San Jose State University

SJSU ScholarWorks

Faculty Research, Scholarly, and Creative Activity

1-1-2024

Creating valid adversarial examples of malware

Matouš Kozák
Czech Technical University in Prague

Martin Jureček
Czech Technical University in Prague

Mark Stamp
San Jose State University, mark.stamp@sjsu.edu

Fabio Di Troia
San Jose State University, fabio.ditroia@sjsu.edu

Follow this and additional works at: https://scholarworks.sjsu.edu/faculty_rsca

Recommended Citation

Matouš Kozák, Martin Jureček, Mark Stamp, and Fabio Di Troia. "Creating valid adversarial examples of malware" *Journal of Computer Virology and Hacking Techniques* (2024). https://doi.org/10.1007/s11416-024-00516-2

This Article is brought to you for free and open access by SJSU ScholarWorks. It has been accepted for inclusion in Faculty Research, Scholarly, and Creative Activity by an authorized administrator of SJSU ScholarWorks. For more information, please contact scholarworks@sjsu.edu.

Journal of Computer Virology and Hacking Techniques
https://doi.org/10.1007/s11416-024-00516-2

ORIGINAL PAPER

Creating valid adversarial examples of malware

Matouš Kozák[1] · Martin Jureček[1] · Mark Stamp[2] · Fabio Di Troia[2]

Received: 9 September 2023 / Accepted: 4 February 2024
© The Author(s) 2024

Abstract

Because of its world-class results, machine learning (ML) is becoming increasingly popular as a go-to solution for many tasks. As a result, antivirus developers are incorporating ML models into their toolchains. While these models improve malware detection capabilities, they also carry the disadvantage of being susceptible to adversarial attacks. Although this vulnerability has been demonstrated for many models in white-box settings, a black-box scenario is more applicable in practice for the domain of malware detection. We present a method of creating adversarial malware examples using reinforcement learning algorithms. The reinforcement learning agents utilize a set of functionality-preserving modifications, thus creating valid adversarial examples. Using the proximal policy optimization (PPO) algorithm, we achieved an evasion rate of 53.84% against the gradient-boosted decision tree (GBDT) detector. The PPO agent previously trained against the GBDT classifier scored an evasion rate of 11.41% against the neural network-based classifier MalConv and an average evasion rate of 2.31% against top antivirus programs. Furthermore, we discovered that random application of our functionality-preserving portable executable modifications successfully evades leading antivirus engines, with an average evasion rate of 11.65%. These findings indicate that ML-based models used in malware detection systems are sensitive to adversarial attacks and that better safeguards need to be taken to protect these systems.

Keywords Validity · Adversarial examples · Malware detection · PE files · Reinforcement learning

1 Introduction

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

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

Using malware detection models based on machine learning yields promising results [3]. Nonetheless, ML models are susceptible to adversarial examples (AEs) that can mislead the models [4]. For example, a minor modification of a malware file can make its feature vector resemble some of the feature vectors of benign files. Consequently, this can cause the malware classifier to make an incorrect prediction.

State-of-the-art antivirus programs incorporate both static and dynamic analysis in their inner workings. Static analysis methods, which do not require the executable to be run, provide us with information such as opcode sequences, program format information, and others. On the other hand, the dynamic analysis consists of studying the program's activ-

✉ Matouš Kozák
matous.kozak@fit.cvut.cz

Martin Jureček
martin.jurecek@fit.cvut.cz

Mark Stamp
mark.stamp@sjsu.edu

Fabio Di Troia
fabio.ditroia@sjsu.edu

1 Faculty of Information Technology, Czech Technical University in Prague, Prague, Czechia

2 Department of Computer Science, San Jose State University, San Jose, CA, USA

Published online: 18 March 2024

ity during execution and recording information such as API calls, registry and memory changes, and more [5].

In this work, we target our attack on static malware analysis for numerous reasons. Firstly, static detection is time-efficient as it does not involve the execution of binary executables. As a result, it is typically the first line of defense against unwanted threats and is thus a critical part of any antivirus program. Secondly, dynamic analysis requires executing malware inside a secure sandbox and recording its behavior, which is both time and technically demanding. Thirdly, malware authors can incorporate sandbox-evading techniques to detect that their malware is running inside a controlled environment and stop its malicious behavior [6, 7].

Our goal is to implement a technique of adversarial attack at the level of samples, i.e., a technique that would create functional AEs. This task is considerably more demanding, as typical machine learning models operate at the level of feature vector, and reliable reverse mapping from a feature vector back to a binary file is difficult to perform. Therefore, we chose to perform adversarial perturbations on the binary level. To verify the functional preservation of our adversarial perturbations, we present a method comparing behavior patterns before and after modifications, which ensures maximum validity of the generated AEs.

Our adversarial attack works in a black-box scenario, mimicking the most difficult scenario where no information about the target classifier apart from the final prediction label is known to the attacker. While the defense methods should be tested in the most open (white-box) scenario to ensure the protection of the system in all settings [8], we believe that the attack methods should be tested in the black-box scenario to provide a worst-case estimate of their abilities.

In this work, we introduce a set of functionality-preserving binary file modifications. Further, we train reinforcement learning agents to use these modifications to modify Windows malware binaries to avoid detection by the target classifier. Additionally, we compare trained and random agents and assess the transferability of our attacks to other malware detectors. This research demonstrates that a large number of malware detection models are vulnerable to adversarial attacks and should be better protected against these threats.

Paper outline:

- In Sect. 2, we establish the necessary background. Starting with an introduction to adversarial machine learning, continuing with a brief dive into reinforcement learning and finishing with a description of the portable executable file format.
- In Sect. 3, we display related work focusing on the area of adversarial malware generation.
- In Sect. 4, we define our method in detail. From modification of binary files and protocol to guarantee they retain their original functionality, to describing our reinforcement learning environment and agents.
- In Sect. 5, we introduce our experiments, present the results achieved, and suggest ideas for future research.
- In Conclusion, we summarize the contributions of our work.

2 Background

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

2.1 Adversarial machine learning

Adversarial machine learning is an area of machine learning focusing on improving ML systems to withstand adversarial attacks both from inside (data poisoning) and outside (evasion attacks). An adversarial attack is a carefully created action to mislead the ML model. The victim model is also called a target model, and the attacker is called an adversary. Nevertheless, both attacker and adversary are used interchangeably in the current literature. The object responsible for misleading the target model is referred to as an *adversarial example (AE)*. Adversarial machine learning is commonly employed in the malware detection domain to mislead AV products into incorrectly classifying malicious files as benign.

The success of an adversarial attack is dependent on the available knowledge of the targeted system [9]. When the adversary has access to the system and can examine its internal settings or training datasets, we call this a *white-box* scenario. Contrarily, we refer to a situation as a *black-box* scenario if the adversary only has access to a limited amount of information, typically just the model's final prediction, such as the malware/benign label for each sample that is presented. In between these two is a *grey-box* scenario where the adversary has higher access to the system than in the black-box scenario, but the access is still limited to some parts. For example, the attacker can use the model's score or feature space but cannot access and modify its training dataset. Since the specific structure of the AV is typically unknown to the attacker, the black-box scenario is the most realistic for producing adversarial malware examples.

2.2 Reinforcement learning

Reinforcement learning (RL) is a branch of machine learning where an *agent* equipped with a set of actions is learning how to reach its goal. The agent can be a bot learning to play a computer game or a physical robot working in a factory. Based on trial and error and appropriate feedback from an interactive element called the *environment*, the agent learns which actions are "good" and "bad" for achieving its goal. The crucial challenge for reinforcement learning is a balance of *exploration* and *exploitation*, i.e., how to explore enough of the environment while maximizing the reward and reaching its goal. This section is based on the book [10], where you can find more details and examples on this topic.

