FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	Application of Reinforcement Learning to Creating Adversarial Malware Samples
Student:	Bc. Matouš Kozák
Supervisor:	Mgr. Martin Jureček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Science
Department:	Department of Theoretical Computer Science
Validity:	until the end of summer semester 2022/2023

Instructions

The aim of the adversarial machine learning technique is the deliberate modification of input data, which causes a reduction in the accuracy of classification. Data modification for the malware detection problem can be done, either at the level of the feature vector or at the level of the samples themselves from which the feature vectors are extracted. The latter modification is technically more demanding but more applicable in practice. The goal of this work would be to implement a chosen method of adversarial learning and test its effectiveness on some malicious code detection systems.

Instructions:
1. Study techniques of adversarial machine learning, focusing mainly on the area of malware detection.
2. Implement chosen technique of adversarial attack and apply it at the level of the samples.
3. Test effectiveness against some malware detectors.
4. Discuss the results and compare them with related work.

Electronically approved by doc. Ing. Jan Janoušek, Ph.D. on 24 January 2022 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master's thesis

Application of Reinforcement Learning to Creating Adversarial Malware Samples

Bc. Matouš Kozák

Department of Theoretical Computer Science
Supervisor: Mgr. Martin Jureček, Ph.D.

January 3, 2023

Acknowledgements

First of all, I want to thank my thesis supervisor Mgr. Martin Jureček, Ph.D., for his guidance and support while working on this thesis and in other academic decisions as well. Secondly, I would like to thank my family for their unlimited support in my life, not only in this academic journey.

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46 (6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity.

In Prague on January 3, 2023 . . . . . .

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Kozák, Matouš. *Application of Reinforcement Learning to Creating Adversarial Malware Samples.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Strojové učení se díky svým prvotřídním výsledkům v mnoha oblastech stává stále více populárnější pro řešení nejrůznějších problémů. Díky tomu vývojáři antivirů začínají začleňovat modely strojového učení i do svých produktů. I když tyto modely zlepšují schopnosti detekce antivirových programů, mají také své nevýhody v podobě citlivosti na adversariální útoky. Ačkoli tato citlivost byla prokázána u mnoha modelů při white-box útocích, pro oblast detekce malwaru je black-box útok využitelnější v praxi. Proto představujeme black-box útok, kde má útočník k dispozici pouze výsledek predikce a vzdává se jakýchkoli dalších informací o cílovém klasifikátoru. S využitím algoritmů zpětnovazebního učení jsme implementovali útok proti GBDT klasifikátoru natrénovaném na EMBER datasetu. Natrénovali jsme několik zpětnovazebních agentů na datové sadě malwaru pro operační systém Windows. Při modifikování jsme kladli velký důraz na zachování původní funkčnosti škodlivých vzorků. Dosáhli jsme úspěšnosti zmýlení cílového klasifikátoru v 58,92 % s využitím PPO algoritmu. Kromě toho, že jsme cílili na tento detektor, jsme studovali, jak se adversariální útok může přenést na jiné modely. Agent dříve natrénovaný proti GBDT klasifikátoru zaznamenal úspěšnost v 28,91 % případů proti MalConv, což je model založený čistě na strojovém učení. Vygenerované adversariální vzorky jsme také otestovali proti špičkovým AV programům a dosáhli jsme úspěšnosti zmýlení v rozmezí od 10,24 % do 25,7 %. Tyto výsledky dokazují, že nejen modely založené pouze na strojovém učení jsou náchylné k adversariálním útokům a že je třeba přijmout lepší opatření k ochraně našich systémů.

Klíčová slova adversariální vzorky, zpětnovazební učení, detekce malwaru, PE soubory, statická analýza, strojové učení

Abstract

Machine learning is becoming increasingly popular as a go-to approach for many tasks due to its world-class results. As a result, antivirus developers are starting to incorporate machine learning models into their products. While these models improve malware detection capabilities, they also carry the disadvantage of being susceptible to adversarial attacks. Although this sensitivity has been demonstrated for many models in white-box settings, a black-box attack is more applicable in practice for the domain of malware detection. Therefore, we present a black-box scenario in which the attacker only has the predicted label at his disposal and forgoes any other information about the target classifier. Using reinforcement learning algorithms, we implemented an attack against the GBDT classifier trained on the EMBER dataset. We trained several RL agents on a dataset of Windows malware with an emphasis on preserving the original functionality of the malicious samples. We achieved an evasion rate of 58.92% against the targeted classifier using the PPO algorithm. In addition to targeting this detector, we studied how the adversarial attack can be transferred to other models. The agent previously trained against the GBDT classifier scored an evasion rate of 28.91% against MalConv, a model based solely on machine learning. We also tested the generated adversarial examples against top AV programs and achieved an evasion rate ranging from 10.24% to 25.7%. These results prove that not only machine learning-based models are vulnerable to adversarial attacks and that better safeguards need to be taken to protect our systems.

Keywords adversarial samples, reinforcement learning, malware detection, PE files, static analysis, machine learning

Contents

List of Figures

List of Tables

Introduction

Malicious soft**ware**, also known as **malware**, conducts unwanted actions on infected systems. Protection of our devices is paramount as more and more of our lives are in the digital world. Cybersecurity professionals are developing new defence mechanisms to improve the detection capabilities of their antivirus (AV) programs. However, their opponents are advancing at the same, if not faster, rate, making the problem of malware detection a never-ending battle between attackers and antivirus developers.

According to the AV-TEST institute, more than 450,000 new malware samples are registered daily, totalling more than 150,000,000 new malicious programs in 2021 [1]. Nowadays, attackers are not focusing only on Windows devices, but other platforms such as Linux, Mac or Android are also targeted. However, Windows remain the go-to target for most attackers [2].

Mydoom from 2004 is considered the most harmful malware in history, with an estimated damage of 38 billion dollars. This program spread itself by emails and added the infected computers into a net of computers (botnet), performing DDoS (distributed denial of service) attacks on various institutions. In its heyday, this malware was responsible for 25% of all email traffic worldwide. A more recent example is the WannaCry ransomware from 2017. Ransomware is a type of malware that encrypts files on a victim's computer and demands a ransom for decrypting them. WannaCry spread rapidly around the world, infecting over 200,000 computers across 150 countries with an estimated impact of four billion dollars. Among the victims were important organizations such as FedEx, NHS in the United Kingdom, O2 Telefónica in Spain or the German railway company Deutsche Bahn [3].

State-of-the-art antivirus programs incorporate both static and dynamic analysis in their inner workings. Traditional static analysis methods are based on byte sequences (signatures) stored in a database. Signatures reliably identify known malicious files and are time-efficient. The main weakness of the signature-based approach is that it cannot classify zero-day malware, and even slight modifications to malware samples can cause the signature to change,

making them undetectable by AVs. On the other hand, the dynamic analysis consists of behaviour-based techniques that look for behavioural patterns and can partially detect unknown and obfuscated malware samples, but with the added inefficiency of running malware in a secure environment [4].

Using malware detection models based on machine learning (ML) yields promising results [5]. ML-based detectors are trying to bridge the gap between traditional static and dynamic analysis in the area of detecting unknown malware. Nonetheless, ML models are susceptible to adversarial attacks that can mislead the models [6]. For example, a minor modification of a malware file can make its feature vector resemble some feature vectors of benign files. Consequently, this can cause the malware classifier to make an incorrect prediction. Therefore, relying on only one type of defence mechanism to stop incoming threats is currently not an acceptable option for antivirus developers.

The goal of this thesis is to implement a technique of adversarial attack at the level of samples, i.e., a technique that would create functional adversarial samples. This task is considerably more demanding, as typical machine learning models operate at the level of feature vector, and reliable reverse mapping from a feature vector back to a binary file is difficult to perform. We chose to use techniques based on reinforcement learning. These techniques require an environment and an agent to be implemented. The environment consists of a manipulator capable of altering binary files and various methods for interacting with the agent. We need to make the modifications at the binary level and ensure that the original functionality remains untouched. For this task, we present a method comparing behaviour reports before and after modification.

Our adversarial attack works in a black-box scenario, meaning that no information about the target classifier apart from the final prediction label is known to the attacker. We train reinforcement learning agents to modify Windows malware binaries with the goal of bypassing detection by the targeted machine learning classifier. Additionally, we test the transferability of our adversarial attacks to another ML classifier. The trained agents are later tested against professional antivirus programs.

In this thesis, we target our attack on static malware analysis for numerous reasons. Firstly, dynamic analysis requires executing malware inside a secure sandbox and recording its behaviour, which is both time and technically demanding. Secondly, malware authors can incorporate sandbox evading techniques to detect that their malware is running inside a controlled environment and stop its malicious behaviour [7, 8]. Thirdly, static detection is usually the first line of defence against unwanted threats and is thus a critical part of any antivirus program.

This work is an extension of our previous research project [9] carried out as part of the Student Summer Research Program 2021 of FIT CTU in Prague.

The outline of the thesis

- In Chapter 1, we establish the necessary background. Starting with an introduction to adversarial machine learning, continuing with a brief dive into reinforcement learning and finishing with a detailed description of the portable executable file format.
- In Chapter 2, we define our method in detail. From modification of binary files and ensuring that they retain their original functionality to describing our reinforcement learning environment and agents.
- In Chapter 3, we introduce our experimental setup and routine. We describe the evaluation metrics and datasets used, and we present the results achieved.
- In Chapter 4, we display related work focusing on the area of adversarial malware generation and discuss how our results compare with state-of-the-art methods.
- In Conclusion, we summarize the contributions of our work, address the shortcomings of our approach to generating adversarial malware examples and suggest ideas for future research.

Chapter 1

Background

In this chapter, we outline the necessary background to comprehend this thesis. Firstly, we briefly introduce adversarial machine learning. Then we follow by describing the fundamental principles of reinforcement learning, and we finish by describing the portable executable file format in detail.

1.1 Adversarial Machine Learning

In recent years, we can see the ever-growing popularity of machine learning algorithms in many domains, such as advertisement recommendation, image classification, or playing Go, where the ML models achieve state-of-the-art results [10, 11]. However, in other areas, such as self-driving cars or disease diagnostics, both the general public and researchers are still reluctant to trust the decisions of these models [12, 13]. One of the possible reasons for doubting ML models is the unexplainable nature of their decisions, and the subsequent potential fragility and bias of the entire system [14]. Consequently, ML systems can be sensitive to small changes exploited by adversarial attacks [6].

Adversarial machine learning is an area of machine learning focusing on improving ML systems to withstand adversarial attacks both from outside (evasion attacks) and inside (data poisoning). An adversarial attack is a carefully created action to mislead the ML model. The victim model is also called a target model, and the attacker is called an adversary. Nevertheless, both attacker and adversary are used interchangeably in the current literature. The object responsible for misleading the target model is referred to as an adversarial sample. The following sections describe the taxonomy of adversarial attacks focusing on the domain of malware detection and possible defence mechanisms against these attacks.

1.1.1 Taxonomy

In this section, we describe the taxonomy of adversarial machine learning. We follow the taxonomy presented by Huang et al. [15] since it is one of the most comprehensive and security-related descriptions on this topic. The adversarial attacks are described by three main properties: influence, security violation and specificity.

Influence The first property describes adversaries' capabilities when attacking a given model. The first category is called *causative* attacks, where the attacker can influence the training process of the model, e.g., camouflage wrongly labelled malware samples inside the training dataset (data poisoning). The second category is called *exploratory* attacks. These attacks do not affect the training phase and thus cannot alter the model itself. Their goal is to find information about the model and bypass its detection mechanisms, e.g., malicious file evading detection and thus infecting the user's device (evasion attack).

Security violation The second property characterizes the type of security violation caused by the adversary. If the adversarial attack is causing an increase in the model's false negative rate, we call it an *integrity* attack, e.g., adversarial malware samples are classified as benign. When the attack causes an increase in both false negative and false positive rates, thus making the model unusable for any prediction, it is called an *availability* attack. The last type is a *privacy* attack, where the goal is to steal the model's confidential information, such as the training dataset or configuration of the model. This type of attack is sometimes also called a model stealing attack.

Specificity Third property indicates the scale of the adversarial attack. Suppose the attack is targeted at a small and specific set of samples. In that case, we denote it as a *targeted* attack, e.g., a particular malware program bypassing the detection and infecting the device. When the goal is the misclassification of any sample, the attack is called *indiscriminate*.

The properties can be combined together to better describe a specific adversarial attack. For example, an exploratory-integrity-targeted attack could be an attack where the adversary modifies a small set of malware programs to bypass detection by the targeted antivirus, increasing the false negative rate of the classifier. Another example could be a causative-availability attack where the adversary hides a group of mislabelled benign and malware files inside the training dataset, therefore decrementing the targeted model's accuracy.

The success of an adversarial attack is dependent on the available knowledge of the targeted system. When the adversary has access to the system and can examine its internal configuration or training datasets, we call

this a **white-box** scenario. On the other hand, if the adversary has limited knowledge, usually only in the form of the model's final prediction, e.g., malware/benign label for each submitted sample, we call this a **black-box** scenario. In between these two is a **grey-box** scenario where the attacker has higher access to the system than in the black-box scenario, but the access is still limited to some parts. For example, the attacker can use the model's score or feature space but cannot access and modify its training dataset.

In real-world scenarios, not all types of attacks are feasible in the domain of malware detection. For example, performing a causative attack against a commercial antivirus program is extremely difficult because the training datasets are well-guarded secrets in most security companies. However, an insider in the development of secure AV could perform a white-box adversarial attack to explore all possible vulnerabilities inside the system by utilizing the maximum knowledge possible.

