DQN	0.999	0.001	108.65	5.91
DQN	0.999	0.0001	108.36	5.95
DQN	0.5	0.001	105.6	6.17
DQN	0.5	0.0001	103.01	6.17

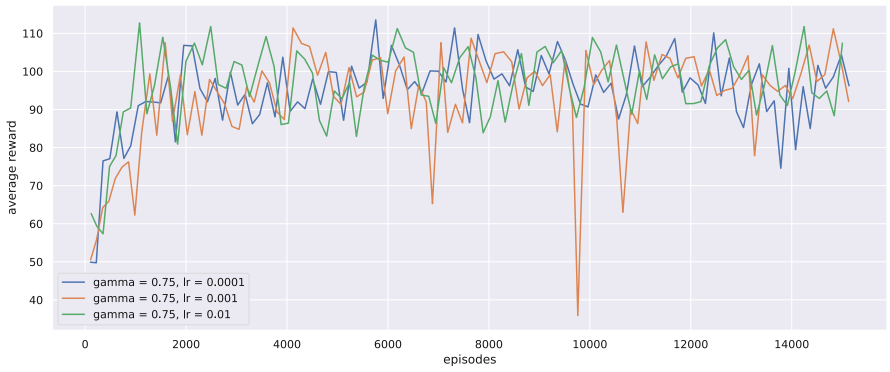

Fig. 4 Training runs of the three most promising agents from the M-3 setup

Table 7 Test results of the three most promising agents from the M-3 setup

Agent	Gamma	lr	Evasion rate [%]
DQN	0.75	0.0001	58.50
DQN	0.75	0.001	**60.37**
DQN	0.75	0.01	57.25

as a targeted model. We marked this approach *M-3*. Interestingly, we did not achieve a higher average reward during training than with our first setup. The results of the search for optimal hyperparameters can be found in Table 6.

We again took the three best performing agents from the ranking and trained them up to 15,000 training episodes. The training process is pictured in Fig. 4. We did not find big differences between configurations during training runs, although the agent with $\gamma = 0.75$ and $lr = 0.001$ recorded the biggest drops in average reward during training.

We evaluate these three agents on the test set and the result are in Table 7 below. At the beginning of Sect. 3 we argued that having an agent with observation space equal to the feature space of the target classifier might give the agent an unfair advantage. However, in our evaluation, we actually did score a lower evasion rate than with our M-1 setup.

Table 8 Overall results of all three setups from the first experiment against seven real-world AVs [%]

	AV-1	AV-2	AV-3	AV-4	AV-5	AV-6	AV-7
M-1	2.76	1.8	5.54	1.47	4.87	1.47	8.06
M-2	**6.12**	**31.69**	**26.52**	**13.03**	**16.94**	**47.09**	**17.67**
M-3	3.74	14.68	3.01	1.13	4.61	31.79	2.03

At the end of the evaluation, we wanted to verify real-world performance. Therefore, we took the best agents from all three setups, modified all samples from the test set, excluding the misclassified samples, and then showed them to commercially available AVs. We used DQN agents with $\gamma = 0.5$ and $lr = 0.001$ from M-1 setup, $\gamma = 0.75$ and $lr = 0.01$ from M-2 setup together with $\gamma = 0.75$ and $lr = 0.001$ from M-3 setting. The results we achieved are shown in Table 8.

From the Table 8 we can see that the overall best setting is the DQN agent ($\gamma = 0.75, lr = 0.01$) from M-2 setup. This is an unforeseen result since when testing against the original target classifier (MalConv), this configuration achieved in most cases lower evasion rate (Table 5) than against real-world AVs.

4.2 *Adversarial Benign Examples*

In the second experiment, we implemented an exploratory integrity attack with interjected classes, i.e., we modified benign files to mislead the target classifier into falsely predicting malware. This is an unusual setup since most researches focus on the opposite scenario. Even though it is less popular, we think that a scheme where one company develops both the AV and other software could potentially modify their software to increase the false positive of rival AV developers.

