Computers & Security 121 (2022) 102869

Contents lists available at ScienceDirect

Computers & Security

journal homepage: www.elsevier.com/locate/cose

Reinforcement learning based adversarial malware example generation against black-box detectors

Fangtian Zhong[a], Pengfei Hu[b,*], Guoming Zhang[b], Hong Li[c], Xiuzhen Cheng[b,1]

[a] Department of Computer Science, The George Washington University, Washington DC, USA
[b] School of Computer Science and Technology, Shandong University, Qingdao, China
[c] Institute of Information Engineering, Chinese Academy of Sciences and School of Cyber Security, University of Chinese Academy of Sciences, Beijing, China

ARTICLE INFO

Article history:
Received 24 April 2022
Revised 11 July 2022
Accepted 6 August 2022
Available online 8 August 2022

Keywords:
Adversarial malware examples
Dynamic programming
Malware
Reinforcement learning
Temporal difference learning

ABSTRACT

Recent advances in machine learning offer attractive tools for sophisticated adversaries. An attacker could transform malware into its adversarial version but retain its malicious functionality by employing a dedicated perturbation method. These adversarial malware examples have demonstrated the effectiveness to bypass antivirus engines. However, recent works only leverage a single perturbation method to generate adversarial examples, which cannot defeat advanced detectors. In this paper, we propose a reinforcement learning-based framework called `MalInfo`, which could generate powerful adversarial malware examples to evade the third-party detectors via an adaptive selection of a perturbation path for each malware in our collected dataset with 1000 diverse malware. To cope with limited computation, `MalInfo` applies either dynamic programming or temporal difference learning to choose the optimal perturbation path where each path is formed by the combination of Obfusmal, Stealmal, and Hollowmal. We provide a proof-of-concept implementation and extensive evaluation of our framework. Both the detection rate and evasive rate have substantially been improved compared with the state-of-art research `MalFox` Zhong et al. (2021). To be specific, The average detection rates for dynamic programming and temporal difference learning are 23.2% (21.9% lower than `MalFox`) and 27.5% (7.4% lower than `MalFox`), respectively, and the average evasive rates are 65.8% (17.1% higher than `MalFox`) and 59.4% (5.7% higher than `MalFox`), respectively.

1. Introduction

Recently industrial and academic communities enthusiastically exploit machine learning techniques as useful tools to detect malicious executable files (malware). However, machine learning models have been proved to be vulnerable to adversarial attacks. For instance, in 2014, Szegedy *et al.* provided an approach to apply a human imperceptible perturbation to an image, which causes the network to misclassify it (Szegedy et al., 2014). Yakura *et al.* proposed a method to generate robust audio adversarial examples which could confuse a state-of-the-art speech recognition model (Yakura and Sakuma, 2019). The audio adversarial examples were generated by simulating the transformations caused by the over-the-air playback through introducing a band-pass filter, an impulse response, and white Gaussian noises. Evaluation and a listening experiment demonstrate that the generated adversarial examples can attack the recognition model without being noticed by humans, which indicates that audio adversarial examples may become a real physical threat.

Given the wide usage of machine learning techniques in malware detection, researchers embarked on generating adversarial malware examples to attack machine learning-based malware detectors. Kolosnjaji *et al.* studied a gradient-based algorithm by altering a few specific bytes at the end of each malware program to evade a deep neural network malware detector (Kolosnjaji et al., 2018a). Grosse *et al.* exploited an augmented adversarial crafting algorithm to dramatically mislead a state-of-the-art classifier for 63% of malware programs (Grosse et al., 2017). Hu *et al.* proposed a generative RNN-based approach to generate irrelevant API strings and inserted them into the original API sequence feature vectors, which can effectively bypass the RNN-based malware detectors (Hu and Tan, 2018).

To defeat the machine learning-based malware detectors (self-developed or open-source), various adversarial example crafting al-

* Corresponding author.
E-mail addresses: squareky_zhong@gwu.edu (F. Zhong), phu@sdu.edu.cn (P. Hu), guomingzhang@sdu.edu.cn (G. Zhang), lihong@iie.ac.cn (H. Li), xzcheng@sdu.edu.cn (X. Cheng).
[1] Fellow, IEEE

https://doi.org/10.1016/j.cose.2022.102869
0167-4048/

gorithms have been proposed to construct a highly effective attack. Most of them rely on a single perturbation method to generate adversarial examples. The perturbation methods proposed in Al-Dujaili et al. (2018); Demetrio et al. (2019); Grosse et al. (2017); Hu and Tan (2018); Kolosnjaji et al. (2018a); Suciu et al. (2019a); Yuan et al. (2020) can work well on their customized detectors but not commercial ones since these commercial detectors normally integrate signature-based methods and code similarity techniques. For instance, they generated the adversarial malware examples by appending random bytes to the end of the malware binary, attaching random bytes to regions of malware not mapped to memory, or modifying a few tens of bytes in the header of a malware file. Moreover, these approaches require good knowledge about the malware detectors and adopt simple evasive techniques, which renders them incapable under a black-box setting to generate a powerful adversarial malware example against a collection of practical antivirus products and online scan engines, *e.g.* VirusTotal (vir, 2020). In this paper, we aim to generate a powerful adversarial malware example for each malware to bypass the detection of third-party detectors under a complete black-box setting. There are three key challenges to designing such a generation scheme. First, since the ultimate purpose of malware is to launch an attack, any generated adversarial malware example should function indistinguishably from its original version. Second, as some scan engines, like VirusTotal, consist of a collection of malware detectors but provide no details of their implementations, the adversarial malware example should be able to evade them without any prior knowledge, *i.e.* under a black-box setting. Third, given a certain number of perturbation methods, the action space to generate adversarial malware examples should be largely traversed to ensure the optimum.

To address the aforementioned challenges, we propose a reinforcement learning-based architecture called `MalInfo`. `MalInfo` consists of three major components: Agent, PE Editor, and Detector. Agent leverages reinforcement learning algorithms to search for optimal action sequences that are employed to generate powerful adversarial malware examples against third-party detectors. With the optimal trace, `MalInfo` could guarantee a promising detection rate and evasive rate. PE Editor generates adversarial malware examples by altering the order of program execution that follows the action sequences output from Agent. An action sequence is the combination of three framework methods, *i.e.* Obfusmal, Stealmal, and Hollowmal (Zhong et al., 2021). PE Editor follows each action in the sequence one by one to pack a malware program into an adversarial malware example without changing its original functionality. We employ VirusTotal as Detector because it includes many practical antivirus products and online scan engines that are well-known among security professionals and public users. VirusTotal may involve more than 82 detection entities, and many of the entities come from famous tech companies, such as F-Secure, McAfee, 360, Tencent, Microsoft, etc.

`MalInfo` seeks optimal action sequences to transform malware programs into foxy ones which can evade the maximum number of malware detectors in VirusTotal. In PE Editor, we adopt three novel framework methods that can dramatically enhance the evasive ability of a malware program when converted into its adversarial versions. The reinforcement learning algorithm used by the agent embodies dynamic programming and temporal difference learning due to their superior performance than deep reinfocement learning with fully connected feed-forward neural network-based function approximators and convolutional neural network-based function approxiamtors. The reason behind this design lies in that the choice of the action sequence has no strict linear or non-linear relationship with the generated adversarial malware examples. The dynamic programming algorithm (DP) is suitable to seek optimal action sequence given a perfect model of the environment when the computation resources are sufficient, while the temporal difference algorithm could achieve the same effect as DP without requiring a perfect model of the environment when the computation resources are limited. We provide the implementation of `MalInfo` and conduct extensive experiments to evaluate its effectiveness against VirusTotal. The main contributions of this work include:

- To the best of our knowledge, `MalInfo` is the first work that seeks the optimal strategy to transform malware into its adversarial example.
- By exploring the optimal combination of three novel actions, `MalInfo` can generate powerful adversarial malware examples that can defeat the third-party black-box malware detectors with high probability without any prior knowledge of their underlying implementation details.
- `MalInfo` can efficiently find the optimal action sequence among a large combination space, even though the complete prior knowledge of the environment's dynamics is not specified due to the limited computing resources.
- We implement `MalInfo` and evaluate it on the dataset with 1000 diverse malware. The adversarial malware examples generated by `MalInfo` can significantly lower the detection rate. Besides, the average evasive rate surpasses 59%, which tells that the generated adversarial malware examples can largely sneak out detection as a regular user often installs only one or a few antivirus entities.

The rest of the paper is organized as follows. Section 2 provides the background knowledge of reinforcement learning and adversarial example. Section 3 introduces some related works about adversarial malware example generation techniques. Section 4 provides the overview of `MalInfo`. Section 5 presents the design and implementation detail of `MalInfo`. Section 6 evaluates the performance of `MalInfo`, and Section 7 concludes the paper with a discussion for future research.

2. Background

We provide a brief introduction to reinforcement learning and adversarial examples in this section.

2.1. Reinforcement learning

Reinforcement learning is one of the basic machine learning techniques, alongside supervised learning and unsupervised learning. It enables an agent to learn in an interactive environment to make a sequence of decisions to maximize its cumulative reward. There are two types of reinforcement learning mechanisms, i.e., model-based and model-free. Model-based reinforcement learning refers to a collection of algorithms making use of transition probability distributions and reward functions associated with a Markov decision process to deal with stochasticity, uncertainty, partial observability, and temporal abstraction in dynamics model learning and then use the learned dynamics model to derive a policy for planning (Moerland et al., 2020). On the contrary, model-free reinforcement learning does not use transition probability distributions and reward functions. Instead, it employs algorithms that can be regarded as explicit trial-and-error ones without any informed model of the environment in both learning and planning (Strehl et al., 2006).