1.1.2 Defence Mechanisms

The most well-known strategy to defend models against adversarial samples is **retraining**. This defence approach needs a set of correctly labelled adversarial samples, which are then used in the model's training stage. Retraining the model with these samples can increase the its ability to detect them. This method was proven to be effective [16, 17, 18] but has its shortcomings when used thoughtlessly. If the quantity of adversarial samples is high, it can cause a shift in the distribution of samples and worsen the performance of the classifier [19].

The second defence mechanism rests on misleading the adversary to craft non-adversarial samples. One of the methods is identifying the incoming file as a probe from the adversarial sample generator and blocking or creating a fake response to fool the generator. The recognition can be done by identifying suspicious IP addresses or finding repeated patterns in the history of queries. Another method was presented in [20], where the target classifier is in the form of an ensemble of models. The predicting model is chosen dynamically based on internal strategy, thus introducing uncertainty to the attacker.

The third method of defence is designing models that do not depend on weak features. Sometimes models can use insignificant parts of the feature vector for their decisions [21]. The interpretability of ML models is a problematic area. Choosing the right features representing the input file and changing accordingly if the file has been modified is vital to preventing adversarial attacks.

1.2 Reinforcement Learning

Reinforcement learning (RL) is a branch of machine learning where an *agent* equipped with a set of actions is learning how to reach its goal. The

agent can be a bot learning to play a computer game or a physical robot working in a factory. The agent learns which actions are "good" and "bad" in the current situation based on trial and error and appropriate feedback from an interactive element called *environment*. This section is based on the book *Reinforcement Learning: An Introduction* by Richard S. Sutton and Andrew G. Barto [22], where you can find more details and examples on this topic.

Reinforcement learning is a separate area of machine learning since, in contrast with *supervised learning*, it does not try to learn from labelled training sets but rather from interaction with the environment. Although it may seem like it, RL is also not a subclass of *unsupervised learning*, as it does not try to find hidden patterns in the training sets but instead maximises the total reward provided as feedback from the environment. The crucial challenge for reinforcement learning is a balance of *exploration* and *exploitation*. This problem is an integral part of any RL algorithm, how to explore enough of the environment while maximising the reward and reaching its goal.

In addition to the agent and environment mentioned above, there are three other key elements of reinforcement learning: the agent's policy, the reward signal, and the value function. A model of the environment is also included for some tasks.

The core part of any reinforcement learning agent is its **policy**. It represents the agent's behaviour at a given time. It is a function mapping from the states of the environment to an action from a set of agent actions. The policy can be not only in the form of a lookup table or a simple function but also in the form of an advanced search algorithm. If learned correctly, it should lead to a strategy that maximises the total rewards the agent receives.

The **reward signal** is an immediate response to a taken action provided by the environment. This signal grades action taken at a given state as good or bad concerning the agent's goal. The function calculating the reward signal is a part of the environment and not under any condition it can be altered by the agent.

The **value function** estimates how rewarding the current state is. The ultimate goal of every RL agent is to achieve the highest total reward, also called return. This goal usually cannot be accomplished by following states and actions with the highest immediate rewards but rather with the highest values, as these maximise the cumulative reward. Whereas the reward signal is usually an easily computable function contained in the environment, the value function must be learned (recalculated) based on past observations during training. Along with the policy, it is an essential part of any RL algorithm.

The optional last part of some RL systems is the **model of the environment**. This model is a reproduction of the environment that simulates future states and rewards. The model provides additional input for the agent to decide on future actions. If a complete model of the environment is known to the agent, the optimal solution to the problem can be found using dynamic programming techniques. Needless to say, this can be computationally infeasible.

If an RL agent requires a model of the environment, we call it a model-based algorithm. Otherwise, we use the term model-free.

Although there are other formal definitions of reinforcement learning, in this work, we follow the one presented in [22]. Reinforcement learning can be defined as repeated interactions between agent and environment at discrete time steps $t = 0, 1, 2, \dots T$. At time step t, the environment is at a state $S_t \in S$ where S is a set of all possible states. After the agent is presented with the state S_t, based on its policy π, creates a mapping $S_t \to A_t \in A(S_t)$, where $A(S_t)$ is a set of all possible actions at state S_t. In many scenarios, $A(S_t)$ can change based on the current state S_t, but in others, it can remain fixed depending on the environment. After deciding on the action A_t, the chosen action is sent to the environment where it gets executed. The subsequent response from the environment gets presented to the agent in the form of a new state S_{t+1}, and reward $R_{t+1} \in R \subset \mathbb{R}$, where R is the set of all possible rewards. Figure 1.1 illustrates the interaction between the agent and the environment.

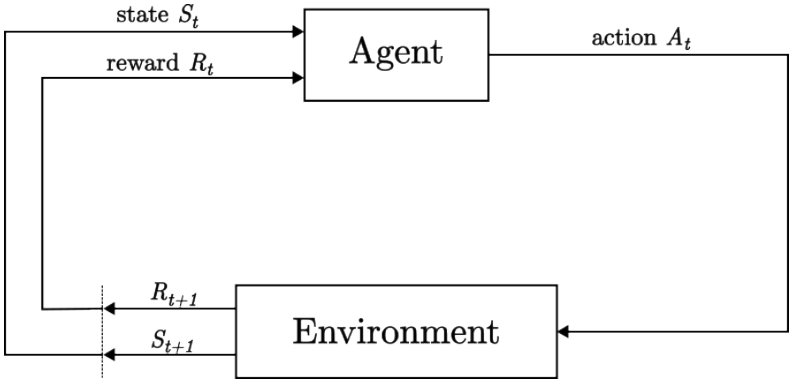

Figure 1.1: Interaction between agent and environment.

We call the exchange of actions, states and rewards between the agent and the environment across time steps $t = 0$ and $t = T$ an *episode*. One episode can be characterised by the following sequence ending in the terminal state S_T, $S_0, A_0, R_1, S_1, A_1, \dots, R_T, S_T$. Subsequently, the environment is reset, and a new independent episode begins.

As stated before, the agent's goal is to maximise the total of rewards $G_t = R_{t+1} + R_{t+2} + \cdots + R_T$, also called expected *return*. For the computation of expected return G_t, it is common to use a technique called discounting. This technique allows control over how far into the future agent should look, i.e., how much value it should assign to the future states. We calculate *discounted return* as follows:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots + \gamma^{T-t-1} R_T = \sum_{i=0}^{T-t-1} = \gamma^i R_{t+i+1} \quad (1.1)$$

where $0 \leq \gamma \leq 1$ is a parameter called *discount rate*. If $\gamma = 0$ agent only considers immediate reward, and the closer the γ is to 1, the higher value the agent gives to the future states.

As mentioned before, the value function represents an estimation of future rewards for a given state. Formally, we distinguish between two value functions, state-value and action-value functions. The **state-value** function portrays the expected future return for an agent in a state s and is defined as:

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] \tag{1.2}$$

where $\mathbb{E}_\pi[\cdot]$ is the expected value of a random variable with regard to policy π. Similarly, we define the **action-value** function that represents the expected return for state s when action a is performed according to policy π as follows:

$$q_\pi(s, a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] \tag{1.3}$$

The relationship between the value of the current state and its subsequent states can be described by the *Bellman equation*. It decomposes the value of the current state into the immediate reward and the value of the expected future state:

$$\begin{aligned} v_\pi(s) &= \mathbb{E}_\pi[G_t|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + \cdots |S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma(R_{t+2} + \gamma^2 R_{t+3} + \cdots)|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma G_{t+1}|S_t = s] \\ &= \mathbb{E}_\pi[R_{t+1} + \gamma v_\pi(S_{t+1})|S_t = s] \end{aligned} \tag{1.4}$$

Both state-value and action-value functions are usually calculated from past interactions. By following a given policy π and calculating average future returns for every state, the agent can eventually estimate a reasonable value of v_π. In the following equations, we denote estimation that converges to v_π as V, and it can be calculated as follows:

$$V(s) = \frac{\sum_{t=0}^{T} \mathbf{1}[S_t = s]G_t}{\sum_{t=0}^{T} \mathbf{1}[S_t = s]} \tag{1.5}$$

where $\mathbf{1}[S_t = s]$ equals 1 if $S_t = s$ and 0 otherwise. Likewise, if the agent keeps separate counts for each action at a given state, it can approximate the action-value function q_π, later denoted as Q.

The averaging update of the state-value function shown in Equation (1.5) is appropriate for environments which do not change in time. However, it is often necessary to give more weight to recent experiences than to old ones. The update then takes the following form:

$$V(S_t) = V(S_t) + \alpha\Big(G_t - V(S_t)\Big) \tag{1.6}$$

where α is the learning rate, G_t is the target value, and $V(S_t)$ is the old estimate. In other words, the agent is updating the value function by steps of size $\alpha(G_t - V(S_t))$, where $(G_t - V(S_t))$ is an error in the old estimate with regards to the target G_t.

The presented estimation approaches are called *Monte Carlo methods*. One of the disadvantages of the Monte Carlo methods is that they have to wait until the end of the episode to update the value functions since the return G_t is not known during the episode.

The methods that do not wait for the end of the episode to update the value functions are called *temporal-difference (TD) methods*. These methods still learn from past experiences, but instead of waiting for complete returns, they use immediate rewards and value function estimates for the update. The most straightforward state-value function update is:

$$V(S_t) = V(S_t) + \alpha\Big(R_{t+1} + \gamma V(S_{t+1}) - V(S_t)\Big) \tag{1.7}$$

where α is the learning rate and $0 \leq \gamma \leq 1$ is the discount rate. In contrast with the Monte Carlo update in Equation (1.6), the target used in TD methods is $R_{t+1} + \gamma V(S_{t+1})$, i.e., the value function gets updated towards immediate reward and added discounted estimate of the subsequent state. By using the immediate reward R_{t+1} and value estimate $V(S_{t+1})$, estimates can be updated during the episode without waiting for the total return G_t as with Monte Carlo methods.

1.2.1 Algorithms

In this part, we briefly describe some of the algorithms popular in reinforcement learning, focusing on those we use later in this work. Detailed descriptions can be found in their original publications.

We start with a simple TD algorithm called **Sarsa** [23]. This algorithm works by iteratively updating the estimation of action-value function $Q(s, a)$, also called Q values, in the following manner:

$$Q^{new}(S_t, A_t) = Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)\Big) \tag{1.8}$$

where α and γ follow the previously mentioned definitions. The agent chooses actions based on the learned Q values, usually using the *ϵ-greedy* strategy. This strategy selects random action with probability ϵ and greedy action $\arg\max_{a \in A(S_t)} Q(S_t, a)$ with probability $1 - \epsilon$. By choosing a random action once upon a time, the agent maintains a balance between exploration and exploitation. Sarsa is part of the family of algorithms called *on-policy*,

where the agent follows the current policy when deciding on the subsequent action.

Another popular TD method called **Q-learning** was introduced in [24] by Watkins. This algorithm works by estimating action-value function using the following formula:

$$Q^{new}(S_t, A_t) = Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma \max_a\{Q(S_{t+1}, a)\} - Q(S_t, A_t)\Big) \quad (1.9)$$

where a belongs to the set of possible actions in the state S_t, $A(S_{t+1})$. The $\max_a\{Q(S_{t+1}, a)\}$ represents the best value in the following state S_{t+1}. The calculated Q values are stored in a so-called Q table. Q-learning follows the same *ϵ-greedy* strategy as the Sarsa algorithm when selecting actions. However, in contrast, it is an *off-policy* algorithm where the agent does not use the current policy for deciding on the next action. That is because the Q value is updated with the best estimate $\max_a\{Q(S_{t+1}, a)\}$ from state S_{t+1}, but when choosing an action in the following state, the agent does not have to follow the action a that led to the best estimate of Q value used for the update.

Improvement of the Q-learning algorithm called **deep Q-network (DQN)**, or deep Q-learning, was introduced by Mnih et al. [25, 26]. DQN replaces the tabular manner of storing all state-action pairs $Q(s, t)$ with a function, usually taking the form of a neural network. The Q value is then defined as $Q(s, a; \xi)$, where ξ can be one or more function parameters. Additionally, it introduces an experience replay buffer and periodically updated target. *Experience replay buffer* stores every episode step in the form of quadruplet (S_t, A_t, R_t, S_{t+1}) in a memory and the Q updates during training are then done by taking random samples from this buffer. This approach helps to improve data efficiency as one entry can be used multiple times and removes the correlation between subsequent episode steps. Another improvement is in the form of a *periodically updated target*, where the target Q values are updated only every Cth step (C is a hyperparameter), thus improving training stability.

All of the above-introduced algorithms learn the value functions and choose appropriate actions based on them. **Policy gradient (PG)** methods optimise the policy directly [27]. The policy is parametrised by weight vector $\theta \in \mathbb{R}^n$. The decision on choosing action a is therefore not only conditioned by the state s but also by the vector θ. The policy can then be defined as $\pi(a|s, \theta)$, i.e., the probability of taking action a given that the agent is in the state s with weight vector θ at time step t. Policy gradient methods use a gradient ascent algorithm to find the best settings for θ to maximise the total return.

Proximal policy optimization (PPO) is an on-policy algorithm introduced by Schulman et al. in [28]. In contrast with vanilla policy gradient methods which perform one gradient ascent update of the target policy per sample, PPO performs multiple repetitions of gradient ascent before updating the policy vector. As stated in the original paper, this improvement is easy-to-implement, yet it brings substantial performance improvements.

1.2.2 Achievements

The popularity of reinforcement learning can be attributed to the successes in mastering games and surpassing world-class players. In 2013 Mnih. et al. published [25] where the DQN agent learned to play some of the Atari games and later further improved already achieved results in the paper [29].