In this experiment, we defined two environments with LightGBM and MalConv classifier as a targeted model. This time we used only the 1024 bytes feature space for the first setup and did not experiment with other than DQN agents. We labelled the first and second settings as *B-1* and *B-2*, respectively.

Table 9 shows results of DQN agents from B-1 setting. The difference between the results from the first experiment and these is striking, with the latter performing significantly worse. On the other hand, results from B-2 setup (Table 10) where we attack the MalConv model are looking better than from the first experiment.

As in experiment one, we took the first three configurations from both setups and trained them up to 15,000 episodes. The first setup with LightGBM model as target classifier is shown in Fig. 5. All agents struggled to increase the false positive rate of the classifier with a mean reward not exceeding 20.0.

The second setup performed a lot better in modifying benign files against the MalConv classifier with two agents configurations that exceeded the mean reward of 70.0. The training runs are shown in Fig. 6.

The test results from Table 11 verified the statement mentioned above about the B-1 setup. The evasion rate of 3.45% is the lowest recorded evasion rate across the

Table 9 Training results of the search for hyperparameters for the B-1 setup. Table is sorted by episode reward in descending order

Agent	Gamma	lr	episode_reward_mean	episode_len_mean
DQN	0.5	0.01	13.76	10.34
DQN	0.999	0.01	13.7	10.35
DQN	0.75	0.01	12.28	10.4
DQN	0.999	0.001	12.0	10.43
DQN	0.5	0.001	10.0	10.54
DQN	0.999	0.0001	8.9	10.56
DQN	0.5	0.0001	7.5	10.61
DQN	0.75	0.001	7.4	10.62
DQN	0.75	0.0001	7.2	10.64

Table 10 Training results of the search for hyperparameters for the B-2 setup. Table is sorted by episode reward in descending order

Agent	Gamma	lr	episode_reward_mean	episode_len_mean
DQN	0.75	0.01	73.28	7.55
DQN	0.75	0.0001	68.36	7.79
DQN	0.5	0.01	67.84	7.74
DQN	0.999	0.0001	67.73	7.84
DQN	0.5	0.0001	65.0	8.07
DQN	0.75	0.001	64.85	7.77
DQN	0.5	0.001	64.27	7.87
DQN	0.999	0.01	62.66	8.0
DQN	0.999	0.001	59.28	8.24

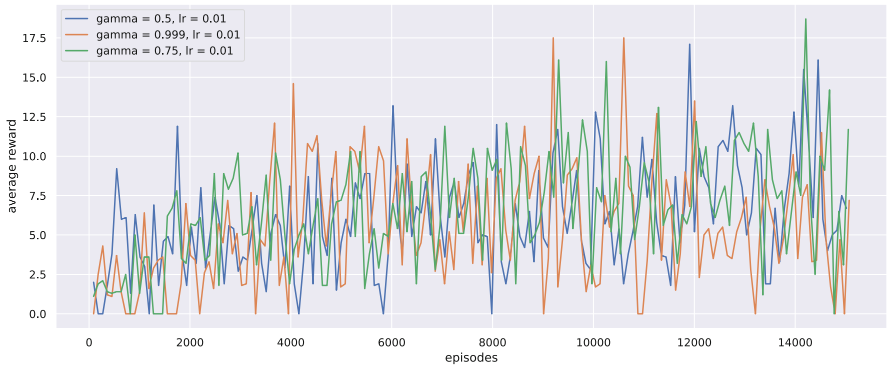

Fig. 5 Training runs of the three most promising agents from the B-1 setup

experiments. This reveals that creating benign files classified as malware against the LightGBM classifier is more complicated than the reversed scenario. However, this is not the case with the MalConv classifier. The results from Table 12 show that

