FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Assignment of master's thesis

Title:	A Comparison of Adversarial Learning Techniques for Malware Detection
Student:	Bc. Pavla Louthánová
Supervisor:	Mgr. Martin Jureček, Ph.D.
Study program:	Informatics
Branch / specialization:	Computer Security
Department:	Department of Information Security
Validity:	until the end of summer semester 2023/2024

Instructions

The aim of the adversarial learning technique is the deliberate modification of input data, which causes a reduction in the accuracy of classification. State-of-the-art methods for creating adversarial examples fall into various areas, such as reinforcement learning, genetic algorithms, or gradient-based techniques. The contribution of this work would be to apply some existing methods in the field of adversarial learning to selected malware detection systems and compare them in terms of accuracy and usability in practice.

Instructions:

1. Study techniques of adversarial machine learning, focusing mainly on the area of malware detection.
2. Use existing tools (e.g., SecML Malware) or implement at least three different techniques for creating adversarial malware samples.
3. Test the effectiveness of the techniques against at least three malware detectors regarding evasion rate and usability in practice.
4. Discuss the results and compare them with related work.

Electronically approved by prof. Ing. Róbert Lórencz, CSc. on 15 January 2023 in Prague.

FACULTY
OF INFORMATION
TECHNOLOGY
CTU IN PRAGUE

Master’s thesis

A Comparison of Adversarial Learning Techniques for Malware Detection

Bc. Pavla Louthánová

Department of Information Security
Supervisor: Mgr. Martin Jureček, Ph.D.

May 4, 2023

Acknowledgements

I would like to thank my supervisor Mgr. Martin Jureček, Ph.D. for his guidance, support, and patience during the work on this thesis.

Declaration

In Prague on May 4, 2023 . . . .

Citation of this thesis

Louthánová, Pavla. *A Comparison of Adversarial Learning Techniques for Malware Detection.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2023.

Abstrakt

Malware je dnes jednou z nejvýznamnějších bezpečnostních hrozeb. Pro účinnou ochranu před malwarem je zásadní jeho včasná detekce. Strojové učení se ukázalo jako užitečný nástroj pro automatickou detekci malwaru. Výzkum však ukázal, že modely strojového učení jsou zranitelné vůči adversariálním útokům. Tato práce se zabývá adversariálními učícími technikami v oblasti detekce malwaru. Cílem bylo aplikovat některé existující metody pro generování vzorků adversariálního malwaru, otestovat jejich účinnost proti vybraným detektorům malwaru, porovnat dosaženou míru úniku a praktickou použitelnost. Práce začíná úvodem do adversariálního strojového učení, následuje popis portable executable formátu souborů a přehled publikací, které se zaměřují na vytváření adversariálních vzorků malwaru. Dále jsou popsány techniky použité k vytvoření vzorků malwaru pro experimentální vyhodnocení. Nakonec jsou popsány provedené experimenty, zahrnující sledování času potřebného k vytvoření vzorků, změn velikosti vzorku po použití generátoru, testování účinnosti proti antivirovým programům, kombinování aplikace více generátorů na vzorek a jejich vyhodnocení. Pro účely experimentů bylo vybráno pět generátorů: Partial DOS, Full DOS, GAMMA padding, GAMMA section-injection a Gym-malware. Výsledky ukázaly, že použití optimalizovaných modifikací, na dříve detekovaný malware, může vést k nesprávnému vyhodnocení klasifikátorem jako benigního souboru. Bylo také zjištěno, že vygenerované vzorky škodlivého softwaru lze úspěšně použít proti jiným detekčním modelům, než které byly použity k jejich vygenerování, a že použitím kombinací generátorů lze vytvořit nové vzorky, které se vyhnou detekci. Experimenty ukazují, že největší potenciál v praxi má generátor Gym-malware, který využívá přístup zpětnovazebního učení. Tento generátor dosáhl průměrné doby generování vzorku 5,73 sekundy a nejvyšší míry úniku 67 %. Při použití v kombinaci se sebou samým, se míra úniku zlepšila na 78 %.

Klíčová slova adversariální vzorky, detekce malwaru, strojové učení, PE soubory

Abstract

Malware is one of the most significant security threats today. Early detection is important for effective malware protection. Machine learning has proven to be a useful tool for automated malware detection. However, research has shown that machine learning models are vulnerable to adversarial attacks. This thesis discusses adversarial learning techniques in malware detection. The aim is to apply some existing methods for generating adversarial malware samples, test their effectiveness against selected malware detectors, and compare the evasion rate achieved and their practical applicability. The thesis begins with an introduction to adversarial machine learning, followed by a description of the portable executable file format and a review of publications that focus on generating adversarial malware samples. The techniques used to generate malware samples for experimental evaluation are then presented. Finally, the experiments performed are described, including observation of the time required to generate samples, changes in sample size after using the generator, testing effectiveness against antivirus programs, combining the use of multiple generators to generate samples, and evaluation of the results. Five generators were selected for the experiments: Partial DOS, Full DOS, GAMMA padding, GAMMA section-injection and Gym-malware. The results showed that applying optimised modifications to previously detected malware can lead to incorrect classification of the file as benign. It was also found that generated malware samples can be successfully used against detection models other than those used to generate them, and that using combinations of generators can create new samples that evade detection. Experiments show that the Gym-malware generator, which uses a reinforcement learning approach, has the greatest practical potential. This generator achieved an average sample generation time of 5.73 seconds and the highest evasion rate of 67%. When used in combination with itself, the evasion rate improved to 78%.

Keywords adversarial examples, malware detection, machine learning, PE files

Contents

List of Figures

List of Tables

Introduction

Technology has become an essential part of our lives. We interact with technology every day in both our personal and professional lives. Unfortunately, as technology continues to evolve and expand its reach into different areas, cyber-attacks are on the rise. As a result, cybersecurity is becoming increasingly important.

One of the most common types of attack is malware [1]. Malware is any malicious software created with malicious intent. Malware can come in many forms. Examples of malware types include viruses, worms, trojan horses, spyware, and ransomware. Today, malware is developed for a variety of operating systems, including Windows, Linux, MacOS, and Android. However, the most common target of attacks is the Windows operating system [2], mainly due to its widespread use in both personal and professional environments.

Early detection of malware is critical for protecting computers and the Internet. However, malware detection is a very challenging area due to a large amount of new malicious codes that are created every day [3]. Since it is not possible to analyse each sample individually, automated mechanisms are needed to detect malware.

Antivirus companies typically use signature-based detection [4]. Signatures are specific patterns that can be used to identify malicious files, for example, a sequence of bytes, a file hash, or a string. When the file is analysed, the antivirus system compares its contents with the signatures of known malware already stored in the database. If a match is found, the file is flagged as malware. Signature-based detection methods are fast and effective in detecting known malware. However, they are limited in that they cannot detect new malware that does not yet have a signature in the antivirus database. Malware authors can modify malicious code to change the signature of the program to avoid later detection. To do this, they use obfuscation techniques such as encryption, oligomorphic, polymorphic, metamorphic, stealth, and packing methods [5]. Obfuscation aims to make a malware code more difficult to analyse by trying to hide its true behaviour. Due to these weaknesses, antivirus companies also use a behaviour-based method to track file behaviour. This method can detect some unknown or obfuscated malware. However, this process can be time-consuming and inefficient.

Today, machine learning models are being applied in various areas of human life. For example, they can be found in language translation systems, image recognition systems, weather forecasting systems, recommendation systems, self-driving cars, and disease diagnosis systems. Machine learning has also proven to be a useful tool in cybersecurity. For example, it has been used to automate malware detection [5]. Machine learning can make analysis easier and potentially help detect malware more efficiently than other approaches. Unlike signature-based methods, it can detect previously unknown or obfuscated malware. However, it can be difficult to explain why the model classifies a particular file as malicious or benign [6], which can leave hidden vulnerabilities for attackers to exploit.

While machine learning models have many potential benefits, they have been shown to be vulnerable to adversarial attacks [7]. Attackers can create adversarial examples, which are deliberately modified input data to a machine learning model, to mislead the model into making an error in its predictions. For example, in malware detection, a small modification to a malware file can often cause it to be misclassified as a benign file. Adversarial machine learning is the field that studies attacks on machine learning models and defences against those attacks.

Malware detection seems to be a never-ending battle between malware authors and defenders [8]. Attackers are constantly looking for new ways to evade detection and compromise systems, whilst defenders develop new methods to detect and prevent these attacks. Each malware detection method has its advantages and disadvantages. In different scenarios, one method may be more successful than another. At the same time, relying on a single detection method may not be sufficient. Creating an effective malware detection method is a very challenging task that requires continuous research and development in the field of cybersecurity. Research is essential to stay ahead of new and emerging threats, as well as to improve the accuracy and effectiveness of existing detection methods.

The aim of this thesis is to introduce adversarial machine learning techniques in the field of malware detection, apply some existing methods to generate adversarial malware samples, and then test the effectiveness of these techniques against selected malware detectors, comparing their evasion rate and practical usability.

The thesis is organised into the following five chapters:

- Chapter 1 introduces the field of adversarial machine learning, including basic terminology and taxonomy. The state-of-the-art techniques used to generate adversarial examples are described, as well as ways to defend against adversarial attacks.
- Chapter 2 contains a description of the format of portable executable files and also describes the different types of manipulation of portable executable malware files to create adversarial malware samples.
- Chapter 3 provides an overview of the publications focused on the creation of adversarial portable executable malware samples. The chapter is divided into several sections based on the approach used to create adversarial samples.
- Chapter 4 describes the techniques used to create adversarial malware samples in this thesis.
- Chapter 5 describes the experiments performed and their evaluation. The purpose of the experiments, their setup, the metrics used for evaluation, and the datasets are presented. In addition, individual experiments are described and their results are discussed.

Chapter 1

Adversarial Machine Learning

In this chapter, we first introduce the field of adversarial machine learning, including its basic terminology and taxonomy. Then, we present state-of-the-art techniques used to create adversarial samples. Finally, we explain how to defend against adversarial attacks.

1.1 Terminology and Taxonomy

Adversarial machine learning (AML) is a subfield of machine learning (ML) that focuses on the vulnerability of ML models to adversarial attacks. An attacker, commonly known as an adversary, can manipulate the input data of an ML model to produce incorrect results, which may have serious consequences. To do this, the adversary creates manipulated input data, called an adversarial example (AE).

The process of creating adversarial examples is called an adversarial attack and can be performed using several different techniques, such as gradient-based methods, reinforcement learning, generative adversarial networks, and evolutionary algorithms. The difference between the original example and its adversarial version is called the adversarial perturbation, which can be very small but still significantly change the behaviour of the ML model.

The transferability refers to the ability of an adversarial example to remain harmful to models other than the one used to generate it. This means that if an attacker successfully creates an adversarial example for one model, that example can also be used to attack other models [9].

Adversarial attacks can be classified according to several different aspects. These categories include the stage of learning at which the attack is executed, the adversary's knowledge of the target model, the adversary's space, and the adversary's goals.

1.1.1 Attack Timing

Machine learning involves a training phase, during which the model learns, and a testing phase, during which the model is tested on a new unlabelled data. Adversarial attacks on a machine learning system can occur during the training or the testing/deployment phase.

Attacks during the training phase are called **poisoning attacks**. An attacker attempts to gain control over some of the training data, its labels, model parameters, or the code of the machine learning algorithm. For example, the attacker can inject malicious data into the training dataset to influence the model parameters and cause the model to misclassify future inputs.

Attacks during the testing/deployment phase can be divided into two different types. The first type is called **evasion attacks**, where the attacker's goal is to modify test samples to create adversarial examples that are similar to the original samples, but change the model's prediction. The second type is called **privacy attacks**, where the attacker tries to infer sensitive information about training data or may try to gain information about the parameters or internal workings of the ML model [10].

1.1.2 Adversarial Knowledge

The attacker may have varying degrees of knowledge of the target model. This knowledge can range from complete knowledge of the target system to zero knowledge. It can include training data, parameters, feature sets, learning algorithms, or objective functions that are minimised during training. Depending on the level of this knowledge; different attack scenarios can be described.

In the case of a **white-box attack** scenario, the attacker has perfect knowledge of the target model, including its architecture, learning method, parameter values, and possibly even training data.

On the contrary, in a **black-box attack** scenario, the attacker has no knowledge of the target model. However, by using the target model as an oracle, the attacker can gain knowledge of the relevant output for each submitted sample (for example, in the case of malware detection, whether the file is classified as malware or benign).

In a **grey-box attack** scenario, the attacker has limited knowledge of the target model. This is a scenario where the attacker has more knowledge than in a black-box attack, but less knowledge than in a white-box attack. Typically, this means that the attacker knows the set of features and the learning algorithm, but does not know the classifier parameters or the training data [7].

1.1.3 Adversarial Space

From the perspective of the space in which an attacker creates adversarial examples, adversarial attacks can be divided into feature-space attacks and problem-space attacks as described in [11].

The problem space $\mathbb{Z}$, also known as the input space, contains samples from a particular domain, such as images or portable executable (PE) files. Each sample $z \in \mathbb{Z}$ in the problem space is assigned a ground-truth label $y \in \mathbb{Y}$, where $\mathbb{Y}$ denotes the corresponding label space. In the context of PE malware detection, the problem space refers to the space of all possible PE files, and the label space denotes the space of detection labels, such as $\mathbb{Y} = \{0, 1\}$, where 0 denotes malware, and 1 denotes goodware.

In the case of **feature-space attack**, we have a classification model f and a given input sample z with a corresponding representation of the features $x \in \mathbb{X}$, where $\mathbb{X}$ denotes the feature space that numerically describes the internal structure of the samples in the problem space. This can be written as $f(x) = y$. The attacker tries to minimise the distance between x' and x in the feature space so that the resulting adversarial example $x' \in \mathbb{X}$ is misclassified by the classification model f:

$$\begin{aligned} &\min_{x'} \text{distance}(x', x) \\ &f(x') = y' \neq y \end{aligned} \tag{1.1}$$

where distance(x', x) is the distance metric between x' and x in the feature space.

In the case of **problem-space attack**, the attacker aims to modify the example $z \in \mathbb{Z}$ with minimal cost so that the resulting adversarial example $z' \in \mathbb{Z}$ in the problem space can also be misclassified by the target model f in the following way:

$$\begin{aligned} &\min_{z'} \text{cost}(z', z) \\ &f(\phi(z')) = y' \neq y \end{aligned} \tag{1.2}$$

where cost(z', z) is the cost function that transforms z into z' in the problem space, and ϕ is the feature mapping function defined as $\phi : \mathbb{Z} \to \mathbb{X}$. In other words, the feature mapping function ϕ maps the problem space to the feature space.

The inverse feature mapping function ϕ^{-1} can then be used to find the corresponding sample z' from the generated adversarial sample x', that is, $\phi^{-1}(x') = z'$.

1.1.4 Adversarial Goals

The aim of the attacker is to fool the target ML model into producing incorrect results. According to [12], the attacker's goals are divided into three categories: Untargeted Misclassification, Targeted Misclassification, and Confidence Reduction.

In the case of **Untargeted Misclassification**, the attacker tries to change the output of the model to a different value from its original prediction. For example, if the ML model predicts that a malware file belongs to the family A, the attacker will try to get the model to misclassify it as a different family.

On the other hand, in the case of **Targeted Misclassification**, the attack is directed at a specific target. The attacker tries to change the output of the model to the target value. For example, if an ML model predicts that a malware file is family A, the attacker's goal is to force the model to misclassify it as family B.

The final category is **Confidence Reduction**, where the attacker's goal is to reduce confidence in the prediction of the ML model. It is not necessary to change the prediction value; a simple confidence reduction is sufficient to achieve the goal.

1.2 Adversarial Example Generation

When creating adversarial examples in various areas of machine learning, it may be necessary to use different techniques and approaches. In the case of images, the data is represented as pixels. By adding imperceptible noise to the input image, an adversarial example can be created to cause the machine learning model to misclassify the image. An example of this attack is shown in Figure 1.1.

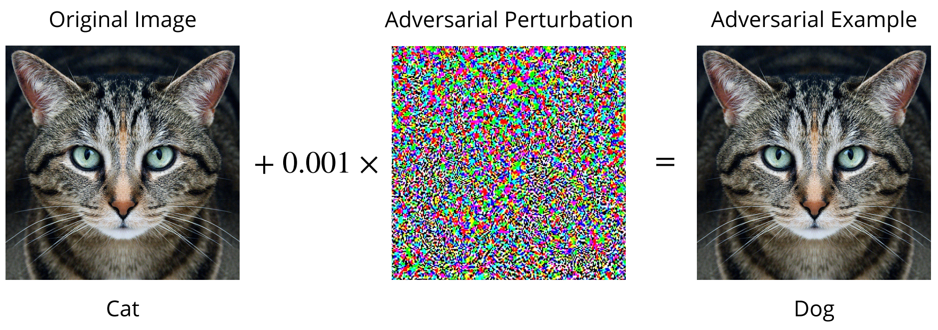

Figure 1.1: Example of an adversarial attack in the image domain [13].

Creating adversarial malware examples requires more sophisticated techniques than creating adversarial images. Executable files have a much more complex structure, as they are composed of a sequence of bytes. It is important to note that changing pixels in an image is not the same as changing bytes in an executable file.

Adversarial attacks in the field of malware are based on strategically modifying, inserting, or removing certain bytes in the original malware file in such a way that its original properties are not affected (preserving the file format, executability, and maliciousness) while generating adversarial malware that is incorrectly classified by the target malware detection model (such as an antivirus tool or a machine learning-based malware classifier) as benign.