Arguably the most significant recent success was achieved by Silver et al. by mastering the game of Go with a program called AlphaGo and eventually beating the highest rank professional players [30]. This accomplishment was significant for reinforcement learning and whole computer science, as this game was considered too complex for computers to dominate [31].

Further progress was made by the authors of OpenAI Five in [32], where they managed to win a 5v5 best-of-three game of Dota2 against the world champion team. In 2019 Vinyals et al. presented a program called AlphaStar [33], which ranked 99.8% above all ranked human players in the online game StarCraft II.

While reinforcement learning may seem mainly used for playing games, these are usually just stepping stones for other real-world applications, such as cooling Google's data centres more efficiently and thus reducing their operating costs [34].

1.3 Portable Executable File Format

Portable executable (PE) is a file format commonly found on Windows operating systems for various types of files, such as executables (EXEs) or dynamically linked libraries (DLLs). This file format is based on the Common Object File Format (COFF) found on Unix operating systems. It contains all the necessary information for the operating system (OS) loader to correctly map the PE file to system memory [35].

In this section, we will focus on describing the PE file format used for EXE files because the usage of some fields might differ from other file types. The PE file format has a rigid structure defined as follows, starting with the MS-DOS header and the MS-DOS stub program. Next is the COFF file header, closely followed by the optional header. The PE file format is concluded with section headers and respective data sections. Most of the information presented in this section comes from the official Windows documentation [36]. Visual representation of the file format is depicted in Figure 1.2.

1.3.1 MS-DOS Header and Stub Program

MS-DOS header and stub program are still part of the PE file format for backward compatibility with older operating systems (MS-DOS). Nowadays, if a modern Windows executable gets executed on MS-DOS, it should display

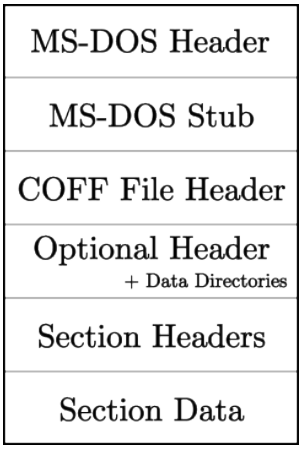

Figure 1.2: PE File Format

some variation of the following message: "This program cannot be run in DOS mode.".

MS-DOS header is 64 bytes long and is located at the beginning of the PE file. The first field of this header is `e_magic`, also called a magic number. This field usually contains a value of `0x5A4D`, a hexadecimal representation of the characters MZ, initials of one of the MS-DOS developers, Mark Zbikowski [37]. This field is followed by several other important fields for the MS-DOS system, which are not relevant for modern systems. The header is concluded with the `e_lfanew` field, which stores a file offset to the COFF file header.

MS-DOS stub program is an actual valid MS-DOS program that would get executed on the MS-DOS operating system. This stub is located right after the MS-DOS header, and its size is variable depending on the program.

1.3.2 COFF File Header

Following the MS-DOS header and stub program is the **COFF File header**. It is located at the offset found in the `e_lfanew` field from the MS-DOS header. Before the actual COFF header starts, there is a 4-byte field called `Signature` that identifies the file as a PE file with a value of `PE\0\0`. The following 20 bytes are the header itself, which contains general information regarding the PE file:

`Machine`: CPU type on which the PE file can be run, e.g., `AMD64` or `i386`.

`NumberOfSections`: Number of entries in the section table.

`TimeDateStamp`: Date when the PE file was created in Unix time.

`PointerToSymbolTable`: Offset of the symbol table inside the file. Usually set to 0.

`NumberOfSymbols`: Number of entries in the symbol table. Usually set to 0.

`SizeOfOptionalHeader`: Optional header size in bytes.

`Characteristics`: 2-byte flag indicating attributes of the file, e.g., the file is executable or debug information was removed.

1.3.3 Optional Header

Right after the COFF file header is located the **optional header**. Although it is called optional, for many files, such as EXEs, it is mandatory. It integrates core information for the OS loader. The header size can be found in the `SizeOfOptionalHeader` field in the COFF header. This header has three main parts: standard fields, Windows-specific fields, and data directories.

Standard fields are eight fields used for every COFF file and are defined as follows:

`Magic`: Indicates the type of optional header (32-bit/64-bit).

`MajorLinkerVersion`, `MinorLinkerVersion`: Linker version numbers.

`SizeOfCode`: The size of the code section, usually called *.text*.

`SizeOfInitializedData`: The size of the initialised data section, traditionally called *.data*.

`SizeOfUninitializedData`: The size of the uninitialized data section, usually called *.bss*.

`AddressOfEntryPoint`: Relative virtual address (RVA) of the entry point (the first instruction when execution begins) after being loaded into memory.

`BaseOfCode`: RVA of the code section after being loaded into memory.

The following are 21 fields belonging to the **Windows-specific fields** that contain unique information for the Windows operating system. Some of the fields are:

`ImageBase`: Preferred address of the beginning of the PE file after being loaded into memory.

`SectionAlignment`: Section alignment in memory.

`FileAlignment`: Section alignment on disk. The section is padded with zeros.

`SizeOfImage`: The size of the PE file. It must be rounded up to multiples of `SectionAlignment`.

`SizeOfHeaders`: Sum of all header sizes rounded up to a multiple of `SectionAlignment`.

`CheckSum`: Checksum value used to validate files such as drivers or DLLs.

`NumberOfRvaAndSizes`: Number of entries in data directories.

At the end of the optional header are placed the **data directories**. These directories form an array of 8-byte structures with two fields: RVA and size of the directory. There are 15 types of data directories, such as *export*, *import*, *debug* or *certificate* tables. The actual content of the directories is stored in their respective sections. Data directories that are not utilized are set to zero.

1.3.4 Section Headers and Data

Immediately following the optional header and data directories is the section table, also known as **section headers**. The position of the section headers can be calculated as a sum of `e_lfanew`, the size of the COFF file header and `SizeOfOptionalHeader`. Each section header has ten fields, totalling 40 bytes:

`Name`: Eight bytes representing the name of the section padded with zeros, e.g., `.text\0\0\0`.

`VirtualSize`: Section size when loaded into memory.

`VirtualAddress`: RVA of the first byte of the section. The headers must be sorted in ascending order by their corresponding virtual addresses. Additionally, the value of `VirtualAddress` must be a multiple of `SectionAlignment`.

`SizeOfRawData`: The size of the section data on disk. It must be multiple of `FileAlignment`. If the size is less than `VirtualSize`, the rest of the section is padded with zeros.

`PointerToRawData`: Pointer to the beginning of the section on disk.

`PointerToRelocations`, `NumberOfRelocations`: Pointer to the relocations and number of relocation entries for the section. Set to zero for EXEs.

`PointerToLinenumbers`, `NumberOfLinenumbers`: Pointer to the line-numbers and number of line-number entries for the section. These fields should be set to zero, as they are deprecated.

`Characteristics`: 4-byte flag indicating section attributes, e.g., the section contains executable code or can be shared in memory.

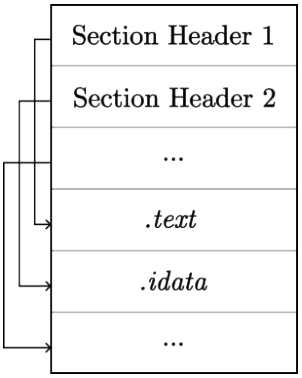

Figure 1.3: Section Headers and Data

The location and size of the relevant **section data** are indicated in the corresponding section header. For executables, section data must be aligned using the `FileAlignment` value found in the Windows-specific fields in the optional header. Figure 1.3 shows the relationship between section headers and section data, i.e., each section header points to the relevant section data. Note that the image is simplified, and the section names used are only examples.

Ordinary PE file usually has several commonly found sections [35]. Although their names may vary from file to file, their intentions are the same. The most notable is the *.text* section, which encapsulates all pieces of code. Naturally, the `AddressOfEntryPoint` points to this section and also marks the end of the import address table, which is also part of this section.

Usually, a program needs some data to perform its intended functionality. Three types of data sections are commonly found in PE files *.bss*, *.rdata*, and *.data*, each storing different kinds of data (uninitialized, read-only, ...).

One of the critical parts of nearly every executable is the import directory table (IDT), or just the import table. This table is usually stored in the *.idata* section and takes the form of an array, with each entry representing one of the imported DLLs. Since the size of the array is not fixed, the last entry is set to zero to indicate the end. Each entry contains the RVAs of the name of the imported DLL and the import address and lookup tables associated with this DLL. The import lookup table (ILT) is a table of imported function names from a given DLL. While stored on disk, the import address table (IAT) has the same structure and content as ILT. However, after the PE file is loaded into memory, the IAT entries contain the addresses of the imported functions instead of the function names.

One of the most evident section names is the *.debug* section, containing debugging information. Five types of debugging information can be stored, each with a unique header structure. This section is not memory-mapped by default, and the PE file format also allows storing the information in a separate debug file.

If a program exports some data, these are stored in the *.edata* section. This section can usually be found in DLLs. The resources needed for the application are stored in a multilevel binary tree within the *.rsrc* section. There are also other predefined section names, such as *.pdata* (exception handling) or *.reloc* (relocation information), which we will not cover in this work.

When a PE file has a certificate, e.g., to ensure the file's origin or immutability, the location is specified in a security data directory inside the optional header. The security data directory points to the beginning of the attribute certificate table, which contains 20-byte entries for each certificate. This certificate table is usually not stored in one of the mapped sections but is appended to the end of the file in a segment typically called an *overlay*.

Chapter 2

Proposed Method

We introduce a complete framework for generating adversarial malware samples called **AMG** (Adversarial Malware Generator). AMG consists of a tested PE file modifier, which can be easily expended with additional modifications, an environment in the Open AI Gym [38] format working with raw binary files and a set of trained and optimised reinforcement learning agents ready to use.

Based on the taxonomy presented in Section 1.1.1, our approach falls into the category of exploratory-integrity attacks. In other words, we are performing an adversarial attack to mislead the targeted model (antivirus) to classify malware samples as benign. Our objective is to execute small modifications on PE files that do not alter the original functionality but can make them unrecognisable to the antivirus. Our attack targets the so-called static malware analysis, where the detector makes decisions without examining the executable's behaviour. We set our adversarial attack in a black-box scenario where only hard labels (malware/benign) from the target classifier are known.

In our work, we modified the existing framework called *gym-malware* by Anderson et al. [39], which provides an environment for training reinforcement learning agents on binary samples. We rewrote most parts because the existing code did not meet our vision and goals. In particular, they used the `LIEF` [40] library for modifying PE files, whereas we used the `pefile` [41] Python library. We found that the `LIEF` library can make unnecessary changes to the original binary and that their modifications did not retain the same functionality as is shown later in Table 2.1. In addition, in their training setup, the agent is presented with an observation space that coincides with the feature space of the targeted classifier. We think that this could bring bias to the training system and could detriment the transferability of trained agents to other classifiers. For this reason, we used a different observation space that is not used by any of the classifiers we targeted. Nonetheless, the work [39] by Anderson et al. is a key stepping stone for future research as it is one of the first complete frameworks for deploying RL agents for adversarial malware

sample generation.

In the following sections, we describe in detail our proposed method, starting with the PE file modifications we use and how we test them. Later, we introduce our RL environment setup and agents.

2.1 PE File Modifications

For implementing PE file modifications, we used the `pefile` Python library. This library provides a simple interface for accessing all parts of the PE file format, such as file and optional header fields or individual sections. The description of the PE file format can be found in Section 1.3. We implemented various modifications of the binary files, all obeying the structure of the PE file format. While we had taken inspiration from state-of-the-art related works, such as the gym-malware mentioned above, we also introduced new modifications. In total, we implemented ten modifications which are described below:

- **Break CheckSum**: Set the `CheckSum` field from the optional header to zero.
- **Append to overlay**: Append a random benign content to the end of the file.
- **Remove debug**: Clear the debug entry in the list of data directories and remove the respective debug information from the file.
- **Remove certificate**: Clear the security entry in the list of data directories and remove the certificate data from the file.
- **Add new section**: Add a new section to the PE file if possible. Firstly, it is necessary to check if there is enough free space between the last section header and the beginning of section data (at least 40 bytes). If so, we can increase the file size and add a new section header and data. To preserve the original PE file structure as much as possible, we also move the old overlay data and, if present, redirect the security data directory to the new address.
- **Append to section**: Append benign content to one of the currently present sections if possible. First, we need to find a section with the possibility of adding extra content, i.e., the virtual size of the section is greater than its raw size. If we encounter one, we fill the empty space with benign content.
- **Rename section**: Choose one section at random and rename it to one of the commonly used names in benign files.

- **Increase TimeDateStamp**: Increase the value of `TimeDateStamp` in the COFF file header by 500 days[1].
- **Decrease TimeDateStamp**: Decrease the value of `TimeDateStamp` in the COFF file header by 500 days[1].
- **Append new import**: Add a new section to the PE file with import data if possible. This process is similar to the preceding add new section modifications with the only change that the section content is not random benign content but import data. If already present, we take the old IDT table from the PE file and append a new entry, a randomly chosen DLL. Then we prepare entries for individual imported functions that get stored in the IAT and ILT tables. All these data are then placed in a newly added section, and the import data directory is pointed to the new IDT table.

We omitted to implement packing of an executable as one of the modifications, as it can be easily detected and reverted with the use of the corresponding packer, such as UPX [42] or Exeinfo PE [43].