There are three key elements of reinforcement learning: the agent's policy, the reward signal, and the value function. A model of the environment is also included for some tasks, but we will not specify this further.

The core part of any reinforcement learning agent is *policy*. It is a function mapping from the states of the environment to an action from a set of agent actions representing the agent's behavior at a given time. If learned correctly, it should lead to a strategy that maximizes the total rewards the agent receives.

The *reward signal* is an immediate response to a taken action provided by the environment. This signal grades action taken at a given state as good or bad concerning the agent's goal.

The *value function* estimates how rewarding the current state is. The ultimate goal of every RL agent is to achieve the highest total reward, also called return. This goal usually cannot be accomplished by following states and actions with the highest immediate rewards but rather with the highest values, as these maximize the cumulative reward.

Although there are other formal definitions of reinforcement learning, in this work, we follow the one presented in [10]. Reinforcement learning can be defined as repeated interactions between agent and environment at discrete time steps $t = 0, 1, 2, \ldots T$. At time step t, the environment is at a state $S_t \in S$ where S is a set of all possible states. After the agent is presented with the state S_t, based on its policy π, creates a mapping $S_t \rightarrow A_t \in A(S_t)$, where $A(S_t)$ is a set of all possible actions at state S_t. In many scenarios, $A(S_t)$ can change based on the current state S_t, but in others, it can remain fixed depending on the environment. After deciding on the action A_t, the chosen action is sent to the environment where it gets executed. The subsequent response from the environment gets presented to the agent in the form of a new state S_{t+1}, and reward $R_{t+1} \in R \subset \mathbb{R}$, where R is the set of all possible rewards. Figure 1 illustrates the interaction between the agent and the environment.

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

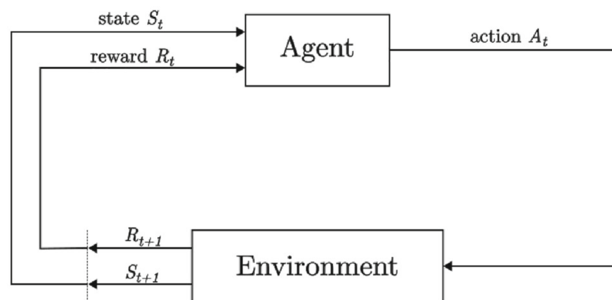

Fig. 1 Interaction between the agent and the environment

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

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

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

2.2.1 Algorithms

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

A popular algorithm called **Q-learning** was introduced in [11] by Watkins. This algorithm works by iteratively estimating action-value function $Q(s, a)$, also called Q values. Q values represent values of individual actions (a) in each state (s) and are calculated using the following formula:

$$Q^{new}(S_t, A_t) = Q(S_t, A_t) + \alpha\Big(R_{t+1} + \gamma \max_a\{Q(S_{t+1}, a)\} - Q(S_t, A_t)\Big) \quad (2)$$

where $a \in A(S_{t+1})$ and α is the learning rate. The $\max_a\{Q(S_{t+1}, a)\}$ represents the best value in the following state S_{t+1}. The newly calculated Q values (Q^{new}) are stored in a so-called Q table.

Improvement of the Q-learning algorithm called **deep Q-network (DQN)**, or deep Q-learning, was introduced by Mnih et al. [12, 13]. DQN replaces the tabular manner of storing all state-action pairs $Q(s, a)$ with a function, usually taking the form of a neural network. The Q value is then defined as $Q(s, a; \xi)$, where ξ can be one or more function parameters.

The above-introduced algorithms learn the value functions and choose appropriate actions based on them. **Policy gradient (PG)** methods optimize the policy π directly [14]. The policy is parameterized by weight vector $\theta \in \mathbb{R}^n$. The decision on choosing action a is therefore not only conditioned by the state s but also by the vector θ. The policy can then be defined as $\pi(a|s, \theta)$, i.e., the probability of taking action a given that the agent is in the state s with weight vector θ at time step t. Policy gradient methods use a gradient ascent algorithm to find the optimal value of θ to maximize the total return.

Proximal policy optimization (PPO) is a variant of the policy gradient methods where the policy vector is updated only after several gradient ascent iterations [15]. This is in contrast with the original PG methods, which perform one gradient ascent update of the target policy per sample. As stated in the original paper, this improvement is easy to implement, yet it brings substantial performance improvements.

2.3 Portable executable file format

Portable executable (PE) is a file format commonly found on Windows operating systems for various types of files, such as executables (EXEs) or dynamically linked libraries (DLLs). This file format is based on the Common Object File Format (COFF) found on Unix operating systems. It contains all the necessary information for the operating system (OS) loader to correctly map the PE file to system memory [16].

In this section, we will describe the PE file format used for EXE files, as the usage of some fields differs from other file types. The PE file format has a rigid structure, as presented in Fig. 2. Most of the information listed in this section comes from the official Windows documentation [17].

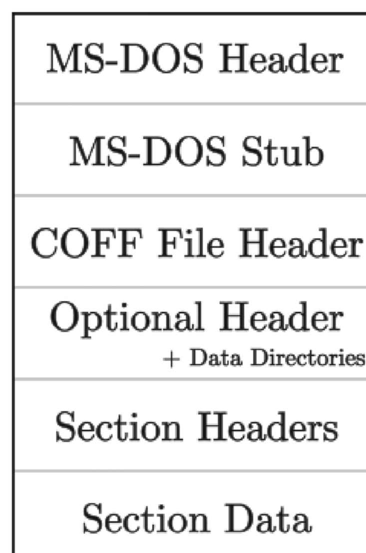

Fig. 2 PE File Format

2.3.1 MS-DOS header and stub program

MS-DOS header and stub program are still part of the PE file format for backward compatibility with older operating systems (MS-DOS). Nowadays, if a modern Windows executable gets executed on MS-DOS, it should display some variation of the following message: "This program cannot be run in DOS mode.".

MS-DOS header is 64 bytes long and is located at the beginning of the PE file. The first field of this header is `e_magic`, also called a magic number. This field usually contains a value of `0x5A4D`, a hexadecimal representation of the characters MZ, initials of one of the MS-DOS developers, Mark Zbikowski [18]. This field is followed by several other important fields for the MS-DOS system, which are not relevant for modern systems. The header is concluded with the `e_lfanew` field, which stores a file offset to the COFF file header.

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

2.3.2 COFF file header

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

2.3.3 Optional header

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

The **standard fields** are eight fields used for each COFF file, which contain items such as:

- `Magic`: Indicates the type of optional header (32-bit/64-bit).
- `SizeOfCode`: The size of the code section, usually called *.text*.
- `AddressOfEntryPoint`: Relative virtual address (RVA) of the entry point (the first program instruction when execution begins) after being loaded into memory.

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

- `SizeOfImage`: The size of the PE file.
- `CheckSum`: Checksum value used to validate files such as drivers or DLLs.
- `NumberOfRvaAndSizes`: Number of entries in the data directories.

At the end of the optional header are placed the **data directories**. These directories form an array of 8-byte structures with two fields: RVA and size of the directory. There are 15 types of data directories, such as *export*, *import*, *debug*, or *certificate* tables.