Reinforcement learning differs from supervised learning in two aspects: it does not need a labeled dataset to be presented and it does not require sub-optimal actions to be explicitly corrected (Kaelbling et al., 1996). To tackle the challenges brought by distinctive problem sizes in terms of time and space, it focuses on finding a balance between the exploration of uncharted territory

and the exploitation of current knowledge. Reinforcement learning intends to capture the most essential aspect of the real problem faced by a learning agent that interacts with its environment over time to achieve a goal. The superiority of reinforcement learning has brought practical solutions to complex sequential decision-making problems.

2.2. Adversarial example

Adversarial examples are dedicated inputs to a machine learning model, which could cause it to make false predictions. Figuratively speaking, adversarial examples are like optical illusions for machines. Various adversarial examples have been developed with different purposes to mislead classifiers and detectors. By placing a picture over a stop sign, an attacker could induce a recognition software equipped on a self-driving vehicle to identify it as a parking prohibition sign, which could cause a serious vehicle collision (Morgulis et al., 2019). Intending to deceive the email recipient, malware authors crafted a spam email to camouflage itself with a normal one, which made the detector fail to classify the email as spam (Molnar, 2021). Recently, a machine learning-based scanner has been deployed in airports to detect weapons in suitcases. However, an attacker could craft a knife with dedicated features to bypass the scanner making the knife being identified as an umbrella (Molnar, 2021).

Perturbation is one of the most important techniques to generate adversarial examples by adding a small number of noises into the original inputs, which could deceive a victim system while remaining quasi-imperceptible for humans (Moosavi-Dezfooli et al., 2017). Goodfellow *et al.* proposed a non-iterative and hence fast gradient-sign method (FGSM) to add adversarial perturbations into images (Goodfellow et al., 2015). FGSM performs one step gradient update along the direction of the sign of gradient at each pixel of the image and then adds the update (perturbation) to the corresponding pixel in the original image, which finally generates an adversarial example. As a result, the generated adversarial example yields the highest increase in the linearized cost function for the categorization model. In Rozsa et al. (2016), Rozsa *et al.* developed the Fast Gradient Value method (FGVM), in which the sign of the gradient is replaced with the raw gradient. FGVM has no constraint on each pixel and can generate an adversarial example with a large local difference in the image.

3. Related work

The most widely-used techniques for adversarial malware example generation include *appending random data, appending goodware strings, changing binary headers*, and *embedding samples in a dropper*, which are summarized as follows.

Appending Random Data. Machine learning-based malware detectors developed on the basis of raw binary inputs are usually sensitive to changes in the inputs. This vulnerability has been exploited by adversaries to create malware that can bypass detection by appending random data to the malware binaries. For example, Ceschin *et al.* produced adversarial malware examples by repeatedly generating growing chunks of random data up to 5MB and inserting them into the code caves of malware files; as a result, the newly generated adversarial malware examples can successfully mislead Malcon, an end-to-end deep learning model that takes raw bytes of a file as inputs to determine its maliciousness (Ceschin et al., 2019). Fang *et al.* created a new empty section at the end of a malware file and filled that section with random binary bytes. At the same time, they also changed the corresponding fields in the newly added section table and PE File Header to guarantee the successful execution of the malware (Fang et al., 2019).

Appending Goodware Strings. Appending goodware strings is a technique that repeatedly retrieves strings presented at goodware files and appends them to malware binaries. It can significantly affect the performance of the models whose detection capability is based on the frequency or information entropy of the occurrences of special strings. Gorelik *et al.* reported that generating a new binary with goodware data appended to malware can yield an adversarial malware example that is capable of bypassing string-based antivirus engines, with the functionality of the malware remaining unchanged (Gorelik, 2020). Aghakhani *et al.* identified the routinely occurred benign sequences of strings in an online gaming program and injected them into a target malware binary, which tricks Cylance's AI-based anti-malware engine to identify that malware (such as WannaCry, Mimikatz, etc.) as benign (Aghakhani et al., 2020).

Changing Binary Headers. A binary header defines the structure of executable files or object files under a particular operating system. Modification of binary headers could typically increase the success rate to evade a malware detector that is capable of checking the static features at header fields. Demetrio *et al.* proposed a novel attack algorithm to generate an adversarial malware example by changing a few tens of bytes in the file header, which can effectively reveal the weaknesses of a convolutional neural network-based detector named MalConv (Demetrio et al., 2020; Suciu et al., 2019b); Castro *et al.* presented a gradient-based approach that can carefully generate valid adversarial executable files by modifying header information, leading to misclassification of the state-of-the-art detectors (Kolosnjaji et al., 2018b; Labaca-Castro et al., 2019).

Embedding Malware in a Dropper. Embedding malware in a dropper can make the samples properly execute. The dropper places a malware file in a newly added section and drops the malware into the runtime space after execution. This approach has been proven capable of bypassing detectors without breaking malware's execution. For example, Murali *et al.* created a dropper that attaches malware at its end to successfully evade detection by Virus Scanner (Murali et al., 2020). Ceschin *et al.* inserted a malware sample in the *Dr0p1t* dropper to effectively escape the detection by Non-Neg MalConv and LightGBM (Ceschin et al., 2019).

Summary. The adversarial malware example generation techniques mentioned above exploit different weaknesses in machine learning-based models. Specifically, appending random data or goodware strings as well as changing binary headers mainly exploit the weakness that these models are sensitive to feature modifications in malware while embedding malware in a dropper relies on a stealthy dodge to prevent malware from being detected.

In this paper, we present `MalInfo`, which utilizes both offline and online reinforcement learning algorithms to seek optimal action sequences for generating adversarial malware examples against VirusTotal. Previous studies (Demetrio et al., 2020; Gorelik, 2020; Murali et al., 2020) largely focus on the effectiveness of a single perturbation method to attack one specific machine learning-based detector. Hence they perform less effectively when going against more sophisticated anti-virus products and online scan engines in a black-box setting. As a comparison, `MalInfo` aspires to attack a collection of anti-virus products and online scan engines that are from well-known third parties instead of specific machine learning-based models, which makes it more practical.

4. MalInfo overview

The objective of `MalInfo` is to search for optimal action sequences that can be followed by PE Editor to generate adversarial malware examples. `MalInfo` consists of 3 components: *Agent, PE Editor*, and *Detector* as shown in Fig. 1. Agent produces an action sequence given an input malware program and a reinforce-

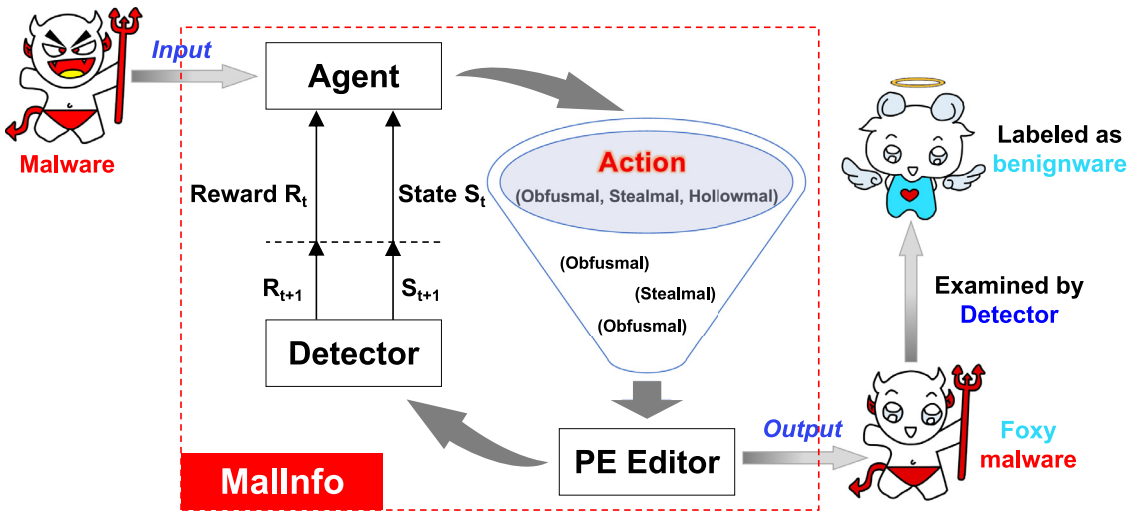

Fig. 1. Procedure of Generating Adversarial Malware Examples.

ment learning algorithm. PE Editor generates an adversarial malware example for the corresponding malware by following the action sequence produced by Agent. Detector is a group of antivirus products and online scan engines from third parties for malware detection. We intend to produce an adversarial malware example that could be recognized as benign by as many third-party malware detectors as possible, e.g., VirusTotal.

Algorithm 1 depicts the procedure of producing a powerful foxy malware, i.e., an adversarial malware example, by `MalInfo`. Assume that Agent has been well-trained (to be detailed later). Once a malware program is fed into the agent, it generates an optimal action sequence that could achieve the maximum cumulative reward (to be explained later). Then PE Editor follows the action sequence to edit the malware program and transform it into an adversarial one. Finally, the generated adversarial malware is examined by Detector to assess its evasiveness.