2.1.1 Preserving Functionality of PE Files

We believe that preserving the original functionality of executable binary files is a critical part of generating adversarial malware samples. Without emphasising this criterion, we cannot guarantee that the resulting adversarial example will still be a working executable with the same functionality as the original file. We have found that more than simply checking the syntax of the PE file format is needed to maintain functionality, so we implement the following procedure to test modifications to our PE files. However, our method is not without flaws, as we mention later in Conclusion.

To ensure that the functionality of the file after modification is as close as possible to the original file behaviour, we used a Cuckoo sandbox. **Cuckoo sandbox** is an open-source automated malware analysis tool that can run malicious files and examine their behaviour. Even though it is predominantly intended for malware analysis, we use it to analyse benign files as well since it provides behavioural analysis, which we utilise to track any changes in the functionality of executables. We decided to use benign files instead of malware executables for testing the modifications. The reasoning behind this decision is that malware authors can insert checks into their programs that monitor whether their malware is running in a sandbox environment and change its behaviour accordingly [7, 8]. By using benign files, we limit the possibility of artificial activity of the tested binaries, and thus we can better analyse the reported behaviour.

[1]We picked 500 days because it is a considerable period of time and it is not a multiple of one year.

From our benign dataset, described later in Section 3.1, we randomly selected 100 benign EXE to test our PE modifications. We made sure that all files were executable in the sandbox environment. We then launched these files inside the Cuckoo sandbox and studied their respective behaviour reports. Namely, we look into three features found in the Cuckoo analysis report: signatures, API calls, and processes:

Signatures Predefined patterns that are used to compare with the analysed file. They are used predominately for malware detection to cluster malware into their respective families. Nevertheless, they can also classify types of actions such as file open/write or access to system files, which are also encountered in benign files.

API calls Function calls by the program to external libraries in the course of program execution.

Processes Main process and sub-processes started by the program.

Unfortunately, we found these features unstable, meaning they can change slightly (or more) during different analysis runs of the same file. To combat this, we conducted three testing rounds and considered the feature stable if it got at least 95% agreement between rounds. In the worst case, two features remained reliable for each of the 100 unmodified benign files used.

The 100 benign files listed above are used as a control dataset to test whether the functionality of the PE file has changed after modification. We run three rounds of Cuckoo analysis and compare modified files according to the same three features as we mentioned before. We consider the modification a failure if the modified file cannot run in the Cuckoo sandbox. If the file is executed successfully, we compare the three generated analysis reports with all three control reports. We look at each feature individually, matching it with its respective control file. The feature is considered matched if it has an agreement of at least 95% with one of its control files. Overall, the modified file is considered successfully modified, i.e., the original functionality is preserved if it matches at least two of its features with control reports.

We compare our modifications with several PE file modifiers from well-known frameworks for generating adversarial malware samples. Namely, we tested gym-malware [39], Pesidious [44] and MAB-malware [45]. Both gym-malware and Pesidous used the `LIEF` library for modifying binaries, whereas MAB-malware same as we used the `pefile` library. Additionally, the authors of Pesidous used the `PE Bliss` [46] C++ library for rebuilding PE files.

We present the results of our functionality preserving testing in Table 2.1. The first column represents different PE modifications, and the following represent the PE modifiers from the respective frameworks. Our framework is denoted as AMG, an abbreviation for Adversarial Malware Generator. The

Table 2.1: Numbers of files with preserved functionality after modification from a total of 100 binaries.

action	gym-malware	Pesidious	MAB-malware	AMG
break checksum	89	×	100	100
create new entry point	17	×	×	×
append new import	20	42	×	66
overlay append	100	99	100	100
remove debug	90	×	100	100
remove certificate	22	×	90	91
add new section	4	85	75	98
append to section	8	×	99	99
rename section	89	89	99	100
upx pack	73	×	×	×
upx unpack	100	×	×	×
increase TimeDateStamp	×	×	×	100
decrease TimeDateStamp	×	×	×	100

symbol × signifies that the operation was not implemented by the given framework. We can see that our PE modifications equal or surpass all other tested frameworks. Apart from the modifications mentioned in Table 2.1, authors of MAB-malware also implemented code randomisation operation. However, we could not reproduce the code locally for our dataset, so we did not include it in our testing. Even though we improved the modification append new import entry to the PE file significantly in comparison with other frameworks, we still did not manage to achieve a similar score as with other modifications, which all preserved functionality in more than 90% of cases. Later, in Conclusion, we propose this as a possible improvement for our framework.

2.2 Malware Environment

As mentioned in Section 1.2, RL algorithms are based on learning through feedback provided by the environment. We worked with a commonly used environment format developed by the OpenAI company called Gym [38]. The `gym` is an open-source Python library equipped with a standardised API for agent-environment interaction.

Several essential methods and properties must be defined to deploy our malware environment. In particular, `reset` and `step` methods. The `reset` method restarts the environment to an initial state, which in our case, is a new malware sample. The return value of this method is an observation. *Observation* is a feature vector that represents the current state of the environment presented to the agent. In our work, we came up with an observation that carries the form of 10,000 raw bytes extracted from the PE binary by taking

the first and last 5,000 bytes from the binary file and merging them together. By taking bytes from both ends of the PE file, we hope to cover most of the binary while keeping a relatively small feature space for the agent to explore.

The `step` method is the pivotal method for agent-environment interaction because it is responsible for performing the actions selected by the agent. This method uses the aforementioned PE file modifications to execute the selected actions. Additionally, it also tracks the length of the episodes and calculates rewards for each action. The reward is either 0 if the sample is not evasive or $100 - penalty$ if it can bypass detection by the target classifier where the *penalty* is a slight handicap reflecting the increase of the executable's size. We introduce this handicap, intending to force the agent to minimise the total size of modifications.

A critical part of the environment is the target classifier, as each action is rewarded with respect to its predictions. We studied two ML classifiers, MalConv and GBDT, both freely available online by their respective authors. MalConv is a deep convolutional network that does not require complex feature extraction procedures because it uses the entire executable (truncated to 2,000,000 bytes) as an input feature vector [47]. On the other hand, GBDT is a gradient-boosted decision tree trained using the LightGBM framework [48] that requires converting the input executable to an array of 2,381 float numbers. We used pre-trained versions of both classifiers by their respective authors. Note that we did not directly target the MalConv classifier but only used it for testing the transferability of adversarial attacks between ML classifiers.

2.3 Reinforcement Learning Agents

In our work, we experimented with three RL agents, deep q-network, vanilla policy gradient and proximal policy optimization. We chose these reinforcement learning algorithms because they are well-known in the reinforcement learning community and represent both on-policy and off-policy approaches. A more detailed description can be found in the previous Section 1.2.1 and in the original publications [26, 27, 28].

For the implementation of RL agents, we used a reinforcement learning library called `Ray RLLib` [49]. This library contains prefabricated implementations of many state-of-the-art algorithms. We decided not to go the route of implementing our own RL agents, i.e., creating unique agents tailored for modifying PE files. Instead, we focused on optimising the hyperparameters of existing state-of-the-art algorithms and creating the best possible environment in which they can learn. Additionally, the `Ray RLLib` provides a parallelism layer encapsulating RL agents, allowing us to parallelise and speed up the training process. Documentation for individual RL algorithms can be found in `RAY RLLib` documentation [50].

Chapter 3

Evaluation

In this chapter, we describe in detail our experimental setup and how we approached the evaluation of our adversarial malware generator. We present our achieved results divided according to the algorithm used and, later, how the generated attacks transfer to other malware detectors.

3.1 Setup

The principal metric we use in this work is called an **evasion rate**. This metric denotes the ratio of misclassified files by the target classifier and is calculated as follows:

$$evasion\ rate = \frac{\#\ missclassified}{total} \tag{3.1}$$

where *total* stands for the total number of files submitted to the target classifier after discarding files that were already incorrectly predicted before modification.

Further, we use a metric called a **mean episode reward**. This metric denotes the mean of all rewards received by the agent during a single episode. It is used predominantly during training to understand how well the agent is performing. Note that this metric does not compare agents across different environments, as every environment can define its own reward function.

Another key term used in the subsequent paragraphs is a **training iteration**. One training iteration consists of one or more episodes depending on the agent's configuration, e.g., the size of the training batch. Each agent's exact number of episodes during one training iteration can be calculated using the configuration files from the `Ray RLLib`.

We use two datasets. A dataset of benign binaries, including more than 4,000 executables, was scrapped from the fresh Windows 10 installation. These benign files are only used while testing the preservation of functionality after modification, as mentioned in Section 2.1.1. Second, a dataset of malware files

was obtained from the VirusShare repository [51]. In total, we operate with 6,000 malware files divided into three parts: a training dataset consisting of 4,000 files and validation and testing sets containing 1,000 files each.

Our experiments were executed on a single computer platform with two server CPUs (Intel Xeon Gold 6136, base frequency 3.0Ghz, 12 cores), one GPU (Nvidia Tesla P100, 12 GB of video RAM) and 754 GB of RAM running the Ubuntu 20.04.5 LTS operating system.

3.2 Experiments

We define the following procedure used in all subsequent experiments for each RL algorithm. The first step is finding the optimal number of modifications, also called the maximal number of steps, for a given RL algorithm. This optimisation is done by limiting the maximum number of calls to the `step` method that the agent can make inside the environment. For this part of the experiment, we leave agent parameters at their default settings as set in the `Ray RLLib`. The range we are testing is between 5 and 200 modifications, and we choose the optimal value based on two criteria. Firstly, we try to maximise the evasion rate achieved by the agent and secondly, we try to minimise the increased size of the adversarial sample.

After determining the maximum number of modifications, we conduct a hyperparameter search using the grid search method over two hyperparameters, the learning rate (lr, α) and the discount rate (gamma, γ), leaving the rest of the parameters at the default settings as defined by the authors of `Ray RLLib`. Based on the highest mean episode reward scored during 100 training iterations, we select the best four agent configurations and let them train for another 900 iterations. After the training finishes, we test these agents on the validation set and determine the best agent configuration for a given RL algorithm.

Subsequently, we introduce our testing dataset, which is presented to the final RL agent. The results obtained on this set of samples are then used to compare different RL algorithms and to verify the success of the entire training process.

In all the experiment steps mentioned above, we consider the GBDT classifier as our target model. In the penultimate phase, we test the transferability of adversarial attacks between GBDT and MalConv classifiers. We use the best-trained agents for each RL algorithm to generate adversarial samples against the MalConv classifier and measure the difference in performance.

The last step of our procedure is the evaluation performance of the best RL agents against commercial AV programs. We conduct this assessment on several well-known AV products such as AVG, Avast, Avira, Bitdefender, G Data, McAfee, Kaspersky, Symantec (nowadays known as Norton) and VIPRE using VirusTotal website [52]. We selected these top antivirus programs based

on the September 2022 antivirus comparative study by the Austrian AV testing laboratory AV-Comparatives [53]. In order to minimise the possible risk of misuse of our work, we anonymise the results of each antivirus. Figure 3.1 gives a complete overview of our experiment workflow.

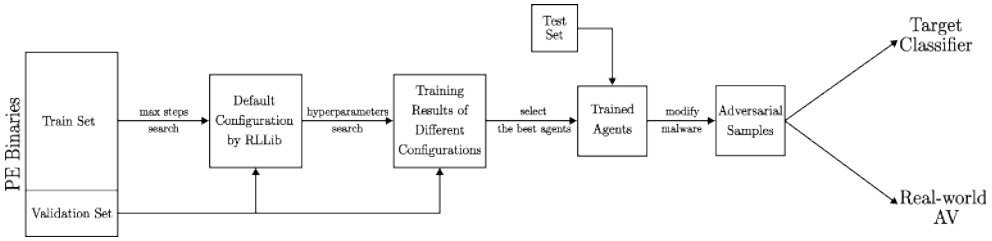

Figure 3.1: Workflow of our training and testing procedure.

3.2.1 Deep Q-Network

The first reinforcement learning algorithm we tested is deep q-network. As mentioned earlier, we started by searching for the optimal maximum number of steps (*max steps*) permitted to the agent. We trained the default configuration of DQN as set by the authors of `Ray RLLib` for 50 training iterations. We repeated this process for each value tested as it required a different environment setup.

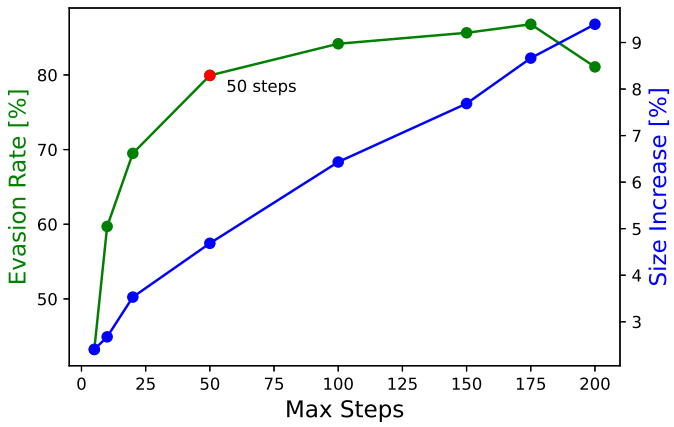

Figure 3.2: Evasion rate and input size increase based on a maximum number of steps used by the DQN agent.

Agent's results on the validation set can be seen in Figure 3.2. In this picture, we can see that the evasion rate increased rapidly until the value of 50, after which it started to decelerate until a decrease in performance for 200. The increasing number of modifications puts higher demands on training thus is possible that 50 episodes were not enough for the agent to train reasonably

well. Unsurprisingly, the size increase between the original file and generated adversarial sample kept steadily increasing throughout the testing. Based on these results, we chose the value of 50 as optimal because the agent recorded a high evasion rate of almost 80% while maintaining the average increased size of the adversarial malware sample below 5%.