2.3.4 Section headers and data

Immediately following the optional header and data directories is the section table, also known as **section headers**. Each section header contains ten fields totaling 40 bytes in size. Examples of the fields include:

- `Name`: Eight bytes representing the name of the section padded with zeros, e.g., `.text\0\0\0`.
- `VirtualSize`: Section size when loaded into memory.
- `Characteristics`: 4-byte flag indicating section attributes, e.g., the section contains executable code or can be shared in memory.

The location and size of the relevant **section data** are indicated in the corresponding section header. An ordinary PE file usually has several commonly found sections [16]. Although their names may vary from file to file, their intentions remain the same. The most notable is the *.text* section, which encapsulates all pieces of code and naturally `AddressOfEntryPoint` points to this section.

One of the critical parts of nearly every EXE is the import directory table (IDT), or just the import table. This table is usually stored in the *.idata* section, with each entry representing one of the imported DLLs. Each entry contains the RVAs of the name of the imported DLL and the import address and lookup tables associated with this DLL. The import lookup table (ILT) encodes the imported function names from a given DLL. While stored on disk, the import address table (IAT) has the same structure and content as ILT. However, after the PE file is loaded into memory, the IAT entries contain the addresses of the imported functions instead of the function names.

The section name *.debug* refers to a section containing debugging information. This section is not memory-mapped by default, and the PE file format also allows the information to be stored in a separate debug file.

When a PE file has a certificate, e.g., to ensure file origin or immutability, the location is specified in a security data directory inside the optional header. The security data directory points to the beginning of the certificate table, which contains 20-byte entries for each certificate. This certificate table is usually appended to the end of the file in a segment typically called an *overlay*.

3 Related work

This section summarizes related publications that focus on creating adversarial malware examples. We break down this section into several subsections depending on the approach used to generate AEs. We begin by describing works based on the same technique as our work, i.e., reinforcement learning attacks. Then, we present the research that exploits the back-propagation algorithm commonly used in training deep neural networks with so-called gradient-based attacks [4, 19]. Note that most of these gradient-based attacks classify as white-box attacks since they work directly with the inner configurations of targeted models. Lastly, we mention several publications related to adversarial malware attacks that do not fit within these two categories. A summary of publications related to generating evasive AEs is presented in Table 1.

3.1 Reinforcement learning-based attacks

Adversarial attacks using reinforcement learning algorithms are based on a fundamental concept in which RL agents are equipped with a set of file manipulations that are used to perturbate malicious binaries to evade detection. The file manipulations are usually self-contained modifications of executable files on a binary level.

One of the first works done in the domain of generating AEs using reinforcement learning was published in 2018 by Anderson et al. [20]. The authors presented a gym-malware framework equipped with RL agents and an OpenAI gym [31] environment. They targeted a gradient-boosted decision tree (GBDT) trained on 100,000 binary files and achieved an evasion rate of up to 24%, depending on the dataset used.

In [21], Fang et al. presented two models, a detector called DeepDetectNet and a generator of AEs called RLAttackNet. In a pure black-box scenario, their generator, based on the

Table 1 Summary of related adversarial attacks

Paper	Knowledge	Problem space	Target model	Functionality
Anderson et al. [20]	Grey-scale, black-box	Binary files	GBDT	×
Fang et al. [21]	Black-box	Binary files	Deep neural network	×
Song et al. [22]	Black-box	Binary files	GBDT, MalConv, AVs	Sandbox
Quertier et al. [23]	Grey-scale, black-box	Binary files	GBDT, MalConv, AVs, Grayscale	Sandbox
Kolosnjaji et al. [24]	White-box	Binary files	MalConv	×
Kreuk et al. [25]	White-box	Binary files	MalConv	×
Demetrio et al. [26]	White-box	Binary files	MalConv	×
Yang et al. [27]	White-box	Binary files	Convolution neural network	Manually
Hu and Tan [28]	White-box, black-box	Extracted features	Deep neural network	×
Ebrahimi et al. [29]	Black-box	Binary files	MalConv	Sandbox
Demetrio et al. [30]	Grey-box, black-box	Binary files	GBDT, MalConv	×

The symbol × denotes that functionality preservation was not empirically verified

DQN algorithm, bypassed their own detector in 19.13% of cases. The authors later used the AEs to retrain their DeepDetectNet model, and the evasion rate of RLAttackNet dropped to 3.1%.

The MAB-Malware framework was presented by Song et al. [22]. As an agent, the authors used a multi-armed bandit (MAB) model while adjusting the agent's action set in real-time by adding successful action-content pairs. Further, the authors introduced an action minimization procedure, which removes unnecessary modifications after successful evasion, reducing the final size of AEs. The authors targeted the GBDT[1] by EMBER [32], MalConv[1] [33], and several commercial AV detectors. They recorded a high evasion rate of 74.4% against GBDT, 97.7% against MalConv, and up to 48.3% against commercial AVs. Further, they validated the functionality preservation of generated AEs by comparing file signatures with genuine malware files.

Quertier et al. in [23] used reinforcement learning algorithms to attack MalConv, GBDT by EMBER and Grayscale (convolutional neural network interpreting PE binaries as images) classifiers in grey-scale settings with available prediction scores for learning. Further, the authors targeted commercial AV in a pure black-box environment as well. They used the DQN and REINFORCE (policy gradient algorithm) agents and achieved a high evasion rate against all targeted models, including an 80% evasion rate with REINFORCE against GBDT, a 100% perfect evasion against MalConv with both algorithms and a 70% evasion rate against commercial AV with REINFORCE. However, Quertier et al. did not specify what commercial AV they were targeting, nor did the authors disclose their models for further use.

3.2 Gradient-based attacks

In [24], Kolosnjaji et al. proposed a gradient-based attack against the MalConv malware detector [33]. Their attack only perturbed the overlay part of the file and achieved an evasion rate of 60% while modifying less than 1% of total bytes.

Kreuk et al. [25] used a gradient-based attack, limited to injecting small-scale chunks of bytes into unused parts or at the end of the file. The authors argued that these modifications do not change the functionality of the file but without further justification. The authors scored a high evasion rate of 99% against the MalConv classifier.

Another attack on MalConv was carried out by Demetrio et al. in [26]. Using an integrated gradient method, the authors studied which sections of binaries stimulate the MalConv classifier and thus are vulnerable to adversarial attacks. They found that MalConv partially bases its prediction on features in the DOS header and achieved an evasion rate of more than 86% by only modifying the DOS header.

A variation on the method introduced in [25] was presented in 2021 by Yang et al. [27]. The authors treated the input EXEs as images, used as input into a convolution neural network. They calculated necessary byte perturbations for evasion and then transformed them into specific byte sequences. Depending on the location of the given perturbation, the resulting byte sequence was either a dead-code or API call instruction. The authors conducted a theoretical examination of the above-mentioned modifications to confirm that their modifications preserve functionality. Their introduced perturbations reduced the accuracy of several ML detectors by up to 94%.

[1] Description of GBDT and MalConv classifier can be found in Sect. 4.2.

3.3 Other methods

A generative adversarial network (GAN) called MalGan was proposed in [28] as a method of generating AEs. The MalGan model was trained to mislead a deep neural network, serving as a substitute detector, by modifying feature vectors representing extracted API calls from malware files. Their results demonstrate high attack transferability between the substitute and target models, with near-perfect evasion against the random forest, decision tree, and linear regression algorithms. While the MalGan model was capable of creating highly evasive adversarial feature vectors, the authors did not provide a method that would convert the generated feature vectors into real-word EXEs to produce functioning AEs.