Preserving the original properties of a PE file is not an easy task. A PE file has a fixed structure that must be followed to properly load the file into memory and execute it. The PE file format only determines its structure, but cannot ensure the correctness of the content of individual file elements. An improper file transformation can cause the modified PE file to crash during runtime and prevent it from running normally. However, file transformation can also damage the original functionality of the file and cause adversarial malware to no longer perform the same malicious behaviour as the original file. Such a file would not meet the objective of an adversarial attack. Therefore, simply preserving the file format does not necessarily mean that it will remain executable, and preserving executability does not necessarily mean that the original maliciousness of the PE malware will be preserved [11]. Figure 1.2 illustrates the relationship between the three challenges of adversarial attacks in the field of PE malware.

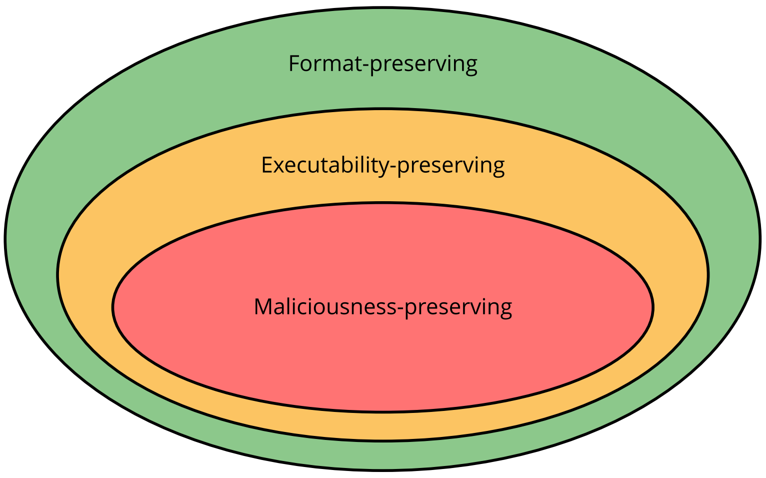

Figure 1.2: Diagram showing the relationship between the three challenges of adversarial attacks on PE malware: format-preserving, executability-preserving, and maliciousness-preserving [11].

As mentioned above, there are several techniques to generate adversarial examples, including reinforcement learning, evolutionary algorithms, generative adversarial networks, and gradient-based techniques. The following sections of this chapter discuss these techniques in more detail.

1.2.1 Gradient-based Attacks

Machine learning algorithms can be classified according to the way they learn: supervised, unsupervised, semi-supervised, and reinforcement learning. In supervised machine learning, the model learns from a set of labelled training examples. Based on the input and expected output, the model creates a mapping equation that it can later use to label the unlabelled input. In unsupervised machine learning, the model learns only from unlabelled input. The model classifies the input data into classes with similar characteristics. The new input is later labelled based on the characteristics it shares with one of these classes. Semi-supervised learning combines elements of these two techniques. Training is done on data where some examples are labelled and others are not. Reinforcement learning is described later in a separate section.

Consider the case of supervised learning, where we have a set of labelled samples, that is, for each sample x we have a true label y. During the model training process, the goal is to find the ideal parameters w of the model by optimising the so-called loss function L. Specifically, we want to minimise this function to reduce the error of the model predictions on the training data:

$$\min_{w} L(w, x, y). \tag{1.3}$$

The training process consists of iterative adjustment of the classifier parameters in the direction of the gradient $\nabla_w L(w, x, y)$.

The concept of gradient-based adversarial attacks is based on the use of the same mechanisms. The goal is to confuse the classifier by maximising the probability of error. This can be achieved by manipulating the input data x to maximise the loss function L instead of adjusting the model parameters w:

$$\max_{x} L(w, x, y). \tag{1.4}$$

The loss function gradient allows one to calculate an adversarial perturbation δ. The perturbation is then added to the input sample x, creating an adversarial sample x' that can fool the model [14]. This can be expressed by the following equation [15]:

$$x' = x + \delta. \tag{1.5}$$

An example of a gradient-based algorithm for generating adverse samples is the Fast Gradient Sign Method (FGSM).

FGSM is a simple method to generate adversarial samples. It uses the gradient of the loss function L with respect to the input sample x to generate an adversarial sample x' that maximises the loss function. First, the loss function gradient is calculated as follows:

$$\nabla_x L(w, x, y) = \frac{\partial L(w, x, y)}{\partial x}. \tag{1.6}$$

The direction of the gradient is then obtained using the sign function, where the value $\text{sign}(\nabla_x L(w, x, y))$ can be 1 if the gradient value is positive, 0 if the gradient value is zero, or -1 if the gradient value is negative. The adversarial perturbation δ is then calculated as follows:

$$\delta = \text{sign}(\nabla_x L(w, x, y)) \times \varepsilon, \tag{1.7}$$

where ε is a small number that ensures that the perturbations are small. The resulting adversarial sample x' can be obtained as:

$$x' = x + \text{sign}(\nabla_x L(w, x, y)) \times \varepsilon. \tag{1.8}$$

and then used as a new input to the classification model f, which returns the classification result $f(x')$ [16].

1.2.2 Reinforcement Learning-based Attacks

Reinforcement Learning (RL) is one of the machine learning techniques, along with supervised and unsupervised learning. Unlike these two techniques, reinforcement learning does not depend on a static data set but learns based on its own experience.

The reinforcement learning model consists of two main parts: an agent and an environment. The agent learns to perform tasks through repeated interactions with the environment through trial and error. The following information and notation are based on [17].

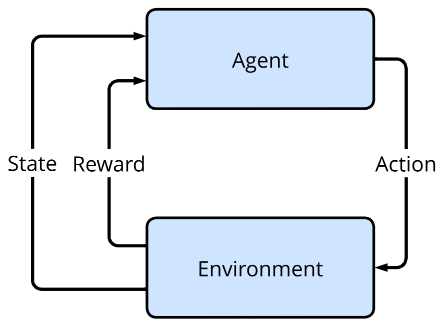

Figure 1.3: A basic structure of reinforcement learning.

Individual interactions occur in discrete time steps $t = 0, 1, 2, 3, \ldots T$. At time step t, the environment is in the state $S_t \in \mathcal{S}$, where $\mathcal{S}$ is the set of all possible states of the environment. The agent receives state S_t and then chooses action $A_t \in \mathcal{A}(S_t)$ based on policy π, where $\mathcal{A}(S_t)$ is the set of all available actions in state S_t. The policy π defines the behaviour of the agent at a given time. In other words, it is the strategy that the agent uses to determine the next action based on the current state. The policy maps environmental states to actions that must be taken in those states to achieve the highest reward.

After the environment receives information about the chosen action A_t from the agent, it calculates a reward $R_{t+1} \in \mathcal{R}$, where $\mathcal{R}$ is the set of all possible rewards and sends it back to the agent as feedback. At the same time, the environment transitions to a new state S_{t+1}. The reward signal $R_t \in \mathbb{R}$ is feedback from the environment to the agent, indicating the success or failure of the agent's action in that state.

When this cycle is complete, we say that the one-time step has elapsed. At each time step, the agent is in a certain state and sends the selected action as its output to the environment, which then returns a new state and a reward signal to the agent. This cycle repeats until the final state or the maximum time step $t = T$ is reached. The time that elapses between $t = 0$ and the end of the environment $t = T$ is called an episode. An episode can be written as the following sequence: $S_0, A_0, R_1, S_1, A_1, R_2, S_2, A_2, R_3, \dots R_T, S_T$.

The goal of the agent is to maximise the total amount of reward it receives. This means not maximising immediate reward, but the cumulative reward it receives in the long-term horizon. While the reward signal shows what is advantageous in the present, the value function describes what is advantageous in the long-term horizon. The value function provides an idea of the expected cumulative reward from the current state of the environment in the future. The goal of the agent is to maximise the total reward. The expected return G_t is defined as the sum of the rewards $G_t = R_{t+1} + R_{t+2} + R_{t+3} + \cdots + R_T$, where T is a finite time step.

1.2.3 Evolutionary Algorithm-based Attacks

Evolutionary Algorithms (EA) are optimisation methods that use the Darwinian principle of evolution. Essentially, a population evolves over multiple generations based on natural selection and the survival of the fittest individuals. These algorithms aim to mimic these natural processes. The goal is to find an optimal or satisfactory solution to a problem through competition between solutions that gradually evolve within the population [18]. Figure 1.4 shows the general principle of an evolutionary algorithm. The following information is based on [19].

There are different versions of evolutionary algorithms, such as evolutionary programming (EP), evolutionary strategy (ES), genetic programming (GP), and genetic algorithm (GA). These versions share the same concept of simulating evolution but differ in their application to a specific problem and in their implementation.

To begin using EA to solve a problem, the first step is to define the representation of candidate solutions within the original problem context. In this context, these candidate solutions are referred to as individuals or phenotypes. Since phenotypes can have complex structures, EA uses coding to represent

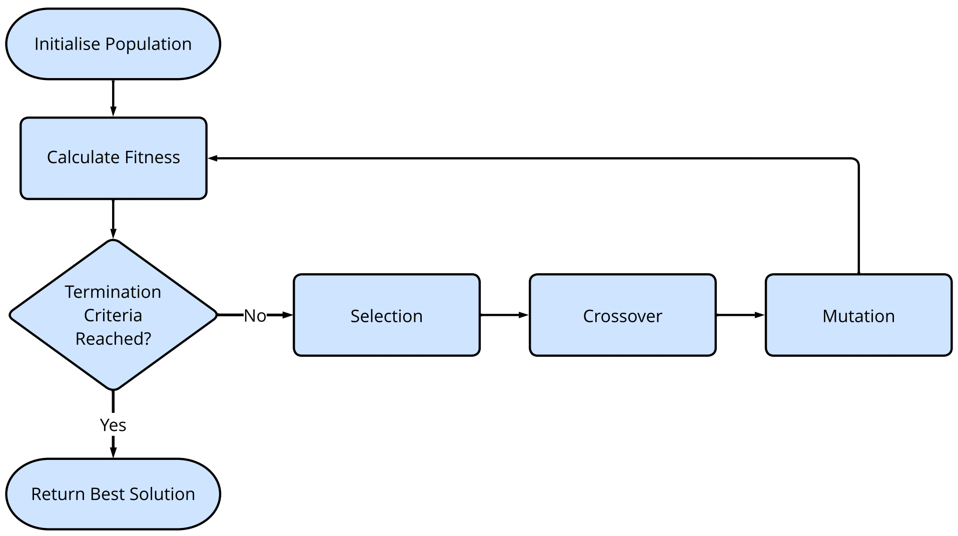

Figure 1.4: A basic structure of evolutionary algorithm.

the appropriate representation of individuals, which we call chromosomes or genotypes. This is a mapping of the set of phenotypes to the set of genotypes. Each chromosome consists of several genes. A gene is a unit of the chromosome and an allele is the value stored in the gene. The set of chromosomes forms a population and each chromosome in the population consists of the same number of genes.

After that, an initial population of individuals is created. Each individual in the population represents an encoded solution to the given problem. This population can be created by randomly selecting from the search space or, if prior knowledge of the problem exists, from several known good solutions.

Once the population is created, each member must be evaluated using a fitness function. This function takes into account the characteristics of the individual and numerically represents the quality of its solution. We then select the individuals with the highest scores and continue with them. The selection operator is based on Darwin's theory of natural selection and represents the principle of survival of the fittest. It identifies the strongest individuals in the population, who then participate in the production of offspring.

After selecting the best members (usually the top two, but this number can vary), these members are used to create the next generation. Using the properties of the selected parents, new offspring are created that are a mixture of the properties of the parents. Crossover operators take two (or more) parents and create offspring by exchanging information between them.

Next, we need to introduce new genetic material into the population. Without this crucial step, we would quickly get stuck in local optima and fail to achieve optimal results. This step is called a mutation, and it involves changing a small portion of the children so that they no longer perfectly reflect subsets of the parents' genes.

The process repeats until a stop condition is met. Typically, this occurs in one of two cases: either the algorithm has run for a maximum amount of time, or it has reached a certain performance threshold. The individual with the highest fitness at the stopping point is considered the output of the algorithm and represents the best solution found.

1.2.4 Generative Adversarial Network-based Attacks

A Generative Adversarial Network (GAN) is a model consisting of two neural networks: a generator G and a discriminator D. These two networks compete with each other, playing a minimax game for two players with a value function $V(G, D)$:

$$\min_G \max_D V(D, G) = \mathbb{E}_{x \sim p_{data}(x)}[\log\, D(x)] + \mathbb{E}_{z \sim p_z(z)}[\log\, (1 - D(G(z)))] \quad (1.9)$$

where x are real data, z is a random noise vector, $G(z)$ is generated data by the generator, $D(x)$ is the probability estimate that D recognises x as real data, $D(G(z))$ is the probability estimate that D recognises the generated sample data $G(z)$ as real data, $\mathbb{E}_x$ is the expected value over all instances of real data and $\mathbb{E}_z$ is the expected value over all instances of fake data generated. G tries to minimise this value function, while D tries to maximise it [20].

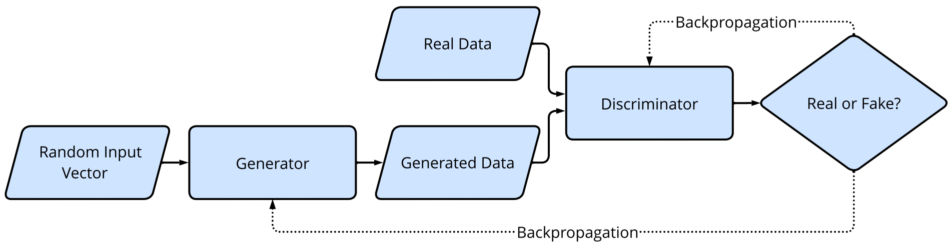

Figure 1.5: A basic structure of generative adversarial network.

The goal of the generator is to create data that are indistinguishable from the real data in the training set. This is done to fool the discriminator. The discriminator's goal is to distinguish fake data created by the generator from real data in the training set, thus preventing it from being fooled by the generator.

The generator constantly tries to improve its ability to generate real samples, while the discriminator constantly tries to improve its ability to distinguish between real and generated samples. This mutual competition forces both networks to continually improve. When $D(G(z)) = 0.5$, the discriminator can no longer distinguish between real and generated samples. This is the point where the model has reached its global optimal solution [21].

The generator and the discriminator have different training processes, so they are trained alternately. During the discriminator training phase, the generator weights do not change. Similarly, during the generator training phase, the discriminator weights do not change [22].

During generator training, the generator takes in a noise vector and uses it to create fake data. The generator outputs these data directly to the discriminator's input. The discriminator penalises the generator for generating unreliable data. Through backpropagation, the discriminator provides a classification signal that the generator uses to update its weights [23].

During discriminator training, the discriminator takes in both real and fake data instances. Then it classifies whether each instance is real or fake data generated by the generator. The discriminator is penalised for incorrectly classifying real instances as fake or fake instances as real. The discriminator then updates its weights using backpropagation [24].

1.3 Defense against Adversarial Machine Learning

Machine learning is widely used in various aspects of human life. Research and development of mechanisms to detect and defend against machine learning attacks are becoming increasingly important. These attacks can compromise the security and reliability of machine learning models, which can have disastrous consequences in areas such as autonomous vehicle driving, medical recognition systems, biometric systems, or malware detection systems. For example, article [25] demonstrates how an attacker can trick a computer vision system into misclassifying a STOP sign as a speed limit sign, simply by applying a few pieces of tape to the sign.

Techniques used to defend against adversarial examples are known as adversarial defences. These techniques are designed to ensure that the machine learning model can correctly classify even deliberately modified examples that cause misclassification. Methods for detecting adversarial examples are called adversarial detection. These methods can determine whether a sample has been altered to cause misclassification. The key property of a machine learning model that determines its ability to resist misclassification from adversarial examples is adversarial robustness [26].

One of the most well-known techniques for defending against adversarial examples is adversarial training. The basic idea of adversarial training is to generate adversarial examples and add them, along with their correct labels, to the training dataset. Training in original and adversarial examples, the model learns to be more robust against adversarial attacks [27].

Quiring et al. introduced a defence system against adversarial examples called PEberus [28], which won first place in the Microsoft Evasion Competition [29]. This system combines several different defensive measures, each targeting different attack strategies. The input PE file is submitted to Semantic Gap Detectors. Semantic gaps are various unused spaces that do not affect the functionality of the program, but affect the machine learning-based system. These detectors check if the PE file has been attacked by an adversarial attack, specifically checking if slack spaces are filled, if overlaps are too large compared to the file size, and if the same sections are used. If the PE file is not detected by the semantic gap detectors, the file is passed to the existing malware detectors, the Skipgram model, and the Signature-based model. Maximum voting is used for evaluation. Then, the stateful nearest neighbor detector is used, which continuously checks if the PE file is similar to any previously detected malware in the history cache. The PE file is classified as malware if any of the system components consider it harmful. Therefore, the attacker must exploit the weaknesses of all components to successfully deceive the system.

Chapter 2

Portable Executable File Format

Malware can be spread using various file formats. According to AV-TEST statistics, the majority of malware targets the Windows operating system [2], and the most common file format used to spread malware on the Windows operating system is the executable file [30].

The PE file format is a binary file format used in Microsoft Windows operating systems. This format is based on the Common Object File Format (COFF), a file format used in Unix operating systems. PE files include executable files (EXE) and dynamically linked libraries (DLL) [31].

The PE file format has a fixed structure. It can be metaphorically divided into two parts: headers and sections. It starts with a DOS Header and a DOS Stub program. This is followed by the NT Headers, which contain the PE signature, the File Header and the Optional Header. All the section headers and section data are at the end of the PE file. The general structure of the PE file is shown in Figure 2.1.

This chapter describes the PE file format with a focus on the executable files. The parts of the file are described in the order in which they appear when the file is traversed. Most of the information in this chapter is taken from the official Microsoft Windows documentation [32].