In the next stage, we optimised the hyperparameters of the deep q-network. The training results after 100 training iterations can be found in Table 3.1. The results are sorted by mean episode reward in descending order, thus putting the most performing configurations in the top rows. We can see that apart from the first configuration ($\gamma = 0.5$ and $\alpha = 0.01$), the following five configurations performed similarly. Even though we set the maximum number of steps to 50, the mean episode length was generally much lower.

Table 3.1: Hyperparameters search results for the DQN agent with the maximum number of steps limited to 50.

γ	α	mean episode reward	mean episode length
0.5	0.01	93.76	10.69
0.5	0.001	87.85	12.17
0.75	0.001	87.84	12.02
0.99	0.001	86.85	13.08
0.99	0.01	86.84	14.3
0.75	0.01	84.87	14.98
0.99	0.0001	80.88	16.97
0.5	0.0001	79.86	17.53
0.75	0.0001	78.9	16.83

We took the best four configurations from Table 3.1 and trained them for further 900 iterations. The training progress can be seen in Figure 3.3. We displayed only the first 250 training iterations for better comprehension of the training process, as more iterations made the plot unreadable. We can see that the training process was similar for all configurations. The only exception is the configurations with $\gamma = 0.99$ and $\alpha = 0.001$, which recorded significant drops in performance during training. Further, we can see a sizeable performance increase was recorded only in the first 100 iterations, with only marginal improvements in the later stages. These results may signify that the total of 1,000 training iterations was not enough and further gains could be reached if trained for longer, or that we hit a plateau and different agent configurations would be needed to achieve a higher reward. Additionally, we can see that the training progress was not stable, as there were significant differences between training iterations. The total training time for each configuration was around 25 hours on our setup.

Finally, we selected the best checkpoints from the 1,000 training iterations based on the highest mean episode reward and measured the performance

Figure 3.3: Training progress of the first four DQN configurations from Table 3.1 during the first 250 training iterations.

of the respective agents on the validation set. The results are displayed in Table 3.2, where we can see that the DQN agent achieved the highest evasion rate with $\gamma = 0.5$ and $\alpha = 0.01$. However, all four configurations performed similarly, as the differences were negligible. It is necessary to mention that these results are only slightly better than the results after 50 iterations, as shown in Figure 3.2. This was already evident during training, as Figure 3.3 indicated that significant improvements were made only in the first iterations.

Table 3.2: Results of the first four DQN configurations from Table 3.1 after 1,000 training iterations.

γ	α	evasion rate [%]	size increase [%]	mean episode length
0.5	0.01	**79.93**	4.64	**10.17**
0.5	0.001	79.77	**4.62**	10.27
0.75	0.001	79.28	4.73	10.28
0.99	0.001	79.77	4.66	10.28

3.2.2 Policy Gradients

The second reinforcement algorithm we studied is the vanilla policy gradients algorithm. The evaluation plan was the same as with DQN. Firstly, we performed a search over values between 5 and 200 to determine the optimal maximum number of modifications permitted to the agent. The results are depicted in Figure 3.4. At first glance, we can see that the results are much more variable than with DQN. Especially for the higher maximum number of steps, we can observe different behaviour with a decline in performance between 50 and 150 modifications. This decrease may be caused by the fact that 50 iterations were insufficient for PG agents to learn a meaningful strategy and thus used more random actions. Based on Figure 3.4, we proceeded with 20 as the maximum number of modifications, offering both a relatively high evasion rate (almost 70%) and a low size increase (around 3%).

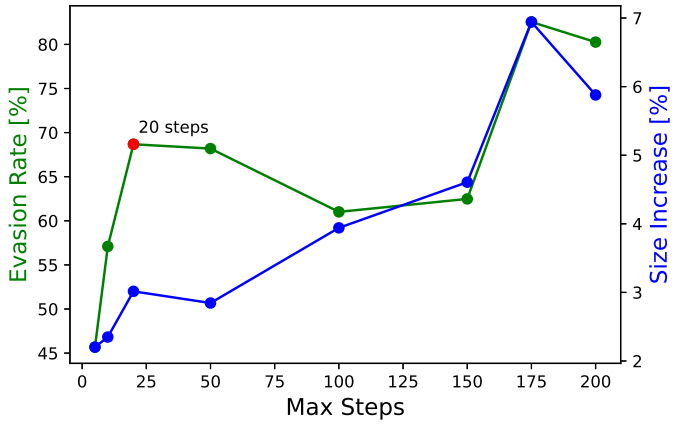

Figure 3.4: Evasion rate and input size increase based on a maximum number of steps used by the PG agent.

After determining the optimal number of steps, we explored the hyperparameter space the same as with DQN. The training results for the PG algorithm after 100 iterations can be seen in Table 3.3. The best performance was recorded by PG configuration with $\gamma = 0.5$ and $\alpha = 0.001$. Apart from the $\gamma = 0.99$ and $\alpha = 0.0001$ configuration, all hyperparameter settings achieved a mean episode reward higher than 70. Same as with DQN, the mean episode length was generally much lower than the cap value.

Table 3.3: Hyperparameters search results for the PG agent with the maximum number of steps limited to 20.

γ	α	mean episode reward	mean episode length
0.5	0.001	85.83	6.25
0.75	0.01	80.89	8.26
0.75	0.001	79.9	8.44
0.99	0.01	79.87	8.19
0.5	0.01	78.9	8.13
0.5	0.0001	77.9	8.36
0.99	0.001	76.86	9.41
0.75	0.0001	71.89	10.49
0.99	0.0001	60.91	12.59

The training progress during 250 training iterations of the first four PG configurations from Table 3.3 can be found in Figure 3.5. We trained the agents for 1,000 iterations but omitted the rest of the iterations from this figure. Overall, the training was more stable than with DQN. However, some configurations experienced significant drops in performance in the first iterations, which later stabilised with continued training. Unlike DQN, the training

time was much lower, as less than 12 hours were needed to fully train one configuration.

Figure 3.5: Training progress of the first four PG configurations from Table 3.3 during the first 250 training iterations.

The trained agents from Figure 3.5 were tested on the same validation set as DQN. From Table 3.4, we can see that the evasion rate was, on average, 10% lower than with DQN. The best result of 69.33% was recorded by the PG agent with $\gamma = 0.75$ and $\alpha = 0.01$. The only improvement to DQN is the smaller increased size of the modified files, which for PG was slightly above 3%. Similarly to DQN, these results are marginally better than the results achieved with only 50 iterations of training.

Table 3.4: Results of the first four PG configurations from Table 3.3 after 1,000 training iterations.

γ	α	evasion rate [%]	size increase [%]	mean episode length
0.5	0.001	69.0	3.31	5.89
0.75	0.01	**69.33**	**3.24**	5.84
0.75	0.001	69.17	3.3	5.88
0.99	0.01	69.0	3.24	**5.83**

3.2.3 Proximal Policy Optimization

The last reinforcement algorithm we tested is another policy gradient algorithm called proximal policy optimization. We followed the aforementioned experiment protocol, starting with an exploration through a maximum number of steps between 5 and 200. From Figure 3.6, we can see that the plot for PPO mimics the same behaviour we saw with DQN in Figure 3.2. Since the evasion rate is increasing rapidly until the value of 50 and then begins to stagnate, we decided to proceed with the maximum number of steps set to 50 same as with DQN.

Secondly, we optimised the gamma and learning rate hyperparameters of PPO. In Table 3.5, we can see the performance on the training set after 100

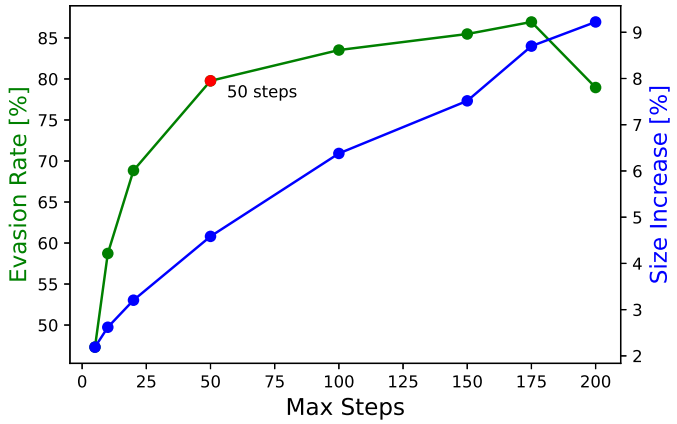

Figure 3.6: Evasion rate and input size increase based on a maximum number of steps used by the PPO agent.

training iterations. Apart from the last configuration with $\gamma = 0.99$ and $\alpha = 0.01$, the differences in mean episode reward were only minor between individual hyperparameters. The highest mean episode reward of 87.66 was recorded by the PPO agent with $\gamma = 0.75$ and $\alpha = 0.0001$. In contrast with the DQN results presented in Table 3.1, the agents with lower learning rates performed better.

Table 3.5: Hyperparameters search results for the PPO agent with the maximum number of steps limited to 50.

γ	α	mean episode reward	mean episode length
0.75	0.0001	87.66	13.08
0.5	0.0001	87.33	13.23
0.99	0.0001	85.91	14.51
0.5	0.001	85.09	13.79
0.99	0.001	84.9	14.04
0.75	0.001	83.76	15.21
0.5	0.01	81.5	17.31
0.75	0.01	79.7	17.4
0.99	0.01	53.47	31.9

We retrained the first four PPO agents from Table 3.5 for 1,000 iterations, and the training progress during the first 250 iterations can be seen in Figure 3.7. At first glance, we can see that different values of gamma and lr had little effect on the training progress and that an increased number of training iterations did not significantly improve the agents' performance. This may indicate that the agent is hitting a local optimum. In contrast with DQN and PG, the training performance was much more stable in most cases, maintaining

itself above the mean episode reward of 70. The only exception was the PPO agent with $\gamma = 0.99$ and $\alpha = 0.0001$, which recorded significant dips in performance during some episodes, but overall performance remained similar to other configurations. Using the setup mentioned above, we averaged over 60 hours of training time for each PPO agent, most of all RL agents tested.

Figure 3.7: Training progress of the first four PPO configurations from Table 3.5 during the first 250 training iterations.

Taking the aforementioned trained configurations of PPO agents, we measured their performances on our validation set of malware samples. The full result can be found in Table 3.6. The highest evasion rate of 90.38% was scored by PPO with $\gamma = 0.5$ and $\alpha = 0.0001$. Overall, all PPO configurations performed better than their DQN (Table 3.2) and PG (Table 3.4) counterparts, recording a minimal evasion rate of 81.4%. It is worth mentioning that apart from the first agent ($\gamma = 0.75, \alpha = 0.0001$), all took longer to build adversarial samples with more than 20 modifications needed on average, increasing the size of the binaries significantly.

Table 3.6: Results of the first four PPO configurations from Table 3.5 after 1,000 training iterations.

γ	α	evasion rate [%]	size increase [%]	mean episode length
0.75	0.0001	81.4	**4.7**	**9.98**
0.5	0.0001	**90.38**	9.2	20.22
0.99	0.0001	87.77	7.95	21.41
0.5	0.001	87.44	9.02	22.41

3.2.4 Test Results

We introduced a separate test set of 1,000 malware binaries for the final phases of our evaluation procedure. This set of samples has never been used in our experiments before, which gives us a clear picture of the agents' performances. Based on the results achieved on the validation set, we selected the following agent configurations DQN ($\gamma = 0.5, \alpha = 0.01$), PG ($\gamma = 0.75, \alpha = 0.01$) and PPO ($\gamma = 0.5, \alpha = 0.0001$) for the subsequent experiments.

Firstly, we evaluated the aforementioned training process by measuring the performance of RL agents on the test set. The results are shown in Table 3.7. We immediately see significant drops in performance for all agents compared to performance on the validation set, suggesting possible overfitting. However, they still achieved decent results. The trend is similar to what we saw on the validation set, as the highest evasion rate of 58.92% was recorded by the PPO and the lowest by the PG agent. For the DQN agent, there was a substantial file size increase, from 4.64% on the validation set to 11.25% on the test set. To conclude, the overall best RL algorithm against GBDT classifier is the PPO with $\gamma = 0.5$ and $\alpha = 0.0001$, striking the highest evasion rate while maintaining a reasonable size increase of adversarial examples.

Table 3.7: Results of the best configuration for each tested RL algorithm on the test set against the GBDT classifier.

	evasion rate [%]	size increase [%]	mean episode length
DQN	55.95	11.25	21.67
PG	40.14	**5.92**	**10.84**
PPO	**58.92**	9.01	21.07

In the following test, we investigated the potential transferability of an attack generated against the GBDT classifier to another ML model, MalConv. This was done by taking the aforestated trained RL agents and introducing them to MalConv as the target classifier without any prior training with this classifier. The comparison of results against GBDT and MalConv can be seen in the subtables of Table 3.8. The last column of the respective tables indicates the increase or decrease of the measured metric relative to the original value recorded for the GBDT classifier. Note that all values presented in these tables are in percentages.

Table 3.8: Transferability of the adversarial attack targeted against GBDT to MalConv.

(a) Evasion Rate [%]

	GBDT	MalConv	change
DQN	55.95	27.17	-51.43
PG	40.14	18.96	-52.77
PPO	58.92	**28.91**	-50.93

(b) Size Increase [%]

	GBDT	MalConv	change
DQN	11.25	19.02	69.13
PG	5.92	**9.58**	61.85
PPO	9.01	18.95	110.24

The average success rate of adversarial samples that bypassed the GBDT classifier was slightly below 50% when facing the MalConv model. The PPO agent maintained the highest evasion rate against MalConv but was closely followed by the DQN agent, achieving evasion rates of 28.91% and 27.17%,

respectively. However, PPO recorded the highest relative size increase, with average size increasing from 9.01% to 18.95%, an increase of more than 110%. Nonetheless, in absolute values, PPO is still the go-to algorithm to attack MalConv in a black-box setting without any prior training against this model. These results indicate the possibility of transferring attacks from one ML classifier to another but with a significant decrease in performance.