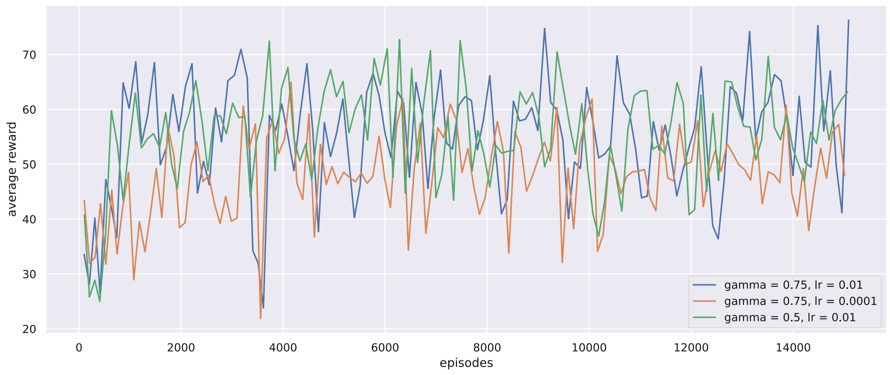

Fig. 6 Training runs of the three most promising agents from the B-2 setup

Table 11 Test results of the three most promising agents from the B-1 setup

Agent	Gamma	lr	Evasion rate [%]
DQN	0.5	0.01	2.19
DQN	0.999	0.01	2.19
DQN	0.75	0.01	**3.45**

Table 12 Test results of the three most promising agents from the B-2 setup

Agent	Gamma	lr	Evasion rate [%]
DQN	0.75	0.01	**36.62**
DQN	0.75	0.0001	26.76
DQN	0.5	0.01	29.58

Table 13 Overall results of two setups from the second experiment against seven real-world AVs [%]

	AV-1	AV-2	AV-3	AV-4	AV-5	AV-6	AV-7
B-1	6.25	**0.33**	0.0	**0.34**	0.0	0.0	0.0
B-2	**14.29**	0.0	0.0	0.0	0.0	0.0	0.0

we achieved higher evasion rates with benign files than with malware in the first experiment. These results might also be partially caused by the differing sizes of benign and malware datasets in our experiments.

In the end, we tested one agent from both setups, which achieved the highest evasion rate against seven AVs programs. The results can be found in Table 13. We can see that for most AVs, our RL agent struggled to mislead antivirus into classifying benign files as malware. Together with our test results, the results against real-world AVs hint that performing exploratory integrity attacks with interjected classes in the domain of malware detection is much harder than traditional scenario.

5 Related Work

In this section, we summarize up-to-date publications which focus on the adversarial machine learning in conjunction with malware detection topic. We divide related work into three sections based on the author's strategies: gradient-based attacks, reinforcement learning based attacks and other methods.

5.1 Gradient-Based Attacks

The gradient-based attack was proposed in [21] to attack the MalConv [25] classifier which utilizes raw bytes of binaries. Their attack modified in average less than 1% of padding bytes at the end of the file and achieved a 60% evasion rate.

Grosse et al. in [16] performed a white-box attack on their neural network classifier. In their attack, they computed necessary perturbation using a gradient of their network and then changed the corresponding features. They successfully mislead their deep neural network in more than 63% of cases.

Another research that is based on gradient-based attacks is presented in [22]. The authors generated small chunks of bytes called payloads which they injected either into unused parts of sections or at the end of the file to ensure the functionality after injection. Their white-box attack targeted against MalConv and achieved an almost perfect evasion rate of 99%.

More attacks on MalConv were carried out in [10] by Luca Demetrio et al. They used a technique called integrated gradients to explain what parts of binaries contribute to prediction. They uncovered that MalConv learns weak features from the DOS header, and perturbing only a few bytes is enough to obscure detection in 52 of 60 malware samples.

Yang et al. in [35] treated binaries as greyscale images. Firstly the authors marked key parts of binaries by "00", then they processed them by a convolutional neural network (CNN). On CNN, a fast gradient sign method was applied to find perturbations within marked sections. The perturbations were then converted to "closest" dead-code instructions (`wait`, `nop`, `...`) or API calls. Their approach recorded a decrease of over 60% in the accuracy of deep learning detectors and an evasion rate of over 30% on VirusTotal.

