Revisiting Adversarial Malware Examples with Reinforcement Learning

Raphael Labaca-Castro ✉[1,2], Sebastian Franz*[3], and Gabi Dreo Rodosek[1,2]

[1] Research Institute CODE, 81739 Munich, Germany**
[2] Universität der Bundeswehr München, 85577 Neubiberg, Germany
[3] Technische Universität München, 85748 Munich, Germany
raphael.labaca@unibw.de

Abstract. Machine learning models have been widely implemented to classify software. These models allow to generalize static features of Windows portable executable files. While highly accurate in terms of classification, they still exhibit weaknesses that can be exploited by applying subtle transformations to the input object. Despite their semantic-preserving nature, such transformations can render the file corrupt. Hence, unlike in the computer vision domain, integrity verification is vital to the generation of adversarial malware examples. Many approaches have been explored in the literature, however, most of them have either overestimated the semantic-preserving transformations or achieved modest evasion rates across general files. We therefore present AIMED-RL, Automatic Intelligent Malware modifications to Evade Detection using Reinforcement Learning. Our approach is able to generate adversarial examples that lead machine learning models to misclassify malware files, without compromising their functionality. We implement our approach using a Distributional Double Deep Q-Network agent, adding a penalty to improve diversity of transformations. Thereby, we achieve competitive results compared to previous research based on reinforcement learning while minimizing the required sequence of transformations.

Keywords: Adversarial Learning · Reinforcement Learning · Malware

1 Introduction

Malicious software, known as malware, has been a prevalent digital threat. Large efforts have been conducted to correctly and efficiently detect malicious applications using Machine Learning (ML) [1,2]. However, ML models can be fooled by tricking the classifier into returning the incorrect label [3]. Subtle transformations, referred to as perturbations, inserted into the file can be responsible for misclassification. For this reason, the generation of adversarial malware examples has become an intensive area of research in the last decade [4]. Unlike in

* Work done at Research Institute CODE while a student at LMU Munich
** This research is partially supported by EC H2020 Project CONCORDIA GA 830927

the computer vision domain, where images can be randomly modified with adversarial perturbations, Windows Portable Executable (PE) files can lose their integrity and functionality following a series of too many or strong injections [5].

Recent advances have shown that ML-based malware classifiers report weaknesses when confronted with gradient-based attacks [6]. Generative Adversarial Networks (GAN) were also successful in generating adversarial examples. In this case, a surrogate model was trained based on the target malware classifier. Both approaches rely heavily on feature-space adversarial examples, thus merely producing a representation of the input object rather than a real file [7,8].

Conversely, further research has been looking at the problem-space on the Windows platform. Anderson et al. [9] were one of the first to show that reinforcement learning (RL) can be successfully used to generate adversarial examples in the problem space for Windows PE. Yet, the use of semantic-preserving perturbations can still lead to corrupt adversarial examples. Hence, an integrity verification is paramount to ensure functionality. Further approaches, including Labaca-Castro et al. [10], explored Genetic Programming (GP) with integrity verification that outperforms similar strategies without rendering the files corrupt. However, the inherent issue of getting stuck in local-minima may prevent the system from finding the best sequence of adversarial transformations.

We, therefore, present **AIMED-RL**: Automatic Intelligent Malware modifications to Evade Detection with Reinforcement Learning. This approach combines integrity analysis with improved reinforcement learning techniques. We assume that an attacker use a toolbox with a set of transformations, which can be injected into an original malware file and prevent a ML-based model from properly classifying it as malicious. Our approach shows that RL can be implemented to increase the success rate of such attacks against malware classifiers and is able to outperform previous research in the field by significantly reducing the efforts. This paper is structured as follows: In §2, we take an extensive look at the existing literature about adversarial machine learning in the malware domain focusing mainly on RL. Next, we describe the methodology in §3 and illuminate the design of our reinforcement learning approach. In §4 we present the results and further explores the experiments conducted. Finally, we conclude this work with a short summary in §6.

2 Related Work

Here we evaluate the existing literature about adversarial machine learning in the malware domain. We discuss related research and elaborate on the current state of the field.

2.1 Reinforcement Learning

Reinforcement learning has become an increasing area of scientific interest in the past decade. The usage of RL has extended beyond traditional applications and entered new fields, such as networking and security [11,12,13,14].

One of the first attempts to generate adversarial malware examples using RL was presented by Anderson et al. [9]. The authors implemented ten different perturbations designed to be semantic- and functionality-preserving; that is, they do not negatively affect the structure of the actual code. A maximum budget of ten turns, equivalent to ten injected perturbations, was allowed before the attempt was cancelled and the next episode was started. The reward function only consisted of the detection result by the classifier, where 0 stands for a detected file and 10 for an evasive. The environment was based on the OpenAI gym framework [15]. The article reports an evasion rate of up to 24%, and an average rate of 16.25% over 200 holdout malware examples. According to the authors, the results must be seen as modest in comparison to white-box based gradient attacks or grey-box attempts. Although the perturbations were intended to be functionality-preserving, they did, in fact, hamper the integrity of the adversarial examples since no integrity verification took place.