The ultimate evaluation step was to measure how the adversarial samples generated during testing from Table 3.7 transfer to previously unseen commercial AV programs. This is the purest black-box scenario, where neither the training nor the generation of samples is done against the targeted classifier.

Table 3.9: Evasion rate of the generated adversarial samples against real-world AV programs.

	AV-1	AV-2	AV-3	AV-4	AV-5	AV-6	AV-7	AV-8	AV-9
DQN	8.9	8.9	14.7	9.99	9.65	10.02	7.38	7.83	9.94
PG	9.25	9.25	15.0	9.35	9.27	9.92	16.86	12.5	9.6
PPO	**10.24**	**10.24**	**25.7**	**11.31**	**11.08**	**10.9**	**19.8**	**13.88**	**11.46**

The results against the top nine antivirus products are presented in Table 3.9. Each row indicates one of the used RL algorithms, and each column represents the evasion rate in percentage achieved against the respective AV. As stated before, the antivirus products' names are anonymised, but all AV programs used are listed in the experiment description above. In line with the previous results, the PPO agent achieves the highest evasion rate against all AV products, topping up the evasion rate of 25.7% against AV-3. However, the performances of DQN and PG were much closer than measured in earlier experiments, with the PG agent beating DQN against 5 AV programs.

Chapter 4

Related Work

This chapter summarises related publications that focus on creating adversarial malware attacks. We break down this chapter into several sections depending on the approach used to generate adversarial samples. We start by describing the works using the same method as we did, attacks based on reinforcement learning. Then we present the researches that exploit the back-propagation algorithm commonly used in training deep neural networks with so-called gradient-based attacks [6]. Note that most of these gradient-based attacks classify as white-box attacks since they work directly with the inner configurations of targeted models. Next, we mention several publications that relate to adversarial malware attacks but do not fit within these two categories. Lastly, we discuss our results with respect to state-of-the-art methods.

4.1 Reinforcement Learning-Based Attacks

One of the first works done in the domain of generating adversarial samples using reinforcement learning was published in 2018 by Anderson et al. [39]. The authors presented a gym-malware framework equipped with RL agents and an OpenAI gym environment. They targeted a gradient-boosted decision tree which was trained on 100,000 binary files and achieved an evasion rate of up to 24%, depending on the dataset used. We already mentioned earlier in the introduction of Chapter 2 some of the shortcomings of this framework, with some of them also mentioned in their discussion.

In contrast with our work, the authors of [17] focused on Android as their platform of choice. They represented Android applications as a feature vector containing permissions such as reading messages or accessing the location granted to the given program. The authors limited the maximum number of modifications to 5 while targeting 8 ML detectors, such as decision tree or deep neural network. In a white-box scenario, they recorded an average evasion rate of 44.28%, which was further improved to 53.20% when they trained the RL agent on various target models in grey-box settings. The authors proposed

retraining the ML detectors as a countermeasure against their attack and recorded a decrease in evasion rate by 15.22%–29.44%.

In [16], Fang et al. presented two models, a detector called DeepDetectNet and a generator of adversarial samples called RLAttackNet. Their detector is based on the DQN algorithm and managed to bypass their own detector in 19.13% of cases in a pure black-box scenario. Later the authors used the adversarial samples to retrain their DeepDetectNet model, and the evasion rate of RLAttackNet dropped to 3.1%.

Already mentioned MAB-malware framework by Song et al. [45] is one of the most detailed works we have found in the area of crafting adversarial malware examples. Like we did, they used the `pefile` library to modify adversarial samples. To control functionality preservation of their generated adversarial samples, the authors used Cuckoo sandbox to validate that the file signatures reported by the sandbox do not change after modification. As an agent, the authors of MAB-malware used a multi-armed bandit, a simple reinforcement learning algorithm that does not consider the order of actions used during an attack. During training, the authors adjusted the agent's action space in real time by adding successful action-content pairs. Further, Song et al. introduced an action minimisation process, which removes unnecessary modifications after successful evasion, thus decreasing the final size of adversarial examples. The authors targeted the GBDT by EMBER, MalConv and several commercial AV detectors. With their approach, they achieved a high evasion rate of 74.4% against GBDT, 97.7% against MalConv and up to 48.3% against commercial AVs. Further, the authors evaluated the transferability of samples between detectors and showed that more than 80% of adversarial examples are evasive between GBDT and MalConv, but merely 7% of samples that were evasive on ML detectors were also evasive on commercial AVs.

Rigaki and Garcia in [54] used the aforementioned MAB-malware framework to compare target and surrogate models. A surrogate model is a substitute model used instead of the target model in scenarios where the original model is unavailable, e.g., it has a high response time or the querying quota is limited. Their surrogate models achieved 99% agreement with pure ML models and 90%–98% with AV products. Their results show that the quality of the surrogate model highly depends on the overlap of training datasets between the surrogate and original models. While testing the evasion rate of adversarial samples generated by MAB-malware on real-world AVs, the authors noticed that the success of bypassing the AVs rests on whether AVs are connected to their respective cloud systems with significant drops in evasion rate in online settings.

The authors of [55] worked with the gym-malware framework while targetting MalConv, LGBM (LightGBM) and random forest classifiers. The authors achieved the highest evasion rate of 43.8% against MalConv with a particular variant of DQN. Nevertheless, after they tested their adversarial samples for functionality preservation by executing them on Windows 7 virtual machine,

their evasion rate dropped significantly, as only 18.6% of executable malware files escaped detection by MalConv after discarding non-executable samples. Their work further strengthens our results shown in Table 2.1, proving that some of the modifications included in the gym-malware framework generate non-functional samples.

Quertier et al. in [56] used reinforcement learning algorithms to attack MalConv, GBDT by EMBER and Grayscale (convolutional neural network interpreting PE binaries as images) classifiers in grey-scale settings with available prediction scores for learning. Further, the authors targeted commercial AV in a pure black-box environment. They used DQN and REINFORCE (policy gradient algorithm) agents and achieved a very high evasion rate against all targeted models. Namely, an 80% evasion rate with REINFORCE against GBDT, 100% perfect evasion against MalConv with both algorithms and the most outstanding, a 70% evasion rate against commercial AV with REINFORCE. However, Quertier et al. did not specify what commercial AV they were targeting, and neither the authors published their work for further use.

4.2 Gradient-Based Attacks

A gradient-based attack on an Android malware detector was proposed in [19]. Using the gradient descent algorithm, the authors calculated the necessary perturbation for the feature vector, which contained features found in the Android manifest file. They targeted a self-made detector and scored an evasion rate of 63%–69%. Further, the authors proposed to retrain the classifier with adversarial samples but cautioned that this could lead to reduced classifier performance in specific scenarios.

In [57], the authors proposed a gradient-based attack against a MalConv malware detector. The feature space of this classifier is in the form of 2,000,000 raw bytes extracted from the PE binary. Their attack only targeted the overlay part of the file and achieved an evasion rate of 60% while modifying less than 1% of total bytes.

The authors of [58] used a gradient-based attack, limited to injecting small-scale chunks of bytes into unused parts or at the end of the file. They argued that these modifications do not change the functionality of the file but without further evidence. The authors scored a high evasion rate of 99% against the MalConv classifier.

Another attack on MalConv was carried out by Demetrio et al. in [21]. Using an integrated gradient method, the authors studied which sections of a binary stimulate the MalConv classifier and thus are sensitive to adversarial attacks. Demetrio et al. found that MalConv is relaying its prediction on features found in the DOS header. This is a surprising realisation because, in modern programs, most of this header is only included for backward compatibility and is not utilised. While the exact numbers are not presented in the

original paper, the authors say that by slightly modifying the DOS header, they can achieve a high probability of evasion against the MalConv detector.

A variation on the method introduced in [58] was presented in 2021 by Yang et al. [59]. The authors treated the input executables as images, which were fed as input into a convolution neural network. They calculated necessary byte perturbations for detection evasion, which were then converted into specific byte sequences. Depending on the location of the given perturbation, the resulting byte sequence was either a dead-code or API call instruction. The authors conducted a theoretical examination of the above-mentioned modifications to confirm that their modifications preserve functionality but without any real-world testing. Their method decreased the accuracy of several deep neural models by more than 60% but performed worse against models such as random forest or linear regression with an accuracy decrease of less than 20%.

4.3 Other Methods

In [18], the authors implemented a greedy search algorithm which modifies feature vectors representing the API calls made by a given file. The authors conducted an analysis to determine which API calls contribute to files being classified as malware or benign and recorded an evasion rate of up to 69.78% while targeting an unknown ML-based classifier.

A generative adversarial network (GAN) called MalGan was proposed in [60] as a method of generating adversarial samples. During training, the authors used a substitute detector in the form of a deep neural network, and their results show high transferability of attack between the substitute and target model by achieving near-perfect evasion against the random forest, decision tree or linear regression algorithms. However, they worked in a feature space of extracted API calls and did not provide a method capable of converting the adversarial feature vectors back to real-world executables.

One of the few works we have found that tackle the data poisoning issue is [61]. Even though this work is focused on the Android operating system, the results could also apply to Windows systems. The authors recorded up to 30% drops in accuracy after injecting their data while targetting pure ML models. As a defence mechanism, they introduced a camouflage detector that detects suspicious samples inside the training dataset and increases the detector's accuracy by at least 15%.

Fleshman et al. in [62] presented three black-box adversarial attacks to validate the strength of targeted classifiers such as MalConv or commercial AV products. The first attack used random action selection with modifications taken from [39]. For the second one, the authors studied which parts of the executable are critical for the decision-making of the targeted classifier. To detect the pivotal parts, they used a binary search algorithm. After locating the critical parts, Fleshman et al. changed the corresponding bytes at random

or used bytes extracted from benign binaries. Adversarial samples generated with these two types of attacks struggled to bypass detection by pure ML classifiers, but surprisingly the performance of tested AVs suffered. The last type of attack was in the form of injecting malicious code inside otherwise benign binaries. This attack proved extremely difficult to detect for all tested malware detectors reaching an evasion rate of more than 67%. Nevertheless, some of these modifications could cause malfunctioning of the generated adversarial malware samples, which was not tested in this work.

The authors of [63] proposed a generative sequence-to-sequence language model in the form of a recurrent neural network. This network was trained on benign binaries to generate adversarial benign bytes. These benign bytes were then appended to malware executables to generate adversarial malware examples. Their black-box attack could successfully evade detection by three state-of-the-art ML malware classifiers, such as MalConv. They achieved an evasion rate of 82.4% with an average append size of 5%. Their results show that increasing padding size increases the evasion rate but with diminishing returns for higher values. The authors conducted a behaviour analysis to validate that the functionality of malicious executables did not change after appending generated benign bytes.

In [64], a heuristic Monte Carlo tree search algorithm was proposed to find the optimal set of modifications to evade detection by the surrogate model in the form of a decision tree trained on the EMBER dataset. The authors managed to bypass the surrogate classifier in more than 56% of cases, while 52% of these cases only needed one modification in the form of changing the certificate signature. When transferring these learned modifications to the target classifier, they recorded a considerable decrease in evasion success, with only 8.79% of adversarial malware samples capable of bypassing the detector. Their work was conducted only inside extracted feature space, not on binary executables.

Another work by Demetrio et al. [65] presents a black-box attack named GAMMA. This attack tackles the problem of creating adversarial samples as an optimisation problem where the main criteria are maximal evasion and minimal size of inserted content. The optimisation problem is solved using a genetic algorithm which uses the traditional selection process, cross-over and mutation. In the training phase, the authors targeted the GBDT classifier to later attack real-world AVs hosted on the VirusTotal website, bypassing 12 out of 70 detectors on average.

4.4 Comparison with State-of-the-art Methods

It is generally difficult to compare our work with others, given the non-existence of standardised benchmarks and datasets. Moreover, not all researchers focus on preserving the original functionality of adversarial files, thus

creating evasive but non-functional samples. Additionally, some authors only created examples in feature space or targeted different platforms, so comparing their work with our exploratory-integrity attack on PE files is impossible. Furthermore, while many try to benchmark their adversarial samples on commercially available antivirus, the results are usually anonymised for security and licencing reasons, thus rendering them incomparable.

Nonetheless, our evasion rate of 58.92% against the GBDT classifier with the PPO agent is comparable, if not better, than most of the studied related works. One of the exceptions is the before mentioned MAB-malware [45]. Their 74.4% evasion rate against GBDT is significantly higher than our recorded results of 58.92%. Although they achieved higher transferability (over 80%) compared to our attack (slightly below 50%) between ML models, our attack transferred better to commercial AVs, bypassing detection more than 10% of the time and peaking at 25.7%. The possible improvements to our work are suggested later in Conclusion.

Conclusion

This thesis aimed to study adversarial machine learning techniques related to the area of malware detection. We crafter an exploratory-integrity attack using reinforcement learning algorithms on the space of PE binaries. We implemented an interactive environment in the OpenAI Gym format for training RL agents. The environment includes a PE file modifier with tested modifications that maximise the conservation of original functionality. We collected a dataset of Windows malware executables that we used for training and testing. We experimented with three reinforcement learning algorithms DQN, PG and PPO. Firstly, we determined the optimal number of modifications for each algorithm. Next, we fine-tuned various hyperparameters for each RL agent. Later, we chose the most promising configuration for extended training. From these configurations, we selected the best representative of each RL algorithm based on their performance on the validation set. Finally, with this selection of RL agents, we conducted a comparative study determining the agent with the highest evasion rate against the GBDT classifier and how the attacks transfer to another ML classifier and commercially available AV programs. Furthermore, we compared the method proposed by us and the results achieved with related research.