One of the few works that addresses the data poisoning issue is [34] by Chen et al. Even though this work is centered on the Android operating system, the results could also apply to Windows systems. The authors saw up to 30% drops in accuracy after injecting their data while targeting pure ML models. As a defense mechanism, they introduced a camouflage detector that detects suspicious samples inside the training dataset and boosts the detector's accuracy by at least 15%.

Ebrahimi et al. proposed a generative sequence-to-sequence language model in the form of a recurrent neural network [29]. This network was trained on benign binaries to generate adversarial benign bytes. These benign bytes were subsequently appended to malware EXEs to produce adversarial malware instances. The authors achieved an average evasion rate of 73.24% against the MalConv classifier. Their findings suggest that increasing padding size increases the evasion rate but with diminishing returns as the value grows. The authors conducted a behavior analysis to validate that the functionality of malicious EXEs did not change after appending the generated benign bytes.

In [30], Demetrio et al. presented a black-box attack named GAMMA. The GAMMA attack tackles the problem of creating AEs as an optimization problem, with the main requirement being maximum evasion and minimal inserted content size. The optimization problem was solved using a genetic algorithm that used the traditional selection process, cross-over, and mutation. These operations are applied to vectors representing modifications of EXEs by injecting benign content. The best solution vector is applied to the original malware file to create AE. In the training phase, the authors targeted the GBDT and MalConv classifiers to later attack real-world AVs hosted on the VirusTotal website, successfully bypassing 12 out of 70 detectors on average.

4 Adversarial malware generator

We introduce a complete framework for generating adversarial malware examples called AMG (Adversarial Malware Generator). AMG consists of a tested PE file modifier, which can be easily expended with additional modifications, an environment in the Open AI Gym format [31] working with raw binary files, and a set of optimized reinforcement learning agents ready to use.

Our approach falls into the category of evasive black-box attacks. In other words, we are performing an adversarial attack to mislead the target model (e.g., antivirus) to classify malware samples as benign. Our objective is to execute small modifications on PE files that do not alter the original functionality but can make them undetectable to the antivirus. Our attack targets static malware analysis, where the detector makes decisions without examining the EXE's behavior. We set our adversarial attack in a black-box scenario where only the target classifier's hard predictions (malware/benign) are known to the attacker, as this is the most difficult scenario for the attacker. In addition, we believe that using the black-box approach limits the likelihood of our attack being successful only against a specific classifier and consequently increases the potential of transferability to other detectors. However, additional research in this field should be conducted in the future.

In our work, we modified the existing framework called gym-malware by Anderson et al. [20], which provides an environment for training reinforcement learning agents on binary samples. We rewrote most parts because the existing code did not meet our vision and goals. In particular, they used the `LIEF` [35] library for modifying PE files, whereas we used the `pefile` [36] Python library. We found that the `LIEF` library can make unnecessary changes to the original binary and that their modifications did not retain the same functionality as is shown later in Sect. 5.2. In addition, in their training setup, the agent is presented with an observation space that coincides with the feature space of the target classifier. As such, it cannot be classified as a pure black-box setup. Furthermore, we think that using the feature space of the target classifier can detriment the transferability of trained agents to other detectors. For this reason, we used a different observation space that is not used by any of the classifiers we targeted. Nonetheless, the work by Anderson et al. [20] is a key stepping stone for future research as it is one of the first complete frameworks for deploying RL agents for adversarial malware example generation.

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

4.1 PE file modifications

For implementing PE file modifications, we used the `pefile` [36] Python library. This library provides a simple interface for accessing all parts of the PE file format, such as file and optional header fields or individual sections. The description of the PE file format can be found in Sect. 2.3. We implemented various modifications of the binary files, all obeying the structure of the PE file format. While we had taken inspiration from state-of-the-art related works, such as the gym-malware mentioned above, we also introduced new modifications. In total, we implemented ten modifications, which are described below:

- *Break CheckSum*: Set the `CheckSum` field from the optional header to zero.
- *Append to overlay*: Append a random benign content to the end of the file.
- *Remove debug*: Clear the debug entry in the list of data directories and remove the respective debug information from the file.
- *Remove certificate*: Clear the security entry in the list of data directories and remove the certificate data from the file.
- *Add new section*: Add a new section to the PE file if possible. Firstly, it is necessary to check if the file has enough free space between the last section header and the beginning of section data (at least 40 bytes). If so, we can increase the file size and add a new section header and data. To preserve the original PE file structure as much as possible, we also move the old overlay data and, if present, redirect the security data directory to the new address.
- *Append to section*: Append benign content to one of the existing sections if possible. First, we need to find a section with the possibility of adding extra content, i.e., the virtual size of the section is greater than its raw size. If we encounter one, we fill the empty space with benign content.
- *Rename section*: Choose one section at random and rename it to one of the section names commonly used in benign files.
- *Increase TimeDateStamp*: Increase the value of `TimeDateStamp` in the COFF file header by 500 days.[2]
- *Decrease TimeDateStamp*: Decrease the value of `TimeDateStamp` in the COFF file header by 500 days[2].
- *Append new import*: Add a new section to the PE file with import data if possible. This process is similar to the preceding add new section modification, with the only change being that the section content is not random benign content but import data. If already present, we append a new entry (randomly chosen DLL) to the IDT. Then, we prepare entries for the imported functions in the IAT and ILT and store these tables in a newly added section. Finally, we modify the import data directory to point to the updated IDT.

4.1.1 Validity of PE file modifications

We believe that preserving the original functionality (i.e., validity) of executable binary files is a critical part of generating adversarial malware samples. Without emphasizing this criterion, we cannot guarantee that the resulting AE will still be a working executable with the same functionality as the original file. We have found that more than simply checking the syntax of the PE file format is needed to maintain functionality, so we designed the following protocol to test the validity of PE file modifications.

To ensure that the functionality of an executable after adversarial perturbations is as close as possible to the original file behavior, we used a Cuckoo Sandbox [37]. Cuckoo Sandbox is an open-source automated malware analysis tool that can run malicious files and examine their behavior. Even though it is predominantly intended for malware analysis, we also used it to analyze benign files as it provides behavioral analysis, which we utilized to track any changes in the functionality of executables. We decided to use benign files instead of malware EXEs for testing the modifications because malware authors can insert checks into their programs that monitor whether their malware is running in a sandbox environment and change its behavior accordingly [6, 7]. Consequently, by using benign files, we limit the possibility of artificial activity of the tested binaries, and thus, we can better analyze the reported behavior.

In contrast with other approaches [22, 23], we do not verify the functionality preservation of generated AEs, but we propose validating each modification individually before the generation process. Therefore, our approach is more time-efficient as it does not require discarding nonfunctional AEs during or at the end of the generation procedure.

For our method, we selected a set of benign EXEs $\mathcal{D}$ that was executable in the sandbox environment and studied their respective behavior reports. Namely, we looked into three features found in the Cuckoo analysis report: signatures, API calls, and processes:

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

- *Signatures*: Predefined patterns that are used to compare with the examined file. They are used predominately for malware detection to cluster malware into their respective families. Nevertheless, they can also classify types of actions, such as file open/write or access to system files, which also occur with benign files.
- *API calls*: Function calls by a program to external libraries during program execution.
- *Processes*: Main process and sub-processes started by a program.

To combat the variability of results reported by the sandbox environment, we conducted three testing rounds and considered the feature matched if it got at least 95% agreement between rounds. We selected the value of 95% to allow a small margin of error and to get all unmodified files reliably matched.

