MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware

Wei Song
wsong008@ucr.edu
University of California, Riverside
Riverside, USA

Xuezixiang Li
xli287@ucr.edu
University of California, Riverside
Riverside, USA

Sadia Afroz
sadia.afroz@avast.com
sadia@icsi.berkeley.edu
Avast, ICSI
San Francisco, USA

Deepali Garg
deepali.garg@avast.com
Avast
Santa Clara, USA

Dmitry Kuznetsov
kuznetsov@avast.com
Avast
Prague, Czech

Heng Yin
heng@cs.ucr.edu
University of California, Riverside
Riverside, USA

ABSTRACT

Modern commercial antivirus systems increasingly rely on machine learning (ML) to keep up with the rampant inflation of new malware. However, it is well-known that machine learning models are vulnerable to adversarial examples (AEs). Previous works have shown that ML malware classifiers are fragile to the white-box adversarial attacks. However, ML models used in commercial antivirus (AV) products are usually not available to attackers and only return hard classification labels. Therefore, it is more practical to evaluate the robustness of ML models and real-world AVs in a pure black-box manner. We propose a black-box Reinforcement Learning (RL) based framework to generate AEs for PE malware classifiers and AV engines. It regards the adversarial attack problem as a multi-armed bandit problem, which finds an optimal balance between exploiting the successful patterns and exploring more varieties. Compared to other frameworks, our improvements lie in three points: 1) limiting the exploration space by modeling the generation process as a stateless process to avoid combination explosions, 2) reusing the successful payload in modeling; and 3) minimizing the changes on AE samples to correctly assign the rewards in RL learning (which also helps identify the root cause of evasions). As a result, our framework has much higher evasion rates than other off-the-shelf frameworks. Results show it has over 74%–97% evasion rate for two state-of-the-art ML detectors and over 32%–48% evasion rate for commercial AVs in a pure black-box setting. We also demonstrate that the transferability of adversarial attacks among ML-based classifiers is higher than that between ML-based classifiers and commercial AVs.

CCS CONCEPTS

• **Security and privacy → Malware and its mitigation**.

KEYWORDS

malware classification, adversarial learning, reinforcement learning

ACM Reference Format:
Wei Song, Xuezixiang Li, Sadia Afroz, Deepali Garg, Dmitry Kuznetsov, and Heng Yin. 2022. MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware. In *Proceedings of the 2022 ACM Asia Conference on Computer and Communications Security (ASIA CCS '22), May 30-June 3, 2022, Nagasaki, Japan.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3488932.3497768

1 INTRODUCTION

Malware attacks continue to be one of the most pressing security issues users face today. Recent research showed that during the first nine months of 2019, at least 7.2 billion malware attacks and 151.9 million ransomware attacks have been reported.[1] The attack rate hit a new high with the COVID-19 pandemic.[2] The traditional signature-based methods cannot keep up with this rampant inflation of novel malware. Hence commercial antivirus companies started using machine learning [7, 54]. Machine learning-based detectors are scalable and efficient at protecting against the huge influx of malware. Since the first paper in 2001 on detecting malware using machine learning [49], there has been an explosion of academic research papers on predicting malicious content using machine learning. Many of them flaunting high accuracy and being able to detect new malware unseen during training [5, 14, 43, 44, 48]. However, research has also demonstrated that machine-learning-based detectors can be easily evaded by making even trivial changes to malware [3, 4, 10, 13, 18–20, 22–24, 28, 30, 35, 37, 41, 45, 46, 52, 56, 57]. Even commercial antivirus systems, such as Cylance, have been shown to be susceptible to trivial adversarial attacks [6].