Building on the same idea, Fang et al., [16] undertook a similar approach using different parameters. The state input was reduced to 513 dimensions and consisted only of byte and entropy histograms. The authors assumed that a smaller and thus more comprehensive input could simplify the training of the agent. The action space was also decreased to four actions, which were expected to maintain improved functionality-preserving properties. A value-based Double Deep Q-Network (DDQN) was trained. In addition, it is reported the use of integrity verification to validate that the created examples remained functional [17]. An evasion rate of 46.56% was reported, which can be considered a strong improvement compared to previous approaches. However, the use of 80 injections based on only four different perturbations could potentially make it easier to detect and identify files that have been respectively modified.

Fang, Zeng, et al., [18] criticized that previous work [9] claimed to use a black-box scenario, while using the same feature-space for the reinforcement learning agent as well as for the detection engine, resembling more a grey-box attack. To avoid this problem, they trained their own classifier to detect PE files using 2,478 dimensions. Still, it remains unclear if this solves the issue since attackers usually have domain knowledge and could be able to anticipate which features are likely to be used by a static malware classifier. They further suggested that the high amount of randomness in the perturbations used in previous work [9,16] could lead to instability during the training process. Instead of picking an import function at random, for instance, they crafted an individual perturbation for each import function. These alterations led to a significantly increased action space with 218 dimensions, which raises the question about whether the agent is able to explore the possible states satisfactorily and register the small differences between the many individual perturbations. A DDQN model in combination with a Dueling DQN (DuDDQN) was implemented and the agent was trained for 3,000 episodes, which then reported an evasion rate of 19.13%.

2.2 Further Approaches

In a stochastic approach [5] perturbations were randomly injected into the malware to explore the potential of automated malware manipulation. An integrity test was implemented using a sandbox [17] to check whether the malware is still functional after the injections. It was reported that an increasing number of injected perturbations reduced the number of functional examples considerably, ranging from 50% functionality for three injected perturbations to only 7.5% for 25. Overall, 18% of manipulated files were reported to be functional on average. The detection was tested by malware scanners on VirusTotal [19]. The best manipulated and functional examples achieved a reduction in the detection rate by about 80%. Interestingly, examples using only five perturbations showed similar results as those relying on an extreme number of 500 perturbations. Moreover, the length of the perturbation sequence proved to be less important than the order of the injected perturbations. However, the approach was slow to find adversarial examples when scaling and, hence, optimization techniques will be needed to improve efficiency.

To address the limitations discussed, another strategy was proposed [10]. This time, a genetic programming algorithm was implemented to find adversarial malware examples. Unlike reinforcement learning, this technique does not require any training time. The fitness function was composed of four parameters, namely functionality, detection, similarity (to the original file on byte-level) and distance (number of generations). Compared to previous approach [5], the current solution was significantly faster, thus requiring less processing time to create files. It also produced fewer corrupted, non-functioning examples. Probing the functional examples against four classifiers, evasion rates of about 24% were reported.

In [20], another similar genetic programming approach to evade static detection was introduced. In this case, the authors only used two perturbations, both of which were meant to be functionality-preserving by design. They, therefore, did not employ an integrity step to check the generated examples for functionality. The perturbations were *padding* (adding bytes at the end of the file) and *section injection*, which were previously used by Anderson et al. and Labaca-Castro et al., among other perturbations. By removing the functionality check, the process of generating adversarial malware was sped up significantly, thus avoiding the most limiting factor regarding the performance in previous approaches [5,10]. However, no evidence was presented to confirm the functionality of the malware, as the files did not appear to be verified a posteriori. In fact, the original perturbations used in [9] were also declared to be functionality-preserving, but turned out to produce corrupted malware as it was acknowledged in the article. Nonetheless, the authors reported that the approach managed to evade, on average, a considerable amount of 12 commercial classifiers.

3 AIMED-RL

In this section, we present how the experiments using reinforcement learning for adversarial malware have been designed. We start providing the theoretical

context and continue defining the experimental settings and environment of our approach.