We accomplished a promising evasion rate against the GBDT classifier, peaking at 58.92% with the PPO agent ($\gamma = 0.5, \alpha = 0.0001$). The DQN followed closely behind with an evasion rate of 55.95%, while the PG agent performed the worst, bypassing the GBDT in only 40.14% of cases. Later, when we tested how our adversarial attack transferred to other malware classifiers, we saw significant performance drops of over 50%. The best agent, PPO, achieved evasion rates of 28.91% and 25.7% against the MalConv and real-world antivirus, respectively.

To summarize this thesis, we successfully performed an adversarial attack against various malware detectors using reinforcement learning algorithms. Our adversarial attack works at the level of samples with a strong emphasis on preserving the original functionality. This is maximised by our testing pro-

tocol, which could be used in future studies. More than half of the generated adversarial samples bypassed the targeted classifier, thus significantly increasing its false negative rate. Without any prior training against commercial AV programs, our attack carried over reasonably well, as shown by more than 10% of generated examples capable of evading detection by top antivirus products.

Discussion and Future work

The goal of this thesis was to implement an adversarial attack against a chosen malware detector. The next logical step would be to introduce a defence mechanism against our attack that could be incorporated into current detectors. A defensive method could be retraining with generated adversarial samples or in the form of a self-contained classifier of adversarial samples.

However, as we already mentioned in some of the previous chapters, our proposed approach to generating adversarial samples still has room for improvements. Firstly, one of our implemented PE modification (append new import) did not meet our criteria of conserving the functionality of the original file. It should be noted that improvements should be made before deploying this operation to other projects.

Next, comparing the original functionality with the Cuckoo sandbox is difficult as reported analyses are unstable, changing with repeated execution. A different comparison method or a more advanced analysis system could give more trust in generated binaries.

Further, our presented modifications could be extended with more and improved modifications to provide the agent with a more potent set of actions to bypass the target classifier. For example, we did not explore the possibilities of modifying the DOS header or changing the executable's entry point.

Likewise, the hyperparameters of reinforcement learning algorithms are known to be challenging to fine-tune. More extensive and granular hyperparameter search could be incorporated to optimize current RL algorithms. Furthermore, a modification or a novel algorithm for working with PE binaries could be presented.

Additionally, vast improvements could be made in the feature engineering of PE executables. While we used only raw extracted bytes from the beginning and end of the file, a set of more descriptive features could be found, e.g., behaviour features. These features could significantly improve malware classifiers' capabilities as they would become less susceptible to adversarial attacks. However, behaviour analyses are time-demanding and thus problematic to include in machine learning models due to prolong training stages.

In this thesis and other related works utilizing reinforcement learning, most authors utilize a set of hardcoded modifications for the agent. While this is a rational approach as it introduces external knowledge of the author about

the subject to the agent, an agent capable of generating its own modification could bring substantial improvements.

Further, attacking malware detectors in a black-box scenario is inherently difficult as the only non-zero reward is provided after the generated sample becomes evasive. This situation puts reinforcement learning agents under a challenging task to determine which modification or combination of modifications truly contributed to the evasion. Although we adopted this restricted approach to simulate real-world situations, model-stealing and subsequent white-box attack is a promising area for future research.

Finally, one of the unexplored areas of adversarial malware generation is altering actual malware behaviour while preserving the intended functionality. Since we only focused on static analysis of malware classifiers, our modifications were not intended to change behaviour. As a result, our attack would likely be powerless against dynamic analysis systems.

However, our work provides a solid implementation of reinforcement learning setup working at the level of samples while generating functional adversarial malware examples. Additionally, our modifications, agents and environment setup can be easily extended for future improvements.

Bibliography

1. INSTITUTE, AV-TEST. *Malware statistics & trends report: AV-TEST* [online]. 2022-01 [visited on 12/28/2022]. Available from: `https://www.av-test.org/en/statistics/malware/`.
2. SOPHOS. *Sophos Threat Report* [online]. 2022-11 [visited on 12/28/2022]. Available from: `https://www.sophos.com/en-us/content/security-threat-report`.
3. GERENCER, Tom. *The top 10 Worst Computer viruses in history* [online]. 2020-11 [visited on 12/28/2022]. Available from: `https://www.hp.com/us-en/shop/tech-takes/top-ten-worst-computer-viruses-in-history`.
4. ASLAN, Ömer Aslan; SAMET, Refik. A Comprehensive Review on Malware Detection Approaches. *IEEE Access* [online]. 2020, vol. 8, pp. 6249–6271 [visited on 12/28/2022]. Available from DOI: `10.1109/ACCESS.2019.2963724`.
5. UCCI, Daniele; ANIELLO, Leonardo; BALDONI, Roberto. Survey of machine learning techniques for malware analysis. *Computers & Security* [online]. 2019, vol. 81, pp. 123–147 [visited on 12/28/2022]. Available from DOI: `https://doi.org/10.1016/j.cose.2018.11.001`.
6. PAPERNOT, Nicolas; MCDANIEL, Patrick; JHA, Somesh; FREDRIKSON, Matt; CELIK, Z Berkay; SWAMI, Ananthram. The limitations of deep learning in adversarial settings. In: *2016 IEEE European symposium on security and privacy (EuroS&P)*. 2016, pp. 372–387. Available from DOI: `10.1109/EuroSP.2016.36`.
7. ERKO, Alexey. *Malware sandbox evasion: Techniques, principles & solutions* [online]. 2022-10 [visited on 12/27/2022]. Available from: `https://www.apriorit.com/dev-blog/545-sandbox-evading-malware`.

8. YUCEEL, Huseyin Can. *Virtualization/sandbox evasion - how attackers avoid malware analysis* [online]. Picus Güvenlik A.Ş., 2022-09 [visited on 12/28/2022]. Available from: https://www.picussecurity.com/resource/virtualization/sandbox-evasion-how-attackers-avoid-malware-analysis.

9. KOZÁK, Matouš; JUREČEK, Martin; LÓRENCZ, Róbert. Generation of Adversarial Malware and Benign Examples Using Reinforcement Learning. In: *Cybersecurity for Artificial Intelligence* [online]. Springer, 2022, pp. 3–25 [visited on 10/30/2022]. Available from DOI: https://doi.org/10.1007/978-3-030-97087-1_1.

10. MARIUS, Hucker. *Overview: State-of-the-art machine learning algorithms per discipline and per task* [online]. Towards Data Science, 2021-12 [visited on 12/06/2022]. Available from: https://towardsdatascience.com/overview-state-of-the-art-machine-learning-algorithms-per-discipline-per-task-c1a16a66b8bb.

11. SILVER, David; SCHRITTWIESER, Julian; SIMONYAN, Karen; HUANG, Aja; GUEZ, Arthur; HUBERT, Thomas; BAKER, Lucas; LAI, Matthew; BOLTON, Adrian; ANTONOGLOU, Ioannis, et al. Mastering the game of go without human knowledge. *Nature* [online]. 2017, vol. 550, no. 7676, pp. 354–359 [visited on 12/06/2022]. Available from DOI: https://doi.org/10.1038/nature24270.

12. JURAVLE, Georgiana; BOUDOURAKI, Andriana; TERZIYSKA, Miglena; REZLESCU, Constantin. Trust in artificial intelligence for medical diagnoses. *Progress in Brain Research* [online]. 2020, vol. 253, pp. 263–282 [visited on 12/06/2022]. Available from DOI: https://doi.org/10.1016/bs.pbr.2020.06.006.

13. EDMONDS, Ellen. *Three-quarters of Americans "afraid" To ride in a self-driving vehicle* [online]. 2020-11 [visited on 12/06/2022]. Available from: https://newsroom.aaa.com/2016/03/three-quarters-of-americans-afraid-to-ride-in-a-self-driving-vehicle/.

14. GILPIN, Leilani H.; BAU, David; YUAN, Ben Z.; BAJWA, Ayesha; SPECTER, Michael A.; KAGAL, Lalana. Explaining Explanations: An Approach to Evaluating Interpretability of Machine Learning. *CoRR* [online]. 2018, vol. abs/1806.00069 [visited on 12/06/2022]. Available from DOI: 10.48550/ARXIV.1806.00069.

15. HUANG, Ling; JOSEPH, Anthony D.; NELSON, Blaine; RUBINSTEIN, Benjamin I.P.; TYGAR, J. D. Adversarial Machine Learning. In: *Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence* [online]. Chicago, Illinois, USA: Association for Computing Machinery, 2011, pp. 43–58 [visited on 07/13/2021]. AISec '11. ISBN 9781450310031. Available from DOI: 10.1145/2046684.2046692.

16. FANG, Yong; ZENG, Yuetian; LI, Beibei; LIU, Liang; ZHANG, Lei. DeepDetectNet vs RLAttackNet: An adversarial method to improve deep learning-based static malware detection model. *Plos one* [online]. 2020, vol. 15, no. 4, e0231626 [visited on 07/15/2021]. Available from DOI: https://doi.org/10.1371/journal.pone.0231626.

17. RATHORE, Hemant; SAHAY, Sanjay K; NIKAM, Piyush; SEWAK, Mohit. Robust Android Malware Detection System Against Adversarial Attacks Using Q-Learning [online]. 2021, vol. 23, no. 4, pp. 867–882 [visited on 07/15/2021]. Available from DOI: https://doi.org/10.1007/s10796-020-10083-8.

18. CHEN, Lingwei; YE, Yanfang; BOURLAI, Thirimachos. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In: *2017 European Intelligence and Security Informatics Conference (EISIC)* [online]. 2017, pp. 99–106 [visited on 08/05/2021]. Available from DOI: 10.1109/EISIC.2017.21.

19. GROSSE, Kathrin; PAPERNOT, Nicolas; MANOHARAN, Praveen; BACKES, Michael; MCDANIEL, Patrick. Adversarial examples for malware detection. In: *Computer Security – ESORICS 2017* [online]. Cham: Springer International Publishing, 2017, pp. 62–79 [visited on 08/03/2021]. ISBN 978-3-319-66399-9. Available from DOI: https://doi.org/10.1007/978-3-319-66399-9_4.

20. RASHID, Aqib; SUCH, Jose. StratDef: a strategic defense against adversarial attacks in malware detection. *arXiv preprint arXiv:2202.07568* [online]. 2022 [visited on 06/16/2022]. Available from DOI: 10.48550/ARXIV.2202.07568.

21. DEMETRIO, Luca; BIGGIO, Battista; LAGORIO, Giovanni; ROLI, Fabio; ARMANDO, Alessandro. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. *ArXiv* [online]. 2019, vol. abs/1901.03583 [visited on 08/05/2021]. Available from DOI: 10.48550/ARXIV.1901.03583.

22. SUTTON, Richard S; BARTO, Andrew G. *Reinforcement learning: An introduction* [online]. MIT press, 2018 [visited on 09/30/2021]. Available from DOI: https://doi.org/10.1016/S1364-6613(99)01331-5.

23. RUMMERY, Gavin A; NIRANJAN, Mahesan. On-line Q-learning using connectionist systems. *Technical Report CUED/F-INFENG/TR 166* [online]. 1994, vol. 37 [visited on 01/06/2022]. Available from: http://mi.eng.cam.ac.uk/reports/svr-ftp/auto-pdf/rummery_tr166.pdf.

24. WATKINS, Christopher John Cornish Hellaby. Learning from delayed rewards [online]. 1989 [visited on 09/25/2021]. Available from: https://www.researchgate.net/publication/33784417_Learning_From_Delayed_Rewards.

25. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; GRAVES, Alex; ANTONOGLOU, Ioannis; WIERSTRA, Daan; RIEDMILLER, Martin. Playing atari with deep reinforcement learning. *CoRR* [online]. 2013, vol. abs/1312.5602 [visited on 10/29/2022]. Available from DOI: 10.48550/ARXIV.1312.5602.

26. MNIH, Volodymyr; KAVUKCUOGLU, Koray; SILVER, David; RUSU, Andrei A; VENESS, Joel; BELLEMARE, Marc G; GRAVES, Alex; RIEDMILLER, Martin; FIDJELAND, Andreas K; OSTROVSKI, Georg, et al. Human-level control through deep reinforcement learning. *Nature* [online]. 2015, vol. 518, no. 7540, pp. 529–533 [visited on 12/25/2021]. Available from DOI: https://doi.org/10.1038/nature14236.

27. SUTTON, Richard S; MCALLESTER, David; SINGH, Satinder; MANSOUR, Yishay. Policy Gradient Methods for Reinforcement Learning with Function Approximation. In: SOLLA, S.; LEEN, T.; MÜLLER, K. (eds.). *Advances in Neural Information Processing Systems* [online]. MIT Press, 1999, vol. 12, pp. 1057–1063 [visited on 11/05/2021]. Available from: https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf.

28. SCHULMAN, John; WOLSKI, Filip; DHARIWAL, Prafulla; RADFORD, Alec; KLIMOV, Oleg. Proximal policy optimization algorithms. *CoRR* [online]. 2017, vol. abs/1707.06347 [visited on 12/25/2021]. Available from DOI: https://doi.org/10.48550/arXiv.1707.06347.