5.2 Reinforcement Learning-Based Attacks

The authors of [13] proposed a deep reinforcement model called RLAttackNet to attack their deep neural classifier. They achieved an evasion rate of 19.13% and used adversary samples to retrain their malware classifier to increase its area under the receiver operating characteristic curve from 0.989 to 0.996.

Anderson et al. presented a reinforcement learning framework called `gym-malware` [2]. They targeted gradient boosted decision tree (GBDT) trained on 100,000 executables. They experimented with both score-based and black-box attacks, with the latter being more successful. After closer inspection of their implementation, we have found that the authors did not perform a complete black-box attack because they used identical feature space both for their RL agent and targeted classifier.

Another reinforcement learning approach is presented in [28]. The authors present stateless RL model, which means that the order of actions applied by RL agent does not matter. They also try to remove unnecessary actions and thus interpret the targeted model. Their work shows promising results with an evasion rate of 74%–97% on ML detectors (LigthGBM Ember [1] and MalConv) and 32%–48% on commercial AVs. They also study the transferability of attacks between targeted models and found out that among ML detectors, it's over 80%. However, between ML and commercial AVs, only up to 7%.

The author of [26] used Android permissions as feature space. This is a key point because modifying permissions doesn't cause any malfunction of a given application. Using a reinforcement learning agent, they achieved an average evasion rate of 44.28% in a white-box scenario against 8 ML classifiers and 53.20% in a grey-box strategy. They managed to reduce this evasion ratio by 15.22%–29.44% by retraining with adversarial samples.

5.3 Other Methods

In [8], the authors used the feature space of their classifier to tailor the attack to evade detection. The feature space was in the form of application programming interface (API) calls of input sample. Greedy search was used to find a set of API calls to add or remove. Later they retrained the classifier with adversarial samples and introduced security regularization to improve their detection ratio further. However, the authors did not propose an algorithm to convert an adversarial set of API calls back to the real-world executable.

Demetrio et al. [11] performed a black-box attack at MalConv and GBDT by injecting small chunks of benign codes into malware binaries either at the end of the file or inside newly created sections. The authors tackled this as an optimization problem to maximize the evasion rate while minimizing the size of injected code. They achieved an evasion ratio of more than 90% on the MalConv classifier and 60%–80% on GBDT. Their attack also bypassed at average 12 out of 70 AVs on VirusTotal [33].

Hu and Tan in [17] proposed a generative adversarial network (MalGAN) to create evasive samples. The authors achieved an almost perfect evasion ratio on random forest, logistic regression and other ML classifiers. They also showed that their model MalGAN could be quickly retrained to bypass new detectors. Needless to say, they worked only with API calls as a representation of executable binary and did not mention how to translate the results back to binaries.

Several authors focused on data poisoning, e.g., Chen et al. poisoned Android training dataset in [9] which led to misclassification of around 70% samples.

In [12], authors trained generative sequence-to-sequence recurrent neural network language model on benign binaries to generate benign bytes, which are later appended at the end of malware executable to bypass detection. This black-box approach led to an evasion rate of more than 72% on three different ML classifiers.

Unique concept so-called grey-box attack is showed in [4] where the attacker has knowledge about feature space of targeted classifier but does not have access to its predictions. They trained their model utilizing the same feature space as the targeted model to substitute it. Using Monte Carlo search, they found a set of operations (limited to size 5) to evade their substitute model in 56% of cases. The authors found out that simple changes such as certificate signature change were enough in 71% of successful mispredictions. Same operations were tried against the targeted classifier but achieved an evasion ratio of less than 9%.

Another paper was published by Fleshman et al. [14] where they presented multiple adversary attacks. First, the authors tried up to 10 random changes. Second, using a binary search algorithm to find critical regions for malware classifications and alter their contents, and last injecting malicious code at the end of otherwise benign binaries. The results varied depending on the modifications performed. The random changes and byte occlusion proved ineffective against ML-based models n-gram and MalConv, whereas the accuracy of four tested AVs suffered. The injection of malicious code successfully bypassed most classifiers, with the lowest evasion rate of 77% recorded by the 4th AV.