Algorithm 1 Foxy Malware Generation by MalInfo

1: Denote by M the malware set, Π_m the action sequence for malware $m \in M$, *Algo* the algorithm for action sequence generation used by Agent, and m' the output adversarial malware example for m;
2: **for** $m \in M$ **do**
3: $\quad \Pi_m \leftarrow \text{Agent}(m, Algo)$;
4: $\quad m' \leftarrow \text{PE EDITOR}(m, \Pi_m)$;
5: **end for**

Agent is trained to seek an optimal action sequence for a single malware program under the reinforcement learning setting. In our implementation, we employ both dynamic programming (DP) and temporal difference (TD) learning. Dynamic programming copes with the limited combinations of the perturbation methods and uses the complete knowledge of the environment and all possible transitions for each malware. Temporal difference learning, on the other hand, handles the scenario with a large number of malware programs or a large number of method combinations, in which case it is not possible for dynamic programming to sweep through all combinations of perturbations for the entire malware set. Note that since perturbations over 3 layers prolong the loading time and thus can easily cause the suspicion of detection entities (Gaudesi et al., 2015), we empirically set the maximum layers of perturbation to be 3, i.e., up to 3 perturbation methods can be sequentially employed to revise a malware program into an adversarial one.

PE Editor manages the process of producing adversarial malware examples by adopting the strategies provided by Agent. The development of PE Editor exploits three framework methods proposed in (Zhong et al., 2021), namely *Obfusmal, Stealmal*, and *Hollowmal*, which are briefly explained as follows.

Obfusmal enciphers the code section of a malware sample, and attaches to its end a dynamic-link library (DLL) named *Shell.dll*. Then, it alters the entry point of the malware's execution to that of *Shell.dll* whose functionality is to decrypt the code section and recover the normal execution of the malware program. Stealmal enciphers the entire malware and appends it to the end of a program named *Shell.exe*, whose responsibility is to decrypt the encrypted malware, create a suspended process, attain the process space, move the decrypted malware to the space, change the context of the process to the start address of the malware's execution and resume the process. Hollowmal enciphers the entire malware and appends it to the end of a benignware program. Then a DLL named *Hollow.dll* is attached to the end of the modified benignware. *Hollow.dll* is responsible for deciphering the malware, creating a suspended process, attaining the process space, moving the decrypted malware into the space, changing the context of the process to the start address of the malware's execution, and resuming the process. Note that none of these three framework methods affects the functionality of the original malware. We say that Obfusmal, Stealmal, and Hollowmal are framework methods because the implementations of the corresponding *Shell.dll*, *Shell.exe*, and *Hollow.dll* are not specifically defined. Additionally, *Hollow.dll* employs a benignware program, which could be any benign one, further extending the search space of the perturbation methods.

In our study, we utilize VirusTotal as the Detector for `MalInfo`. VirusTotal is currently maintained by Google Inc., which incorporates many antivirus products and online scan engines. Many of them such as F-Secure, McAfee, 360, Tencent, and Microsoft, are popular tools, which have been broadly deployed on laptops and mobile devices. Users can upload a file with a maximum size of 550 MB to the website (https://www.virustotal.com/gui/) and receive the detective outcome. In this study, we use VirusTotal to provide detective outcomes for Agent and to validate the performance of `MalInfo`. Each detective outcome specifies whether one of the anti-virus products or scanners in VirusTotal succeeds in the recognition of the malware or adversarial malware example.

5. MalInfo implementation

In this section, we present our implementation of `MalInfo`. Recall that `MalInfo` consists of Agent, PE Editor, and Detector. Since VirusTotal is employed to serve as Detector, we only need to detail the implementations of Agent and PE Editor. The interactions among Agent, PE Editor, and Detector are illustrated in Fig. 1.

Initially, Agent relies on reinforcement learning to select an action with the maximum expected cumulative reward for the input malware program. Note that an action refers to a certain perturbation method applied to the malware to generate its adversarial variant. PE Editor follows the action to process the malware for developing an intermediate adversarial variant, which is sent to Detector to retrieve a reward. Reinforcement learning then updates the expected cumulative reward of the input malware for future use of the next iteration based on the returned reward. For the input malware at the state of the intermediate adversarial variant, If there is an action that could lead to a higher expected cumulative reward and is selected by Agent, PE Editor processes the intermediate adversarial variant by following the action to generate a more powerful adversarial variant, which is more effective in escaping the detection of Detector than the previous intermediate adversarial variant (that is, a fewer number of detectors in Detector can recognize the newly generated intermediate adversarial variant). The newly generated intermediate adversarial variant is similarly sent to Detector, and reinforcement learning retrieves the reward to update the expected cumulative reward for the original intermediate adversarial variant. Otherwise, no action is taken by Agent. It can further be processed to generate adversarial variants until the times of perturbation are reaching to a threshold. `MalInfo` repeats this procedure from the beginning until an optimal action sequence that can lead to the lowest detection rate R_D[2] and the highest evasive rate R_E[3] is obtained. One can see that through iterative interactions among Agent, PE Editor, and Detector over time, `MalInfo` can finally generate an adversarial malware example with the optimal strategy.

5.1. Agent

Agent incorporates two reinforcement learning algorithms, *i.e.* dynamic programming and temporal difference learning, in our implementation. Depending on the environment dynamics, Agent chooses one of the two methods to search for an optimal action sequence. The parameters and their definitions of the two algorithms are presented in Table 1.

5.1.1. Dynamic programming

Dynamic programming (DP) refers to a collection of algorithms that are used to compute optimal policies given a perfect model of the environment as a Markov decision process. Note that DP is of limited utility because of its huge computational expense (Sutton and Barto, 2015). It consists of states, actions, reward sets, value functions, and optimal policies. The key idea of DP is the use of value functions to organize and structure the search for good policies. Given the perfect environment dynamics for a limited number of combinations of perturbation methods, one can use DP to sweep through all options for each malware program to find out the optimal actions that lead to the lowest expected R_D and highest expected R_E for the given dataset. In Algorithm 2 we describe the process of the DP algorithm employed by our study. The

Algorithm 2 Optimal Action Sequence Determination via Dynamic Programming for malware m

1: Denote by Π_m the optimal action sequence to output, A the action set, S_m the state set with s_0 being the original malware m.
2: Compute S_m based on S_0 and A by exhaustive search;
3: Initialize $V_s^0 = 0$ for $\forall s \in S_m$;
4: **while** True **do**
5: **for** each $s \in S_m$ **do**
6: $V_s^1 = \max_a\{\text{prob}(a/s) \cdot r(a/s) + V_{s'}^0 \mid a \in A, s' =$ next state(s,a)$\}$;
7: a_s=arg $\max_a\{\text{prob}(a/s) \cdot r(a/s) + V_{s'}^0 \mid a \in A, s' =$ next state(s,a)$\}$;
8: **end for**
9: **if** $\max\{V_s^1 - V_s^0 \mid s \in S_m\} \le \epsilon$ **then**
10: break; ▷ ϵ is a small positive real number
11: **end if**
12: $V_s^0 = V_s^1$ for $\forall s \in S_m$;
13: **end while**
14: ▷ Optimal Action Sequence Generation
15: $i = 1$; $\Pi_m = \emptyset$;
16: **while** $i \le k$ **do**
17: ▷ k is the designated perturbation path length
18: Π_m.Append($a_{s_{i-1}}$);
19: s_i = next state($s_{i-1}, a_{s_{i-1}}$);
20: **end while**
21: Output Π_m

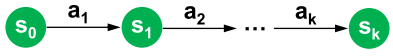

Fig. 2. Action Sequence via Dynamic Programming.

structure of the algorithm contains 5 major components, namely S, A, $r(A/S)$, V, and Π, which are detailed in Table 1.

Algorithm 2 is an iterative one (the `while` loop in lines 4–13) with the value function of each state being updated during each iteration until no value can be significantly changed compared to the previous iteration (the difference is no larger than ϵ, see line 9). For each state at each iteration, only the best action that can maximize the value function, which is recorded in a_s, leads to the update of the value function (see lines 6 and 7). The value function of a state s taking action a, i.e., $V_{a/s}$, is the sum of the expected reward $\text{prob}(a/s) \cdot r(a/s)$ and $V_{s'}^0$, the value function of the next state s' in the previous iteration. The second `while` loop (lines 16–20) selects the first k best actions that form the policy to be taken by PE Editor to generate the adversarial malware example (see Fig. 2). In practice, k is usually small, e.g., $k = 3$, as a big value of k may alert malware detectors and search engines to discern the produced adversarial malware example.

The convergence of Algorithm 2 can be proven according to (Littman and Szepesvári, 1996). There are four properties in our algorithm similar to those presented in Littman and Szepesvári (1996). First, s' is randomly selected with respect to the probability distribution defined by prob(a/s). Second, the max operation is a non-expansion, and both the expected value and the variance of $V_{s'}^0$ exist given the way s' is sampled since the number of states and the options of the actions are finite in our design. Third, $\text{prob}(a/s) \cdot \text{r}(a/s)$ has a finite variance and expected value given s and a. Finally, the algorithm updates the values synchronously over the entire state space. Thus the convergence of Algorithm 2 can be approached. The optimality can be justified as follow. One can see that if there exist better actions for some states, these actions are always chosen to update their correspond-

[2] Detection rate is the proportion of entities (n) that detect the malware or adversarial malware example over all entities (N) in Detector.

[3] Evasive rate is the proportion of $N_{orig} - N_{adv}$ over N_{orig}, where N_{orig} and N_{adv} are respectively the number of entities that detect the original malware and that of entities that detect the current version of the perturbed malware in Detector.

Table 1
Components of Dynamic Programming and Temporal Difference Learning.