29. JADERBERG, Max; MNIH, Volodymyr; CZARNECKI, Wojciech Marian; SCHAUL, Tom; LEIBO, Joel Z; SILVER, David; KAVUKCUOGLU, Koray. Reinforcement learning with unsupervised auxiliary tasks. *CoRR* [online]. 2016, vol. abs/1611.05397 [visited on 10/29/2022]. Available from DOI: 10.48550/ARXIV.1611.05397.

30. SILVER, David; HUANG, Aja; MADDISON, Chris J; GUEZ, Arthur; SIFRE, Laurent; VAN DEN DRIESSCHE, George; SCHRITTWIESER, Julian; PANNEERSHELVAM, Veda; LANCTOT, Marc; ANTONOGLOU, Ioannis, et al. Mastering the game of Go with deep neural networks and tree search. *Nature* [online]. 2016, vol. 529, no. 7587, pp. 484–489 [visited on 10/29/2022]. Available from DOI: https://doi.org/10.1038/nature16961.

31. LEVINOVITZ, Alan. *The Mystery of Go, the Ancient Game That Computers Still Can't Win* [online]. Conde Nast, 2014-05 [visited on 12/19/2022]. Available from: https://www.wired.com/2014/05/the-world-of-computer-go/.

32. BERNER, Christopher; BROCKMAN, Greg; CHAN, Brooke; CHEUNG, Vicki; DĘBIAK, Przemysław; DENNISON, Christy; FARHI, David; FISCHER, Quirin; HASHME, Shariq; HESSE, Chris, et al. Dota 2 with large

scale deep reinforcement learning. *CoRR* [online]. 2019, vol. abs/1912.06680 [visited on 10/29/2022]. Available from DOI: 10.48550/ARXIV.1912.06680.

33. VINYALS, Oriol; BABUSCHKIN, Igor; CZARNECKI, Wojciech M; MATHIEU, Michaël; DUDZIK, Andrew; CHUNG, Junyoung; CHOI, David H; POWELL, Richard; EWALDS, Timo; GEORGIEV, Petko, et al. Grandmaster level in StarCraft II using multi-agent reinforcement learning. *Nature* [online]. 2019, vol. 575, no. 7782, pp. 350–354 [visited on 10/30/2022]. Available from DOI: https://doi.org/10.1038/s41586-019-1724-z.

34. LAZIC, Nevena; BOUTILIER, Craig; LU, Tyler; WONG, Eehern; ROY, Binz; RYU, MK; IMWALLE, Greg. Data center cooling using model-predictive control. In: BENGIO, S.; WALLACH, H.; LAROCHELLE, H.; GRAUMAN, K.; CESA-BIANCHI, N.; GARNETT, R. (eds.). *Advances in Neural Information Processing Systems* [online]. Curran Associates, Inc., 2018, vol. 31, pp. 3818–3827 [visited on 10/29/2022]. Available from: https://proceedings.neurips.cc/paper/2018/file/059fdcd96baeb75112f09fa1dcc740cc-Paper.pdf.

35. KOWALCZYK, Krzysztof. *Portable Executable File Format* [online]. 2018-07 [visited on 07/13/2022]. Available from: https://blog.kowalczyk.info/articles/pefileformat.html.

36. KARL BRIDGE, Microsoft. *PE Format - Win32 apps* [online]. 2019-08 [visited on 07/13/2022]. Available from: https://docs.microsoft.com/en-us/windows/win32/debug/pe-format.

37. PIETREK, Matt. *An In-Depth Look into the Win32 Portable Executable File Format* [online]. 2008-02 [visited on 07/14/2022]. Available from: https://docs.microsoft.com/en-us/previous-versions/bb985992(v=msdn.10)?redirectedfrom=MSDN.

38. BROCKMAN, Greg; CHEUNG, Vicki; PETTERSSON, Ludwig; SCHNEIDER, Jonas; SCHULMAN, John; TANG, Jie; ZAREMBA, Wojciech. OpenAI Gym. *CoRR* [online]. 2016, vol. abs/1606.01540 [visited on 10/25/2021]. Available from DOI: 10.48550/ARXIV.1606.01540.

39. ANDERSON, Hyrum S; KHARKAR, Anant; FILAR, Bobby; EVANS, David; ROTH, Phil. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. *CoRR* [online]. 2018, vol. abs/1801.08917 [visited on 09/25/2021]. Available from DOI: https://doi.org/10.48550/arXiv.1801.08917.

40. THOMAS, Romain. *LIEF - Library to Instrument Executable Formats* [online]. 2017-04 [visited on 09/26/2021]. Available from: https://lief.quarkslab.com/.

41. CARRERA, E. *Pefile* [online]. 2017 [visited on 07/13/2022]. Available from: https://github.com/erocarrera/pefile.

42. MARKUS, Oberhumer; LASZLO, Molnar; JOHN, Reiser. *UPX, The Ultimate Packer for eXecutables* [online]. GitHub, 2012 [visited on 12/11/2022]. Available from: https://github.com/upx/upx/releases/download/v4.0.1/upx-4.0.1-amd64_linux.tar.xz.

43. SOFT, A.S.L. *Exeinfo PE* [online]. GitHub, 2016 [visited on 12/23/2022]. Available from: https://github.com/ExeinfoASL/ASL.

44. CHADNJ, Vaya; BEDANG, Sen. *Pesidious, Malware Mutation using Deep Reinforcement Learning and GANs* [online]. GitHub, 2019 [visited on 08/15/2021]. Available from: https://github.com/CyberForce/Pesidious.

45. SONG, Wei; LI, Xuezixiang; AFROZ, Sadia; GARG, Deepali; KUZNETSOV, Dmitry; YIN, Heng. Mab-malware: A reinforcement learning framework for attacking static malware classifiers. *arXiv preprint arXiv:2003.03100* [online]. 2020, vol. abs/2003.03100 [visited on 06/17/2022]. Available from DOI: 10.48550/ARXIV.2003.03100.

46. RUKAIMI. *PE Bliss, Cross-Platform Portable Executable C++ Library* [online]. GitHub, 2012 [visited on 12/11/2022]. Available from: https://github.com/BackupGGCode/portable-executable-library.

47. RAFF, Edward; BARKER, Jon; SYLVESTER, Jared; BRANDON, Robert; CATANZARO, Bryan; NICHOLAS, Charles. Malware Detection by Eating a Whole EXE [online]. 2017 [visited on 12/01/2021]. Available from DOI: 10.48550/ARXIV.1710.09435.

48. ANDERSON, Hyrum S.; ROTH, Phil. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. *CoRR* [online]. 2018, vol. abs/1804.04637 [visited on 12/01/2021]. Available from DOI: 10.48550/ARXIV.1804.04637.

49. LIANG, Eric; LIAW, Richard; NISHIHARA, Robert; MORITZ, Philipp; FOX, Roy; GONZALEZ, Joseph; GOLDBERG, Ken; STOICA, Ion. Ray RLLib: A Composable and Scalable Reinforcement Learning Library. *CoRR* [online]. 2017, vol. abs/1712.09381 [visited on 08/15/2021]. Available from DOI: https://doi.org/10.48550/arXiv.1712.09381.

50. *Ray RLLib Documentation* [online]. [N.d.] [visited on 01/11/2022]. Available from: https://docs.ray.io/en/latest/rllib/index.html.

51. *VirusShare* [online]. [N.d.] [visited on 08/08/2021]. Available from: https://virusshare.com/.

52. *VirusTotal* [online]. [N.d.] [visited on 12/01/2022]. Available from: https://www.virustotal.com/.

53. AV-COMPARATIVES. *Malware protection test September 2022* [online]. 2022-10 [visited on 12/11/2022]. Available from: https://www.av-comparatives.org/tests/malware-protection-test-september-2022/.

54. RIGAKI, Maria; GARCIA, Sebastian. Stealing Malware Classifiers and AVs at Low False Positive Conditions. *arXiv preprint arXiv:2204.06241* [online]. 2022 [visited on 06/09/2022]. Available from DOI: 10.48550/ARXIV.2204.06241.

55. DO THI THU, Hien; PHAN, The Duy; LE ANH, Hao; NGUYEN DUY, Lan; NGHI HOANG, Khoa; PHAM, Van-Hau. A Method of Mutating Windows Malwares using Reinforcement Learning with Functionality Preservation. In: *The 11th International Symposium on Information and Communication Technology* [online]. 2022, pp. 142–149 [visited on 12/07/2022]. ISBN 9781450397254. Available from DOI: https://doi.org/10.1145/3568562.3568631.

56. QUERTIER, Tony; MARAIS, Benjamin; MORUCCI, Stéphane; FOURNEL, Bertrand. MERLIN–Malware Evasion with Reinforcement LearnINg. *arXiv preprint arXiv:2203.12980* [online]. 2022 [visited on 06/15/2022]. Available from DOI: 10.48550/ARXIV.2203.12980.

57. KOLOSNJAJI, Bojan; DEMONTIS, Ambra; BIGGIO, Battista; MAIORCA, Davide; GIACINTO, Giorgio; ECKERT, Claudia; ROLI, Fabio. Adversarial malware binaries: Evading deep learning for malware detection in executables. In: *2018 26th European signal processing conference (EUSIPCO)* [online]. 2018, pp. 533–537 [visited on 08/05/2021]. Available from DOI: 10.23919/EUSIPCO.2018.8553214.

58. KREUK, Felix; BARAK, Assi; AVIV-REUVEN, Shir; BARUCH, Moran; PINKAS, Benny; KESHET, Joseph. Deceiving End-to-End Deep Learning Malware Detectors using Adversarial Examples. *CoRR* [online]. 2019, vol. abs/1802.04528 [visited on 08/05/2021]. Available from DOI: 10.48550/ARXIV.1802.04528.

59. YANG, Chun; XU, Jinghui; LIANG, Shuangshuang; WU, Yanna; WEN, Yu; ZHANG, Boyang; MENG, Dan. DeepMal: maliciousness-Preserving adversarial instruction learning against static malware detection. *Cybersecurity*. 2021, vol. 4, no. 1, pp. 1–14. Available from DOI: https://doi.org/10.1186/s42400-021-00079-5.

60. HU, Weiwei; TAN, Ying. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. *CoRR* [online]. 2017, vol. abs/1702.05983 [visited on 08/05/2021]. Available from DOI: 10.48550/ARXIV.1702.05983.

61. CHEN, Sen; XUE, Minhui; FAN, Lingling; HAO, Shuang; XU, Lihua; ZHU, Haojin; LI, Bo. Automated poisoning attacks and defenses in malware detection systems: An adversarial machine learning approach. *Computers & Security* [online]. 2018, vol. 73, pp. 326–344 [visited on 08/05/2021]. ISSN 0167-4048. Available from DOI: https://doi.org/10.1016/j.cose.2017.11.007.

62. FLESHMAN, William; RAFF, Edward; ZAK, Richard; MCLEAN, Mark; NICHOLAS, Charles. 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)* [online]. 2018, pp. 1–10 [visited on 10/02/2022]. ISBN 978-1-7281-0155-2. Available from DOI: 10.1109/MALWARE.2018.8659360.

63. EBRAHIMI, Mohammadreza; ZHANG, Ning; HU, James; RAZA, Muhammad Taqi; CHEN, Hsinchun. Binary Black-box Evasion Attacks Against Deep Learning-based Static Malware Detectors with Adversarial Byte-Level Language Model. *CoRR* [online]. 2020, vol. abs/2012.07994 [visited on 08/05/2021]. Available from DOI: 10.48550/ARXIV.2012.07994.

64. BOUTSIKAS, John; EREN, Maksim E.; VARGA, Charles; RAFF, Edward; MATUSZEK, Cynthia; NICHOLAS, Charles. Evading Malware Classifiers via Monte Carlo Mutant Feature Discovery. *CoRR* [online]. 2021, vol. abs/2106.07860 [visited on 08/05/2021]. Available from DOI: https://doi.org/10.48550/arXiv.2106.07860.

65. DEMETRIO, Luca; BIGGIO, Battista; LAGORIO, Giovanni; ROLI, Fabio; ARMANDO, Alessandro. Functionality-Preserving Black-Box Optimization of Adversarial Windows Malware. *IEEE Transactions on Information Forensics and Security* [online]. 2021, vol. 16, pp. 3469–3478 [visited on 10/06/2022]. ISSN 1556-6021. Available from DOI: 10.1109/TIFS.2021.3082330.

Appendix A

Acronyms

API Application Programming Interface

AV Antivirus

COFF Common Object File Format

DDoS Distributed Denial of Service

DLL Dynamically Linked Library

DQN Deep Q Network

EXE Executable (.exe)

FNR False Negative Rate

FPR False Positive Rate

GAN Generative Adversarial Network

GBDT Gradient Boosted Decision Tree

IAT Import Address Table

IDT Import Directory Table

ILT Import Lookup Table

IP Internet Protocol

LGBM Light Gradient-Boosting Machine (LightGBM)

lr Learning Rate

ML Machine Learning

OS Operating System

PE Portable Executable

PPO Proximal Policy Optimization

RL Reinforcement Learning

RVA Relative Virtual Address

Appendix B

Contents of enclosed DVD

- `README.md` ..... Markdown file with thesis description
- `src` ..... Directory with implementation source codes
  - `AMG` ..... Directory with AMG framework
  - `PE_file_modifications` ..... Directory with testing of modifications
  - `README.md` ..... Markdown file with implementation description
- `text` ..... Directory with thesis text and LaTeX source codes
  - `bib` ..... Directory with bibliography files
  - `fig` ..... Directory with figures
  - `tex` ..... Directory with LaTeX source codes
  - `DP_Kozak_Matous_2022.tex` ..... Main LaTeX document
  - `DP_Kozak_Matous_2022.pdf` ..... Thesis in PDF format