3.1 Framework & Notation

The idea behind reinforcement learning is to find the best decision or action for a given input *state*. Because every state will be linked to a certain *reward*, a machine learning model based on reinforcement learning is programmed to maximize this reward over a sequence of states. The entity that is deciding and executing the actions is called *agent* in RL-terms. By exploring a lot of possible states over the course of many *episodes*, the agent will eventually learn to link actions and states to some amount of reward. After the exploratory *training* stage, it should now be able to perform the best possible action for each state, leading to the highest reward. This process can be formalized as a *Markov Decision Process* (MDP), consisting of a 4-tuple:

$$MDP := (S, A, \gamma, R(S, A)) \tag{1}$$

For S being a finite set of possible states, A the set of possible actions, γ a discount factor for future rewards and $R(S, A)$ the reward function. A transition from one state to another can formally be described as:

$$(s_t, a_t, r_{t+1}, s_{t+1}) \tag{2}$$

This makes clear that for every timestep (*turn*) t an action has to be chosen by the agent to transition to the next state s. The rule by which the agent decides which action to take is called the *policy* of the agent. It can be formalized as a probability distribution over the available set of actions given a certain state:

$$\pi(a|s) = P[A_t = a|S_t = s] \tag{3}$$

It is necessary to describe the policy of the agent as non-deterministic, because it has some random component in the training stage.

After all, the most important goal in reinforcement learning is for the agent to learn a (near) optimal policy.

Q-learning and Deep Q-learning Q-learning has already been introduced in the early 1990s [21]. Following the reinforcement learning process described in Eq. 1– 3, it becomes clear that each different state has its own value determined by its current reward and the possible future reward that the next states can deliver. The function that ascribes these values to the state is called the *value* function $V(s)$. However, for practical reasons, it is simpler to consider the actions associated with the state transitions. This relation is defined by the eponymous Q-function:

$$Q(s, a) = r(s, a) + \gamma \max_{a' \in A} Q(s', a') \tag{4}$$

The discount factor γ controls hereby, how much the agent is grinding for a long-term reward (high value for γ), or is just greedily considering the current reward (low value for γ). This means that the Q-value is the expected discounted reward for executing action a at state s and following policy π thereafter [21].

To track and update the Q-values, they have to be stored together with their associated state and action pair. In a simple case, with only few states and actions available, a 2D table can be used to accomplish the task. This approach can be regarded similarly to dynamic programming. Even if one only stores the visited and thus relevant states, complexity and storage limits are exceeded very fast. The application of deep neural networks allows to handle this problem.

Instead of directly calculating the Q-values and storing them, a deep neural network with weights θ can be used as a non-linear function approximator. The network can be trained by minimizing the loss function $L(\theta)$ over the course of training episodes with stochastic gradient descent. The size of the output layer of the network must match the number of possible actions. This mimics a supervised learning process, where the reward defines the labeled data for a state. The network has to learn to predict these rewards correctly in order to minimize the loss function. From these predictions, the optimal actions to achieve the highest reward can eventually be inferred [22].

3.2 Experimental Setting

Attacker Knowledge Following previous work [4,23], training data knowledge is defined by $\mathcal{D}$ and feature set $\mathcal{X}$, algorithm g, and hyperparameters w.

Limited Knowledge (LK) Based upon $\theta_{LK} = \{\hat{\mathcal{D}}\}$, attackers can query the model in unlimited fashion and receive binary outputs labelling the adversarial examples into malicious or benign. Moreover, they could also transfer the results of the queries into a surrogate classifier in case the attackers have additional knowledge of the learning algorithm and feature set $\theta_{LK} = \{\hat{\mathcal{D}}, \hat{\mathcal{X}}, \hat{g}, \hat{w}\}$. In our scenario, the LK capability fits the situation appropriately since the agent is only able to assign a reward based on the output of the classifier. None of the underlying architecture from the model nor its training set are relevant for the attackers.

Target Model A LightGBM [24] model is implemented, which was trained on 600,000 benign and malicious software files. In terms of performance, the model scores an ROC-AUC of 0.993 [9]. After analyzing an input file, the classifier returns a value between 0 and 1 for benign or malicious examples respectively. Ergo, a larger value corresponds to a higher confidence in the examined file being malicious. As used in the literature [9,5], for the evaluation, we keep the threshold set to 0.9 to label a file as malware and, hence, be able to benchmark performance against different approaches. Regarding the training stage however, we decided to lower the confidence rate to 0.8, providing a bigger challenge for the RL agent. Note that this threshold is only known to the detection model and

is not used by the RL agent, in order to keep the characteristics of a black-box scenario and attack.

Injection strategy Within the literature, a number of publications used varying numbers to define the maximum of allowed perturbations for the agent, ranging from 10 [9] to 100 [18]. However, in [10] the authors suggested that a smaller number of around five perturbations showed similar results and that the order of perturbations could be more relevant than the actual quantity. We therefore limited the number of allowed injections to five perturbations.

Furthermore, we introduced an additional *reset* strategy to enhance the idea of *order* matters more than *numbers*. Our environment is allowed to reset the malware example back to its original state if the classifier is still able to detect it after five perturbations. This allows the agent a second shot at the same file, if the first attempt failed to successfully generate an adversarial example.