The set of untampered benign files $\mathcal{D}$ is used as a control dataset to test whether the functionality of the PE file has changed after the modification M. Firstly, the modification M is applied on each file $f \in \mathcal{D}$, creating a dataset of modified files $f^M \in \mathcal{D}^M$. Next, the modified dataset $\mathcal{D}^M$ is compared with the unmodified dataset $\mathcal{D}$ according to the same three features we mentioned earlier by performing three rounds of Cuckoo analysis. We consider the modification a failure if the modified file cannot be run in the Cuckoo Sandbox. If the file executes successfully, we compare the three generated test analysis reports with all three control reports. We look at each feature individually, matching it with each control file. The feature is considered matched if it has an agreement of at least 95% with one of the control files. Overall, the modified file is considered successfully modified (i.e., the original functionality has been preserved) if it matches at least two of its features with control reports. A more detailed pseudo-code on how we evaluate each modified file f^M is shown in Appendix A (Algorithm 1).

By testing only a single modification M rather than resulting AEs spanning multiple changes, we can better focus on each modification and trace potential errors. In addition, our protocol is time-efficient because the evaluation can be performed before generating evasive AEs.

4.2 Malware environment

As mentioned in Sect. 2.2, RL algorithms are based on learning through feedback provided by the environment. We worked with a commonly used environment format developed by the OpenAI company called Gym [31]. The `Gym` is an open-source Python library equipped with a standardized API for agent-environment interaction. The source codes of our implementation are available in the GitHub repository.[3]

A critical part of the environment is the target classifier, as each action is rewarded with respect to its predictions. We studied two ML classifiers, MalConv and GBDT, both publicly available on GitHub with pre-trained configurations.[4] MalConv is a deep convolutional neural network that does not require complex feature extraction procedures because it uses the entire EXE (truncated to 2,000,000 bytes) as an input feature vector [33]. On the other hand, GBDT is a gradient-boosted decision tree trained using the LightGBM framework that requires converting the input executable to an array of 2,381 float numbers [32]. Note that we did not directly target the MalConv classifier but only used it to explore whether adversarial attacks can be transferred between ML classifiers.

4.3 Reinforcement learning agents

In total, we experimented with three RL agents: deep Q-network (DQN), vanilla policy gradient (PG), and proximal policy optimization (PPO). We chose these reinforcement learning algorithms because they are well-known in the reinforcement learning community and represent both on-policy and off-policy approaches. A more detailed description of these algorithms can be found in the previous Sect. 2.2.1. We made use of the implementations that were offered by the `Ray RLLib` [38] reinforcement learning library.

5 Evaluation

In this section, we first describe our experimental setup. Secondly, we evaluate the validity of various PE file modifiers. Next, we present how we optimized individual RL algorithms to achieve the highest possible evasion rate against the GBDT target classifier. Additionally, we evaluate how generated AEs transfer to other malware detectors. In real-world circumstances, the transferability of AEs between detectors is paramount, as the victim's defenses can be unknown to the attacker. Finally, we discuss the results and propose ideas for future work. Note that the numbers in bold represent the highest evasion rate and the lowest size increase for the respective columns of Tables 3, 4, 5, and 6.

5.1 Setup

Datasets: We use two datasets. A dataset of benign binaries, including more than 4,000 executables, was scrapped from the fresh Windows 10 installation. These benign files are only used for testing the preservation of functionality after modification, as mentioned in Sect. 4.1.1. Second, a dataset of malware files was obtained from the VirusShare[5] repository. For training, we used 5,000 malware files, from which

[3] https://github.com/matouskozak/AMG.

[4] https://github.com/endgameinc/malware_evasion_competition.

[5] https://virusshare.com/.

we selected 4,000 to train the models and 1,000 to validate the training progress. An additional 2,000 malicious binaries were used as a test set for the final evaluation.

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

$$evasion\ rate = \frac{\#\, missclassified}{total} \cdot 100\% \quad (3)$$

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

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

5.2 Evaluation of the preservation of functionality

We used a set of 100 benign EXEs, complying with the requirements introduced in Sect. 4.1.1, to evaluate our PE file modifications described in Sect. 4.1, and to compare them with several PE file modifiers from well-known frameworks for generating adversarial malware examples. Namely, we tested gym-malware,[6] Pesidious[7] and MAB-Malware[8] generators. Both gym-malware and Pesidous use the `LIEF` library for modifying binaries, whereas MAB-Malware, the same as our approach, uses the `pefile` library. Additionally, Pesidous uses the `PE Bliss` [39] C++ library for rebuilding PE files. We chose these frameworks because they are all publicly available on GitHub and share the reinforcement learning approach with our work.

We evaluated each PE file modification in an isolated setting, i.e., in each run of Algorithm 1, the input file was perturbed by a single modification. We present the results of functionality preservation testing in Table 2, where the first column represents different PE modifications, and the following ones represent the number of valid files for PE modifiers from the respective frameworks. The symbol ✕ denotes that the operation was not implemented by the framework. Apart from the modifications mentioned in Table 2, authors of MAB-Malware also implemented code randomization operation. However, we could not reproduce the code locally for our dataset, so we did not include it in our testing.

The implemented append new import modification performed significantly worse than other AMG perturbations, possibly corrupting the underlying functionality of modified executables. However, we can see that the AMG modifications equaled or surpassed all other tested frameworks. In contrast, the gym-malware framework recorded the worst results with some of the modifications, such as adding a new section or appending to a section, created valid binaries in only 4 and 8 cases, respectively. Overall, we can see that MAB-Malware and AMG, the PE file modifiers that use the `pefile` library, better preserve the original functionality than the PE modifiers using the `LIEF` library, such as gym-malware and Pesidious.

5.3 Generating adversarial malware examples

In the following experiment, we focus on optimizing our generator of malicious AEs (AMG) against the GBDT classifier. We define the following procedure used for each RL algorithm. The first step is finding the optimal maximal number of modifications for the RL algorithm. For this part of the experiment, we leave the agent's parameters at their default settings as set in the `Ray RLLib`. The range we test is between 5 and 200 modifications, and we choose the optimal value based on two criteria. Firstly, we try to maximize the evasion rate achieved by the agent, and secondly, we try to minimize the increased size of evasive AEs.

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

Subsequently, we introduce our testing dataset, which is presented to the final RL agent. The obtained results are then used to compare different RL algorithms and to verify the success of the training stage. The complete overview of our optimization and evaluation workflow and how we used our dataset can be found in Fig. 3.

5.3.1 Optimization and selection of RL agents

Firstly, we followed the optimization process described earlier to tune the maximum number of steps, learning rate, and discount rate hyperparameters. Based on the results achieved by the respective RL algorithms on the validation set, we selected the following configurations listed in Table 3. Note that the size increase column refers to the average increase in file size of generated AEs.

[6] https://github.com/endgameinc/gym-malware.

[7] https://github.com/CyberForce/Pesidious.

[8] https://github.com/weisong-ucr/MAB-malware.