Since 2014, there have been more than 1400 papers on adversarial attacks and defense[3]. However, these works mainly focus on the image domain. The adversarial attacks on malware samples are different from attacks in the image domain. For images, adversaries can alter the value of any pixel, as long as the changes are bounded with a L_p-norm. But for malware samples, a one-byte change can break the format of a valid PE, or break the original malicious

[1] https://www.msspalert.com/cybersecurity-research/sonicwall-research-malware-attacks-2019/
[2] https://labs.bitdefender.com/2020/04/coronavirus-themed-threat-reports-havent-flattened-the-curve/
[3] https://nicholas.carlini.com/writing/2019/all-adversarial-example-papers.html

ASIA CCS '22, May 30-June 3, 2022, Nagasaki, Japan

ACM ISBN 978-1-4503-9140-5/22/05.
https://doi.org/10.1145/3488932.3497768

functionality. This is why adversaries usually do not directly modify the raw bytes of the PE file. Instead, they construct a set of actions. Each action can transform the malware sample without breaking the original functionality. For example, the action could be adding a new redundant section (adding a new entry in the section table and appending the section content at the end). Then, the adversarial example generation problem is transformed into finding correct actions and corresponding contents that lead to misclassification.

Adversarial attacks against static malware classifiers are not new. Researchers have proposed a variety of techniques to generate evasive samples (the terms "evasive samples" and "adversarial examples" are used exchangeably in this paper), including genetic programming [17, 56], Monte Carlo tree search [42], and deep Q-learning [4]. Although some of these attempts [42, 56] are dealing with PDF malware and source code authorship respectively, the general algorithms can be applied to PE malware.

Although these techniques have been demonstrated to be effective, we have identified several limitations. First, the existing techniques model the AE generation in a stateful manner. However, it is hard to train a stateful model given that the search space is huge. Therefore, we chose a stateless modeling approach, which can significantly reduce the learning difficulty and result in more productive AE generation. Second, most of the existing techniques only learn a decision-making policy that decides what action to take in the next step and randomly picks content if needed. We found that contents are as important as actions. If the content associated with certain action has proved to be useful in generating one AE, the same action-content pair will likely be useful for some other samples as well. Third, when an AE is successfully generated, these techniques will assign rewards to all the actions involved. In our evaluation, we observe that when AEs are generated, only a small number of actions applied to these AE are essential. The rest are redundant and can be removed. Assigning rewards to these redundant actions will confuse the learning process.

Based on these insights, we propose a reinforcement learning framework, called MAB-Malware to generate AEs for PE malware. Its name comes from our modeling of the AE generation problem as a classic multi-armed bandit (MAB) problem.

In summary, the contributions of this paper are as follows:

- We examine the existing algorithms in blackbox AE generation and provide key insights for stateful vs. stateless modeling, content-aware vs. content-agnostic modeling, and redundant vs. essential actions.
- We argue that a stateless and content-aware modeling is more suitable for generating adversarial PE malware, and an action minimization process is essential.
- To meet these design choices, we propose and implement a novel MAB-based reinforcement learning framework for generating adversarial PE malware.
- We conduct an extensive evaluation on two popular machine learning models and three commercial AV engines. MAB-Malware outperforms the existing blackbox AE generation algorithms by large margins.
- Based on our action minimization, we further look into the root cause of these evasions. Our experiment results suggest the static classifiers in the commercial AV engines are vulnerable to trivial changes to malware samples.

To facilitate the follow-up research on this topic, we released the source code of our framework in a GitHub repository[4].

2 PROBLEM

2.1 Threat Model

We follow the study by Carlini et al. [8] to describe our threat model, from three aspects: adversarial goal, adversarial capabilities, and adversarial knowledge.

Adversarial Goal. The adversary's goal is to manipulate malware samples to evade the detection of static PE malware classifiers. Other types of malware like PDF malware or Android malware are not within the scope of this study. This is an untargeted attack because we only consider a binary classification (benign or malicious) not specific malware families in this classification task and we are only interested in causing the malicious samples to be classified as benign.

Adversarial Capabilities. In this work, we assume that the adversary does not have access to the training phase of the malware classifiers. For instance, the adversary cannot inject poisonous data into the training dataset.