3.3 Environment

State The state input for the agent presented by the environment consists of both, handcrafted PE features extracted from the bytes of the binary file as well as structure-agnostic byte(-entropy) histograms. To extract the PE-specific information, the LIEF library [25] is employed. The state relies on the feature space defined by [9]:

PE-specific features i) *Metadata from PE header file information* (62 dim.): Extracted features from the PE header, such as OS information, linker version or the magic number. ii) *Metadata about the PE sections* (255 dim.): Stores information about section names, sizes and entropy. A hash function is used to compress these values into 255 dimensions for every PE file. iii) *Metadata about Import Table* (1280 dim.): Contains information about the names of imported functions and libraries in the import table of the data directory. The names are stored up to a maximum amount of 10,000 characters. iv) *Metadata about Export Table* (128 dim.): Stores the names of exported functions. The length of the stored value is also limited to 10,000 characters. v) *Counts of human-readable strings* (104 dim.): Counts the number of certain strings like URLs (*https*), registry entries (*HKEY_*) or paths (*c:/*), and creates a histogram that stores the distribution of characters within the strings. vi) *General file information* (10 dim.): General metadata about the file. For instance, whether it has a debug section or a signature, and the length of export and import tables. It also stores the size of the whole file.

Structure-independent features i) *Byte histogram* (256 dim.): Creates a histogram with byte occurrences over the whole binary file. ii) *2D byte-entropy histogram* (256 dim.): To compute the byte-entropy histogram, windows with size 2048 bytes are slided over the raw bytes from the file with a step size of 1024 bytes. For every block created in this way, the entropy is calculated as the base 2

logarithm of the bytes in the block. After that, the byte-entropy histogram is created with these computed values and flattened into a 256 dimensional feature vector. The method is based on the work by [26]; this original work used both a smaller window (1024 bytes) and step size (256 bytes) than applied here.

Both, PE-specific and structure-independent information sum up to a 2,351 dimensional feature vector.

Reward The reward is one of the most important aspects of the environment, as it directly influences the policy of the agent. In our implementation, the reward consists of a linear function of three individual parameters. We hereby decided to set $R_{max} = 10$ as the maximum reward for each.

In [9], only *detection* R_{det} was used to calculate the reward R, returning $R_{det} = 0$ for detected and $R_{det} = 10$ for adversarial examples.

More recent approaches [5,16,18], also included the *distance* R_{dis} from the original file in their reward function. R_{dis} is expressed by the number of turns that have passed. We multiplied it with a factor that gives the maximum number of allowed perturbations $t_{max} = 5$ the highest reward. Thus we incentivised our agents to use our domain knowledge that five perturbations seem to be the most promising in terms of evasion and functionality. R_{dis} can therefore be defined as follows:

$$R_{dis} = \frac{R_{max}}{t_{max}} * t \tag{5}$$

This work further includes the *similarity* R_{sim} of a manipulated file compared to the original one for the reward function, which is inspired by a genetic programming approach [10]. The similarity value is calculated based on a byte-level comparison of the two respective files. A bigger distance between the two files results in a larger value, leading to a higher diversity caused by the injected perturbation.

Given that the value can vary within a larger range, we decided to calculate a ratio between the modified file size, S_{mod}, and the original, S_{orig}, aiming to maintain consistency across the adversarial examples. Based on empirical examination, we determined that a percentage value of $S_{best} = 40\%$ should work best to create the most promising modifications. That is why we calculated R_{sim} according to the difference to this value:

$$R_{sim} = (1 - |S_{best} - \frac{S_{mod}}{S_{orig}}|) * R_{max} \tag{6}$$

In order to be able to tune the model based on importance of each parameter, we introduced weights, ω, for each of the rewards. We therefore present the different weight distributions in Table 1.

The following equation summarizes the reward, R, for our environment:

$$R = R_{det} * \omega_{det} + R_{sim} * \omega_{sim} + R_{dis} * \omega_{dis} \tag{7}$$

Table 1. Weight distribution strategies for the reward function. Standard sets the same weight to each parameter whilst Incremental shifts the attention towards detection.