Components	Definitions
Action Set A	A is the set of all possible perturbations (i.e., actions) that an agent can employ to generate adversarial malware examples.
State Set S	S of a malware program is the set containing the original malware program, denoted by s_0, and all its adversarial variants.
Reward $r(a/s)$	A reward is the feedback of the scan engines by which we measure the effectiveness of a malware variant at a given state s when taking action a. The reward is defined as $r(a/s) = R_E(a/s)$, where $R_E(a/s)$ is the evasive rate derived from the analytical results of the scan engines, e.g., VirusTotal.
Optimal Policy Π	The optimal policy is the action sequence taken by the agent to generate the final adversarial malware example.
Value function $V_{a/s}$ and V_s	$V_{a/s}$, the value function of state s taking action a, is the expected cumulative reward for state s taking action a, and V_s is the value function of state s, which is defined to be the maximum value of $V_{a/s}$ considering all possible actions, i.e., $V_s = \max_a\{V_{a/s} \mid a \in A\}$. V_s^0 denotes the value from the previous iteration while V_s^1 the updated value of the current iteration.
State-Action Value $Q(s, a)$	State-Action value, which is similar to V. It refers to the expected cumulative reward from the current state s when the agent takes a certain action a.

ing state values until no better action can lead to the change of the value function. Therefore, after the value functions stabilize, each state has the best action leading to the lowest detection rate and highest evasive rate for its adversarial variant.

5.1.2. Temporal difference learning

In the dynamic programming setting, we target limited combinations of perturbation methods for the optimal solution. However, such a combination space cannot delineate all the environment dynamics. With the enlarged number of malware programs and the exponentially increased perturbation paths as the number of options of the perturbation methods is enriched, it's impossible to iterate all the potential states to get a promising outcome with constraint computation resources.

To address this issue, one can take advantage of temporal difference (TD) learning to approximate the optimal result via sampling a certain number of episodes. Each episode involves a process from the first state to the terminal state. Temporal difference learning combines Monte Carlo with dynamic programming. Therefore, it can learn directly from raw experience without establishing a model of the environment dynamics in advance, just as Monte Carlo does. In addition, it can update estimates partially based on other learned estimates, as DP does (Sutton and Barto, 2015; Ueno et al., 2011). The structure of the TD algorithm involves 5 major components (S, A, $Q(s, a)$, Π), which are defined in Table 1.

The learning phase to seek an optimal policy by sampling from the environment is depicted in Algorithm 3, which iteratively updates the state-action value by exploring a series of action sequences. The algorithm relies on the state-action value to find an optimal policy. To start with, line 3 randomly selects an action from the action set. Lines 4–29 are the key steps that update the latest state-action value for each chosen state-action pair by measuring the difference between the old state-action value and a better estimate to approach the actual reward. Specifically, lines 7–11 identify the next state s_i for the current state s_{i-1} and select its action a_{i+1} with ε-greedy policy, which determines the state-action value to be updated. The ε-greedy policy($Q(s, a)$) is a strategy that behaves as follows: (a) it picks an action randomly and uniformly among the action set A with probability ε for a state s, or (b) with probability 1-ε, takes the action that yields the maximum value in the state-action value for a state s. The usage of ε-greedy policy is to balance the exploration and exploitation, and therefore one can vary the value of ε to accommodate the computation resources. Lines 13–20 calculate the value t that is used to update the current state-action value $Q(s_{i-1}, a_i)$. More specifically, line 13 exploits the existing state-action values $Q(s_i, a)$ in the state s_i to decide the best action set $A_{\max}$ ($Q(s_i, a)$ may have multiple maximum values). Next, lines 14–19 compute the value t by the weighted sum of state-action values of all possible state-action pairs in the state s_i. One can see that each state-action pair is assigned with the equal probability $\frac{\varepsilon}{len(A)}$, but each best state-action pair has an

Algorithm 3 Optimal Action Sequence Selection Based on Temporal Difference Learning for malware m

1: Denote by Π_m the optimal action sequence to output, S the state set, and A the action set;
2: initilize $Q(s_0, a) = 0$ for all $a \in A$; $\Pi_m = \emptyset$;
3: randomly select an action a_1;
4: **for** each episode **do**
5: $i = 1$; $A_s = \emptyset$; ▷A_s denotes an action sequence
6: **while** $i \leq k$ **do**
7: $s_i \leftarrow$ next state(s_{i-1}, a_i);
8: **if** s_i exists for the first time **then**
9: initialize $Q(s_i, a) = 0$ for all $a \in A$;
10: **end if**
11: $a_{i+1} \leftarrow \varepsilon$-greedy policy$(Q, s_i)$;
12: $t = 0$;
13: $A_{max} = \arg\max_{a \in A}\{Q(s_i, a) | a \in A\}$;
14: **for** $\forall a \in A$ **do**
15: **if** $a \in A_{max}$ **then**
16: $t+ = (\frac{(1-\varepsilon)}{len(A_{max})} + \frac{\varepsilon}{len(A)})Q(s_i, a)$;
17: **else**
18: $t+ = \frac{\varepsilon}{len(A)}Q(s_i, a)$;
19: **end if**
20: **end for**
21: $Q(s_{i-1}, a_i) = Q(s_{i-1}, a_i) + \alpha[r_c + \gamma t - Q(s_{i-1}, a_i)]$;
22: A_s.Append(a_i)
23: i++;
24: **end while**
25: **if** A_s is a new action sequence **then**
26: $m' \leftarrow$ PE Editor(m, A_s);
27: $Q(s_{k-1}, a_k) \leftarrow$ Detector(m');
28: **end if**
29: **end for**
30: $i = 1$;
31: **while** $i \leq k$ **do**
32: a_i = $\arg\max_{a \in A}\{Q(s_{i-1}, A)\}$; ▷ break ties randomly
33: Π_m.Append(a_i);
34: s_i = next state(s_{i-1}, a_i);
35: i++;
36: **end while**
37: Output Π_m;

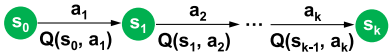

Fig. 3. Temporal Difference Learning.

additional probability $\frac{(1-\varepsilon)}{len(A_{\max})}$ since actions in the best action set $A_{\max}$ are more preferable, where $len(\cdot)$ is the cardinality of a set. Line 21 bases the new state-action value $Q(s_{i-1}, a_i)$ update to approach the real reward of itself on an existing state-action value of the state-action pair (s_{i-1}, a_i), an observed reward (r_c), and a discounted better estimate γt by discounting the weighted expected reward in all possible next state-action pairs. Therein, the observed reward is the possible reward that can be directly received when taking action a_i in state s_{i-1}, α is the learning rate that determines the rate of convergence, γ is the tendency of looking forward to the expected reward received from the future states. Besides, in line 22, all actions in the updated state-action pairs are stored in the action sequence A_s. When the length of the action sequence A_s reaches k, lines 25–28 generate an adversarial malware example m' by PE Editor with the malware m and A_s as parameters. The actual reward is retrieved from Detector to update the state-action value for the last state-action pair. Lines 31–36 identify an optimal action and append it to policy Π_m by referring to the maximum state-action values (as shown in Fig. 3).

The convergence of temporal difference learning to select the right actions is proved in Dayan and Sejnowski (1994). In this study, since Algorithm 3 uses similar equations as presented in Dayan and Sejnowski (1994), the convergence of Algorithm 3 can be derived from that of temporal difference learning (Dayan and Sejnowski, 1994).

5.2. PE editor

By following the strategy generated by Agent, PE Editor can transform a malware program into an adversarial one. A strategy consists of successive actions that correspond to three perturbation framework methods, namely Obfusmal, Stealmal, and Hollowmal, which were originally presented in Zhong et al. (2021) and are sketched here for completeness.

5.2.1. Obfusmal

An adversarial malware example generated by Obfusmal consists of 2 components (as shown in Fig. 4(a)): (*i*) *malware.exe*, which is the original malware program but with an encrypted code section; and (*ii*) *Shell.dll*, which is responsible for taking *malware.exe* back to the normal execution. The detailed implementation of Obfusmal was presented in Zhong et al. (2021). In our study, if the action selected by Agent is 1, *malware.exe* is processed by Obfusmal to generate an adversarial malware example whose execution process incorporates 3 phases as illustrated in Fig. 5(a). In Phase I, its OEP (an OEP is the start address for program execution) is headed to the OEP of *Shell.dll*. In Phase II, *Shell.dll* performs the decryption of the code section, alters the relocation information, and moves the program execution to the start address of *malware.exe*. In Phase III, the normal execution of the code section of *malware.exe* is recovered.

5.2.2. Stealmal

Similar to Obfusmal, an adversarial malware example generated by Stealmal consists of two components (see Fig. 4(b)): (*i*) *malware.exe*, which is the encrypted malware body; and (*ii*) *Shell.exe*, which manages the recovery of *malware.exe* execution. The implementation details of Stealmal were presented in Zhong et al. (2021). In our study, if the action selected by Agent is 2, *malware.exe* is processed by Stealmal to generate an adversarial malware example whose execution process includes 3 phases

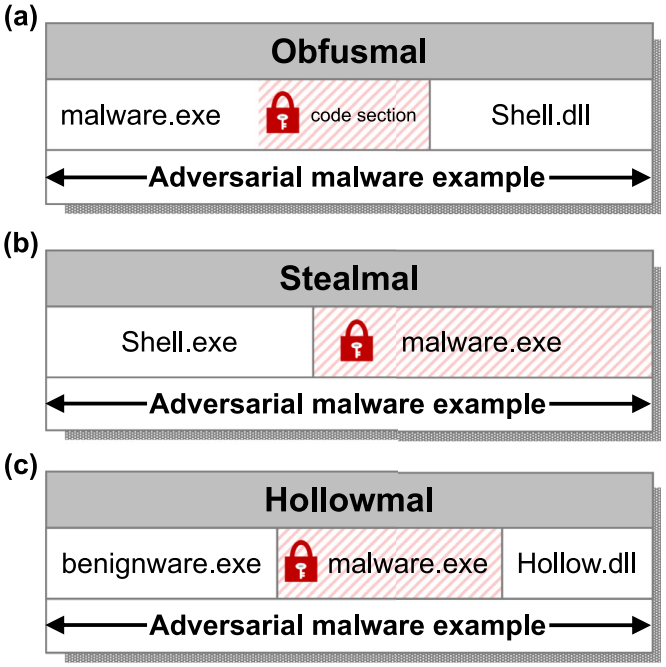

Fig. 4. Components of the Adversarial Malware Example.