Table 2 Number of valid files after modification out of a total of 100

Action	Gym-malware	Pesidious	MAB-malware	**AMG**
Break checksum	89	×	100	100
Create new entry point	17	×	×	×
Append new import	20	42	×	66
Overlay append	100	99	100	100
Remove debug	90	×	100	100
Remove certificate	22	×	90	91
Add new section	4	85	75	98
Append to section	8	×	99	99
Rename section	89	89	99	100
upx pack	73	×	×	×
upx unpack	100	×	×	×
Increase TimeDateStamp	×	×	×	100
Decrease TimeDateStamp	×	×	×	100

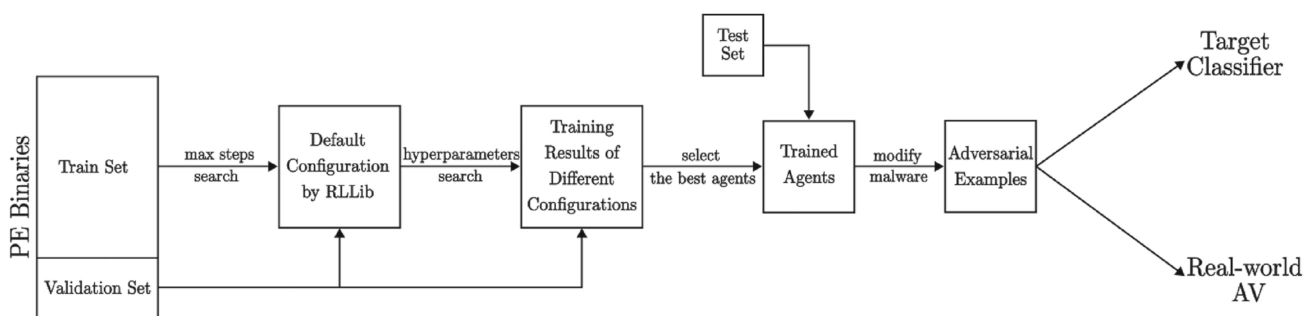

Fig. 3 Workflow of our training and testing procedure for generating adversarial malware examples

Table 3 The best configuration of each tested RL algorithm and their results against the GBDT classifier on the validation set

Agent	Max steps	α	γ	Evasion rate (%)	Size increase (%)
DQN	50	0.01	0.5	79.93	4.64
PG	20	0.01	0.75	69.33	**3.24**
PPO	50	0.0001	0.5	**90.38**	9.2

The values reported show that PPO outperformed the other RL algorithm tested, although it was offset by a 9.2% size increase. Overall, the PG method performed the poorest, as it did not exceed 70% evasion rate, but it increased the resulting AEs the least, by only 3.24% on average.

5.3.2 Evaluation of RL agents

Following that, we used our test dataset to evaluate the best configuration of each RL algorithm from Table 3. Furthermore, we included a so-called random agent in the following results. This agent represents a repeated random application of the modifications described in Sect. 4.1 until the GBDT classifier is bypassed or 50 alterations are reached. Including the random agent allows us to better understand the learning capabilities of the trained RL algorithms.

Table 4 Results of the best configuration of each tested RL algorithm and random agent against the GBDT classifier on the test set

Agent	Evasion rate (%)	Size increase (%)
DQN	52.72	4.63
PG	46.75	3.11
PPO	**53.84**	3.57
Random	36.88	**1.07**

The test results are shown in Table 4. We immediately see significant performance decreases for all agents compared to the validation set, suggesting possible overfitting. Overfitting occurs when an ML model predicts more accurately on data it used for training than on unseen examples. This may be due to a training dataset that does not represent real-world distribution, excessive training of the model, or other factors

Table 5 Transferability of adversarial attacks targeted against GBDT to MalConv

	GBDT	MalConv
MAB-Malware	**76.12**	**60.1**
AMG-PPO	53.84	11.41
AMG-random	36.88	7.65

[40]. However, the agents still achieved decent results by overcoming the GBDT detector in approximately half of the cases. The trend is similar to what we saw on the validation set, with PPO recording the highest evasion rate of 53.84% and the PG agent recording the lowest. The measured results represent an improvement of up to 17% over the random agent, which, on the other hand, had the lowest average size increase of 1.07%. To summarize our results, the overall best RL algorithm against the GBDT classifier is PPO with $\gamma = 0.5$ and $\alpha = 0.0001$, striking the highest evasion rate while maintaining a reasonable size increase of generated AEs.

5.4 Transferability of adversarial attack

In the previous experiment, we considered the GBDT classifier as our target model. In the final evaluation phase, we test the transferability of adversarial attacks between GBDT and other malware detectors. We included the MAB-Malware generator in our transferability experiment as it is a state-of-the-art model based on RL, obtained a similar validity of modified files (Table 2) as our AMG framework and targets the same GBDT classifier.

5.4.1 Transferability to the MalConv classifier

To begin, we examine the transferability of adversarial attacks from GBDT to the MalConv classifier. We employed the best-trained AMG agent (PPO with $\gamma = 0.5$ and $\alpha = 0.0001$), the random agent with the AMG modifications, and MAB-Malware to generate AEs against the GBDT detector and then test them against the MalConv classifier.

The recorded evasion rates against the GBDT and MalConv classifiers are listed in Table 5. From these results, we can conclude that MAB-Malware outperforms our AMG agents in terms of GBDT evasion as well as transferability to the MalConv classifier. The MAB-Malware recorded evasion rates of 76.12% and 60.1% against the GBDT and MalConv models, respectively. In comparison, the AMG-PPO agent struggled to transfer the performance recorded against GBDT to MalConv, with a considerable decrease in performance from 53.84% down to 11.41%. Similarly, AEs generated by the AMG-random agent bypassed the MalConv classifier only in 7.65% of cases.

5.4.2 Transferability to real-world antiviruses

In the final stage of our experiments, we compare the performance of the above-mentioned generators of adversarial malware against leading AV programs. The transferability of generated AEs from lightweight ML detectors to commercially deployed AVs is an essential feature that AE generators should have, as developing AEs against AVs is time-consuming and could reveal the attacker.

We conduct this assessment on a selection of antivirus programs based on the March 2023 antivirus comparative study by the Austrian AV testing laboratory AV-Comparatives [41]. We used the VirusTotal[9] website as a substitute for local instances of selected AV engines.

The recorded evasion rates of generated AEs against AV detectors are presented in Table 6. We anonymized the recorded results to minimize the possible risk of misuse of our work and to comply with the VirusTotal policies.[10] At first glance, we can see that AMG-PPO and MAB-Malware decreased their performance to 2.31% and 2.61%, respectively. This represents a significant drop in evasion rates compared to the original ones recorded against the targeted GBDT classifier. This decrease could indicate that employing the GBDT classifier as a substitute model for generating malicious AEs does not lead to successful evasion against real-world AV programs and that better surrogate models should be utilized.

The AMG-random agent, on the other hand, achieved surprising results against all top-tier AV products, outperforming both the trained PPO agent and MAB-Malware framework in each scenario. While the random application of our AMG modifications did not produce satisfactory results against pure ML models such as GBDT and MalConv, it did produce evasive AEs against commercially available AVs. The random AMG agent recorded evasion rates ranging from 2.15% to 34.48%, with an average of 11.65% among the tested AVs.

5.5 Discussion and future work

The results in Sect. 5.4.1 show that our model is outperformed by MAB-Malware in evading GBDT, a task both systems were designed for. This could be caused by numerous reasons. Firstly, while our modifications better preserve the original functionality, this could also mean that the perturbation area is smaller, thus affecting the predictions by GBDT or MalConv less. Secondly, it is possible that even though the MAB-Malware model is trained better against ML models, it does not generalize well to models not based

9 https://www.virustotal.com/.

10 https://blog.virustotal.com/2012/08/av-comparative-analyses-marketing-and.html.

Table 6 Transferability of adversarial attacks targeted against GBDT to AVs

	AV-1	AV-2	AV-3	AV-4	AV-5	AV-6	AV-7	AV-8	average
MAB-Malware	1.22	0.41	1.66	1.9	1.53	3.0	7.76	3.38	2.61
AMG-PPO	2.39	0.41	2.84	2.75	1.79	2.41	3.96	1.9	2.31
AMG-random	**9.37**	**2.15**	**9.86**	**11.74**	**9.22**	**12.88**	**34.48**	**3.54**	**11.65**

purely on ML (e.g., antivirus engines). This is supported by our findings in Sect. 5.4.2, where our PPO agent and MAB-Malware perform almost the same after transferring attacks to real-world AVs.