2.1 DOS Header

At the beginning of each PE file, there is a 64-byte long DOS Header. This structure is present for backward compatibility, as it allows the file to be executed in MS-DOS system by running its DOS Stub instead of the actual program. The DOS Header is important for the PE loader in MS-DOS system, but for the PE loader in Windows NT systems, only two pieces of information contained in it are relevant. These are the `e_magic` field and the `e_lfanew` field.

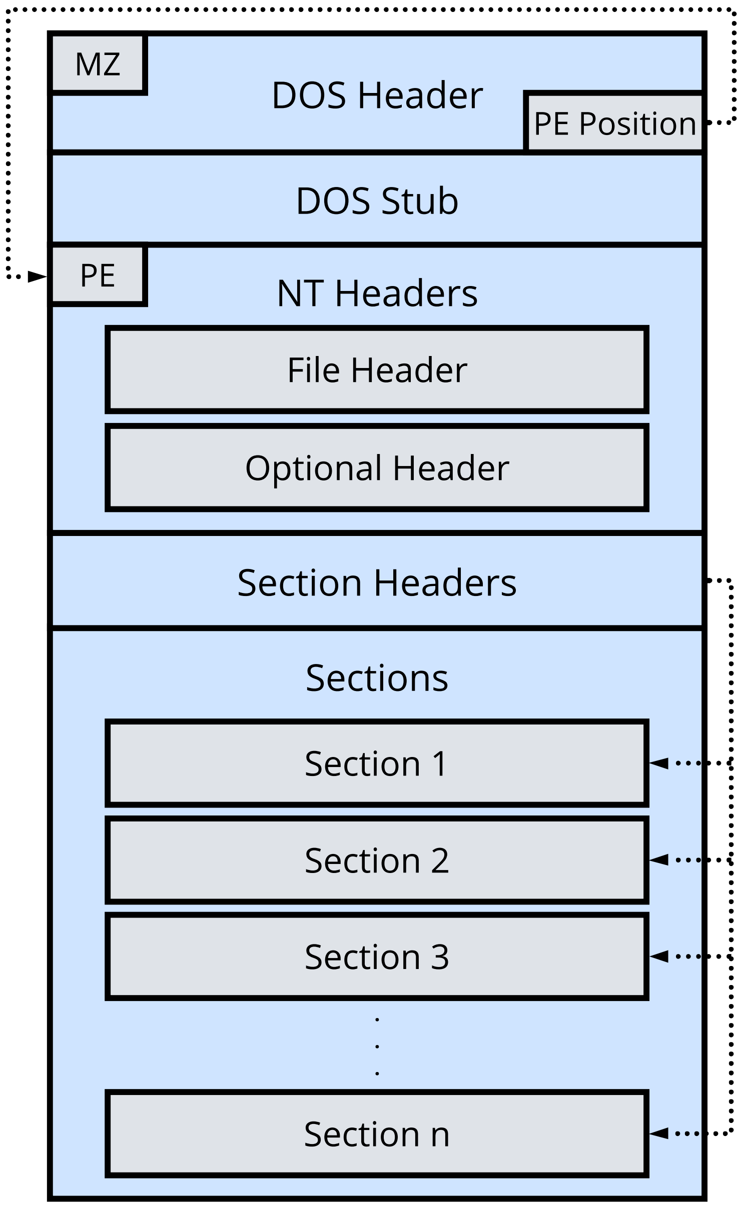

Figure 2.1: Portable Executable File Format.

- `e_magic`: The first 2 bytes of the DOS Header represent the so-called magic number. This entry has a fixed value of `0x5A4D`, or MZ in American Standard Code for Information Interchange (ASCII), which represents the initials of Mark Zbikowski, one of the MS-DOS developers [33].
- `e_lfanew`: The last 4 bytes of the DOS Header, located at offset `0x3C`, contain the offset to the beginning of the NT Headers. This field is important for the PE loader in Windows NT systems because it tells the loader where to locate the PE Header in the file.

2.2 DOS Stub

After the DOS Header, there is a DOS Stub program. This program is designed for use with the MS-DOS system. If a Windows NT executable file is attempted to run on the MS-DOS system, an error message is displayed and the program is terminated. The error message that appears by default says "This program cannot be run in DOS mode". However, if the file is run on the Windows NT system, the DOS Stub program is skipped, and the NT Headers are read instead to obtain information on how to load and execute the executable file.

2.3 Rich Header

Between the DOS Stub program and NT Headers in a PE executable file lies a section of data known as the Rich Header. Although not officially part of the PE file format, Rich Header can be found in PE executable files created with the Microsoft Visual Studio toolset. This undocumented structure contains important information about the tools used to create the executable file, such as the names, types, specific versions, and build numbers of these tools. However, the content of the Rich Header can be completely zeroed out without affecting the functionality of the executable file [34].

2.4 NT Headers

The NT Headers structure is a data structure that starts at an offset defined in the `e_lfanew` field in the DOS Header. There are two versions of this structure: one for 32-bit executable files and one for 64-bit executable files. The main difference between these two versions is the version of the Optional Header structure used. The NT Headers structure consists of three main parts:

- `Signature`: The first 4 bytes of the NT Headers contain a PE signature that identifies the file as a PE format file. This field has a fixed value of `0x50450000` or `PE\0\0` in ASCII (where `\0\0` represents two null bytes).

- **File Header**: This structure contains basic information about the PE file.
- **Optional Header**: This structure provides information to the PE loader for loading and executing the executable file.

2.5 File Header

The File Header, also known as the COFF File Header, is a 20-byte structure with seven fields that contain essential information about the PE file. These fields include:

- `Machine`: Indicates the CPU type on which the file can be executed. For example, `0x14C` for Intel 386 or `0x8664` for AMD64.
- `NumberOfSections`: The number of sections recorded in the Section Table.
- `TimeDateStamp`: Unix timestamp indicating when the file was created.
- `PointerToSymbolTable`: Symbol Table offset. Typically set to 0, meaning that there is no COFF symbol table present.
- `NumberOfSymbols`: The number of records in the symbol table. Typically set to 0, meaning there is no COFF symbol table present.
- `SizeOfOptionalHeader`: The size of the Optional Header. This field is set to a non-zero value for executable files and to zero for object files.
- `Characteristics`: Logical sum of flags that indicate certain attributes of the file. An attribute could indicate that the file is executable, a dynamic link library, or a system file. It could also indicate that debugging information has been removed from the file.

2.6 Optional Header

The Optional Header provides information to the PE loader to load and execute an executable file. It is called an Optional Header because certain types of file, such as object files, do not have it. However, for image files (such as EXE files), this header is mandatory. The size of the Optional Header is not fixed, but it can be found in the `SizeOfOptionalHeader` field located in the File Header. There are two different versions of the Optional Header, depending on whether the file is 32-bit or 64-bit. The header is composed of three main parts: Standard fields, Windows-specific fields, and Data directories.

2.6.1 Standard Fields

The first 8 or 9 (depending on the value of the `Magic` field) items in the Optional Header structure are standard fields defined for every implementation of the COFF file format. These fields contain general information useful for loading and running the executable file.

- `Magic`: The value of this field determines whether the executable file is 32-bit (also known as the PE32 executable) or 64-bit (also known as the PE32 + executable). The value `0x10B` identifies the PE32 format and the value `0x20B` identifies the PE32+ format.

- `MajorLinkerVersion`: The major version number of the linker used to create the executable file.

- `MinorLinkerVersion`: The minor version number of the linker used to create the executable file.

- `SizeOfCode`: The size of the code section (.text). If there are multiple code sections, this value represents the sum of the sizes of all such sections.

- `SizeOfInitializedData`: The size of the initialised data section (.data). If there are multiple initialised data sections, this value represents the sum of the sizes of all such sections.

- `SizeOfUninitializedData`: The size of the uninitialised data section (.bss). If there are multiple uninitialized data sections, this value represents the sum of the sizes of all such sections.

- `AddressOfEntryPoint`: The relative virtual address (RVA) of the entry point when the executable file is loaded into memory. For executable files, this is the initial address where the program begins execution.

- `BaseOfCode`: The RVA of the beginning of the code section when the executable file is loaded into memory.

In the PE32 executable file format, there is an extra field called BaseOfData that is not present in the PE32+ executable file format.

- `BaseOfData`: The RVA for the beginning of the data section when the executable file is loaded into memory.

2.6.2 Windows-specific Fields

The following 21 fields of the Optional Header are marked as Windows-specific fields. They contain additional useful information required by the Windows PE loader and linker.

- `ImageBase`: Preferred address of the first byte of the file when loaded into memory. This value must be a multiple of 64 kB. The default value for executable files is `0x00400000`. However, due to memory protection techniques such as Address Space Layout Randomization (ASLR), the address specified by this field is rarely used.

- `SectionAlignment`: Determines the alignment of sections loaded into memory. The value of this field must be greater than or equal to the value of the FileAlignment field. The default value is the page size for the system.

- `FileAlignment` Determines the alignment of sections in the file. The value should be a power of 2 between 512 and 64 kB (inclusive). The default value is 512. If the `SectionAlignment` member is less than the system page size, this member must be the same as `SectionAlignment`.

- `MajorOperatingSystemVersion`: Major version number of the operating system required to run the executable file.

- `MinorOperatingSystemVersion`: Minor version number of the operating system required to run the executable file.

- `MajorImageVersion`: Major version number of the executable file.

- `MinorImageVersion`: Minor version number of the executable file.

- `MajorSubsystemVersion`: Major version number of the operating system subsystem required to run the executable file.

- `MinorSubsystemVersion`: Minor version number of the operating system subsystem required to run the executable file.

- `Win32VersionValue`: This field is reserved and must be set to 0.

- `SizeOfImage`: Specifies the total size of the executable file in memory. The value of this field must be a multiple of the `SectionAlignment` value.

- `SizeOfHeaders`: Specifies the size of the file headers, including the DOS Stub, NT Headers, and Section Headers. The value of this field is rounded up to the nearest multiple of the `FileAlignment` field.

- `CheckSum`: Contains the checksum value for the entire file.

- `Subsystem`: Specifies the Windows subsystem required to run the file, such as Windows Graphic User Interface (GUI) or Windows Console.

- `DllCharacteristics`: This field contains flags indicating some characteristics of the executable file. For example, whether it can be moved during runtime, whether it is compatible with Data Execution Prevention (DEP), or whether it does not use Structured Exception Handling (SEH).

- `SizeOfStackReserve`:Specifies the number of bytes to reserve for the stack. On loading, only the memory specified by the `SizeOfStackCommit` field is committed; the rest is available in one page increments until this reserve size is reached.

- `SizeOfStackCommit`: Specifies the number of bytes to commit for the stack.

- `SizeOfHeapReserve`: Specifies the number of bytes to reserve for the heap. On loading, only the memory specified by the `SizeOfHeapCommit` field is committed, the rest is available in one page increments until this reserve size is reached.

- `SizeOfHeapCommit`: Specifies the number of bytes to commit for the heap.

- `LoaderFlags`: This member is obsolete and should not be used.

- `NumberOfRvaAndSizes`: The value of this field specifies the number of items in the Data Directories.

2.6.3 Data Directories

At the end of the Optional Header, there is a field called Directory Entries. Each directory entry is an 8-byte structure containing two 4-byte fields:

- `VirtualAddress`: The RVA at which the data directory starts.

- `Size`: The size of the data directory.

Data directories contain useful information that the loader needs and can have different structures. The structure of a directory is determined by its type. Some examples of data directory types include the Export Table, Import Table, and Import Address Table.

2.7 Section Headers

After the Optional Header and Data Directories, the Section Headers, also known as the Section Table, follow. Each record in the Section Table represents a section header that contains information about the section to which it refers. The number of records in the Section Table is indicated in the `NumberOfSections` field, which appears in the File Header. Each section header is 40 bytes long and contains the following fields:

- `Name`: 8-byte long section name. If the name is shorter than 8 bytes, the remaining bytes are filled with null values. For example, `.text\0\0\0` or `.data\0\0\0`.
- `VirtualSize`: Total size of the section when loaded into memory. If this value is greater than `SizeOfRawData`, the section will be padded with zeros.
- `VirtualAddress`: RVA of the first byte of the section when loaded into memory.
- `SizeOfRawData`: Size of the initialised data on the disk. It must be a multiple of the `FileAlignment` member of the Optional Header. If the section contains only uninitialised data, this field should be zero. If this is less than `VirtualSize`, the rest of the section is filled with zero.
- `PointerToRawData`: Pointer to the first byte of the section. The value must be a multiple of the `FileAlignment` member of the Optional Header structure. If the section contains only uninitialized data, this field is set to zero.
- `PointerToRelocations`: Pointer to the beginning of the relocation entries for the section in the file. If there are no relocations, this value is zero.
- `PointerToLineNumbers`: Pointer to the beginning of COFF line number entries for the section in the file. If there are no COFF line numbers, this value is zero.
- `NumberOfRelocations`: Number of relocation entries for the section. This value is zero for executable images.
- `NumberOfLinenumbers`: Number of COFF line number entries for the section.
- `Characteristics`:This field includes the combined flags that indicate the attributes of the section. This includes whether the section contains executable code, initialised data, or uninitialised data, or whether it can be executed as code, read from, or written to.

The values of fields `SizeOfRawData` and `VirtualSize` may differ. The value of `SizeOfRawData` must be a multiple of the value of `FileAlignment` specified in the Optional Header. If the section size is smaller than this value, the rest of the section is filled with zeros, and the field value `SizeOfRawData` is rounded up to the nearest multiple of the value `FileAlignment`. When the section is loaded into memory, this alignment is not taken into account, and only the actual size of the section is occupied. In this case, the value of `SizeOfRawData` will be greater than the value of `VirtualSize`. However, if the section contains uninitialized data, these data are not reflected on the disk. But when the section is mapped into memory, it expands to reserve memory space for later initialisation and use of uninitialized data. This means that the section on disk will take up less space than in memory. In this case, the value of `VirtualSize` will be greater than the value of `SizeOfRawData`.

2.8 Sections

Section data immediately follow Section Headers and occupy the rest of the file. Each section contains things like the actual code of the program, as well as data and resources that the program uses. Data for each section are located at an offset given by the `PointerToRawData` field in the Section Header. The size of these data is determined by the field `SizeOfRawData`. There are several special sections, each with its own purpose, such as:

- `.text`: Contains the executable code of the program.
- `.data`: Contains initialised (non-zero) data.
- `.rdata`: Contains initialised data for read-only purposes only.
- `.bss`: Contains uninitialized (zeroed) data.
- `.idata`: Contains import tables.
- `.edata`: Contains export tables.
- `.debug`: Contains compiler information for debugging.
- `.rsrc`: Contains resources used by the program, including images, icons, or even embedded binary files.
- `.reloc`: Contains relocation information.
- `.tls`: Provides storage for each thread of the program being executed.

2.9 Windows PE Format Manipulations

Manipulation of PE files to create an adversarial malware sample that remains functional can be a complex task. A single byte can be changed inappropriately and the file can be corrupted.

A PE file is made up of many parts, which can be divided into two groups: modifiable and immutable. The modifiable parts of the file can be changed without changing the functionality of the file. For this reason, these parts can be used to create adversarial examples. On the other hand, modifying the immutable parts of a file can either break the functionality of the file or change its behaviour. These parts are important for the correct functioning of the file and must therefore remain unchanged.

Adversarial techniques exploit the redundancy and technical characteristics of the file format to modify files. An attacker looks for suitable places to modify bytes without breaking the structure or creating space for injecting a malicious payload [35].

This section presents some of these manipulations. The manipulations described in this section are mainly based on the work of Demetrio et al. [35, 36, 38].

Manipulating DOS Header and Stub

Partial DOS and Full DOS manipulations involve modifying bytes in the DOS Header and Stub program. These sections can be modified without damaging the file structure because they are only preserved in the PE file for backward compatibility with older Microsoft operating systems, as described in Section 2.1 and Section 2.2 . The magic number MZ, located in the first two bytes of the file, and the 4-byte integer at offset `0x3C`, which contains the address of the PE signature, must not be modified. An attacker can use all other bytes as space to store the adversarial payload. Partial DOS modifies 58 bytes between the magic number and the PE signature offset, in the range from `0x02` to `0x3C`. Full DOS extends this range by adding all bytes in the range from `0x40` to the PE signature location. The number of these bytes varies in different executable files.

Section Injection

The section injection technique allows an attacker to insert new sections into a binary file, providing additional space to store an adversarial payload. The attacker creates a new section which is inserted into the target executable along with a new section entry in the section table.

Padding

Padding consists of adding bytes to the end of the file. Since all the adversarial content is added to the end of the file, the structure of the original file is not changed.

Perturb Header Fields

Manipulation Header Fields modify specific fields within NT Headers and Section Headers. This includes, for example, changing section names, breaking a checksum, or changing debugging information.

Filling Slack Space

As described in Section 2.7, the values of the SizeOfRawData and VirtualSize fields may differ to maintain alignment within the file. This creates a space between sections, called a slack space. The bytes in this space can be freely modified because the executable code never references these bytes, so their existence is ignored. Slack space manipulation takes advantage of these spaces and inserts malicious adversarial payloads into them.

Extend the DOS Header

The Extend manipulation allows the creation of a new space inside the executable, specifically by increasing the size of the DOS Header, where an adversary can then insert an adversarial payload. To keep the file structure uncorrupted, the adversary must: choose a size that does not interfere with the file's alignment, increase the field containing the header size, and increase the offset of each section entry. After this modification, the loader skips the adversarial payload when looking for the PE signature, without changing the rest of the execution flow.