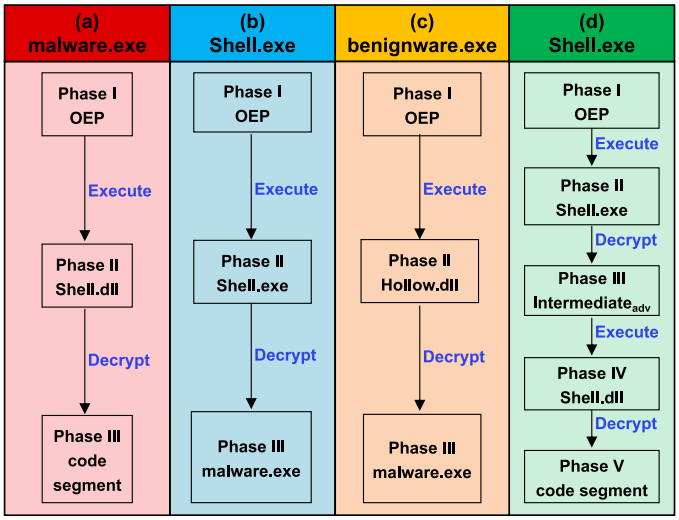

Fig. 5. Processes of Program Execution.

that are depicted in Fig. 5(b). In Phase I, its OEP is headed to the start address of *Shell.exe*. In Phase II, *Shell.exe* accomplishes the following operations: decipher *malware.exe*, create a suspended process, vacate the process space, move *malware.exe* to the space, alter the relocation information, and switch the context of the process to the OEP of *malware.exe*. In Phase III, the normal execution of *malware.exe* is recovered.

5.2.3. Hollowmal

An adversarial malware example generated by Hollowmal is slightly different from Obfusmal and Stealmal. It is comprised of 3 components (see Fig. 4(c)): (*i*) *benignware.exe*, which is a piece of benignware to conceal the malware to improve its surreptitiousness; (*ii*) *malware.exe*, which is the malware program used by Hollowmal to generate an adversarial malware example; and (*iii*) *Hollow.dll*, which embraces similar functionality as *Shell.exe*. The implementation details of Hollowmal were proposed in Zhong et al. (2021). In our study, if the action selected by Agent

is 3, *malware.exe* is processed by Hollowmal to generate an adversarial malware example, whose execution flow is exhibited in Fig. 5(c). In Phase I, its OEP is directed to the start address of *Hollow.dll*. In Phase II, *Hollow.dll* proceeds with the following operations: decipher *malware.exe*, create a suspended process, vacate the process space, move the *malware.exe* to the process space, alter the relocation information to the desired position, and switch the context of the process to the OEP of *malware.exe*. In Phase III, the normal execution of *malware.exe* is recovered.

5.2.4. Combination

Once Agent outputs an action sequence that includes more than one action, PE Editor iterates over each action in the sequence and performs the corresponding action on the malware. After a series of editing, the original *malware.exe* is transformed into its adversarial version. Since more than 3 layers' perturbations obviously extend the loading time and easily cause suspicion of antivirus products and scan engines, the maximum number of layers of perturbation is empirically set to be 3. For example, suppose the *optimal action sequence* $= (1, 0, 2)$ for a malware program m_i, PE editor checks the action sequence and employs Obfusmal (1) on m_i to produce an intermediate version of the adversarial variant m_i^{int} at first. As the second action in the sequence is 0, which means no action needs to be taken, PE Editor skips the second action and moves on to the next action. At last, PE Editor imposes Stealmal (3) on the previous m_i^{int} to produce the ultimate version of the adversarial malware example m_i^{final}. The execution process of m_i^{final} consists of 5 phases (see Fig. 5(d)). In Phase I, its *OEP* is headed to the OEP of *Shell.exe*. In Phase II, *Shell.exe* conducts the following operations on m_i^{int}: the decryption of m_i^{int}, the creation of a suspended process, the vacation of the process space, the move of m_i^{int} to the space, the alteration of the relocation information, and the switch of the process context to the OEP of m_i^{int}. In Phase III, the OEP of m_i^{int} is headed to the OEP of *Shell.dll*. In Phase IV, *Shell.dll* finishes the decryption of the code section and the alteration of the relocation information, and jumps to the OEP of *malware.exe*. In Phase V, the decrypted code section is recovered for execution.

6. Evaluation

6.1. Performance metrics

Two popular metrics, *i.e.* detection rate and evasive rate, have been employed to evaluate the performance of `MalInfo`. The detection rate R_D is defined as follows,

$$R_D = \frac{n}{N} \tag{1}$$

where N is the total number of entities (malware scan engines) in VirusTotal, and n is the number of entities that could identify the malware or adversarial malware example. A lower detection rate implies a better capability of the malware generation scheme to defeat scan engines.

The evasive rate is defined as follows,

$$R_E = \frac{\frac{N_{orig}}{N} - \frac{N_{adv}}{N}}{\frac{N_{orig}}{N}} = \frac{N_{orig} - N_{adv}}{N_{org}} \tag{2}$$

where N_{orig} is the number of entities that can identify the original malware, and N_{adv} is the number of entities that can identify the corresponding adversarial malware example. Particularly, a higher evasive rate suggests a higher probability that the adversarial example can escape from detection by malware scan engines. It demonstrates the effectiveness of `MalInfo` to generate a stealthy adversarial malware example.

6.2. Experimental setup

All the experiments are performed on a laptop with 6-core and 32-GB of RAM, which could be affordable for most users.

Datasets. Our dataset for evaluation includes 1000 malware samples that are active in the wild. They are drawn from VirusShare[4] Vir (2020). All the malware samples target the Windows platform and have complete PE structures. Besides, VirusTotal is a powerful tool for security analysis and has been widely used by security researchers for labeling malware data (Afroz, 2020; Huang et al., 2021; Zhu et al., 2020). In our experimental study, we employ 82 entities in VirusTotal to evaluate the performance of `MalInfo`. All of the malware samples submitted to VirusTotal are Windows executables.

Perturbation Methods. As mentioned early, Obfusmal, Stealmal, and Hollowmal are three framework perturbation methods that could be infinitely instantiated, resulting in a large search space for the action sequence. In our experimental study, we consider a small search space that consists of one instantiation for each framework method (implemented in Zhong et al. (2021)) because this small search space is sufficient to demonstrate the effectiveness of `MalInfo`, as evidenced by the following evaluation results. But we are confident in claiming that a larger search space can produce adversarial examples that have an even lower detection accuracy and a higher evasive rate.

Algorithmic Parameters. In the dynamic programming method, since each action a is selected at a state s with an equal probability, the probability prob(a/s) is equal to 0.25 as the search space size is 4 (including the *Null* action). The threshold ϵ with a value 1e-3 is used in our experiments, which can be justified as follows: the difference of the values V_s between two iterations is at least $0.25 * \frac{1}{82}$, as each iteration takes the best possible action for a state, indicating that fewer detectors in VirusTotal can recognize the adversarial malware example. In an extreme case in which the number of detection engines that identify the adversarial malware example is reduced by 1, the value difference would be $\frac{1}{4} * \frac{1}{82}$, where 82 is the total number of detection engines. Therefore, the threshold ϵ with a value 1e-3 is sufficiently small to halt the update of the value function to find a better policy. In the temporal difference learning method, the greedy probability ε, reward r_c, and step size α are empirically set to 0.3, 0, and 0.95, respectively, since a low greedy probability, a constant reward, and a large step size can be helpful to more quickly search for optimal results (Sutton and Barto, 2015). Besides, temporal difference learning uses a discount rate γ with value 1 since the value of the next state s' has a direct impact on the current state s. The length of the action sequence k used in both dynamic programming and temporal difference learning is empirically set to 3 to avoid the suspicion of scan engines.

6.3. Evaluation results

In this subsection we report our evaluation results in terms of detection rate and evasive rate: the lower the detection rate and the higher the evasive rate, the better the `MalInfo` performs as the produced malware examples can better escape from detection.

For each malware program under our study, we first transform it into the corresponding adversarial malware examples based on the policies generated by the dynamic programming method and the temporal difference learning method; then we consider the detection rate and evasive rate with the collected detection results from VirusTotal; finally, we calculate the average detection

[4] VirusShare is an open-source website for researchers with authorized access to samples of live malicious code.