R_{det}	R_{sim}	R_{dist}	STRATEGY
0.33	0.33	0.33	Standard
0.50	0.20	0.30	Incremental

We have established that agents tend to inject the same perturbation repeatedly and, thus, we introduced a penalization to the reward function. The change consists in a *reward penalty* if the agent uses duplicated perturbations, ρ, within the same file:

$$R = \begin{cases} R & \text{for } \rho = 0 \\ R * 0.8 & \text{for } \rho = 1 \\ R * 0.6 & \text{for } \rho > 1 \end{cases} \tag{8}$$

Actions The agent's task is to decide on each turn which perturbation should be injected into the PE file. The actions are injected sequentially, so that every turn builds on the modified file from the previous injection. The perturbations injected [9], with the exception of *identity* and *create_new_entry_point*, which were left out because of technical problems [5], are described as follows: i) *overlay_append*: Appends a sequence of bytes at the end of the PE file (overlay); length and entropy are random. ii) *imports_append*: Adds an unused function to the import table in the data directory. The function is chosen randomly from a predefined list of DLL imports. iii) *section_rename*: Manipulates an existing section name. For all section perturbations the section name is chosen at random from a list of known benign section names. iv) *section_add*: Creates a new unused section in the section table. v) *section_append*: Appends bytes at the end of a section. The length and entropy of the injected bytes is again chosen at random. vi) *upx_pack*: Uses the UPX [27] packer to pack the whole PE file. Note that the compression level (between 1 and 9) is also chosen at random. vii) *upx_unpack*: Unpacks the file using the UPX packer. viii) *remove_signature*: Removes the signer information in the certificate table of the data directory. ix) *remove_debug*: Manipulates the debug information in the data directory. x) *break_optional_header_checksum*: Modifies and thus breaks the optional header checksum by setting it to 0. Note that the first six perturbations use randomization. The implications of this have already been discussed in section §2.1.

Agent While an Actor Critic model with Experience Replay (ACER) has been used [9] as a policy-based approach for generating adversarial malware examples, it has been shown [16,18] that value-based networks are also suited for RL problems in the malware context. Hence, we implement *Deep Q-Networks* with

additional enhancements [28] that account for data efficiency and performance as can be observed in Table 2.

Table 2. Overview of RL-based approaches and its parameters. While related work implemented ACER, DDQN and Dueling DDQN (DuDDQN), we use a Distributional DDQN (DiDDQN) agent and Noisy Nets as exploration strategy.

Approach	Agent	Optimizer	D. Factor	Exploration
Fang et al., 2019	DDQN	Adam	0.99	$\epsilon - greedy$
F., Zeng et al., 2020	DuDDQN	RMSProp	N/A	Boltzmann
Anderson et al., 2018	ACER	Adam	0.95	Boltzmann
AIMED-RL	**DiDDQN**	**Adam**	**0.95**	**Noisy Nets**

Our experiments with baseline DQN showed a concentrated distribution of Q-values whilst Distributional DQN allowed more precise decisions and improvements in the learning process.

Instead of a regular DQN, we used a *Distributional DQN* [29] with two hidden layers and 64 nodes each and $V_{min} = -10$, $V_{max} = 10$, $N_{atoms} = 51$ that focus in learning the distribution of rewards rather than the expected reward value.

In addition, we implemented a *Double DQN* [30], which is a well-known extension to Q-Learning that solves the problem of action-value overestimation. The neural network uses *Adam* [31] as optimizer with the following parameters: $\alpha = 0.001, \beta_1 = 0.9, \beta_2 = 0.999$ and $\epsilon = 0.01$. Another enhancement was to apply *prioritized experience replay* [32] with $\alpha = 0.6, \beta_0 = 0.4, betasteps = t_{max} * episodes, capacity = 1,000$, instead of usual replay buffers. This allows to select episodes from the replay buffer with higher information for the agent more often, which leads to a more efficient training process. For exploration, we chose *Noisy Nets* [33] with $\sigma = 0.5$ since it allowed for better exploration of the action space and more diversified transformation vectors.

4 Experimental Results

In this section we discuss the results from the experiments. Motivated by limitations from previous work, we focus on answering the following research questions:

RQ1: Is it possible to *increase diversity* in the sequence of adversarial perturbations? (§4.1)

RQ2: Can RL-based agents *efficiently* learn to evade *malware classifiers* with shorter sequences of perturbations? (§4.2)

During the training stage of our agents, we sampled 4,187 portable executable files from VirusShare [34]. The experiments have been evaluated on a holdout set of 200 malware examples that were not included in the training set. The integrity is verified by executing the adversarial example in a protected environment [17].

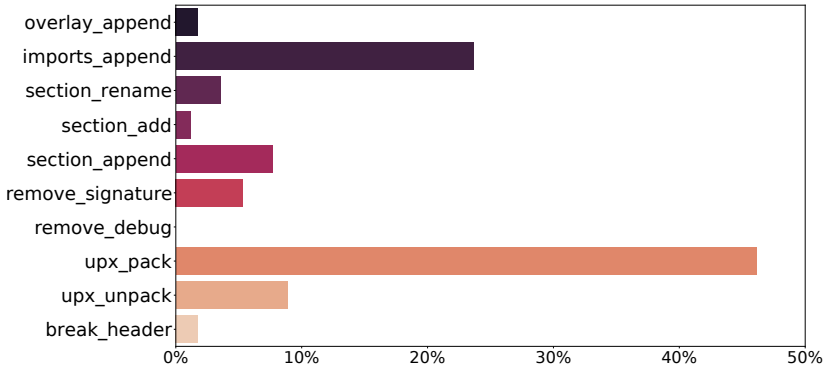

Fig. 1. Usage of perturbations of best agent to create adversarial files. While two perturbations are particularly dominant, a broad range of actions can be observed.