The surprising results of the AMG-random agent against leading AVs are in contrast with its poor performance against ML-based detectors GBDT and MalConv. One possible explanation could be that GBDT and MalConv are vulnerable to specific modifications, whereas commercially available AVs rely on a broader set of features. For that reason, random application of functionality-preserving modification has a higher chance of exploiting the vulnerabilities of real-world AVs.

While our goal was to implement a functionality-preserving adversarial attack, the next natural step would be to introduce a defense mechanism that could be incorporated into existing detectors. Retraining with generated AEs or a self-contained AE classifier could be used as a defensive technique, but more research must be done in this area.

Additionally, our proposed approach to generating AEs still has room for improvement. To begin, one of our implemented PE modifications (append new import) did not sufficiently preserve the validity of modified files. More improvements should be made before deploying this operation to other projects. Following that, different modifications (e.g., obfuscation) or target classifiers could be introduced to improve the AMG framework further.

The experimental part could be enhanced by evaluating AEs directly on real-world AVs instead of using VirusTotal API. This could help us understand the differences between reported results from individual AVs and propose ideas for improvements.

One of the strong points of our evaluation of the modification validity is its time efficiency, which stems from the usage of hard-coded adversarial perturbation found in RL-based generators of adversarial malware. While predefined modifications are common nowadays, learned perturbations could prevail in the future. As a result, a reliable evaluation of the functionality of PE files should be developed, which could be incorporated into the generation of AEs.

6 Conclusion

In this paper, we presented a black-box evasion attack using reinforcement learning algorithms in the space of PE binaries. To achieve that, we implemented an interactive environment in the OpenAI Gym format for training RL agents. The environment includes a PE file modifier with tested modifications that maximize the preservation of original functionality. Our PE modifier registered the highest validity of modified binaries compared to modifiers from frameworks such as gym-malware or MAB-Malware.

Further, we collected a dataset of 7,000 Windows malware EXEs and experimented with three RL algorithms: DQN, PG, and PPO. We optimized the maximum number of modifications and various hyperparameters for each RL agent. Based on the recorded results, the PPO algorithm with $\gamma = 0.5$ and $\alpha = 0.0001$ achieved the highest evasion rate of 53.84% against the GBDT classifier while increasing the AE size by 3.57% on average. Furthermore, we tested the transferability to other malware detectors where our PPO agent achieved an evasion rate of 11.41% against MalConv and an average evasion rate of 2.31% against leading AV engines.

We compared these results to a random agent using the same set of PE modifications as our trained PPO agent and to MAB-Malware, a state-of-the-art generator of malicious AEs. MAB-Malware bypassed the GBDT and MalConv in 76.12% and 60.1% of cases, respectively, outperforming both the random and PPO agents. When transferring the generated AEs to real-world AVs, MAB-Malware achieved an average evasion rate of 2.61%, which is comparable to our trained PPO agent. The random agent, on the other hand, recorded evasion rates ranging from 2.15 to 34.48%, with an average of 11.65%, significantly outperforming both MAB-Malware and our trained PPO agent. These findings show that utilizing GBDT as a substitute model to generate AEs does not result in evasive AEs against real-world AVs and that even leading AVs are vulnerable to well-crafted, yet randomly applied, adversarial perturbations.

This work provides a solid implementation of a reinforcement learning generator working at the level of binary samples while generating functional adversarial malware examples. Additionally, our modifications, agents, and environment setup can be easily extended for future improvements and are freely available to the public.

Acknowledgements This work was supported by the Grant Agency of the Czech Technical University in Prague, grant No. SGS23/211/OHK3/3T/18 funded by the MEYS of the Czech Republic and by the OP VVV MEYS funded project CZ.02.1.01/0.0/0.0/16 019/0000765 "Research Center for Informatics".

Funding Open access publishing supported by the National Technical Library in Prague.

Declarations

Conflict of interest The authors have no relevant financial or non-financial interests to disclose.

Open Access This article is licensed under a Creative Commons Attribution 4.0 International License, which permits use, sharing, adaptation, distribution and reproduction in any medium or format, as long as you give appropriate credit to the original author(s) and the source, provide a link to the Creative Commons licence, and indicate if changes were made. The images or other third party material in this article are included in the article's Creative Commons licence, unless indicated otherwise in a credit line to the material. If material is not included in the article's Creative Commons licence and your intended use is not permitted by statutory regulation or exceeds the permitted use, you will need to obtain permission directly from the copyright holder. To view a copy of this licence, visit http://creativecommons.org/licenses/by/4.0/.

Appendix A

Algorithm 1 Evaluation of file validity after modification

1: $f \leftarrow$ original file
2: $f^M \leftarrow$ modified file
3: $CRs, TRs \leftarrow$ empty lists

4: **for** $i \leftarrow 1$ to 3 **do** ▷ Create lists of control (CRs) and test (TRs) reports
5: $CRs_i \leftarrow$ Cuckoo-analysis(f)
6: $TRs_i \leftarrow$ Cuckoo-analysis(f^M)
7: **end for**

8: $c \leftarrow 0$ ▷ Number of matched features
9: **for** $TR \in TRs$ **do**
10: **if** TR is failure **then return** FAILURE
11: **end if**
12: **for** $feature \in$ [signatures, API calls, processes] **do**
13: **for** $CR \in CRs$ **do**
14: **if** match-feature($feature, TR, CR$) **then**
15: $c \leftarrow c + 1$ ▷ More than 95% agreement between TR and CR
16: **break**
17: **end if**
18: **end for**
19: **end for**
20: **end for**

21: **if** $c >= 2$ **then return** SUCCESS ▷ At least two features are matched
22: **else return** FAILURE
23: **end if**