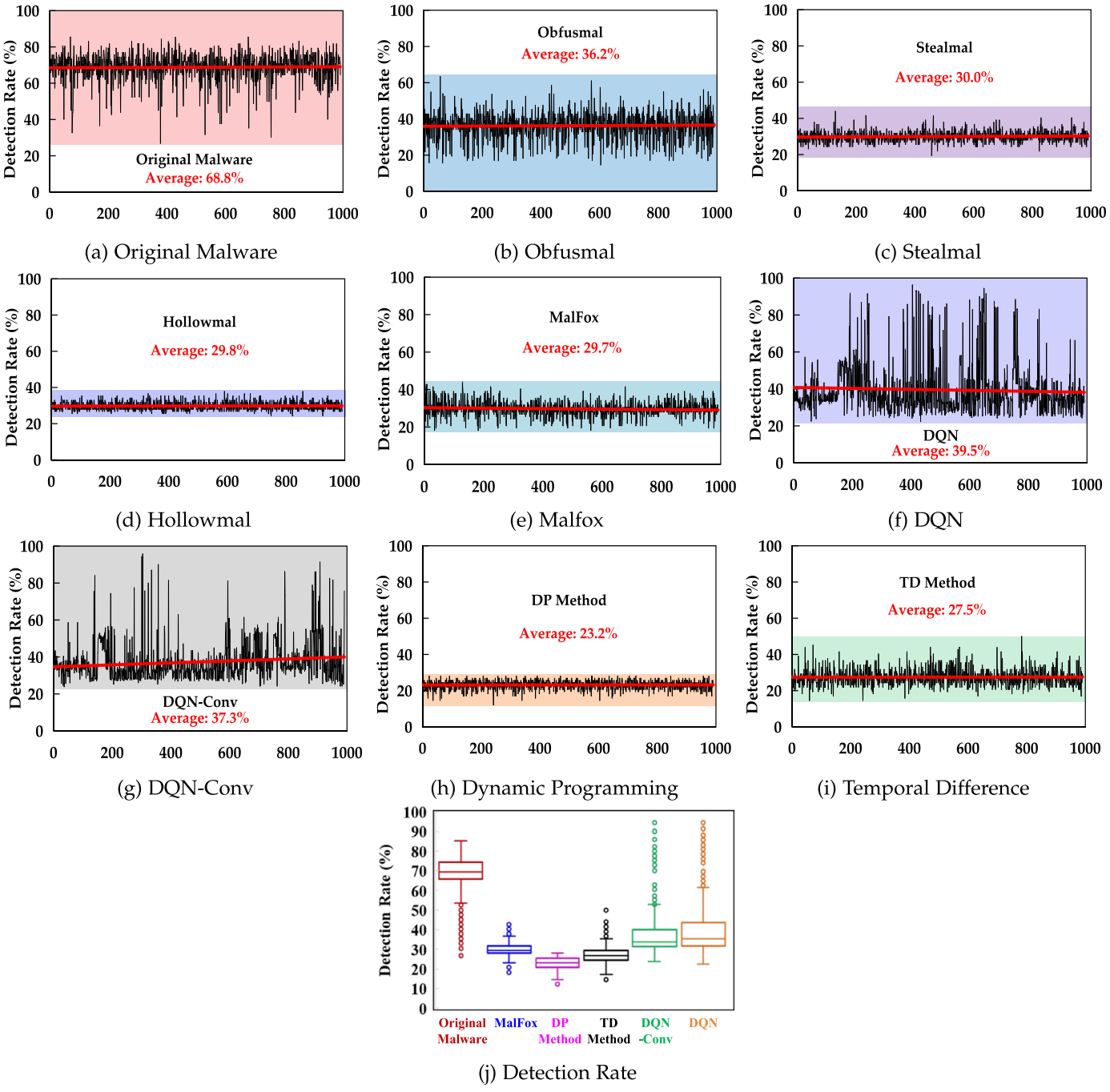

Fig. 6. Detection Rate.

rate and evasive rate for all malware programs in the dataset. To demonstrate the effectiveness of MalInfo, we compare it with MalFox (Zhong et al., 2021), the three individual perturbation methods, namely Obfusmal, Stealmal and Hollowmal, and the two deep reinforcement learning methods (DQN and DQN-Conv). MalFox is a convolutional generative adversarial network-based framework targeting adversarial malware example generation against third-party black-box malware detectors. It was motivated by the rival game between malware authors and malware detectors and adopts a confrontational approach to produce perturbation paths, with each formed by up to three methods (e.g., Obfusmal, Stealmal, and Hollowmal) to generate adversarial malware examples. MalFox has a drawback – its optimal way to generate powerful adversarial malware examples is based on the whole dataset, which means that the action sequence of a malware program doesn't rely specifically on the malware itself. As a comparison, MalInfo selects the best action sequence based on the malware program itself, which obviously could lead to better results due to less noise.

6.3.1. Detection rate

As shown in Fig. 6a, the average detection rate of the original malware is 68.8% instead of 100%. This is because some anti-virus products and scan engines in VirusTotal cannot correctly recognize all malware programs. Figs. 6b-6d indicate that the detection rate of each single perturbation method is 36.2%, 30.0%, and 29.8%, which is significantly improved compared to that of the original malware. `MalFox` relies on these three perturbation methods, which further decreases the detection rate to 29.70%. Nevertheless, as

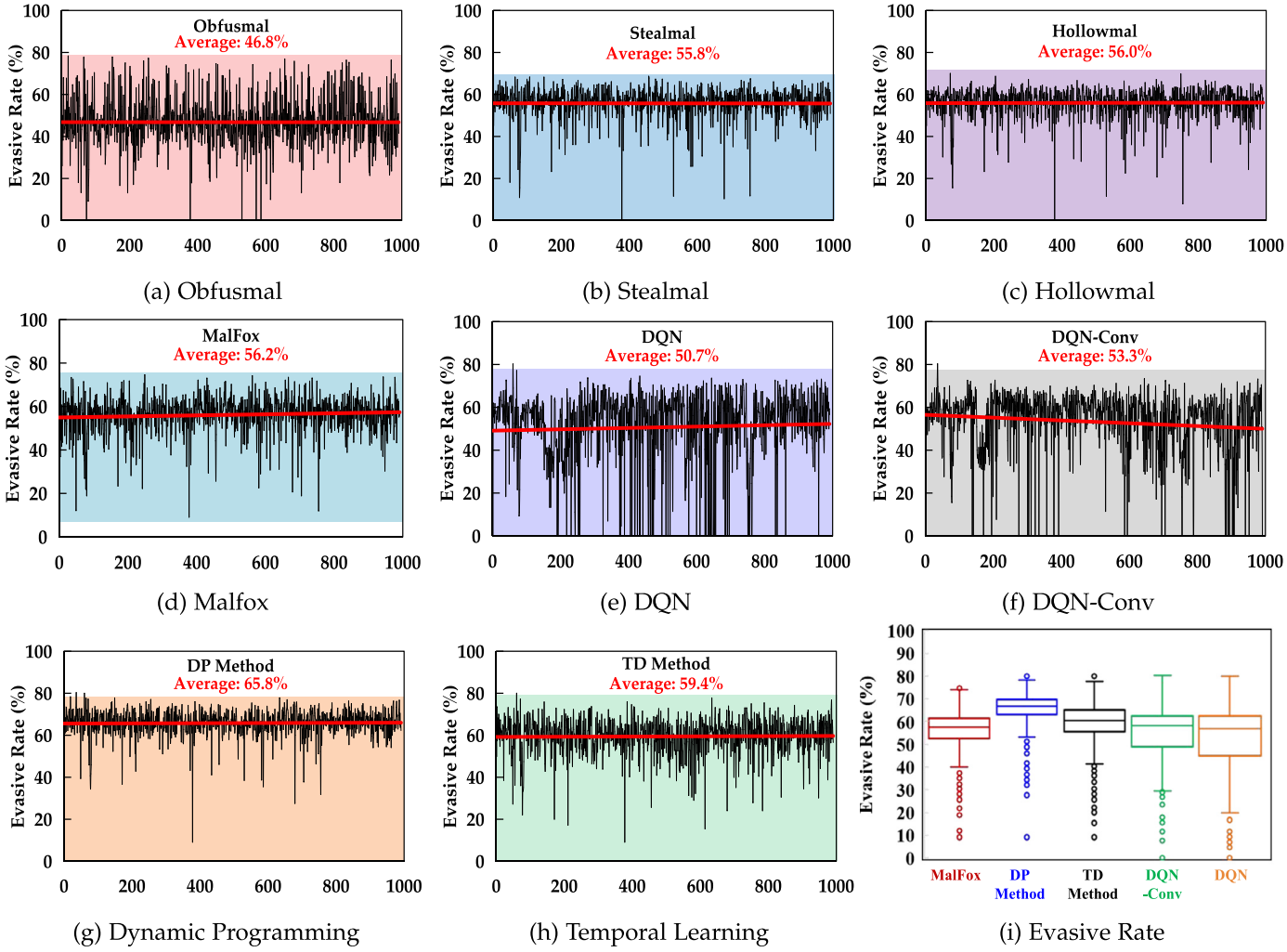

Fig. 7. Evasive Rate.

shown in Fig. 6f and Fig. 6g, the detection rates of the two deep reinforcement learning methods (DQN and DQN-Conv) are 39.5% and 37.3%, respectively, which is no better than any of the perturbation method, even though they are more complicated and have demonstrated superior performance in other applications.

The `MalInfo` framework that utilizes dynamic programming has an average detection rate of 23.2% as shown in Fig. 6h and that employs temporal difference learning has an average detection rate of 27.5% as shown in Fig. 6i. Compared with the original detection rate in Fig. 6a, these two schemes dramatically decrease the average detection rate, with a decrease of about 66.3% and 60.0%, respectively. `MalInfo` outperforms each single perturbation method and `MalFox` with an improvement by 24.0% (Obfusmal), 8.3% (Stealmal), 7.7% (Hollowmal), and 7.4% (`MalFox`). Moreover, it is supperior over the other two deep reinforcement learning methods as shown in Fig. 6f and Fig. 6g, descending at least 30.3% and 26.3%.