6 Conclusion

We successfully implemented a reinforcement learning approach to adversarial machine learning on the space of PE binaries. We tested numerous agent and environment configurations which we evaluated in two separate experiments. Firstly we focused on generating malware adversarial samples that would evade detection, i.e., exploratory integrity attack. In the first experiment, we achieved an evasion rate of 68.64% by the DQN agent ($\gamma = 0.5, lr = 0.001$) with observation states consisting of 1024 raw bytes from the beginning of PE binary and LightGBM model as target classifier. We compared this result with environment setup mimicking `gym-malware` setup with which we recorded an evasion rate of 60.37%, thus exceeding this setting. Further, we reached an evasion rate of only 13.32% with setup consisting of the DQN agent ($\gamma = 0.75, lr = 0.01$), 2381 features extracted from the binary as the agent's observation space and MalConv classifier as a targeted model. However, when testing the same models against real-world AVs, we accomplished the best result with the last mentioned setup, scoring as high as 47.09% evasion rate against AV-6 and consistently outperforming the other setups.

In the second experiment, we applied the opposite scheme where we created benign adversarial samples, thus increasing the false positive rate of the target model. We recorded an evasion rate of 36.62% with DQN agent ($\gamma = 0.75, lr = 0.01$) with observation space made by 2381 features extracted from the PE binary and MalConv as target model. The lower result was achieved when targeting the LightGBM classifier, only 3.45%. These relatively low results were confirmed when testing against real-world AVs. The highest evasion rate of 14.29% was accomplished by the agent targeting the MalConv classifier. This result was registered against AV-1 with other AVs unaffected by the modifications. The agent targeting the LightGBM model managed to mislead additionally AV-2 and AV-4 but in less than 0.5% of cases. These results indicate that creating false positive samples is far more demanding than the typical approach of creating false negative adversarial samples. This is almost certainly caused by the design of antivirus programs which typically focus on maintaining good accuracy while minimizing false positive rate [20].

6.1 Future Work

In the future, we would like to explore reinforcement learning in more depth, trying different agent implementations or broadening hyperparameter search space. We aim to implement a better protocol for testing the executability of binaries since we have found that our testing method is sensitive, e.g., to installed library version on our virtual machine or folder path of the exe. More work should be devoted to designing modifications of PE files and studying which are responsible for evasion. To improve our results against real-world AVs, we would like to target specific antivirus in the training process directly. An exciting area of adversarial machine learning that we did not cover is retraining the classifier with adversarial samples to increase its resistance against such attacks. Further in the future, we wish to publish the source code for this work as open-source.

Acknowledgments This work was supported by the Student Summer Research Program 2021 of FIT CTU in Prague and by the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16 019/0000765 “Research Center for Informatics” and by the Grant Agency of the CTU in Prague, grant No. SGS21/142/OHK3/2T/18 funded by the MEYS of the Czech Republic.

References

1. H. S. Anderson and P. Roth. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *ArXiv e-prints*, April 2018.
2. Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917*, January 2018.