Content Shifting

Shift manipulation allows a new space to be created within the executable by shifting the contents of a section to make space for an adversarial payload. Each section entry in the section table specifies an offset within the binary where the loader can find the contents of that section. Each of these offsets is a multiple of the file alignment specified in the Optional Header. The loader reads these entries and loads the contents of each section into memory by looking inside the binary at the offset specified by that entry. An attacker can increase the offset of each section to create a space before the start of each section and fill the newly created spaces with chunks of bytes. In this way, the loader will look for the contents of each section and ignore any unfavourable distortions introduced between them.

API Injection

The goal of the Application Programming Interface (API) Injection technique is to add records to the Import Address Table of the executable. This table specifies which function from which library must be included during the loading process. An attacker cannot remove the API because this would break the functionality of the program, but they can inject new API imports. Adding entries to this table will increase the number of APIs that the operating system will include during the loading process, but that the program will not use.

Binary Rewriting

Binary rewriting techniques allow various ways of modifying the code of a program. These include manipulations such as replacing a set of instructions with others that are semantically equivalent (e.g. replacing addition with subtraction and changing the sign of the values); adding dead code that will never be executed; encrypting or encoding the contents of one binary file within another binary file and decoding it at runtime; setting an entry point in a new executable part that jumps back to the original code; saving the code of one file as a resource of another binary file that is then loaded at runtime.

Chapter 3

Related Work

This chapter provides an overview of publications that focus on creating attacks against the detection of malicious PE software. The chapter is divided into several sections based on the approach used to create adversarial examples.

3.1 Gradient-based Attacks

Kolosnjaji et al. in [39] presented a gradient-based white-box attack against the MalConv network. The basic idea of the attack is to systematically manipulate some bytes in the malware file to maximise the probability that the modified sample will be classified as harmless. The attack allows manipulation of any byte in the file, but to ensure that the malware file remains functional, the authors only use manipulation of padding bytes appended to the end of the file in their work. The appended bytes are generated using the gradient descent method, where each byte is optimised one byte at a time. The authors conducted an experiment in which bytes were added at the end of each file using two different strategies. The first strategy was a random attack, where random byte values were inserted. The second strategy was a gradient-based attack. Random bytes were shown to be ineffective in bypassing the network. However, the gradient-based attack outperforms random byte injection; the authors were able to achieve a 60% evasion rate against the MalConv network using the gradient-based attack simply by modifying 1% of the bytes in the PE file.

Kreuk et al. proposed an improved gradient-based white-box attack against MalConv in [40]. Unlike [39], where bytes are only inserted at the end of the file, Kreuk et al. used two methods to insert a sequence of bytes, the adversarial payload, into the original malware file. The first method is based on inserting the adversarial payload into a slack region, which is the region of unused bytes of file sections whose physical size is larger than the virtual size. The second method is based on adding the adversarial payload to the end of the file, where the adversarial payload is considered a new section and appended to the file. When generating the adversarial version of the malware sample, the adversarial payload is first appended or inserted, which is randomly and uniformly initialised. The inserted adversarial payload is then perturbed using iterative FGSM until the file is misclassified.

Suciu et al. describe and compare several attack strategies in [41]. These fall into two categories: append attacks and slack attacks. The authors propose improved attacks based on FGSM, called append-FGSM and slack-FGSM attacks, and compare their effectiveness against MalConv. In the append-FGSM attack, random bytes are added to the malware file and optimised using a one-step FGSM. To address the non-differentiability problem, the added bytes are updated in the embedding space using the gradient and then mapped to the nearest byte value. The slack-FGSM attack identifies slack regions. These are areas that are not mapped into memory and can be used to inject adversarial noise. The appropriate bytes are modified using an approach similar to the append-FGSM attack. Experimental results show that the slack-FGSM attack is more effective with fewer modified bytes than the append-FGSM attack.

Demetrio et al. [42] used an integrated gradient technique to identify the main characteristics by which MalConv distinguishes malware from benign files. The results showed that MalConv relies primarily on the characteristics of the PE file header to differentiate between malware and benign samples. As a result, MalConv mostly ignores the data and text sections where the malicious content is often hidden. Based on this observation, Demetrio et al. introduced the Partial DOS attack, which is similar to the attack in [39]. However, the main difference is that [39] inserts adversarial bytes at the end of the PE file, while this attack modifies the bytes in the DOS header. This approach is more effective because it requires the manipulation of much fewer bytes to bypass the detector. In total, 58 bytes located between the `e_magic` field and the value of the `e_lfanew` field at offset `0x3C` were marked as editable. These two values were not included because they should not be modified to maintain file functionality. The results showed that only a few perturbed bytes are sufficient to bypass MalConv with high probability.

Demetrio et al. introduced the RAMEN framework [35], which provides a unified approach to white-box and black-box adversarial attacks against PE malware detectors. In addition to generalising and describing previous attacks against machine learning models, this framework includes three new attacks based on modifications to the PE file format that preserves its functionality. The first attack, Full DOS, extends the previous Partial DOS attack [42]. Full DOS allows modification of all available bytes in the entire DOS header up to the PE signature. However, it still has limitations and cannot modify the magic number MZ and the four-byte value at offset `0x3C` that contains the PE signature offset. The second attack, Extend, allows the DOS header to be expanded, creating new space for inserting malicious bytes without disrupting the structure of the executable. The third attack, Shift, allows the contents of the first section to be shifted to create a new space for inserting an adversarial payload.

3.2 Reinforcement Learning-based Attacks

Anderson et al. introduced the Gym-malware framework [43], which is based on a reinforcement learning approach. It is a black-box attack where the attacker does not know the target model, except for the ability to obtain labels for input samples whether they are malicious or benign files. The environment consists of a malware sample and the target of the attack, which is an anti-malware engine, specifically a gradient-boosted decision tree (GBDT) model. The agent is the algorithm used to modify the environment and has a set of operations that can be applied to the PE file. The default agent is Actor-Critic with Experience Replay (ACER). Through a series of games against the anti-malware engine, the agent learns which sequence of operations is likely to evade the detector for a given malware sample.

Song et al. proposed the MAB-Malware framework [44], which approaches the problem of adversarial attack as a multi-armed bandit problem, to find a balance between exploiting successful patterns and exploring new variants. The authors propose a stateless modelling approach that can significantly reduce the learning effort and lead to more productive AE generation. Many actions to transform PE malware are independent and, according to the results, often only one or two actions are needed to generate an AE. They also suggested that modelling actions and content as integral units for manipulating PE files. If the content associated with an action proved useful in one AE, the same action-content pair is likely to be useful for some other samples. The authors also found that most actions are redundant and assigning rewards to these actions complicates the learning process. Therefore, they proposed an action minimisation process that minimises AEs by removing redundant actions and further reducing significant actions into even smaller actions, called micro-actions. Rewards are then assigned only to these essential micro-actions. The results showed that MAB-Malware achieved an evasion rate of 74.4% against GBDT, 97.7% against MalConv, and up to 48.3% against commercial antiviruses. Furthermore, the authors demonstrated that the transferability of adversarial attacks between GBDT and MalConv detectors is greater than 80%, while the transferability of attacks between these detectors and commercial AV is only 7%.

3.3 Evolutionary Algorithm-based Attacks

Castro et al. introduced the AIMED [45] system, which uses genetic programming algorithms to generate adversarial examples. Nine types of file modification operations are defined. Modifications are first applied to the original malware to create a population. Then, each modified example in the population is evaluated using a fitness function, which includes four phases: functionality (checking the file for corruption), detection (scanning the file using commercial engines or research models to determine whether it is considered malicious), similarity (calculating the difference between the malware mutation and the original file it came from), and generation (using the current generation number to prefer the newest members of the population). Each of these phases contributes a value to the fitness function. The following steps are selection, crossover, and mutation. This process is repeated until the generated malware can evade the malware classifier.

Wang and Miikkulainen proposed the MDEA model [46] to detect malware. MDEA consists of a detection model based on the MalConv network and an evolutionary optimisation algorithm. To create adversarial malware samples, an action space is defined that contains 10 different methods to modify binary programs. The evolutionary algorithm generates different sequences of actions by selecting actions from the action space until the generated adversarial malware can evade the target malware detector. Once the action sequences are found successfully, they are applied to the corresponding malware samples. All newly generated malware samples are added to the training set, and the detection model is retrained. This process is a form of adversarial training. The results demonstrate that retraining the model with evolutionary malware samples significantly improves its performance.

Demetrio et al. introduced GAMMA, a black-box attack framework that includes two types of attacks [36]. The attacks are based on injecting harmless content, extracted from goodware, into malicious files. This involves either inserting harmless content at the end of a file (padding attack) or into newly created sections (section-injection attack). To make the attack difficult to detect, it is formalised as a constrained minimisation problem that optimises the trade-off between the probability of evading detection and the size of the injected payload. The problem is solved using a genetic algorithm.

3.4 Generative Adversarial Network-based Attacks

Hu and Tan introduced a model called MalGAN in their article [47], based on generative adversarial networks. The authors consider adding only some irrelevant API calls to the original malware sample to generate adversarial malware samples in the feature space. For each program, a 160-sized binary feature vector is created based on 160 system APIs. MalGAN includes two feedforward neural networks: a generator and a substitute detector. The input to the generator is the malware feature vector and a noise vector. The generator is used to transform the vector of malware features into its adversarial version. A black-box detector is also used, which receives adversarial examples created by the generator and benign examples from the training set. The black-box detector evaluates the input examples and determines whether the sample is benign or malicious. The resulting decision from the black-box detector, along with the adversarial examples from the generator and the benign examples, is fed into the substitute detector. The substitute detector is used to mimic the black-box detector by learning the classification rules of malware and goodware classified by the black-box detector. Adversarial examples are dynamically generated according to the feedback from the black-box detector. The generator and the substitute detector work together to attack the black-box detector.

Later, Kawai et al. presented an improved MalGAN [48]. In their paper, the authors discuss the problems of MalGAN and try to improve them. The issues include importing the malware detector into MalGAN, reducing the number of features used in MalGAN to 128, using the same API list for both MalGAN and the detector, and using multiple malware to learn in MalGAN. The improvements include executing malware detectors externally, adding all the APIs used for the malware to the feature quantities, creating MalGAN and detector API lists from different training data, and using only one malware for MalGAN.

3.5 Combination of Techniques

Vaya and Sen created the Pesidious [49] tool to generate adversarial PE malware samples. Pesidious is based on a combination of two techniques, namely generative adversarial networks and reinforcement learning, and is implemented using MalGAN [47] and Gym-malware [43]. In this case, the generative adversarial network is used to generate benign looking imports and sections, and reinforcement learning is used to train the agent to select the best mutation sequence for the malware sample to evade the malware classifier. The default malware classifier is the gradient boosting classifier, and the following mutations are available for generating adversarial samples: add imports received from GAN, add sections received from GAN, append bytes to sections, rename sections, UPX pack, UPX unpack, add/remove the signature, and append a random number of bytes.

Kozák and Jureček [50] proposed an adversarial attack strategy that combines existing malware adversarial example generators to increase their potential and create more sophisticated adversarial examples that are more likely to evade malware classifiers. Unlike [49], this method combines two separate AE generators. The authors tested combinations of five different AE generators: MAB-Malware [44], AMG (PPO and random agent) [51], FGSM [40] and Partial DOS [42]. First, the authors created samples using individual generators and tested them against nine selected antivirus programs on VirusTotal. They then used the unsuccessful samples from the first generator as input to the second generator, creating new samples that were again tested against selected antivirus products. The results showed that combining different generators can significantly improve their effectiveness against antivirus programs. The most effective combination of generators was AMG-random and MAB-malware, which achieved an average detection rate of 15.9% against anti-virus products. This is an average improvement of over 36% and 627% respectively compared to using AMG-random and MAB-malware generators alone.

Chapter 4

Selected Methods

This chapter describes the tools and techniques used to create adversarial malware samples in this thesis. A description of how to install, run, and use the libraries and scripts is provided in a separate document included in the files accompanying this thesis.

4.1 Malware Classifiers

In this section, we describe two popular machine learning-based malware detectors that we used as target detectors in the creation of adversarial malware samples. Both models were trained in the EMBER dataset [52].

4.1.1 MalConv

MalConv is a convolutional neural network (CNN) developed by Raff et al. [53], whose architecture is shown in Figure 4.1. MalConv distinguishes between malicious and benign files purely based on their byte representation, without extracting any features. First, the size of the input file is limited to 2 MB. If the file size is less than 2 MB, the file is padded with a value of 256 (the padding value does not correspond to a valid byte to properly represent the absence of information); otherwise, the file is truncated and only the first 2 MB are analysed. For our experiments, we used the MalConv model provided by Anderson et al. [52], which differs from the original model by setting the maximum input file size at 1 MB. The first layer of the network is the embedding layer, which is defined by the function $\phi : \{0, 1, \ldots, 256\} \rightarrow \mathbb{R}^{d \times 8}$, which maps each input byte to an eight-dimensional vector in the embedding space. The goal of this step is to learn to represent bytes that have semantically similar behaviour as closer points in this space, thus providing a meaningful measure of the distance between bytes. The embedding layer is learnt during training and then used as a look-up. Create an eight-dimensional embedding for each byte, which is then fed into two convolutional layers. Each of the convolutional layers iterates over nonoverlapping windows of 500 bytes with a total of 128 convolutional filters. The output of the two convolutional layers is then multiplied, and the result is fed into the max-pooling layer. The result is a set of the 128 most activated features of all convolution windows. The results of these operations are passed as input to the fully connected layer for classification. The last layer produces a prediction value f(x) between 0 and 1. It is given by the softmax function applied to the results of the dense layer and classifies the input x as malicious if $f(x) \geq 0.5$, otherwise it is benign [42].

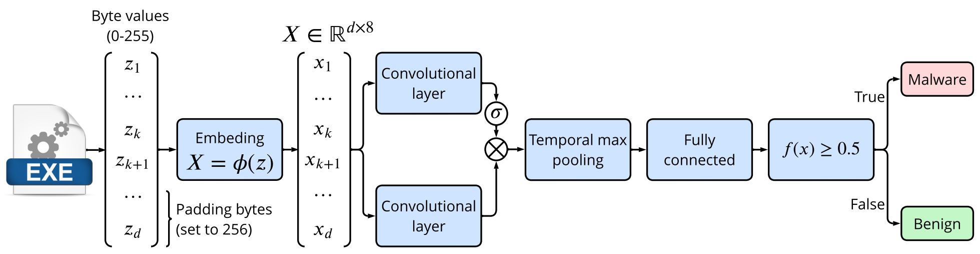

Figure 4.1: The architecture of MalConv [37].

4.1.2 Gradient Boosting Decision Tree

Anderson et al. trained a Gradient Boosting Decision Tree (GBDT) model [52, 54] on their EMBER dataset. GBDT is a model consisting of a set of sequentially trained decision trees. Unlike MalConv, GBDT uses a set of 2,381 manually created features derived from static analysis of binary files using the Library to Instrument Executable Formats (LIEF). These features come from, for example, the following sources [36]: general file information (such as virtual file size, number of imported and exported functions, presence of debug sections), header information (executable file properties, target architecture, version), byte histogram (number of occurrences of each byte divided by the total number of bytes), byte entropy histogram, information extracted from strings (number of occurrences of each string and how many special characters they contain, such as \, HKEY, http and https), section information (name, length, entropy, and virtual size of each section), imported and exported functions.

4.2 Generators of Adversarial Malware Examples

An adversarial malware example refers to a type of malicious software that has been intentionally modified to avoid detection. This is achieved by strategically modifying, inserting, or removing certain bytes from the original malware file while preserving its original characteristics. The goal is to maintain file format, executability, and maliciousness while also ensuring that the resulting adversarial malware is incorrectly classified as harmless by the target malware detection model, such as an antivirus tool or machine learning-based malware classifier.

To create adversarial malware examples, we used three different techniques: gradient-based techniques, evolutionary algorithm-based techniques, and reinforcement learning-based techniques. Specifically, from gradient-based techniques, we chose Partial DOS and Full DOS generators, which are part of the secml-malware Python library [55]. From evolutionary algorithm-based techniques, we selected GAMMA padding and GAMMA section-injection generators, which are also part of the secml-malware library. From reinforcement learning-based techniques, we chose the Gym-malware generator, which is implemented in the Python Gym-malware library [43]. Both libraries use LIEF [56] to analyse and manipulate PE files. This section describes each of these attacks in detail.

4.2.1 Partial DOS and Full DOS

The Partial DOS and Full DOS generators focus on changing the bytes in the DOS header of the PE file. As mentioned in Chapter 2, the DOS header is included in the file for backward compatibility with MS-DOS, but for Windows, it contains only two important pieces of information, namely the first 2 bytes, which represent the magic number MZ, and the last 4 bytes at offset `0x3C`, which indicate the location of the PE signature in the NT Header. All other bytes can be used to inject a malicious payload. Specifically, when the program is executed, control is passed to the loader, which begins to parse the executable. After checking the magic number, it reads the PE offset and proceeds to the PE header metadata, skipping the DOS header parsing and stub, thus preserving the functionality of the input program. Thus, in addition to the two limitations mentioned above, the entire DOS header can be manipulated without damaging the program.

The Partial DOS attack modifies only 58 bytes, specifically bytes in the range `0x02` to `0x3B` inclusive, ignoring the first two bytes containing the magic number MZ and the four-byte offset between `0x3C` and `0x3F`.

The Full DOS attack extends the Partial DOS attack by expanding the range of modified bytes to include all but the two aforementioned restrictions up to the PE signature. The location of this signature can vary from file to file, but is located at offset `0x3C`. This amount can vary from sample to sample: in our test dataset it ranges from 106 to 506 bytes.