Fig. 6j illustrates a clearer comparison of the detection rates of the original malware and the adversarial malware examples produced by `MalInfo`, deep reinforcement learning methods, and `MalFox`. One interesting observation shown in Fig. 6j is that temporal difference learning has a higher variance than dynamic programming, which implies that dynamic programming has more stable performance. One possible reason is that, given the perfect environment dynamics (i.e. finite states, finite actions, a clear transition function, and a complete reward set), dynamic programming can select the best actions by referring to the past knowledge. However, temporal difference learning, which is not provided with perfect environment dynamics (due to limited computing resources and insufficient knowledge), prefers to explore more traces and acquires the statistically best actions that sometimes are not the best in practice.

6.3.2. Evasive rate

As shown in Fig. 7 a-7c, the average evasive rates of using a single perturbation method are 46.8% (Obfusmal), 55.8% (Stealmal), and 56.0% (Hollowmal), which suggest that roughly half of the detectors can't characterize the original malware samples processed by them. `MalFox` increases the evasive rate to 56.2% by relying on combinations of these methods, given in Fig. 7d. Our proposed framework, `MalInfo`, further improves the evasive rate to about 65.8% for dynamic programming and 59.4% for temporal difference (see Fig. 7g-7h), which leaves deep reinforcement learning methods far behind. The average evasive rates of the latter are 50.7% and 53.3% as shown in Fig. 7e-Fig. 7f. One can see that dynamic programming has a visible improvement in the average evasive rate from Obfusmal, Stealmal, Hollowmal, `MalFox`, DQN, and DQN-Conv with an enhancement of 40.6%, 17.9%, 17.50%, 17.1%, 29.8%, and 23.5% respectively. Temporal difference learning noticeably increases the corresponding evasive rate by 26.9%, 6.5%, 6.1%, 5.7%, 17.2%, and 11.4%.

Fig. 7i provides a more intuitive comparison over the evasive rates between `MalInfo`, the two deep reinforcement learning meth-

Table 2
Action Sequence Selection in Different Methods.

Malware Name	R_D, R_E, and Action Sequence (MalFox)	R_D, R_E, and Action Sequence (DP)	R_D, R_E, and Action Sequence (TD)
*VirusShare_*00184*c*8076*d*6*c*2*cf*240*ba*644120*f*4*db*2	0.354, 0.482 (2, 0, 3)	0.244, 0.643, (1 1 1)	0.244, 0.643, (3, 2, 1)
*VirusShare_*0178*ef*69*b*88*afd*7312077*a*576*a*127*e*18	0.415, 0.370, (1, 2, 3)	0.256, 0.611, (2, 2, 2)	0.281, 0.574, (1, 1, 1)
*VirusShare_*191*f*0226*ba*850*cb*4*bcc*21376673*a*0212	0.293, 0.539, (0, 2, 0)	0.268, 0.577, (1, 0, 3)	0.281, 0.558, (3, 2, 1)
*VirusShare_*202*bcba*7948*c*059549*d*70315*eb*892456	0.293, 0.613 (1, 0, 3)	0.220, 0.710, (2, 2, 1)	0.232, 0.694, (1, 3, 1)
*VirusShare_*00233*a*088*b*98089*e*54699*f*71*beda*5*ec*0	0.268, 0.627, (1, 2, 3)	0.183, 0.746, (1, 1, 2)	0.183, 0.746, (1, 1, 3)
*VirusShare_*0214*b*8*fbf*48771*b*374044*f*21*b*3*a*2892*e*	0.281, 0.635 (1, 2, 3)	0.256, 0.667, (0, 2, 1)	0.293, 0.619, (1, 3, 1)
*VirusShare_*00253*a*322732*c*797*fcc*2*cd*87*c*5*cfcde*6	0.317, 0.536, (0, 0, 3)	0.207, 0.696, (2, 2, 2)	0.207, 0.696, (1, 2, 3)
*VirusShare_*0236*af*719*b*5*e*6*b*67*c*9*d*98*e*3*ff*96297*ab*	0.207, 0.712, (1, 2, 3)	0.207, 0.712, (1, 2, 3)	0.207, 0.712, (1, 2, 3)
*VirusShare_*00261*ae*215599*d*0848*cac*48891*a*674*ca*	0.329, 0.491, (0, 0, 3)	0.268, 0.585, (3, 2, 1)	0.293, 0.547, (1, 1, 1)
*VirusShare_*278*efb*7*cf*8*b*406*dbb*4*ae*38*eb*89*eb*3*d*15	0.317, 0.559, (0, 0, 3)	0.244, 0.661, (2, 3, 1)	0.293, 0.593, (2, 1, 1)

ods, and MalFox. Notice that temporal difference learning has a higher mean average evasive rate when compared to MalFox, which indicates that through more diverse explorations, temporal difference learning can gain more insights to know better opportunities to achieve its goal. Besides, dynamic programming can employ past knowledge to select the best actions. One can see that though deep reinforcement learning methods adopt similar theoretical foundation as MalInfo, they have significantly worser performance. The reason behind this is that the choice of the action sequence has no strict linear or non-linear relationship with generating adversarial malware examples, resulting in neural networks inappropriate in such application scenarios. Additionally, the maximum evasive rate of dynamic programming and DQN even reaches 80.3%, while those of MalFox, DQN-Conv, and temporal difference learning are 74.6%, 80%, and 80.0%, respectively. Note that temporal difference learning has a higher variance as it may not always select the right actions, but choose the statistically correct ones.

6.4. Example illustration on action sequence selection

Table 2 delineates the resulting action sequences of 10 randomly selected malware examples after applying MalFox, dynamic programming, and temporal difference. One can see that dynamic programming yields the lowest detection rate and the highest evasive rate for the same malware in all cases. Let's examine an example, see the ninth malware sample *VirusShare_*00261*ae*215599*d*0848*cac*48891*a*674*ca*, to gain more insights. For this malware program, MalFox chooses the perturbation path $(0, 0, 3)$, which indicates that the original malware processed by Hollowmal can result in the lowest R_D and highest R_E under the MalFox setting. Temporal difference learning selects a different perturbation path $(1, 1, 1)$ to generate a more powerful adversarial malware example with a lower $R_D = 0.293$ and a higher $R_E = 0.547$. Dynamic programming applies an even more complicated generation scheme $(2, 3, 1)$, which edits the original malware by Stealmal, Hollowmal, and then Obfusmal. It yields the best performance with $R_D = 0.244$, and $R_E = 0.661$. Since MalFox is trained on a collection of malware samples, the perturbation path generated by it may not be optimal for each malware sample. However, dynamic programming and temporal difference learning enable the exploration of various perturbation paths for a single malware sample and therefore can achieve a lower R_D and a higher R_E.

7. Conclusion and future research

In this paper, we make use of a reinforcement learning-based framework (titled MalInfo) to look for optimal action sequences (perturbation paths) to generate adversarial malware examples via dynamic programming and temporal difference learning. Given either perfect environment dynamics with a limited number of options of the perturbation paths or no informed model of environment with more diverse options of the perturbation paths, MalInfo can transform a collection of malware programs into foxy ones with a significantly higher chance of evading detection, greatly enhancing the adversarial attacks targeting a collection of third-party antivirus products and online scan engines. The performance of MalInfo is extensively studied in terms of detection rate and evasive rate and a comparison study is also provided to compare with a generative adversarial network-based adversarial malware generation framework (MalFox).

Our future research will be carried out in the following directions. Literally, the number of perturbation methods is relevant to the increase of the evasive rate that an adversarial malware example can obtain. Therefore we intend to combine more diverse methods of generating adversarial malware examples to enrich the actions for an enhanced evasive capability. Besides, we intend to conclude the statistical relationship between the action selection and the performance of MalInfo and investigate heuristic-inspired reinforcement learning algorithms to reduce the burden of computation in dynamic programming and temporal difference learning. We also intend to enrich the area of defense to adversarial malware examples.

Declaration of Competing Interest

The authors declare that they have no known competing financial interests or personal relationships that could have appeared to influence the work reported in this paper.

CRediT authorship contribution statement

Fangtian Zhong: Conceptualization, Methodology, Software, Validation, Formal analysis, Investigation, Data curation, Writing – original draft. **Pengfei Hu:** Formal analysis, Writing – review & editing, Visualization. **Guoming Zhang:** Writing – review & editing. **Hong Li:** Writing – review & editing. **Xiuzhen Cheng:** Writing – review & editing, Visualization, Supervision, Project administration.

References

Afroz S.. How to build realistic machine learning systems for security?2020.
Aghakhani, H., Gritti, F., Mecca, F., Lindorfer, M., Ortolani, S., Balzarotti, D., Vigna, G., Kruegel, C., 2020. When Malware is Packin' Heat; limits of machine learning classifiers based on static analysis features. In: Proceedings of Symposium on Network and Distributed System Security (NDSS).
Al-Dujaili, A., Huang, A., Hemberg, E., O'Reilly, U., 2018. Adversarial deep learning for robust detection of binary encoded malware. In: 2018 IEEE Security and Privacy Workshops (SPW), pp. 76–82. doi:10.1109/SPW.2018.00020.
Ceschin, F., Botacin, M., Gomes, H.M., Oliveira, L.S., Gregio, A., 2019. Shallow security: on the creation of adversarial variants to evade machine learning-based malware detectors. In: Proceedings of the 3rd Reversing and Offensive-oriented Trends Symposium (ROOTS' 19). Association for Computing MachineryNew YorkNYUnited States, Vienna, Austria, pp. 1–9.
Dayan, P., Sejnowski, T.J., 1994. Td(λ) converges with probability 1. Mach. Learn. 14 (1573-0565), 295–301. doi:10.1023/A:1022657612745.