4.1 Diversity of Perturbations

As we can observe in Figure 1, the agent employs a broad variety of perturbations throughout the evaluation to generate adversarial examples. In line with previous research [9,18,20], *upx_pack* turned out to be the most dominant perturbation in our environment.

Table 3. Comparison of evasion rates among agents using two different strategies: incremental and standard weights with (WP) and with no penalty (NP). An additional set was included to compare RL-based agents with random results.

Strategy	Episodes	Avg. Evasion	Best Agent
Incremental (WP)	1000	23.52%	40.00%
	1500	20.35%	33.84%
Incremental (NP)	1000	18.78%	30.81%
	1500	23.06%	35.35%
Standard (WP)	1000	21.07%	35.86%
	1500	26.29%	43.15%
Standard (NP)	1000	21.6%	30.0%
	1500	23.74%	41.41%
Random Agent	—	21.21%	24.62%

Since packing strongly impacts the structure of the file and this is an important feature for static malware classifiers, its dominance over other perturbations appears reasonable. In fact, packing is a common practice amongst commercial (benign) software vendors to obfuscate their code or to reduce the size of their executable files. Further research [20] suggests that these kind of perturbations

(i.e., packing) increase the probability of a file to be flagged as malicious. However, from the classifier perspective, considering compressing with UPX packing a malicious behavior itself would necessarily increase the number of false-positive results and is therefore not encouraged. At this point, we must note that some attacks will always be more prominent and, therefore, we advise to focus on the diversity of perturbations instead of concentrating on the most dominant. Most of the agents created used heterogeneous sequences of perturbations, indicating the success of our enhancements to the environment.

While some transformations may be more prevalent, others may be flagged by security techniques such as pre-analysis. For instance, *overlay_append* can indeed be a strong sign of a modified, probably malicious, file. For a benign PE it would be unusual to have bytes randomly appended after the overlay, as these would only increase its file size without adding any value. Packing or unpacking the file before presenting it to the ML-model could also be applied as a pre-analysis technique to avoid packers to fool the classifier.

4.2 Evasion Rate

In Table 3 we observe the comparison of evasion results among agents taking into account different strategies. In each case, 10 agents were trained for 1000 and 1500 episodes respectively. A random agent has also been added to compare the generation of adversarial examples with the use of reinforcement learning. Note that some combinations of perturbations can render the adversarial examples corrupt. These files were excluded from both the training and the evaluation sets, resulting in non-uniform values for some evasion rates. While the best *average evasion rate* improvement scores 7%, the best *agent* is improved by more than 20%. Both weight distributions returned agents that scored significantly better than a purely random approach. The best agents, however, were trained with the help of our reward penalty strategy. In fact, the agents implementing the penalty technique always outperform their non-penalized counterparts, as depicted in Figure 2 where a comparison among four different configuration of agents is displayed.

The best agent, trained within $1,500$ episodes using *standard* strategy with penalty, managed to score an evasion rate of 43.15% on the holdout set. The adversarial examples created were 97.64% functional, leading to an overall evasion rate of 42.13%. Even considering the reset strategy that de-facto doubles the amount of episodes to $3,000$, this number of episodes is still arguably small. In fact, the agent was updated only about 13,900 times during training. This is equivalent to the number of modifications created during training and is considerably lower than the budget of 50,000 modifications used by [9] in their previous approach. Thus, such an agent can be trained without highly powerful hardware in a short period of time.

Table 4 summarizes the results of AIMED-RL compared to previous work. While Fang et al. [16] report a higher evasion rate of 46.56% with functional files, it is important to note that we were not able to reproduce these results given that no artifact was made available.

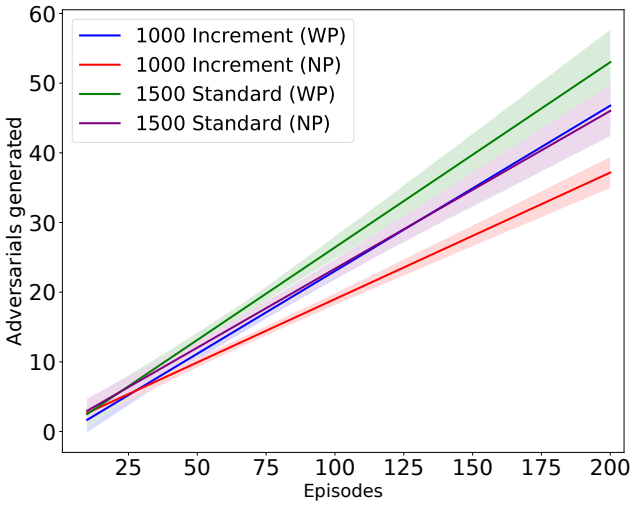

Fig. 2. Comparing our best results regarding the reward penalty strategy. The lines represent the average of generated adversarial files over ten trained agents. For both numbers of training episodes, the agents with the penalty outperform their counterparts. This concerns both the best and the average evasion rate.