The Partial DOS and Full DOS are gradient-based attacks. This means that the attacker has white-box type access to the target model to calculate the gradient of the loss function. In these attacks, an initial perturbation is applied to the input malware file, resulting in a modified file. The manipulated file is then encoded in feature space. The next step is to modify the representation based on the features of the manipulated input sample to minimise the loss function. The solution is obtained by iterative updating the feature-based representation using the loss function gradient. The algorithm stops either when evasion is achieved or when the maximum allowed number of iterations is reached. The feature-based representation obtained after this step does not necessarily correspond to any input sample. For this reason, it is necessary to use appropriate reconstruction strategies that ensure that the reconstructed sample is a valid program and preserves the intended functionality of the original malware. File reconstruction is then based on selecting the closest byte in the embedded space.

4.2.2 GAMMA padding and GAMMA section-injection

GAMMA padding generator and GAMMA section-injection generator are based on inserting parts extracted from harmless files into files containing malicious code. These are black-box attacks in which only the target model is queried and its output is observed, without access to its internal structure and parameters. GAMMA attacks are formalised as a constrained optimisation problem:

$$\begin{aligned} &\underset{\boldsymbol{s}\in\mathcal{S}}{\text{minimize}}\ F(\boldsymbol{s}) = f(\boldsymbol{x}\oplus\boldsymbol{s}) + \lambda\cdot\mathcal{C}(\boldsymbol{s}), \\ &\text{subject to } q \leq T \end{aligned} \tag{4.1}$$

where $\boldsymbol{x}$ is the input malware file, $\boldsymbol{s}$ is one of the manipulations that can be applied to the input file. $F(\boldsymbol{s})$ is the objective function, which consists of two contradictory parts: $f(\boldsymbol{x}\oplus\boldsymbol{s})$, which is the output of the classifier in the manipulated program, and $\mathcal{C}(\boldsymbol{s})$, which is the penalty function that evaluates the number of bytes inserted into the input malware file. The hyperparameter $\lambda > 0$ adjusts the trade-off between these two parts, thus supporting solutions with fewer inserted bytes at the expense of reducing the probability of misclassifying the sample as benign. Since black-box optimisation of the target requires repeated queries of the target model f, a constraint $q \leq T$ is used, which is an upper bound on the maximum number of queries q that can be performed with the query budget T.

Therefore, the goal is to minimise the probability of detection and also minimise the size of the injected content. This optimisation problem is solved using a black-box genetic optimiser. First, a random matrix is generated that represents the initial population of N manipulation vectors. The algorithm then iterates in three steps: selection, crossover, and mutation. During selection, the objective function is evaluated and the N best candidate manipulation vectors are selected from the current population and the population created in the previous iteration. These are the candidate manipulation vectors associated with the lowest values of F. This is followed by the crossover function, which modifies the candidates from the previous step by mixing the values of pairs of randomly selected candidate vectors, and returns a new set of N candidates. The final operation is a mutation, which randomly changes the elements of each vector with low probability. In each iteration, N queries are performed on the target model to evaluate the objective function of new candidates and maintain the population of best candidates. When the maximum number of queries is reached or no further improvement in the objective function value is observed, the best manipulation vector from the current population is returned. The resulting sample of adversarial malware $\boldsymbol{x}^*$ is obtained by applying the optimal manipulation vector $\boldsymbol{s}^*$ to the input sample of malware $\boldsymbol{x}$ using the manipulation operator $\oplus$, such that $\boldsymbol{x}^* = \boldsymbol{x} \oplus \boldsymbol{s}^*$.

4.2.3 Gym-malware

The Gym-malware generator uses a reinforcement learning approach, where the environment consists of a malware sample and an anti-malware engine (a gradient-boosted decision tree model). The engine was trained on 50,000 malicious and 50,000 benign samples using extracted features such as byte-level data (e.g., histogram and entropy), headers, sections, imports, or exports. The agent is the algorithm used to modify the environment, with the default agent being Actor-Critic with Experience Replay (ACER) using the chainer-rl library. The agent has a set of functionality-preserving operations it can perform on a PE file. Through a series of games against the anti-malware engine, the agent learns which sequences of operations are likely to evade the detector for a given malware sample.

File mutations are actions available to the agent within the environment where a relatively small number of changes can be made to a PE file without affecting its format or performance. These actions include adding a function to the import address table, manipulating existing section names, creating new unused sections, adding bytes to the end of sections, creating a new entry point, manipulating debug information, packing or unpacking the file, changing the header checksum, and adding bytes to the end of a PE file.

At each step, the agent receives feedback from the environment in the form of a reward value and a function vector summarising the state of the environment. The reward function is measured by the antivirus engine, with a reward of 0 for a modified sample judged malicious, and R (set to 10) for a benign sample. The environment outputs the state as a feature vector, which is a 2350-dimensional vector consisting of PE header metadata, section metadata, import and export table metadata, human-readable string counts, byte histogram and 2D byte entropy histogram.

Based on the feedback, the agent selects mutations from the action set, and this process is repeated in multiple rounds. Each round starts with a malware sample that is modified through a series of mutations during the round. Rounds can be terminated prematurely if the agent evades the anti-malware engine before completing ten allowable mutations.

Chapter 5

Experiments

This chapter describes the experiments we performed and our results we obtained. We introduce the purpose of each experiment and present the hardware and software used to perform and evaluate them. Then we describe the datasets used and explain the individual metrics used to evaluate the experiments. Finally, we discuss each experiment individually and present and discuss the results.

5.1 Purpose of Experiments

Since the goal of this thesis is to compare different methods for generating adversarial examples, several experiments need to be performed. These experiments will help us to verify and compare the different characteristics, properties, and efficiency of the methods used to generate adversarial examples.

One of the basic characteristics we want to verify is the running time of the algorithm used to generate each sample. An experiment that examines this variable will help us to estimate the time requirements of each generation method. In addition to the time requirements of the algorithms; we should not forget the memory requirements. Some generation methods are expected to have a significant impact on the size of the original programs. Some may increase significantly (e.g. GAMMA padding and GAMMA section-injection attacks), while others may decrease (e.g. Gym-malware attack), depending on the type of file manipulation used. However, these assumptions need to be verified experimentally.

One of the most important properties that we are interested in is the success rate of each adversarial attack strategy. Therefore, we need to perform an experiment that will help us determine how successful each method is against different antivirus programs.

Studying the aforementioned properties of each method can help us to evaluate the entire process of generating examples using a particular adversarial attack strategy. However, the question is how these properties would change if multiple strategies were used simultaneously within a single sample generation process. We will not examine the time and memory requirements, as they should not be significantly different from the results measured for each method in the combination used. Much more interesting is the success rate of the combination of strategies. It is expected that by using a combination of strategies during a single generation process, we could theoretically achieve better results than by using each strategy separately. It is also possible that some combinations will not be as effective. However, these assumptions also need to be tested experimentally.

We, therefore, decided to perform four different experiments, the list of which is given below. A detailed description of the experiments can be found in the following sections.

1. **Sample Generation Time**: to test the time complexity of generating individual examples using different adversarial example generators.
2. **Sample Size**: to investigate the change in size when generating individual examples using different adversarial example generators.
3. **Bypassing Commercial AV Products**: to evaluate the success of selected methods used to generate adversarial malware examples.
4. **Combination of Multiple Techniques**: to test the effectiveness of using a combination of methods to generate adversarial malware examples.

5.2 Experimental Setup

The experiments were carried out on an NVIDIA DGX Station A100 server running Ubuntu 20.04.5 LTS. The server is equipped with an AMD EPYC 7742 processor running at 2.25GHz with 64 cores and 512 GiB of RAM. We also used virtual machines running Windows 11 and Kali Linux for testing and analysis.

We used two datasets for our experiments. The first dataset contains 3625 benign executables obtained from a fresh Windows 11 installation. The second dataset contains 3625 malicious executables obtained from the VirusShare repository [57]. It is important to note that all 7250 samples are portable executables.

In this study, we compared five adversarial attack strategies. Partial DOS and Full DOS attacks were performed in a white-box environment against the MalConv detector, using a maximum of 50 iterations. GAMMA padding and GAMMA section-injection attacks were performed in a black-box setting against the MalConv detector, using a maximum of 500 queries. The injection content was obtained by extracting 100 sections of a given type (.data, .text, .rdata) from benign programs. The regularisation parameter was set to 10^{-5}.

Gym-malware attack was performed in a black-box environment against the GDBT detector using its default settings. The Gym-malware model was trained on a dataset of 3000 malicious samples and a validation set of 1000 files that are not part of the experimental dataset described above. These samples were also obtained from the VirusShare repository.

5.3 Evaluation Metrics

We used several metrics to evaluate the experiments. These metrics are described in detail in this section.

The detection rate is the proportion of malware files that were correctly classified by the target malware classifier and can be calculated as follows:

$$detection\ rate = \frac{\#\ correctly\ classified}{\#\ total} \tag{5.1}$$

where $\#\ correctly\ classified$ is the number of malware samples correctly detected as malware and $\#\ total$ is the total number of files submitted to the target classifier.

The evasion rate is the proportion of malware files misclassified by the target malware classifier and can be calculated as follows:

$$evasion\ rate = \frac{\#\ misclassified}{\#\ total} \tag{5.2}$$

where $\#\ misclassified$ is the number of malware samples misclassified as benign and $\#\ total$ is the total number of files submitted to the target classifier after discarding files that were already incorrectly predicted before modification.

The evasion rate mentioned above is a universal metric that can be used to evaluate both single methods and combinations of methods. In both cases, we are interested in the percentage of malware that escaped detection by the antivirus program. In addition, we used the following metrics to evaluate the combination of attacks.

The first metrics that we chose to evaluate the success of the combination are the absolute and relative improvement in the evasion rate when using the second attack in the combination compared to the first attack. Absolute improvement can be described by the following formula:

$$improvement_A = evasion\ rate_C - evasion\ rate_1 \tag{5.3}$$

where $evasion\ rate_C$ is the total evasion rate when using a combination of methods, and $evasion\ rate_1$ is the evasion rate after using the first attack in the combination alone. The result is the percentage increase in evasion rate between the first and second attack in the combination. For example, if the evasion rate after the first attack in the combination is 0.01 and after the second attack is 0.1, then the absolute improvement is 0.09, meaning that the second attack improved the overall evasion rate by 9%.

Similarly, the relative improvement can be expressed using the formula:

$$improvement_R = \frac{evasion\ rate_C - evasion\ rate_1}{evasion\ rate_1} \tag{5.4}$$

where the meaning of the variables is the same as in the previous formula 5.3. However, in the previous case, the result expressed a percentage increase over all samples tested. In the case of relative improvement, we limit ourselves to the set of samples that escaped the antivirus program after the first attack. For example, if the evasion rate after the first attack of the combination is 0.01 and after the second attack it is 0.1, then the relative improvement is 9, i.e. the second attack improved the evasion rate of the first attack by 900%.

Next, we need to compare the combination of attacks to performing the attacks separately to see if the combination of attacks adds any value. To do this, we use two metrics, the first of which we will call the evasion rate benefit, and it is defined as follows:

$$evasion\ rate\ benefit = \frac{\#\ mis_C - \#\ mis_{C,1} - \#\ mis_{C,2} + \#\ mis_{C,1,2}}{\#\ total} \tag{5.5}$$

where $\#\ mis_C$ is the number of malware not detected as malware after the combination of attacks, $\#\ mis_{C,1}$ represents the number of malware that were not detected as malware after executing both the attack combination and the first attack alone, similarly $\#\ mis_{C,2}$ for the second attack. Furthermore, $\#\ mis_{C,1,2}$ is the number of malware counted under both $\#\ mis_{C,1}$ and $\#\ mis_{C,2}$. The denominator $\#\ total$ is the number of all malware in the test set. Thus, the result generally reflects the percentage of samples that escaped detection due to a combination of attacks; attacks performed only on these samples would have failed. If the result is 0, then the combination did not add anything over the first and second attacks performed separately.

The final metric is a simple comparison of the evasion rate of the combination of attacks to the evasion rate of the attacks performed separately. We call it *evasion rate cmp*, and it has the following calculation:

$$evasion\ rate\ cmp = evasion\ rate_C - MAX(evasion\ rate_1, evasion\ rate_2) \tag{5.6}$$

where $evasion\ rate_C$ is the evasion rate of the combination attack, while $evasion\ rate_1$ and $evasion\ rate_2$ are the evasion rates of the first and second attacks, respectively. If the result is positive, it means that the combination of attacks performed better than the combination of the two attacks in the combination that would have been performed alone. If the result is negative or zero, it means that the execution of the attack combination was pointless because one of the attacks that were part of the combination performed better or was equal to the combination in terms of evasion rate.

5.4 Experiments and Results

This section describes experiments in which we investigate various characteristics of selected methods for adversarial attacks. We provide the results of each experiment and compare the individual methods with each other.

5.4.1 Sample Generation Time

In this experiment, we measured the time required to generate individual samples using all of the algorithms listed. The attacker's goal is to generate functional adversarial examples in the shortest possible time. Table 5.1 shows the average time and standard deviation in seconds required to generate a sample using different methods.

Table 5.1: Average time to generate the sample for each sample generator.

Attack	Average Duration [s]	Standard Deviation [s]
Partial DOS	99.27	31.13
Full DOS	169.08	104.53
GAMMA padding	87.61	39.28
GAMMA section-injection	118.47	69.44
Gym-malware	5.73	7.52

Figure 5.1 displays a box plot graph that shows the time required to generate a sample for each of the five tested methods of adversarial attacks. The vertical axis represents time in seconds, and the horizontal axis represents the individual methods. The boxes in the graph represent the range of values measured for each method, and the red line inside the boxes represents the median. Outliers are displayed as individual points outside the boxes.

The image shows that the Gym-malware adversarial attack was the fastest, taking an average of less than 6 seconds, with some outlier values taking less than 100 seconds. In contrast, the Full DOS attack took the longest time to create adversarial examples, with an average duration of more than 160 seconds. The other three methods took similar amounts of time, around 100 seconds, although the GAMMA section-injection attack had several outlier values with extremely long durations of over 300 seconds.

The measured times may be affected by the settings of individual algorithms. For example, changing the number of iterations or queries could reduce the time required to generate a sample. However, such changes could significantly degrade the quality of the samples. Therefore, it is important to strike a balance between quality and time.

5.4.2 Sample Size

This experiment investigates how the size of original malware samples changes when using various adversarial malware generators. The attacker's goal is to

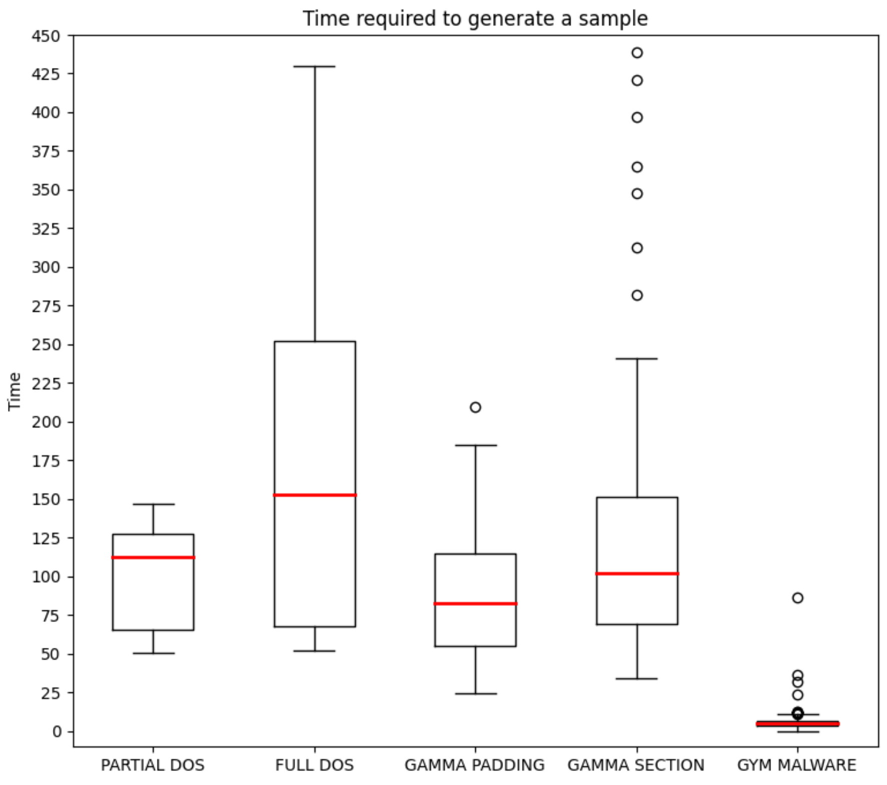

Figure 5.1: Time required to generate a sample for each sample generator.

minimise the increase in the size of the generated adversarial files, making them harder to distinguish from the original malware samples. Table 5.2 shows the average change in file size resulting from the use of various adversarial attack methods, measured in kilobytes.

Table 5.2: Average increase in sample size for each sample generator.

Attack	Average Size Increase [kB]	Standard Deviation [kB]
Partial DOS	0	0
Full DOS	0	0
GAMMA padding	223.60	48.40
GAMMA section-injection	1940.35	78088.98
Gym-malware	−149.27	754.66

If partial DOS and Full DOS attacks are used, which are based on strategic byte changes in the DOS header, the size of the resulting files does not change. This results in an average size change of zero.

In the case of GAMMA padding and GAMMA section-injection attacks, parts extracted from harmless files are inserted into the malicious file. The file size increases on average by 223,605 bytes in the case of GAMMA padding attack and by 1,940,352 bytes in the case of GAMMA section-injection attack.

The Gym-malware attack uses various types of file manipulation. The file size may be decreased, increased, or remain unchanged depending on the chosen modification. On average, the file size decreases by 149,273 bytes.