3. Ömer Aslan Aslan and Refik Samet. A comprehensive review on malware detection approaches. *IEEE Access*, 8:6249–6271, 2020.
4. John Boutsikas, Maksim E. Eren, Charles Varga, Edward Raff, Cynthia Matuszek, and Charles Nicholas. Evading malware classifiers via monte carlo mutant feature discovery, 2021.
5. Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *CoRR*, abs/1606.01540, 2016.
6. E Carrera. Pefile, 2017.
7. Bedang Sen Chandni Vaya. Pesidious, malware mutation using deep reinforcement learning and gans. https://github.com/CyberForce/Pesidious#malware-mutation-using-deep-reinforcement-learning-and-gans, 2020.
8. Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)*, pages 99–106. IEEE, 2017.
9. Sen Chen, Minhui Xue, Lingling Fan, Shuang Hao, Lihua Xu, Haojin Zhu, and Bo Li. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *computers & security*, 73:326–344, 2018.
10. Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Explaining vulnerabilities of deep learning to adversarial malware binaries, 2019.
11. Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware. *IEEE Transactions on Information Forensics and Security*, 16:3469–3478, 2021.
12. Mohammadreza Ebrahimi, Ning Zhang, James Hu, Muhammad Taqi Raza, and Hsinchun Chen. Binary black-box evasion attacks against deep learning-based static malware detectors with adversarial byte-level language model, 2020.
13. Yong Fang, Yuetian Zeng, Beibei Li, Liang Liu, and Lei Zhang. Deepdetectnet vs rlattacknet: An adversarial method to improve deep learning-based static malware detection model. *Plos one*, 15(4):e0231626, 2020.
14. William Fleshman, Edward Raff, Richard Zak, Mark McLean, and Charles Nicholas. Static malware detection amp; subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*, pages 1–10, 2018.
15. Leilani H. Gilpin, David Bau, Ben Z. Yuan, Ayesha Bajwa, Michael A. Specter, and Lalana Kagal. Explaining explanations: An approach to evaluating interpretability of machine learning. *CoRR*, abs/1806.00069, 2018.
16. Kathrin Grosse, Nicolas Papernot, Praveen Manoharan, Michael Backes, and Patrick McDaniel. Adversarial examples for malware detection. In *European symposium on research in computer security*, pages 62–79. Springer, 2017.
17. Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan, 2017.
18. Ling Huang, Anthony D. Joseph, Blaine Nelson, Benjamin I.P. Rubinstein, and J. D. Tygar. *Adversarial Machine Learning*. AISec '11. Association for Computing Machinery, New York, NY, USA, 2011.
19. Microsoft Karl Bridge. Pe format - win32 apps. "https://docs.microsoft.com/en-us/windows/win32/debug/pe-format", 8 2019.
20. Eugene Kaspersky. Doing the homework. https://eugene.kaspersky.com/2012/06/20/fighting-false-positives/, 2012.
21. Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European signal processing conference (EUSIPCO)*, pages 533–537. IEEE, 2018.
22. Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. Deceiving end-to-end deep learning malware detectors using adversarial examples, 2019.

23. Richard Liaw, Eric Liang, Robert Nishihara, Philipp Moritz, Joseph E Gonzalez, and Ion Stoica. Tune: A research platform for distributed model selection and training. *arXiv preprint arXiv:1807.05118*, 2018.
24. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin A. Riedmiller. Playing atari with deep reinforcement learning. *CoRR*, abs/1312.5602, 2013.
25. Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles Nicholas. Malware detection by eating a whole exe, 2017.
26. Hemant Rathore, Sanjay K Sahay, Piyush Nikam, and Mohit Sewak. Robust android malware detection system against adversarial attacks using q-learning. *Information Systems Frontiers*, pages 1–16, 2020.
27. John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *CoRR*, abs/1707.06347, 2017.
28. Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. Mab-malware: A reinforcement learning framework for attacking static malware classifiers, 2021.
29. Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.
30. Romain Thomas. Lief - library to instrument executable formats. https://lief.quarkslab.com/, April 2017.
31. Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.
32. Virusshare dataset. https://virusshare.com/.
33. Virustotal. https://www.virustotal.com/.
34. Christopher John Cornish Hellaby Watkins. Learning from delayed rewards, 1989.
35. Chun Yang, Jinghui Xu, Shuangshuang Liang, Yanna Wu, Yu Wen, Boyang Zhang, and Dan Meng. Deepmal: maliciousness-preserving adversarial instruction learning against static malware detection. *Cybersecurity*, 4(1):1–14, 2021.
36. Yanfang Ye, Dingding Wang, Tao Li, and Dongyi Ye. Imds: Intelligent malware detection system. In *Proceedings of the 13th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1043–1047, 2007.