References

1. Institute, A.-T.: Malware statistics & trends report: AV-TEST (2022). https://www.av-test.org/en/statistics/malware/
2. Sophos: Sophos Threat Report (2022). https://www.sophos.com/en-us/content/security-threat-report
3. Ucci, D., Aniello, L., Baldoni, R.: Survey of machine learning techniques for malware analysis. Comput. Secur. **81**, 123–147 (2019). https://doi.org/10.1016/j.cose.2018.11.001
4. Papernot, N., McDaniel, P., Jha, S., Fredrikson, M., Celik, Z.B., Swami, A.: The limitations of deep learning in adversarial settings. In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P), pp. 372–387 (2016). https://doi.org/10.1109/EuroSP.2016.36. IEEE
5. Damodaran, A., Troia, F.D., Visaggio, C.A., Austin, T.H., Stamp, M.: A comparison of static, dynamic, and hybrid analysis for malware detection. J. Comput. Virol. Hack. Tech. **13**, 1–12 (2017). https://doi.org/10.1007/s11416-015-0261-z
6. Erko, A.: Malware sandbox evasion: techniques, principles and solutions (2022). https://www.apriorit.com/dev-blog/545-sandbox-evading-malware
7. Yuceel, H.C.: Virtualization/sandbox evasion—how attackers avoid malware analysis. Picus Güvenlik A.Ş (2022). https://www.picussecurity.com/resource/virtualization/sandbox-evasion-how-attackers-avoid-malware-analysis
8. Kerckhoffs, A.: La cryptographie militaire. J. Sci. Militaires **9**(4), 5–38 (1883)
9. Huang, L., Joseph, A.D., Nelson, B., Rubinstein, B.I.P., Tygar, J.D.: Adversarial machine learning. In: Proceedings of the 4th ACM Workshop on Security and Artificial Intelligence. AISec '11, pp. 43–58. Association for Computing Machinery, New York, NY, USA (2011). https://doi.org/10.1145/2046684.2046692
10. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press (2018). https://doi.org/10.1016/S1364-6613(99)01331-5
11. Watkins, C.J.C.H.: Learning from delayed rewards. King's College, Cambridge United Kingdom (1989). https://www.researchgate.net/publication/33784417_Learning_From_Delayed_Rewards
12. Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., Riedmiller, M.: Playing atari with deep reinforcement learning. CoRR arXiv:1312.5602 (2013). https://doi.org/10.48550/ARXIV.1312.5602
13. Mnih, V., Kavukcuoglu, K., Silver, D., Rusu, A.A., Veness, J., Bellemare, M.G., Graves, A., Riedmiller, M., Fidjeland, A.K., Ostrovski, G., et al.: Human-level control through deep reinforcement learning. Nature **518**(7540), 529–533 (2015). https://doi.org/10.1038/nature14236
14. Sutton, R.S., McAllester, D., Singh, S., Mansour, Y.: Policy gradient methods for reinforcement learning with function approximation. In: Solla, S., Leen, T., Müller, K. (eds.) Proceedings of the 12th International Conference on Neural Information Processing Systems. NIPS'99, vol. 12, pp. 1057–1063. MIT Press, Cambridge, MA, USA (1999). https://proceedings.neurips.cc/paper/1999/file/464d828b85b0bed98e80ade0a5c43b0f-Paper.pdf
15. Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O.: Proximal policy optimization algorithms. CoRR arXiv:1707.06347 (2017). https://doi.org/10.48550/arXiv.1707.06347
16. Kowalczyk, K.: Portable Executable File Format (2018). https://blog.kowalczyk.info/articles/pefileformat.html
17. Karl Bridge, M.: PE Format - Win32 apps (2019). https://docs.microsoft.com/en-us/windows/win32/debug/pe-format
18. Pietrek, M.: An In-Depth Look into the Win32 Portable Executable File Format (2008). https://docs.microsoft.com/en-us/previous-versions/bb985992(v=msdn.10)?redirectedfrom=MSDN

19. Goodfellow, I.J., Shlens, J., Szegedy, C.: Explaining and harnessing adversarial examples. In: 3rd International Conference on Learning Representations (ICLR) (2015). https://doi.org/10.48550/ARXIV.1412.6572. arxiv:1412.6572
20. Anderson, H.S., Kharkar, A., Filar, B., Evans, D., Roth, P.: Learning to evade static pe machine learning malware models via reinforcement learning. CoRR arXiv:1801.08917 (2018). https://doi.org/10.48550/arXiv.1801.08917
21. Fang, Y., Zeng, Y., Li, B., Liu, L., Zhang, L.: Deepdetectnet vs rlattacknet: an adversarial method to improve deep learning-based static malware detection model. Plos one **15**(4), 0231626 (2020). https://doi.org/10.1371/journal.pone.0231626
22. Song, W., Li, X., Afroz, S., Garg, D., Kuznetsov, D., Yin, H.: Mab-malware: a reinforcement learning framework for attacking static malware classifiers. arXiv preprint arXiv:2003.03100 (2020). https://doi.org/10.48550/ARXIV.2003.03100
23. Quertier, T., Marais, B., Morucci, S., Fournel, B.: Merlin–malware evasion with reinforcement learning. arXiv preprint (2022). https://doi.org/10.48550/ARXIV.2203.12980 arXiv:2203.12980
24. Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F.: Adversarial malware binaries: Evading deep learning for malware detection in executables. In: 2018 26th European Signal Processing Conference (EUSIPCO), pp. 533–537 (2018). https://doi.org/10.23919/EUSIPCO.2018.8553214 . IEEE. arXiv:1803.04173
25. Kreuk, F., Barak, A., Aviv-Reuven, S., Baruch, M., Pinkas, B., Keshet, J.: Deceiving end-to-end deep learning malware detectors using adversarial examples. CoRR arXiv:1802.04528 (2019) https://doi.org/10.48550/ARXIV.1802.04528
26. Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A.: Explaining vulnerabilities of deep learning to adversarial malware binaries. arXiv:1901.03583 (2019) https://doi.org/10.48550/ARXIV.1901.03583
27. Yang, C., Xu, J., Liang, S., Wu, Y., Wen, Y., Zhang, B., Meng, D.: Deepmal: maliciousness-preserving adversarial instruction learning against static malware detection. Cybersecurity **4**(1), 1–14 (2021). https://doi.org/10.1186/s42400-021-00079-5
28. Hu, W., Tan, Y.: Generating adversarial malware examples for black-box attacks based on gan. CoRR arXiv:1702.05983 (2017). https://doi.org/10.48550/ARXIV.1702.05983
29. Ebrahimi, M., Zhang, N., Hu, J., Raza, M.T., Chen, H.: Binary black-box evasion attacks against deep learning-based static malware detectors with adversarial byte-level language model. CoRR arXiv:2012.07994 (2020). https://doi.org/10.48550/ARXIV.2012.07994
30. Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A.: Functionality-preserving black-box optimization of adversarial windows malware. IEEE Trans. Inf. Forensics Secur. **16**, 3469–3478 (2021). https://doi.org/10.1109/TIFS.2021.3082330
31. Brockman, G., Cheung, V., Pettersson, L., Schneider, J., Schulman, J., Tang, J., Zaremba, W.: Openai gym. CoRR arXiv:1606.01540. https://doi.org/10.48550/ARXIV.1606.01540 (2016)
32. Anderson, H.S., Roth, P.: Ember: an open dataset for training static pe malware machine learning models. CoRR arXiv:1804.04637 (2018). https://doi.org/10.48550/ARXIV.1804.04637
33. Raff, E., Barker, J., Sylvester, J., Brandon, R., Catanzaro, B., Nicholas, C.: Malware detection by eating a whole exe (2017). https://doi.org/10.48550/ARXIV.1710.09435
34. Chen, S., Xue, M., Fan, L., Hao, S., Xu, L., Zhu, H., Li, B.: Automated poisoning attacks and defenses in malware detection systems: an adversarial machine learning approach. Comput. Secur. **73**, 326–344 (2018). https://doi.org/10.1016/j.cose.2017.11.007
35. Thomas, R.: LIEF—Library to Instrument Executable Formats (2017). https://lief.quarkslab.com/
36. Carrera, E.: Pefile (2017). https://github.com/erocarrera/pefile
37. Guarnieri, C.: Cuckoo Sandbox—Automated Malware Analysis (2012). https://cuckoosandbox.org/
38. Liang, E., Liaw, R., Nishihara, R., Moritz, P., Fox, R., Gonzalez, J., Goldberg, K., Stoica, I.: Ray rllib: A composable and scalable reinforcement learning library. CoRR arXiv:1712.09381 (2017). https://doi.org/10.48550/arXiv.1712.09381
39. rukaimi: PE Bliss, Cross-Platform Portable Executable C++ Library. GitHub (2012). https://github.com/BackupGGCode/portable-executable-library
40. IBM: what is overfitting? (2022). https://www.ibm.com/topics/overfitting
41. AV-Comparatives: Malware Protection Test March 2023 (2023). https://www.av-comparatives.org/tests/malware-protection-test-march-2023/

Publisher's Note Springer Nature remains neutral with regard to jurisdictional claims in published maps and institutional affiliations.