Demetrio, L., Biggio, B., Lagorio, G., Roli, F., Armando, A., 2019. Explaining vulnerabilities of deep learning to adversarial malware binaries. In: 3rd Italian Conference on Cyber Security, ITASEC 2019, volume 2315. IOS Press, Pisa, Italy, pp. 1–7.
Demetrio L., Biggio B., Lagorio G., Roli F., Armando A.. Explaining vulnerabilities of deep learning to adversarial malware binaries. https://arxiv.org/pdf/1901.03583.pdf; 2020.
Fang, Z., Wang, J., Li, B., Wu, S., Zhou, Y., Huang, H., 2019. Evading anti-malware engines with deep reinforcement learning. IEEE Access 7 (18615933), 48867–48879. doi:10.1109/ACCESS.2019.2908033.
Gaudesi, M., Marcelli, A., Sanchez, E., Squilero, G., 2015. Malware obfuscation through evolutionary packers. In: Proceedings of the Companion Publication of the 2015 Annual Conference on Genetic and Evolutionary Computation, pp. 757–758. doi:10.1145/2739482.2764940.
Goodfellow, I.J., Shlens, J., Szegedy, C., 2015. Explaining and harnessing adversarial examples. In: 3rd International Conference on Learning Representations (ICLR'15). The Hilton San Diego Resort & Spa.
Gorelik M. Machine learning can't protect you from fileless attacks. https://securityboulevard.com/2020/05/machine-learning-cant-protect-you-from-fileless-attacks/; 2020.
Grosse, K., Papernot, N., Manoharan, P., Backes, M., McDaniel, P., 2017. Adversarial examples for malware detection. In: Foley, S.N., Gollmann, D., Snekkenes, E. (Eds.), Computer Security – ESORICS 2017. Springer International Publishing, Cham, pp. 62–79.
Hu, W., Tan, Y., 2018. Adversarial examples for malware detection. In: Black-Box Attacks against RNNBased Malware Detection Algorithms. The Workshops of the Thirty-Second AAAI Conference on Artificial Intelligence, Hilton New Orleans Riverside, New Orleans, Louisiana, USA, pp. 1–7.
Huang, H., Zheng, C., Zeng, J., Zhou, W., Zhu, S., Liu, P., Molloy, I., Chari, S., Zhang, C., Guan, Q., 2021. A large-scale study of android malware development phenomenon on public malware submission and scanning platform. IEEE Trans. Big Data 7 (2), 255–270. doi:10.1109/TBDATA.2018.2790439.
Kaelbling, L.P., Littman, M.L., Moore, A.W., 1996. Reinforcement learning: a survey. J. Artific. Intell. Res. 4, 1–49. doi:10.1613/jair.301.
Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018a. Adversarial malware binaries: evading deep learning for malware detection in executables. In: 2018 26th European Signal Processing Conference (EUSIPCO), pp. 533–537. doi:10.23919/EUSIPCO.2018.8553214.
Kolosnjaji, B., Demontis, A., Biggio, B., Maiorca, D., Giacinto, G., Eckert, C., Roli, F., 2018b. Adversarial malware binaries: Evading deep learning for malware detection in executables. In: 2018 26th European Signal Processing Conference (EUSIPCO), pp. 533–537. doi:10.23919/EUSIPCO.2018.8553214.
Labaca-Castro, R., Biggio, B., Rodosek, G.D., 2019. Poster: attacking malware classifiers by crafting gradient-attacks that preserve functionality. In: Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security (CCS' 19), pp. 2565–2567. doi:10.1145/3319535.3363257.
Littman, M.L., Szepesvári, C., 1996. A generalized reinforcement-learning model: Convergence and applications. In: ICML, volume 96. Citeseer, pp. 310–318.
Moerland T.M., Broekens J., Jonker C.M.. Model-based reinforcement learning: a survey. https://arxiv.org/abs/2006.16712; 2020.
Molnar C.. Interpretable machine learning a guide for making black box models explainable. https://christophm.github.io/interpretable-ml-book/adversarial.html; 2021.
Moosavi-Dezfooli, S.M., Fawzi, A., Fawzi, O., Frossard, P., 2017. Universal adversarial perturbations. In: Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR). IEEE.
Morgulis N., Kreines A., Mendelowitz S., Weisglass Y.. Fooling a real car with adversarial traffic signs. https://arxiv.org/ftp/arxiv/papers/1907/1907.00374.pdf; 2019.
Murali, R., Ravi, A., Agarwal, H., 2020. A malware variant resistant to traditional analysis techniques. In: 2020 International Conference on Emerging Trends in Information Technology and Engineering (ic-ETITE), pp. 1–7. doi:10.1109/ic-ETITE47903.2020.264.
Rozsa, A., Rudd, E.M., Boult, T.E., 2016. Adversarial diversity and hard positive generation. In: 2016 IEEE Conference on Computer Vision and Pattern Recognition Workshops (CVPRW), pp. 410–417. doi:10.1109/CVPRW.2016.58.
Strehl, A.L., Li, L., Wiewiora, E., Langford, J., Littman, M.L., 2006. Pac model-free reinforcement learning. In: Proceedings of the 23rd international conference on Machine learning (ICML06'). ACM, pp. 881–888. doi:10.1145/1143844.1143955.
Suciu, O., Coull, S.E., Johns, J., 2019a. Exploring adversarial examples in malware detection. In: 2019 IEEE Security and Privacy Workshops (SPW), pp. 8–14. doi:10.1109/SPW.2019.00015.
Suciu, O., Coull, S.E., Johns, J., 2019b. Exploring adversarial examples in malware detection. In: 2019 IEEE Security and Privacy Workshops (SPW), pp. 8–14. doi:10.1109/SPW.2019.00015.
Sutton R.S., Barto A.G.. Reinforcement learning: an introduction. https://web.stanford.edu/class/psych209/Readings/SuttonBartoIPRLBook2ndEd.pdf; 2015.
Szegedy, C., Zaremba, W., Sutskever, I., Bruna, J., Erhan, D., Goodfellow, I., Fergus, R., 2014. Intriguing properties of neural networks. In: International Conference on Learning Representations. http://arxiv.org/abs/1312.6199
Ueno, T., Maeda, S.i., Kawanabe, M., Ishii, S., 2011. Generalized TD learning. J. Mach. Learn. Res. 12 (56), 1977–2020.
VirusShare Virusshare.com-because sharing is caring. https://virusshare.com/; 2020.
Virustotal. https://www.virustotal.com/gui/; 2020.
Yakura, H., Sakuma, J., 2019. Robust audio adversarial example for a physical attack. In: Proceedings of the Twenty-Eighth International Joint Conference on Artificial Intelligence, IJCAI-19. International Joint Conferences on Artificial Intelligence Organization, pp. 5334–5341. doi:10.24963/ijcai.2019/741.
Yuan, J., Zhou, S., Lin, L., Wang, F., Cui, J., 2020. Black-box adversarial attacks against deep learningbased malware binaries detection with GAN. In: 24th European Conference on Artificial Intelligence - ECAI 2020, volume 2315. CEUR Workshop Proceedings, Santiago de Compostela, Spain, pp. 1–7.
Zhong, F., Cheng, X., Yu, D., Gong, B., Song, S., Yu, J., 2021. Malfox: camouflaged adversarial malware example generation based on conv-GANs against black-box detectors. IEEE Trans. Comput. 1–14.
Zhu, S., Zhang, Z., Yang, L., Song, L., Wang, G., 2020. Benchmarking label dynamics of virustotal engines. Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS' 20) 2081–2083. doi:10.1145/3372297.3420013.

Fangtian Zhong received his Ph.D. in Computer Science and B.E. degree in Software Engineering from The George Washington University, and Northeast Normal University respectively. Currently he is a Postdoctoral Scholar in College of Information Science and Technology at The Pennsylvania State University. His research interests span the broad scope of software security, system security and machine learning for cybersecurity.

Pengfei Hu is a professor in the School of Computer Science and Technology at Shandong University. He received the Ph.D. in Computer Science from UC Davis. His research interests are in the areas of cyber security, data privacy, mobile computing. He has published more than 30 papers in reputed conferences and journals on these topics, including IEEE S&P, ACM CCS, IEEE INFOCOM, ACM CoNEXT, IEEE TMC, IEEE TDSC, etc. Dr. Hu also holds 5 patents. He served as reviewer/PC member for numerous journals and conferences including TIFS, TMC, MASS, MILCOM, etc.

Guomin Zhang received his Ph.D. degree from the Department of Electrical Engineering at Zhejiang University, supervised by Prof. Wenyuan Xu and Donglian Qi, his Master degree in School of Mechanical Engineering of Beijing Institute of Technology, supervised by Prof. Jie Hu, and his Bachelor degree in College of Transportation from Ludong University in 2013. His current research interests include IoT security and acoustic communications. He won the best paper awards of ACM CCS 2017 and Qshine 2019.

Hong Li received the B.S. degree in computer science from Xi'an Jiao Tong University in 2011 and the Ph.D. degree in cyber security from University of Chinese Academy of Sciences in 2017. Since 2017, he is an associate professor at the Institute of Information Engineering, Chinese Academy of Sciences. His current research interests include IoT security, ICS Security and Blockchain.

Xiuzhen Cheng received her PhD degree in computer science from University of Minnesota - Twin Cities in 2002. She was a faculty member at the Department of Computer Science, The George Washington University, from 2002 to 2020. Currently she is a professor of computer science at Shandong University, Qingdao, China. Her research focuses on blockchain computing, IOT Security, and privacy-aware computing. She is a Fellow of IEEE.