We experienced a similar situation with F., Zeng et al. [18]. In this case only the malware set was published.

Table 4. Comparison of evasion results of AIMED-RL against different approaches in the literature. The LGBM model is employed widely across the literature and serves as a benchmark. Only one approach implemented DeepDetectNet (DDNet), which makes their evasion rates less comparable. The functionality test (FT) returns a binary output.

Approach	Space	Reward	Perts.	Model	FT	Evasion
Fang et al., 2019	4	R_{det}, R_{dist}	80	LGBM	Yes	46.56%
F., Zeng et al., 2020	218	R_{det}, R_{dist}	100	DDNet	Yes	19.13%
Anderson et al., 2018	11	R_{det}	10	LGBM	No	16.25%
AIMED-RL	**10**	$R_{det}, R_{dist}, R_{sim}$	**5**	**LGBM**	**Yes**	**42.13%**

Therefore, in order to evaluate the data, we proceeded to acquire their pool of malware files and modified them with our best agent. By adding only five perturbations we were able to get 42 out of 50 to be functional, ergo 84%. In their work, however, they were injecting up to 100 transformations. Even if their results keep the functionality rate that we had with five perturbations, the evasion rate with our integrity tests would be around 16.07%. Since this is an extrapolation we

do not strive for formal comparison. Nevertheless, we believe these reproducible steps are important given that highly-perturbed PE files are reported to break after a large number of modifications [5]. This argument similarly applies to [16] which also used a high budget of 80 perturbations.

On the other hand, further approaches [16,18] also reported the use of IDA Pro [35] to generate control flow graphs as a means of checking whether the examples still showed the exact same behavior. Although this approach may seem compelling, in order to be thoroughly implemented, it is likely to require manual verification and, therefore, strongly increase the cost of generating fully-functional adversarial examples. Regarding the remaining approach by Anderson et al. [9], while the environment was published, integrity verification did not take place.

With respect to the reward, the distribution of the parameters contributed in different degrees towards the total reward. *Similarity* accounts for 24%, *Detection* 32%, and *Distance* 44%. In our approach, distance is updated on every turn and hence has stronger role. However, further room for optimization may still be available in terms of how parameters are updated.

Overall, the agent that reports the best evasion result on the holdout dataset needs $1,500$ episodes of training with *standard* strategy and *penalization* activated. Unlike what can generally be observed in the literature, the evasion rate for successful adversarial examples seems to improve by a better combination of small factors rather than a larger sequence of adversarial perturbations.

5 Availability

In order to foster further research in this area we are releasing AIMED-RL[4]. While we are aware that the work could be misused by adversaries, we believe that enforcing security to protect from adversarial examples outweighs the potentially negative impact. Malicious actors have available resources to generate sophisticated attacks and even legitimate software can be exploited by committed adversaries. However, releasing the code to the community can enable researchers to protect towards adaptive attacks and therefore increase the level of defenses against adversarial malware.

6 Conclusion

In this paper we presented AIMED-RL, which aims to extend the capabilities of existing approaches to generate fully functional adversarial examples in the malware domain. We redefined the reward function and evaluated different weight strategies to maximize the output. To address the limitation of homogeneous sequences of perturbations, which are a widely discussed limitation in reinforcement learning approaches, we introduced and demonstrated the importance of a penalty technique. Moreover, we showed that is possible to train a competitive

[4] https://github.com/zRapha/AIMED

agent that generates adversarial examples with a shorter sequence of transformations, which leads to less manipulated adversarial malware, without compromising its functionality.