5.4.3 Bypassing Commercial AV Products

In this experiment, we analysed the effectiveness of created examples of adversarial malware against real AV detectors. Based on a comparative study by AV-Comparatives [58], we selected the top ten rated AV programs for the experiment, whose names we intentionally anonymised to minimise possible misuse of this work. In the following results of the experiment, we present only nine antivirus products because we gained identical results for two selected AVs from the same company.

Modified malware files from various adversarial algorithms were sent to the VirusTotal server [59] to obtain the evasion rate for each individual antivirus product. To avoid skewed results, we only analysed malware samples that achieved a 100% detection rate against all selected AV products before modification using adversarial techniques. Malware samples for which the generators, with the appropriate settings, failed to generate adversarial malware, were omitted from the final analysis. The aim was to have the same number of samples generated by each generator and the same set of original malware files modified by these individual generators. In total, 894 original malware samples and their modified versions were obtained for each selected generator.

The results of testing the evasion rate of modified malware samples against selected AV products are provided in Table 5.3. Each row in this table represents one of the sample modification methods used, each column in the table shows one of the selected AV products, and the values in the table represent the percentage of evasion achieved by the given algorithm against the given AV product. The last column of the table displays the average evasion rate for a given attack.

Table 5.3: The evasion rate achieved by adversarial examples generated against real antivirus products.

Attack	AV1	AV2	AV3	AV4	AV5	AV6	AV7	AV8	AV9	Average
GAMMA padding	0.00	1.79	0.45	0.22	0.45	0.90	1.34	0.56	0.45	0.68
Partial DOS	0.78	2.57	0.78	1.01	0.78	0.78	1.90	1.45	0.78	1.21
Full DOS	0.67	1.34	0.78	0.90	0.78	0.78	4.14	1.23	0.78	1.27
GAMMA section-injection	18.46	5.37	6.38	4.36	4.47	9.06	43.62	1.23	5.37	10.92
Gym-malware	45.53	19.02	44.86	67.23	41.61	53.58	53.80	26.51	44.86	44.11

From the results shown in the table, it is clear that the Gym-malware attack achieved the highest evasion rate against all selected AV products. Specifically, it achieved evasion rates ranging from 19% to 67.2%. The second best result was achieved by the GAMMA section-injection attack, with evasion rates ranging from 1.2% to 43.6%. However, the GAMMA padding attack achieved the worst results, as it did not fool any of the detectors tested in more than 1.8% cases. The Full DOS and Partial DOS attacks achieved slightly better results than the GAMMA padding attack, with the Full DOS attack slightly outperforming the Partial DOS attack.

Next, we investigate whether the type of section extracted from benign files affects the resulting evasion rate in GAMMA padding and GAMMA section-injection attacks. We chose three types of sections: .data, .text, and .rdata. The resulting evasion rate is recorded in Table 5.4.

Table 5.4: Comparison of evasion rate with different attack settings.

Attack	AV1	AV2	AV3	AV4	AV5	AV6	AV7	AV8	AV9	Average
GAMMA padding (.data)	0.00	1.79	0.45	0.22	0.45	0.90	1.34	0.56	0.45	0.68
GAMMA padding (.rdata)	0.11	2.35	0.22	0.22	0.22	0.11	3.69	1.01	0.22	0.91
GAMMA padding (.text)	0.11	3.58	0.34	0.22	0.34	0.34	3.80	1.01	0.34	1.12
GAMMA section-injection (.rdata)	7.27	7.16	9.73	4.25	4.25	8.39	31.77	1.57	8.50	9.21
GAMMA section-injection (.text)	4.36	7.27	12.98	3.47	6.82	7.94	42.84	1.79	10.52	10.89
GAMMA section-injection (.data)	18.46	5.37	6.38	4.36	4.47	9.06	43.62	1.23	5.37	10.92

The measured results show that no significant changes were observed. Furthermore, when comparing all attacks in Table 5.3, the resulting order of attacks by the average evasion rate achieved, did not change.

5.4.4 Combination of Multiple Techniques

In this experiment, we studied the effect of applying multiple techniques to a malware sample on the resulting evasion rate. We selected three adversarial example generators that performed the best in the previous test. Specifically, we used Full DOS, GAMMA section-injection (extraction of .data sections from benign files), and Gym-malware. The goal of this experiment was to demonstrate whether the application of multiple adversarial example generators to a malware sample would significantly increase its evasion rate.

An overview of the experiment is shown in Figure 5.2. First, the original malware samples are processed by the first generator. These modified samples are then tested against real AV detectors that are not part of the generator. This step was already done in the previous experiment. The measured results are shown in Table 5.3. The result is a set of samples divided into two sets. The first set consists of *evasive examples* that successfully evaded the given malware detector, and this set is no longer processed. In contrast to *adversarial examples*, which are generated against the target classifier, *evasive examples* are samples that have evaded detection by the AV program, although this AV program was not used to generate these samples. The second set consists of *failed examples* that failed to evade the detector and are used as input to the second generator. The second generator processes the *failed examples* from the input, and the resulting modified samples are again

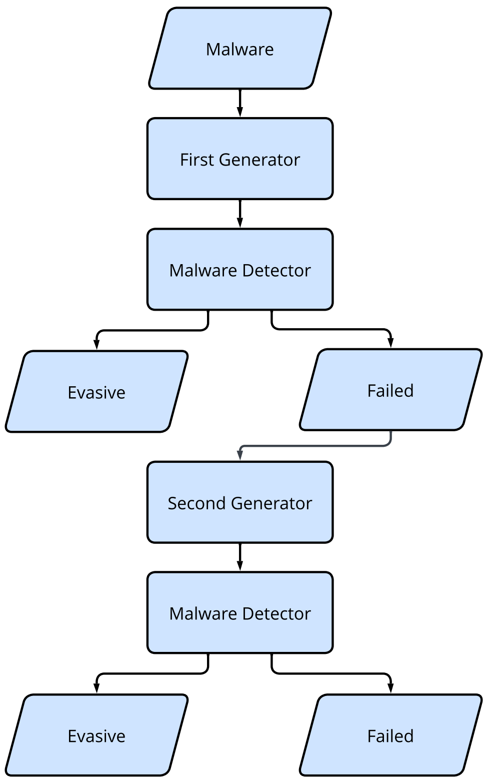

Figure 5.2: Method for generating adversarial examples by combining two generators.

tested against real AV detectors. The result is again a set of samples divided into two sets: *evasive* and *failed*. The set of *evasive examples* produced by the first generator and the set of *evasive examples* produced by the second generator together form the set of resulting successful adversarial examples produced by combining these two generators. The *failed examples* obtained after using the second generator are the resulting samples that did not evade detection.

In the following parts of this section, we present the results of our experiment, divided according to the evaluation metrics used. All metrics are described in Section 5.3. The original data used to calculate the following aggregated data can be found in the files attached to this thesis and in Appendix C.

This section contains two types of tables. The first type of table lists the measured minimum, average, and maximum values for a particular AVs across the nine combinations of generators. The AVs selected and labelled here correspond to those used in the previous experiment described in Section 5.4.3. The second type of table lists the measured minimum, average, and maximum values for a particular combination of generators across the nine AVs. The First Generator column contains the first generator used, and the Second Generator column contains the second generator used.

Evasion Rate

First, we present the results of the evasion rate metric. Table 5.5 shows the results of the evasion rate for each AV. Table 5.6 shows the results for each generator combination. Both tables contain aggregated data, specifically the minimum, average, and maximum values across all AVs and all generator combinations, respectively. The original tables with the detailed results of the evasion rates obtained for each AV using each generator combination can be found in the Appendix of this thesis.

Table 5.5: Evasion rate for each AV using all combinations of generators.

AV	Minimum	Average	Maximum
AV1	0.78	32.39	55.26
AV2	1.45	17.79	29.53
AV3	0.90	31.15	63.09
AV4	1.45	38.59	78.19
AV5	0.78	26.76	57.61
AV6	0.90	35.91	73.60
AV7	5.26	49.32	74.50
AV8	1.57	17.80	41.39
AV9	0.78	30.79	62.75

For all AVs, we examine the minimum, average, and maximum of the results of all combinations of generators tested in the experiment. These results can be found in Table 5.5. For the minimum values of the evasion rate, we can see that none of the antivirus programs reached a detection rate of 100%. On the other hand, all these values are relatively low compared to the average and maximum values, which tells us that some of the 9 generator combinations were not very successful. The average evasion rates range from about 18% to 49%, and the maximum values range from 30% to 78%. This means that if we use all combinations in the experiment, we achieve an average evasion rate of at least 18% for all selected AVs, and some of the combinations achieve an evasion rate of around 30% for all AVs. The best result was achieved by combinations of generators against AV7, where the average evasion rate is around 49%, while the least successful was against AV2, where the average evasion rate is around 18%.

Table 5.6 shows the results of the evasion rate achieved by each combination of generators. Here, we can see that the most successful combination was the one in which the Gym-malware generator was used twice in a row. This achieved a minimum evasion rate of around 30% for all AVs. The average value for this combination is around 58%, and for at least one AV we achieved

Table 5.6: Evasion rate for each generator combination against all AVs.

First Generator	Second Generator	Minimum	Average	Maximum
Full DOS	Full DOS	0.78	1.54	5.26
Full DOS	GAMMA section-injection	1.57	10.48	43.96
Full DOS	Gym-malware	23.15	38.69	61.63
GAMMA section-injection	Full DOS	6.26	15.05	45.30
GAMMA section-injection	GAMMA section-injection	1.90	14.03	44.52
GAMMA section-injection	Gym-malware	25.39	46.97	74.50
Gym-malware	Full DOS	26.51	46.16	67.34
Gym-malware	GAMMA section-injection	27.18	49.22	67.79
Gym-malware	Gym-malware	29.53	58.34	78.19

an evasion rate of around 78% with this combination. On the other hand, the worst combination in terms of evasion rate is the one in which the Full DOS generator was used twice. In this case, we have an average evasion rate of about 2%, while for all other combinations, this value exceeds 10%, and for some even significantly. The maximum value of the evasion rate for this combination is about 5%, for the others we have at least about 44%.

To determine the effectiveness of using generator combinations instead of individual generators alone, we can compare these results with the values from the previous experiment. However, we have additional metrics for this, which are described in Section 5.3. We analyse the results of these metrics in the following parts of this section.

Absolute Improvement

Next, we evaluate the metric that we identified as an absolute improvement in Section 5.3. In short, it is the percentage difference in the evasion rate between the evasion rate achieved by combining both generators and the evasion rate achieved by using only the first generator.

First, we focus on Table 5.7, which shows the absolute improvement values for each AV in the nine combinations of generators. Here we can see that for some AVs we were unable to improve the evasion rate using the second generator application. Specifically, this refers to the minimum value for AV4, AV5, and AV9. Table 5.8 helps us to identify the relevant generators. We can see that they are only combinations in which we use the same generator twice, namely, the Full DOS generator or the GAMMA section-injection generator. This may indicate that the use of these combinations is not entirely effective. For the average values in Table 5.7, we can see that they do not differ significantly, in all cases between about 8% and 15%. For the maximum values, we have a slightly higher range, approximately 22% to 61%. This means that if we use a second generator, we will improve the evasion rate by about 10% on average after using the first generator. If we use all combinations, we will achieve an improvement in the evasion rate of at least 22% for all AVs.

Table 5.7: Absolute improvement for each AV using all combinations of generators.

AV	Minimum	Average	Maximum
AV1	0.11	10.84	29.98
AV2	0.11	9.21	22.48
AV3	0.11	13.81	41.16
AV4	0.00	14.43	60.74
AV5	0.00	11.14	35.01
AV6	0.11	14.77	50.22
AV7	0.90	15.46	40.72
AV8	0.34	8.14	26.51
AV9	0.00	13.78	41.39

However, significantly more intriguing is Table 5.8, which shows the absolute improvement for each generator combination in all AVs. Here, we can see that the use of Full DOS as the second generator does not result in a significant absolute improvement in the evasion rate for the minimum, average, and maximum values. This means that using Full DOS as the second generator in a combination is the least effective. We could also see this in the previous section regarding the evasion rate, where these combinations did not achieve the best result in any case. On the other hand, we can see that we get the best absolute improvement by using the Gym-malware method as the second generator in the combination. The GAMMA section-injection as the second generator does not have the best or worst results.

Table 5.8: Absolute improvement for each generator combination against all AVs.

First Generator	Second Generator	Minimum	Average	Maximum
Full DOS	Full DOS	0.00	0.27	1.12
Full DOS	GAMMA section-injection	0.67	9.21	39.82
Full DOS	Gym-malware	21.92	37.42	60.74
GAMMA section-injection	Full DOS	0.78	4.13	5.93
GAMMA section-injection	GAMMA section-injection	0.00	3.11	10.29
GAMMA section-injection	Gym-malware	20.02	36.04	51.23
Gym-malware	Full DOS	0.11	2.05	7.49
Gym-malware	GAMMA section-injection	0.56	5.11	10.63
Gym-malware	Gym-malware	7.72	14.23	20.02

In Table 5.8, we can observe another interesting fact. As we have already mentioned, the use of two identical generators in combination is not very effective. However, this statement does not apply to the Gym-malware generator. In contrast, if we use the Gym-malware generator as the first generator in the combination, then the best choice to select the second generator seems to be the Gym-malware generator again. Full DOS and GAMMA section-injection generators have minimal absolute improvements in the role of the second generator. This means that the Gym-malware generator is very successful even when used alone, combining generators does not improve it much, in the case that this generator was used as the first generator in the combination.

The best absolute improvement values are achieved when we choose Full DOS as the first generator and Gym-malware as the second. This results in an absolute improvement in the evasion rate in all AVs of at least around 22%, on average 37% and up to a maximum of 61%. On the other hand, the worst results in terms of absolute evasion rate improvement are obtained when we choose Full DOS as both generators. In this case, we obtain a minimum absolute improvement of 0% across all antivirus programs, an average of 0.3% and a maximum of 1.1%. It follows that the Full DOS generator is likely to be the least successful generator, both when used alone and in combination.

Relative Improvement

We will now examine the metric known as the relative improvement in the evasion rate. Analogously to the improvement in the absolute evasion rate, we can also see in the table 5.9 AVs that we did not improve the evasion rate with the second generator. These are still the same AVs, specifically AV4, AV5, and AV9 and combinations of generators consisting of two identical generators Full DOS or GAMMA section-injection. Table 5.9 shows that AV7 performed the best in terms of relative improvement of the evasion rate. Conversely, AV4 performed the worst, where in some cases the second generator managed to increase the number of successfully modified samples (which evaded detection) by more than 67 times.

Table 5.9: Relative improvement for each AV using all combinations of generators.

AV	Minimum	Average	Maximum
AV1	0.61	788.88	4466.67
AV2	8.33	307.66	1675.00
AV3	3.74	732.66	5014.29
AV4	0.00	914.67	6787.50
AV5	0.00	633.29	4257.14
AV6	1.46	846.29	6000.00
AV7	2.05	232.70	983.78
AV8	3.80	510.39	2154.55
AV9	0.00	758.01	5285.71

Table 5.10 provides confirmation that the Full DOS generator, which was used as the second generator in combination, does not appear to be an entirely ideal choice. On the other hand, if this generator is chosen as the first generator in the combination, the GAMMA section-injection or Gym-malware generator can increase the number of samples that evade detection by up to 67 times. This shows that the Full DOS generator is not a very strong generator

on its own. Regarding the Gym-malware generator, we can see that when it is used as the first generator in a combination, the second generator does not significantly increase the result, even when Gym-malware is used again as the second generator. However, this is not a significant difference when compared to the absolute improvement in the evasion rate. The Gym-malware generator is very successful on its own, so as the first generator in combination, it achieves a significantly high evasion rate. Therefore, a relative improvement in the evasion rate from 39% to 56% can cause a drastic increase in the total evasion rate of the combination (up to 20%). We can also see that when the Gym-malware generator is used as a second generator, different from the first generator used, there is a huge relative improvement over the first generator. Again, we can see that Gym-malware is very effective when used in any combination of generators.

Table 5.10: Relative improvement for each generator combination against all AVs.

First Generator	Second Generator	Minimum	Average	Maximum
Full DOS	Full DOS	0.00	18.93	62.50
Full DOS	GAMMA section-injection	75.00	715.51	2400.00
Full DOS	Gym-malware	983.78	4027.99	6787.50
GAMMA section-injection	Full DOS	3.85	104.90	409.09
GAMMA section-injection	GAMMA section-injection	0.00	58.50	181.25
GAMMA section-injection	Gym-malware	70.77	741.93	2154.55
Gym-malware	Full DOS	0.17	7.31	39.41
Gym-malware	GAMMA section-injection	0.83	13.60	42.94
Gym-malware	Gym-malware	16.31	35.90	56.12

Evasion Rate Benefit

In this section, we introduce a metric we have labeled the Evasion Rate Benefit. This metric expresses the percentage of samples that the combination modified in such a way that the new samples evade AV detection and these samples were not successfully created using individual generators separately. The metric allows us to determine whether the combination of generators successfully modified samples that individual generators failed to modify, and whether it produced novel results.

In Table 5.11 we can see that we can find AVs where some combinations of generators do not bring any benefit compared to using individual generators separately. These are AV4, AV5, and AV9. For the others, the combinations always produced some successfully modified samples (in terms of malware detection) that the individual generators would not have produced, even if it was a very small percentage of samples. AV1 performed worse on this metric, with generator combinations producing at most about 8% samples capable of evading detection by AV1 in addition to the individual generators

Table 5.11: Evasion rate benefit for each AV using all combinations of generators.

AV	Minimum	Average	Maximum
AV1	0.11	3.42	7.72
AV2	0.11	6.28	14.99
AV3	0.11	7.51	18.23
AV4	0.00	4.05	10.96
AV5	0.00	6.19	16.00
AV6	0.11	6.86	20.02
AV7	0.90	5.18	11.86
AV8	0.34	5.77	15.55
AV9	0.00	7.52	17.90

used separately. In contrast, AV3 and AV6 performed the worst, with some combinations successfully altering up to 20% additional samples.

Table 5.12: Evasion rate benefit for each generator combination against all AVs.

First Generator	Second Generator	Minimum	Average	Maximum
Full DOS	Full DOS	0.00	0.27	1.12
Full DOS	GAMMA section-injection	0.00	0.81	2.80
Full DOS	Gym-malware	5.93	12.96	15.88
GAMMA section-injection	Full DOS	0.67	3.59	5.26
GAMMA section-injection	GAMMA section-injection	0.00	3.11	10.29
GAMMA section-injection	Gym-malware	5.82	13.32	19.46
Gym-malware	Full DOS	0.11	1.88	7.16
Gym-malware	GAMMA section-injection	0.34	2.60	4.92
Gym-malware	Gym-malware	7.72	14.23	20.02

Table 5.12 demonstrates that the combination of two identical Full DOS generators is not an effective approach. This combination does not provide any significant benefit. However, the use of the Gym-malware generator twice brings an average of up to 14% of new successfully modified samples, with a maximum of around 20%. In general, this combination appears to be the best in terms of the evasion rate benefit metric. Even so, other combinations in which Gym-malware is used as a second generator are also very successful. The least successful combinations are those where Full DOS is used as the second generator.

Overall, this metric shows that by using combinations of generators, we can create a significant percentage of new samples that are able to evade AV detection. From this point of view, the experiment confirms the usefulness of this method in modifying malware samples. At the same time, the tables show that this sense is greater for some combinations of attacks and less for others.

Evasion Rate Comparison

The last metric examined, Evasion Rate Comparison, helps us find the answer to the question of whether it is better to use individual generators separately or to use them in a combination. This metric also measures the level of leakage and simply observes whether the result is better when a better generator from the combination is used separately or when a combination of generators is used. A negative value of this metric indicates that we have achieved a better evasion rate by using the better of the two generators in that combination separately (better in terms of evasion rate). On the contrary, a positive value indicates that we achieved a better result using the combination of both generators.

Table 5.13: Relative improvement for each AV using all combinations of generators.

AV	Minimum	Average	Maximum
AV1	-14.88	0.87	9.73
AV2	0.11	5.28	10.52
AV3	-4.81	4.02	18.23
AV4	-11.63	-0.31	10.96
AV5	-7.49	2.06	16.00
AV6	-5.82	3.03	20.02
AV7	-8.95	4.43	20.69
AV8	-3.36	2.52	14.88
AV9	-2.69	3.99	17.90

Looking at the minimum values in table 5.13, we can see that almost always there was a combination of generators that were not effective due to the negative values in this column. The exception is AV2, where we achieved a better evasion rate in all combinations of generators used. However, the average and maximum values show that in most cases the combination is more effective than the better of the two generators used separately. Only AV4 has a negative value, which means that it performed best. On the other hand, AV2 performed the worst when comparing these programs on average.

Table 5.14 shows that combining generators using Full DOS as the first generator and either GAMMA section-injection or Gym-malware as the second generator yields negative results, as evidenced by both the minimum and average values being negative. Based on the results of the previous parts of this section, we can say that it is more advantageous to use GAMMA section-injection and Gym-malware generators separately in such cases. We can also see other negative values in the minimum value of this metric for the combination of GAMMA section-injection and Gym-malware generators used in this order. However, the average value of the combination is positive, indicating that it may be reasonable to use this combination. However, further research is needed to confirm this conclusion. For the remaining combinations, we can say that using them will result in a better level of leakage than using

Table 5.14: Relative improvement for each generator combination against all AVs.

First Generator	Second Generator	Minimum	Average	Maximum
Full DOS	Full DOS	0.00	0.27	1.12
Full DOS	GAMMA section-injection	-2.80	-0.45	1.57
Full DOS	Gym-malware	-14.88	-5.42	4.81
GAMMA section-injection	Full DOS	0.78	4.13	5.93
GAMMA section-injection	GAMMA section-injection	0.00	3.11	10.29
GAMMA section-injection	Gym-malware	-11.63	2.86	20.69
Gym-malware	Full DOS	0.11	2.05	7.49
Gym-malware	GAMMA section-injection	0.56	5.11	10.63
Gym-malware	Gym-malware	7.72	14.23	20.02

the better of the two generators separately. Once again, the combination of two Gym-malware generators was the best in this evaluation.

Conclusion

This thesis investigates the use of adversarial learning techniques in the field of malware detection. Our goal was to apply existing methods to generate adversarial malware samples, test their effectiveness against selected malware detectors, and compare the evasion rate achieved and the practical applicability of these methods.

For our experiments, we chose five adversarial malware sample generators: Partial DOS, Full DOS, GAMMA padding, GAMMA section-injection, and Gym-malware. We compared techniques based on gradient, evolutionary algorithms, and reinforcement learning, and tested them using nine different antivirus products.

To validate and compare the different characteristics and properties of the methods used to generate adversarial malware samples, we performed four experiments. These included tracking the time taken to generate samples, changes in sample size after using the generator, testing effectiveness against antivirus programs, and combining the use of multiple generators for a sample.

The results indicate that making optimised modifications to previously detected malware can cause the classifier to misclassify the file and label it as benign. Furthermore, the study confirms that generated malware samples can be successfully used against detection models other than those used to generate them. Using combination attacks, a significant percentage of new samples can be created that can evade detection by antivirus programs.

Experiments show that the Gym-malware generator, which uses a reinforcement learning approach, has the greatest practical potential. This generator produces malware samples in the shortest time, with an average sample generation time of 5.73 seconds. The Gym-malware generator achieved the highest evasion rate among all selected antivirus products, with the highest evasion rate recorded at 67%. In addition, the Gym-malware generator was found to be effective when used in combination with other generators. However, when we used Gym-malware in combination twice in a row, we achieved the highest evasion rate, up to 78%.

The results suggest that the use of generator combinations may be an effective method of modifying malware samples to evade detection by antivirus programs. However, the effectiveness of this approach depends on the specific combination of generators used. The Gym-malware generator is effective, whether used alone or in combination with other generators. The Full DOS generator is generally the least effective, whether used alone or in combination. The GAMMA section-injection generator is also effective when used alone in some combinations, but significantly less so than the Gym-malware generator.

Our work highlights the importance of developing effective techniques to detect malware and identify adversarial attacks. More research in this area is needed to successfully combat new threats and attacks.

Bibliography

[1] What Is a Cyberattack?. In: *Cisco* [online]. Cisco Systems, 2023. [visited on 2023-01-10]. Available from: https://www.cisco.com/c/en/us/products/security/common-cyberattacks.html

[2] Distribution of Malware and PUA by Operating System. In: *AV-ATLAS* [online]. AV-TEST, 2023. [visited on 2023-01-10]. Available from: https://portal.av-atlas.org/malware

[3] Malware Statistics & Trends Report. In: *AV-TEST* [online]. AV-TEST, 2023. [visited on 2023-01-11]. Available from: https://www.av-test.org/en/statistics/malware/

[4] WRESSNEGGER, Christian et al. Automatically Inferring Malware Signatures for Anti-Virus Assisted Attacks. In: *Proceedings of the 2017 ACM on Asia Conference on Computer and Communications Security* [online]. New York, NY, USA: Association for Computing Machinery, 2017, pp. 587-598. [visited on 2023-01-11]. ISBN 9781450349444. Available from: doi: https://doi.org/10.1145/3052973.3053002

[5] ASLAN, Omer and Refik SAMET. A Comprehensive Review on Malware Detection Approaches. In: *IEEE Access* [online]. 2020, vol. 8, pp. 6249-6271. [visited on 2023-01-12]. ISSN 2169-3536. Available from: doi: https://doi.org/10.1109/ACCESS.2019.2963724

[6] DOLEJŠ, Jan and Martin JUREČEK. Interpretability of Machine Learning-Based Results of Malware Detection Using a Set of Rules. In: *Artificial Intelligence for Cybersecurity* [online]. Springer, 2022, pp. 107-136. [visited on 2023-01-14]. ISBN 978-3-030-97087-1. Available from: doi: https://doi.org/10.1007/978-3-030-97087-1_5

[7] BIGGIO, Battista and Fabio ROLI. Wild Patterns: Ten Years After the Rise of Adversarial Machine Learning. In: *Proceedings of the 2018 ACM*

SIGSAC Conference on Computer and Communications Security [online]. New York, NY, USA: Association for Computing Machinery, 2018, pp. 2154–2156. [visited on 2023-01-14]. ISBN 9781450356930. Available from: doi: `https://doi.org/10.1145/3243734.3264418`

[8] CHEN, Lingwei, Yanfang YE and Thirimachos BOURLAI. Adversarial Machine Learning in Malware Detection: Arms Race between Evasion Attack and Defense. In: *2017 European Intelligence and Security Informatics Conference (EISIC)* [online].Athens, Greece: IEEE, 2017, pp. 99-106. [visited on 2023-01-18]. ISBN 978-1-5386-2385-5. Available from: doi: `https://doi.org/10.1109/EISIC.2017.21`

[9] HU, Yang et al. Transferability of Adversarial Examples in Machine Learning-based Malware Detection. In: *2022 IEEE Conference on Communications and Network Security (CNS)* [online]. Austin, TX, USA: IEEE, 2022, pp. 28-36. [visited on 2023-01-20]. ISBN 978-1-6654-6255-6. Available from: doi: `https://doi.org/10.1109/CNS56114.2022.9947226`

[10] OPREA, Alina and Apostol VASSILEV. *Adversarial Machine Learning: A Taxonomy and Terminology of Attacks and Mitigations.* NIST AI 100-2e2023 ipd [online]. Gaithersburg, MD: National Institute of Standards and Technology, 2023. [visited on 2023-04-13]. Available from: doi: `https://doi.org/10.6028/NIST.AI.100-2e2023.ipd`

[11] LING, Xiang et al. Adversarial attacks against Windows PE malware detection: A survey of the state-of-the-art. In: *Computers & Security* [online]. 2023, vol. 128 [visited on 2023-03-23]. ISSN 0167-4048. Available from: doi: `https://doi.org/10.1016/j.cose.2023.103134`

[12] ARYAL, Kshitiz, Maanak GUPTA and Mahmoud ABDELSALAM. A survey on adversarial attacks for malware analysis. In: *ArXiv* [online]. 2022, arXiv:2111.08223v2. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.2111.08223`

[13] CARLINI, Nicholas. Attacking Machine Learning: On the Security and Privacy of Neural Networks. In: *Nicholas Carlini* [online]. San Francisco, Moscone Center: RSA Conference, 4-8 March 2019. [visited on 2023-03-22]. Available from: `https://nicholas.carlini.com/slides/2019_rsa_attacking_ml.pdf`

[14] BIGGIO, Battista. Machine Learning Security: Adversarial Attacks and Defenses. In: *Battista Biggio* [online]. Prague, Czech Republic: CyberSec & AI Prague, 2019. [visited on 2023-03-23]. Available from: `https://battistabiggio.github.io/talks/`

[15] GOODFELLOW, Ian J., Jonathon SHLENS and Christian SZEGEDY. Explaining and Harnessing Adversarial Examples. In: *ArXiv* [online]. 2015, arXiv:1412.6572v3. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.1412.6572`

[16] FANG, Zihua. Analysis of Adversarial Attack Based on FGSM. In: *Third International Conference on Intelligent Computing and Human-Computer Interaction (ICHCI 2022)* [online]. SPIE, 2023. [visited on 2023-04-25]. Available from: doi: `https://doi.org/10.1117/12.2656064`

[17] SUTTON, Richard S. and Andrew G. BARTO. *Reinforcement learning: An introduction.* [online]. Second edition. MIT press, 2020 [visited on 2023-04-25]. Available from: doi: `http://incompleteideas.net/book/RLbook2020.pdf`

[18] EIBEN, Agoston E. and Marc SCHOENAUER. Evolutionary computing. In: *ArXiv* [online]. 2005, arXiv:cs/0511004. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.cs/0511004`

[19] EIBEN, A.E. and J.E. SMITH. *Introduction to Evolutionary Computing* [online]. Berlin, Heidelberg: Springer Berlin Heidelberg, 2015. [visited on 2023-04-25]. ISBN 978-3-662-44874-8. Available from: doi: `https://doi.org/10.1007/978-3-662-44874-8`

[20] GOODFELLOW, Ian J. et al. Generative Adversarial Networks. In: *ArXiv* [online]. 2014, arXiv:1406.2661. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.1406.2661`

[21] CRESWELL, Antonia et al. Generative Adversarial Networks: An Overview. In: *IEEE Signal Processing Magazine* [online]. IEEE, 2018, pp. 53-65 [visited on 2023-04-05]. ISSN 1558-0792. Available from: doi: `https://doi.org/10.1109/MSP.2017.2765202`

[22] GAN Training. In: *Machine Learning: Google Developers* [online]. Google, 2022. [visited on 2023-04-05]. Available from: `https://developers.google.com/machine-learning/gan/training`

[23] The Generator. In: *Machine Learning: Google Developers* [online]. Google, 2022. [visited on 2023-04-05]. Available from: `https://developers.google.com/machine-learning/gan/generator`

[24] The Discriminator. In: *Machine Learning: Google Developers* [online]. Google, 2022. [visited on 2023-04-05]. Available from: `https://developers.google.com/machine-learning/gan/discriminator`

[25] EYKHOLT, Kevin et al. Robust Physical-World Attacks on Deep Learning Visual Classification. In: *2018 IEEE/CVF Conference on Computer Vision and Pattern Recognition* [online]. IEEE, 2018, pp. 1625-1634 [visited on 2023-04-05]. ISBN 978-1-5386-6420-9. Available from: doi: `doi:10.1109/CVPR.2018.00175`

[26] MADRY, Aleksander et al. Towards Deep Learning Models Resistant to Adversarial Attacks. In: *ArXiv* [online]. 2019, arXiv:1706.06083v4. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.1706.06083`

[27] REN, Kui et al. Adversarial Attacks and Defenses in Deep Learning. In: *Engineering* [online]. 2020, vol. 6(3), pp. 346-360 [visited on 2023-04-05]. ISSN 20958099. Available from: doi: `https://doi.org/10.1016/j.eng.2019.12.012`

[28] QUIRING, Erwin et al. Against All Odds: Winning the Defense Challenge in an Evasion Competition with Diversification. In: *ArXiv* [online]. 2020, arXiv:2010.09569. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.2010.09569`

[29] STANLEY, Jarek. Machine Learning Security Evasion Competition 2020 Invites Researchers to Defend and Attack. In: *Microsoft* [online]. Microsoft, 2020. [visited on 2023-04-25]. Available from: `https://msrc.microsoft.com/blog/2020/06/machine-learning-security-evasion-competition-2020-invites-researchers-to-defend-and-attack/`

[30] Windows File Types. In: *AV-ATLAS* [online]. AV-TEST, 2023. [visited on 2023-01-21]. Available from: `https://portal.av-atlas.org/malware/statistics`

[31] KOWALCZYK, Krzysztof. Portable Executable File Format. In: *Krzysztof Kowalczyk* [online]. Krzysztof Kowalczyk, 2018. [visited on 2023-01-21]. Available from: `https://blog.kowalczyk.info/articles/pefileformat.html`

[32] PE Format. In: *Microsoft* [online]. Microsoft, 2023. [visited on 2023-03-21]. Available from: `https://learn.microsoft.com/en-us/windows/win32/debug/pe-format`

[33] PIETREK, Matt. Inside Windows: An In-Depth Look into the Win32 Portable Executable File Format. In: *Microsoft* [online]. MSDN Magazine, 2008. [visited on 2023-03-21]. Available from: `https://learn.microsoft.com/en-us/previous-versions/bb985992(v=msdn.10)?redirectedfrom=MSDN`

[34] WEBSTER, George D. et al. Finding the Needle: A Study of the PE32 Rich Header and Respective Malware Triage. In: *Detection of Intrusions and Malware, and Vulnerability Assessment* [online]. Springer, 2017, pp. 119-138. [visited on 2023-01-18]. ISBN 978-3-319-60876-1. Available from: doi: `https://doi.org/10.1007/978-3-319-60876-1_6`

[35] DEMETRIO, Luca et al. Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection. In: *ACM Transactions on Privacy and Security* [online]. 2021, vol. 24(4), no. 27, pp. 1-31. [visited on 2023-04-08]. ISSN 2471-2566. Available from: doi: `https://doi.org/10.1145/3473039`

[36] DEMETRIO, Luca et al. Functionality-Preserving Black-Box Optimization of Adversarial Windows Malware. In: *IEEE Transactions on Information Forensics and Security* [online]. 2021, vol. 16, pp. 3469-3478. [visited on 2023-04-01]. ISSN 1556-6021. Available from: doi: `https://doi.org/10.1109/TIFS.2021.3082330`

[37] DEMETRIO, Luca. *Formalizing evasion attacks against machine learning security detectors* [online]. Italy, 2020. Doctoral Thesis. Università degli studi di Genova, Dipartimento di Informatica, bioingegneria, robotica e ingegneria dei sistemi. [visited on 2023-04-01]. Available from: doi: `https://dx.doi.org/10.15167/demetrio-luca_phd2021-01-20`

[38] DEMETRIO, Luca, Battista BIGGIO and Fabio ROLI. Practical Attacks on Machine Learning: A Case Study on Adversarial Windows Malware. In: *IEEE Security & Privacy* [online]. 2022, vol. 20, no. 5, pp. 77-85. [visited on 2023-04-01]. Available from: doi: `https://doi.org/10.1109/MSEC.2022.3182356`