References

1. Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. Survey of machine learning techniques for malware analysis. *Computers & Security*, 81:123–147, 2019.
2. Edward Raff and Charles Nicholas. Survey of machine learning methods and challenges for windows malware classification. *arXiv:2006.09271*, 2020.
3. Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv*, 2013.
4. Battista Biggio and Fabio Roli. Wild patterns: Ten years after the rise of adversarial machine learning. *Pattern Recognition*, 84:317–331, 2018.
5. Raphael Labaca-Castro, Corinna Schmitt, and Gabi Dreo Rodosek. Armed: How automatic malware modifications can evade static detection? In *2019 5th International Conference on Information Management (ICIM)*, pages 20–27, 2019.
6. Raphael Labaca-Castro, Battista Biggio, and Gabi Dreo Rodosek. Poster: Attacking malware classifiers by crafting gradient-attacks that preserve functionality. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, pages 2565–2567, 2019.
7. Weiwei Hu and Ying Tan. Generating adversarial malware examples for black-box attacks based on gan. *ArXiv*, 2017.
8. Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo Rodosek. Poster: Training gans to generate adversarial examples against malware classification. *IEEE Security and Privacy*, 2019.
9. Hyrum S. Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. Learning to evade static pe machine learning malware models via rl. *ArXiv*, 2018.
10. Raphael Labaca-Castro, Corinna Schmitt, and Gabi Dreo Rodosek. Aimed: Evolving malware with genetic programming to evade detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*, pages 240–247, 2019.
11. Tong Chen, Jiqiang Liu, Yingxiao Xiang, Wenjia Niu, Endong Tong, and Zhen Han. Adversarial attack and defense in reinforcement learning-from AI security view. *Cybersecurity*, 2(1):11, 2019.
12. N. C. Luong, D. T. Hoang, S. Gong, D. Niyato, P. Wang, Y. Liang, and D. I. Kim. Applications of deep reinforcement learning in communications and networking: A survey. *IEEE Communications Surveys Tutorials*, 21(4):3133–3174, 2019.
13. Thanh Thi Nguyen and Vijay Janapa Reddi. Deep reinforcement learning for cyber security. *arXiv preprint arXiv:1906.05799*, 2019.
14. Y. Qian, J. Wu, R. Wang, F. Zhu, and W. Zhang. Survey on reinforcement learning applications in communication networks. *Journal of Communications and Information Networks*, 4(2):30–39, 2019.
15. Greg Brockman, Vicki Cheung, Ludwig Pettersson, Jonas Schneider, John Schulman, Jie Tang, and Wojciech Zaremba. Openai gym. *ArXiv*, 2016.
16. Z. Fang, J. Wang, B. Li, S. Wu, Y. Zhou, and H. Huang. Evading anti-malware engines with deep reinforcement learning. *IEEE Access*, 7:48867–48879, 2019.

17. Claudio Guarnieri, Alessandro Tanasi, Jurriaan Bremer, and Mark Schloesser. Cuckoo sandbox - automated malware analysis. *Cuckoo*, 2021.
18. 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.
19. VirusTotal. Analyze suspicious files and URLs to detect types of malware, automatically share them with the security community. `https://virustotal.com`, 2021, last access Feb. 25, 2021.
20. Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. Functionality-preserving black-box optimization of adversarial windows malware. *ArXiv*, 2020.
21. Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine learning*, 8 (1992):279–292, 1992.
22. Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *ArXiv*, 2013.
23. Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian J. Goodfellow, Aleksander Madry, and Alexey Kurakin. On evaluating adversarial robustness. *CoRR*, abs/1902.06705, 2019.
24. Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. Lightgbm: A highly efficient gradient boosting decision tree. In I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, pages 3146–3154. Curran Associates, Inc, 2017.
25. Quarkslab. Lief: Library to instrument executable formats. *QuarksLab*, 2020.
26. Joshua Saxe and Konstantin Berlin. Deep neural network based malware detection using two dimensional binary program features. *ArXiv*, 2015.
27. Markus F.X.J. Oberhumer, László Molnár, and John F. Reiser. Upx: the ultimate packer for executables - homepage. *GitHub*, 2020.
28. Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 32(1):3215–3222, 2018.
29. Marc G. Bellemare, Will Dabney, and Rémi Munos. A distributional perspective on reinforcement learning. *ArXiv*, 21.07.2017.
30. Hado van Hasselt, Arthur Guez, and David Silver. Deep reinforcement learning with double q-learning. *Proceedings of the AAAI Conference on Artificial Intelligence*, 30(1), 2016.
31. Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *ArXiv*, 2014.
32. Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. *ArXiv*, 2015.
33. Meire Fortunato, Mohammad Gheshlaghi Azar, Bilal Piot, Jacob Menick, Ian Osband, Alexander Graves, Vlad Mnih, Remi Munos, Demis Hassabis, Olivier Pietquin, Charles Blundell, and Shane Legg. Noisy networks for exploration. In *Proceedings of the International Conference on Representation Learning (ICLR 2018)*, Vancouver (Canada), 2018.
34. VirusShare. VirusShare: A repository of malware samples for security researchers. `https://virusshare.com`, 2021, last access Mar. 12, 2021.
35. Hex-Rays. IDA Pro: A powerful disassembler and a versatile debugger. `https://www.hex-rays.com/products/ida/`, 2021, last access Mar. 29, 2021.