[39] KOLOSNJAJI, Bojan et al. Adversarial Malware Binaries: Evading Deep Learning for Malware Detection in Executables. In: *2018 26th European Signal Processing Conference (EUSIPCO)* [online]. Rome, Italy: IEEE, 2018, pp. 533-537. [visited on 2023-04-08]. ISBN 978-9-0827-9701-5. Available from: doi: `https://doi.org/10.23919/EUSIPCO.2018.8553214`

[40] KREUK, Felix et al. Deceiving End-to-End Deep Learning Malware Detectors using Adversarial Examples. In: *arXiv* [online]. 2019, arXiv:1802.04528v3. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.1802.04528`

[41] SUCIU, Octavian, Scott E. COULL and Jeffrey JOHNS. Exploring Adversarial Examples in Malware Detection. In: *2019 IEEE Security and Privacy Workshops (SPW)* [online]. San Francisco, CA, USA: IEEE,

2019, pp. 8-14. [visited on 2023-04-08]. ISBN 978-1-7281-3508-3. Available from: doi: `https://doi.org/10.1109/SPW.2019.00015`

[42] DEMETRIO, Luca et al. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. In: *ArXiv* [online]. 2019, arXiv:1901.03583v2. [visited on 2023-04-08]. Available from: doi: `https://doi.org/10.48550/arXiv.1901.03583`

[43] ANDERSON, Hyrum S. et al. Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning. In: *ArXiv* [online]. 2018, arXiv:1801.08917v2. [visited on 2023-04-08]. Available from DOI: `https://doi.org/10.48550/arXiv.1801.08917`

[44] SONG, Wei, et al. MAB-Malware: A Reinforcement Learning Framework for Attacking Static Malware Classifiers. In: *ArXiv* [online]. 2021, arXiv:2003.03100v3. [visited on 2023-04-08]. Available from DOI: `https://doi.org/10.48550/arXiv.2003.03100`

[45] CASTRO, Raphael Labaca, Corinna SCHMITT and Gabi DREO. AIMED: Evolving Malware with Genetic Programming to Evade Detection. In: *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)* [online]. Rotorua, New Zealand: IEEE, 2019, pp. 240-247. [visited on 2023-04-01]. ISBN 978-1-7281-2777-4. Available from: doi: `https://doi.org/10.1109/TrustCom/BigDataSE.2019.00040`

[46] WANG, Xiruo and Risto MIIKKULAINEN. MDEA: Malware Detection with Evolutionary Adversarial Learning. In: *2020 IEEE Congress on Evolutionary Computation (CEC)* [online]. Glasgow, UK: IEEE, 2020, pp. 1-8. [visited on 2023-04-01]. ISBN 978-1-7281-6929-3. Available from: doi: `https://doi.org/10.1109/CEC48606.2020.9185810`

[47] HU, Weiwei and Ying TAN. Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN. In: *Data Mining and Big Data* [online]. Springer, 2023, pp. 409-423. [visited on 2023-04-01]. ISBN 978-981-19-8991-9. Available from: doi: `https://doi.org/10.1007/978-981-19-8991-9_29`

[48] KAWAI, Masataka, Kaoru OTA and Mianxing DONG. Improved MalGAN: Avoiding Malware Detector by Leaning Cleanware Features. In: *2019 International Conference on Artificial Intelligence in Information and Communication (ICAIIC)* [online]. Okinawa, Japan: IEEE, 2019, pp. 40-45. [visited on 2023-04-01]. ISBN 978-1-5386-7822-0. Available from: doi: `https://doi.org/10.1109/ICAIIC.2019.8669079`

[49] CHADNJ, Vaya and Sen BEDANG. Pesidious: Malware Mutation using Deep Reinforcement Learning and GANs. In: *GitHub* [online]. 2020. [visited on 2023-04-22]. Available from: `https://github.com/CyberForce/Pesidious`

[50] KOZÁK, Matouš and Martin JUREČEK. Combining Generators of Adversarial Malware Examples to Increase Evasion Rate. In: *ArXiv* [online]. 2023, arXiv:2304.07360. [visited on 2023-04-22]. Available from: doi: `https://doi.org/10.48550/arXiv.2304.07360`

[51] KOZÁK, Matouš. *Application of Reinforcement Learning to Creating Adversarial Malware Samples* [online]. Prague, 2023. [visited on 2023-04-22]. Master's thesis. Czech Technical University in Prague, Faculty of Information Technology. Available from: `https://dspace.cvut.cz/handle/10467/107246`

[52] ANDERSON, Hyrum S. and Phil ROTH. EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models. In: *ArXiv* [online]. 2018, arXiv:1804.04637v2. [visited on 2023-04-26]. Available from: doi: `https://doi.org/10.48550/arXiv.1804.04637`

[53] RAFF, Edward et al. Malware Detection by Eating a Whole EXE. In: *ArXiv* [online]. 2017, arXiv:1710.09435. [visited on 2023-04-27]. Available from: doi: `https://doi.org/10.48550/arXiv.1710.09435`

[54] KE, Guolin et al. LightGBM: A Highly Efficient Gradient Boosting Decision Tree. In: *Advances in Neural Information Processing Systems 30 (NIPS 2017)* [online]. Long Beach, CA, USA: NeurIPS Proceedings, 2017. [visited on 2023-04-27]. Available from: `https://proceedings.neurips.cc/paper_files/paper/2017/file/6449f44a102fde848669bdd9eb6b76fa-Paper.pdf`

[55] DEMETRIO, Luca and Battista BIGGIO. secml-malware: Pentesting Windows Malware Classifiers with Adversarial EXEmples in Python. In: *ArXiv* [online]. 2021, arXiv:2104.12848v2. [visited on 2023-04-27]. Available from: doi: `https://doi.org/10.48550/arXiv.2104.12848`

[56] THOMAS, Romain. *LIEF – Library to Instrument Executable Formats* [online]. 2017 [visited on 2023-04-27]. Available from: `https://lief.quarkslab.com/`

[57] *VirusShare* [online]. [visited on 2023-03-20]. Available from: `https://virusshare.com/`

[58] AV-COMPARATIVES. Malware Protection Test September 2022. *AV-Comparatives* [online]. AV-Comparatives, 2022. [visited on 2023-03-20]. Available from: `https://www.av-comparatives.org/tests/malware-protection-test-september-2022/`

[59] *VirusTotal* [online]. [visited on 2023-03-20]. Available from: `https://www.virustotal.com/`

Appendix A

Acronyms

ACER	Actor-Critic with Experience Replay
AE	Adversarial Example
AML	Adversarial Machine Learning
API	Application Programming Interface
ASCII	American Standard Code for Information Interchange
ASLR	Address Space Layout Randomization
AV	Antivirus
CNN	Convolutional Neural Network
COFF	Common Object File Format
DEP	Data Execution Prevention
DLL	Dynamically Linked Library
DOS	Disk Operating System
EA	Evolutionary Algorithm
EP	Evolutionary Programming
ES	Evolutionary Strategy
EXE	Executable
FGSM	Fast Gradient Sign Method
GA	Genetic Algorithm
GAN	Generative Adversarial Network

GBDT	Gradient Boosted Decision Tree
GP	Genetic Programming
GUI	Graphic User Interface
ML	Machine Learning
OS	Operating System
PE	Portable Executable
RL	Reinforcement Learning
RVA	Relative Virtual Address
SEH	Structured Exception Handling

Appendix B

Content of Attachments

```
README.pdf ........................... PDF file with thesis description
data ................................... Directory with measured data
src...................................... Directory with source codes
text ...................................... Directory with thesis text
    bib ............................ Directory with bibliography files
    fig ....................................... Directory with images
    tex ............................ Directory with LaTeX source codes
    txt ......................... Directory with thesis text source files
    DP_Louthanova_Pavla_2023.pdf .............. Thesis in PDF format
```

Appendix C

Combination of Multiple Techniques

This appendix contains the measured results of the experiment described in detail in Section 5.4.4. The results are divided according to the metric used for the evaluation. All the tables presented have the same general structure, but differ in the metric used for evaluation. In the upper left corner of the table, the label of the specific AV is given, which corresponds to the label in Chapter 5. The first column lists the names of the generators used first within a given generator combination. The first row lists the names of the second generators used within a given generator combination. The values listed in the table correspond to the result of the specific metric for the generator combination in the row and column.

Evasion Rate

Table C.1: Evasion rates of combinations of AE generators against AV1.

AV1	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.78	16.78	30.65
GAMMA section-injection	22.60	18.57	46.42
Gym-malware	47.20	55.26	53.24

Table C.2: Evasion rates of combinations of AE generators against AV2.

AV2	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	1.45	6.94	23.83
GAMMA section-injection	11.30	7.94	25.39
Gym-malware	26.51	27.18	29.53

Table C.3: Evasion rates of combinations of AE generators against AV3.

AV3	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.90	5.71	40.05
GAMMA section-injection	11.30	16.67	47.54
Gym-malware	46.53	48.55	63.09

Table C.4: Evasion rates of combinations of AE generators against AV4.

AV4	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	1.45	1.57	61.63
GAMMA section-injection	9.40	4.36	55.59
Gym-malware	67.34	67.79	78.19

Table C.5: Evasion rates of combinations of AE generators against AV5.

AV5	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.78	4.14	34.12
GAMMA section-injection	9.28	7.83	39.49
Gym-malware	43.40	44.18	57.61

Table C.6: Evasion rates of combinations of AE generators against AV6.

AV6	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.90	8.50	47.76
GAMMA section-injection	9.84	9.40	59.28
Gym-malware	54.36	59.51	73.60

Table C.7: Evasion rates of combinations of AE generators against AV7.

AV7	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	5.26	43.96	44.86
GAMMA section-injection	45.30	44.52	74.50
Gym-malware	55.37	64.43	65.66

Table C.8: Evasion rates of combinations of AE generators against AV8.

AV8	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	1.57	2.46	23.15
GAMMA section-injection	6.26	1.90	27.74
Gym-malware	28.19	27.52	41.39

Table C.9: Evasion rates of combinations of AE generators against AV9.

AV9	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.78	4.25	42.17
GAMMA section-injection	10.18	15.10	46.76
Gym-malware	46.53	48.55	62.75

Absolute Improvement

Table C.10: Absolute improvements of combinations of AE generators against AV1.

AV1	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	16.11	29.98
GAMMA section-injection	4.14	0.11	27.96
Gym-malware	1.68	9.73	7.72

Table C.11: Absolute improvements of combinations of AE generators against AV2.

AV2	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	5.59	22.48
GAMMA section-injection	5.93	2.57	20.02
Gym-malware	7.49	8.17	10.52

Table C.12: Absolute improvements of combinations of AE generators against AV3.

AV3	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	4.92	39.26
GAMMA section-injection	4.92	10.29	41.16
Gym-malware	1.68	3.69	18.23

Table C.13: Absolute improvements of combinations of AE generators against AV4.

AV4	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.56	0.67	60.74
GAMMA section-injection	5.03	0.00	51.23
Gym-malware	0.11	0.56	10.96

Table C.14: Absolute improvements of combinations of AE generators against AV5.

AV5	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	3.36	33.33
GAMMA section-injection	4.81	3.36	35.01
Gym-malware	1.79	2.57	16.00

Table C.15: Absolute improvements of combinations of AE generators against AV6.

AV6	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	7.72	46.98
GAMMA section-injection	0.78	0.34	50.22
Gym-malware	0.78	5.93	20.02

Table C.16: Absolute improvements of combinations of AE generators against AV7.

AV7	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	1.12	39.82	40.72
GAMMA section-injection	1.68	0.90	30.87
Gym-malware	1.57	10.63	11.86

Table C.17: Absolute improvements of combinations of AE generators against AV8.

AV8	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.34	1.23	21.92
GAMMA section-injection	5.03	0.67	26.51
Gym-malware	1.68	1.01	14.88

Table C.18: Absolute improvements of combinations of AE generators against AV9.

AV9	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	3.47	41.39
GAMMA section-injection	4.81	9.73	41.39
Gym-malware	1.68	3.69	17.90

Relative Improvement

Table C.19: Relative improvements of combinations of AE generators against AV1.

AV1	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	16.67	2400.00	4466.67
GAMMA section-injection	22.42	0.61	151.52
Gym-malware	3.69	21.38	16.95

Table C.20: Relative improvements of combinations of AE generators against AV2.

AV2	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	8.33	416.67	1675.00
GAMMA section-injection	110.42	47.92	372.92
Gym-malware	39.41	42.94	55.29

Table C.21: Relative improvements of combinations of AE generators against AV3.

AV3	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	14.29	628.57	5014.29
GAMMA section-injection	77.19	161.40	645.61
Gym-malware	3.74	8.23	40.65

Table C.22: Relative improvements of combinations of AE generators against AV4.

AV4	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	62.50	75.00	6787.50
GAMMA section-injection	115.39	0.00	1174.36
Gym-malware	0.17	0.83	16.31

Table C.23: Relative improvements of combinations of AE generators against AV5.

AV5	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	428.57	4257.14
GAMMA section-injection	107.50	75.00	782.50
Gym-malware	4.30	6.18	38.44

Table C.24: Relative improvements of combinations of AE generators against AV6.

AV6	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	14.29	985.71	6000.00
GAMMA section-injection	8.64	3.70	554.32
Gym-malware	1.46	11.07	37.37

Table C.25: Relative improvements of combinations of AE generators against AV7.

AV7	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	27.03	962.16	983.78
GAMMA section-injection	3.85	2.05	70.77
Gym-malware	2.91	19.75	22.04

Table C.26: Relative improvements of combinations of AE generators against AV8.

AV8	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	27.27	100.00	1781.82
GAMMA section-injection	409.09	54.55	2154.55
Gym-malware	6.33	3.80	56.12

Table C.27: Relative improvements of combinations of AE generators against AV9.

AV9	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	442.86	5285.71
GAMMA section-injection	89.58	181.25	770.83
Gym-malware	3.74	8.23	39.90

Evasion Rate Benefit

Table C.28: Evasion rate benefits of combinations of AE generators against AV1.

AV1	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	2.80	5.93
GAMMA section-injection	3.58	0.11	5.82
Gym-malware	1.45	3.24	7.72

Table C.29: Evasion rate benefits of combinations of AE generators against AV2.

AV2	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	0.90	14.99
GAMMA section-injection	5.26	2.57	11.52
Gym-malware	7.16	3.47	10.52

Table C.30: Evasion rate benefits of combinations of AE generators against AV3.

AV3	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	0.11	14.99
GAMMA section-injection	4.25	10.29	16.00
Gym-malware	1.45	2.13	18.23

Table C.31: Evasion rate benefits of combinations of AE generators against AV4.

AV4	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.56	0.34	10.63
GAMMA section-injection	4.36	0.00	9.17
Gym-malware	0.11	0.34	10.96

Table C.32: Evasion rate benefits of combinations of AE generators against AV5.

AV5	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	0.00	13.87
GAMMA section-injection	4.14	3.36	15.21
Gym-malware	1.57	1.57	16.00

Table C.33: Evasion rate benefits of combinations of AE generators against AV6.

AV6	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	0.34	15.44
GAMMA section-injection	0.67	0.34	19.46
Gym-malware	0.67	4.70	20.02

Table C.34: Evasion rate benefits of combinations of AE generators against AV7.

AV7	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	1.12	1.79	11.75
GAMMA section-injection	1.57	0.90	11.19
Gym-malware	1.57	4.92	11.86

Table C.35: Evasion rate benefits of combinations of AE generators against AV8.

AV8	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.34	0.67	13.20
GAMMA section-injection	4.36	0.67	15.55
Gym-malware	1.45	0.78	14.88

Table C.36: Evasion rate benefits of combinations of AE generators against AV9.

AV9	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	0.34	15.88
GAMMA section-injection	4.14	9.73	16.00
Gym-malware	1.45	2.24	17.90

Evasion Rate Comparison

Table C.37: Evasion rate comparisons of combinations of AE generators against AV1.

AV1	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	-1.68	-14.88
GAMMA section-injection	4.14	0.11	0.90
Gym-malware	1.68	9.73	7.72

Table C.38: Evasion rate comparisons of combinations of AE generators against AV2.

AV2	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	1.57	4.81
GAMMA section-injection	5.93	2.57	6.38
Gym-malware	7.49	8.17	10.52

Table C.39: Evasion rate comparisons of combinations of AE generators against AV3.

AV3	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	-0.67	-4.81
GAMMA section-injection	4.92	10.29	2.69
Gym-malware	1.68	3.69	18.23

Table C.40: Evasion rate comparisons of combinations of AE generators against AV4.

AV4	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.56	-2.80	-5.59
GAMMA section-injection	5.03	0.00	-11.63
Gym-malware	0.11	0.56	10.96

Table C.41: Evasion rate comparisons of combinations of AE generators against AV5.

AV5	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	-0.34	-7.49
GAMMA section-injection	4.81	3.36	-2.13
Gym-malware	1.79	2.57	16.00

Table C.42: Evasion rate comparisons of combinations of AE generators against AV6.

AV6	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.11	-0.56	-5.82
GAMMA section-injection	0.78	0.34	5.71
Gym-malware	0.78	5.93	20.02

Table C.43: Evasion rate comparisons of combinations of AE generators against AV7.

AV7	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	1.12	0.34	-8.95
GAMMA section-injection	1.68	0.90	20.69
Gym-malware	1.57	10.63	11.86

Table C.44: Evasion rate comparisons of combinations of AE generators against AV8.

AV8	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.34	1.23	-3.36
GAMMA section-injection	5.03	0.67	1.23
Gym-malware	1.68	1.01	14.88

Table C.45: Evasion rate comparisons of combinations of AE generators against AV9.

AV9	Full DOS	GAMMA section-injection	Gym-malware
Full DOS	0.00	-1.12	-2.69
GAMMA section-injection	4.81	9.73	1.90
Gym-malware	1.68	3.69	17.90