Also, the adversary cannot arbitrarily change the input data. In most scenarios of adversarial attacks, such as image recognition, the adversary is required to make only "small" changes to the original sample to keep the manipulation visually imperceptible. However, when attacking malware classification, the restriction is not on the number or size of changes, but on the preservation of malicious functionality. If "small" changes on a malware sample indeed confuse a malware classifier but prevent the malware from acting maliciously, this manipulation is not considered successful.

Adversarial Knowledge. Based on the knowledge an adversary can obtain, an attack can be divided into two types: 1) whitebox attacks where the adversary has unlimited access to the model; and 2) blackbox attacks where the adversary has no knowledge about the model and can obtain the classification results only through a limited number of attempts. A classification result can be a score or simply a label.

In this work, we consider an adversary with only blackbox access. The adversary does not know anything about the internals of the deployed classifiers, can perform a limited number of attempts to the classifiers, and can observe the classifiers' actions when the samples are considered malicious.

2.2 Problem Definition

In this paper, we focus on three state-of-the-art machine learning classifiers and the static classifiers of 3 top commercial antivirus products. We aim to automatically generate adversarial examples for malware classifiers and explain the root cause of the evasions. The problem can be divided into two sub-problems: adversarial example generation and feature interpretation.

[4] https://github.com/bitsecurerlab/MAB-malware.git

We aim to manipulate a malware sample such that malware classifiers misclassify it as benign, and do not break its malicious functionalities. For whitebox attacks in the image domain, changes to original images are bounded with L_2 and L_∞ norms. It ensures that the pixel changes are imperceptible to humans. However, in the malware classification domain, as long as the binary behaviors remain the same, normal users are unlikely to notice the differences between the original sample and the modified one. That is why previous blackbox attacks [4, 11, 21] on malware do not try to minimize changes when generating AEs. However, we find that the minimal change requirement is still crucial for three main reasons: 1) it reveals which actions and the corresponding payloads are essential to generate evasive samples that can be applied to other samples to create successfully evasive samples; 2) it unveils which feature changes caused the evasion to ensure that the classifier does not rely on superficial features; and 3) it reduces the chance of creating broken binaries. In the blackbox setting, instead of minimizing added noises in feature space, we minimize action sequences applied to generate AEs. It includes removing redundant actions and replacing actions that cause large changes to the features used for detection.

Let $\mathcal{X}$ be a malware dataset, f be a malware classifier that maps a sample $x \in \mathcal{X}$ to a classification label $y \in \{0, 1\}$ (0 represents benign, 1 represents malicious). We implement an action set $\mathcal{A} = \{a_1, a_2, \dots a_n\}$ that can be used to perturb malware samples. We define an objective function for adversarial example generation in (1). An adversarial example $x' = t(x)$ is generated by applying a transformation function t, which is a sequence of actions sampled from set $\mathcal{A}$. $\mathcal{L}(f(t(x)), \bar{y})$ measures the difference between the predicted label of $f(t(x))$ and benign label $\bar{y}$. The transformation function t subjects to the constraint that $t(x)$ does not change the functionality of x, i.e. the functionality difference $\delta(x, t(x))$ before and after transformation equals to 0.

$$\begin{aligned} &\underset{t}{\operatorname{argmin}}\ \mathcal{L}(f(t(x)), \bar{y}), \\ &s.t.\ \ \delta(x, t(x)) = 0 \\ &\qquad\qquad y \neq \bar{y} \end{aligned} \tag{1}$$

3 MOTIVATION

In this section, we first discuss the existing reinforcement learning-based and genetic programming-based approaches on AE generation and their limitations, and then we present our insights that motivate our MAB-based approach. More related works can be found in Section 7.

3.1 Existing Approaches

Deep Q-learning. Anderson et al. [4] propose to apply deep reinforcement learning (RL) to generate AE for PE malware to bypass machine learning models. They first define a set of actions (file mutations), including changing PE headers, appending overlay bytes, packing, and unpacking. Then the agent selects the next action based on a policy and an environmental state. When an evasive sample is generated, all applied actions (including early actions that produce no immediate reward) get promoted for a given state.

Monte Carlo Tree Search. Quiring et al. [42] propose a Monte Carlo Tree Search (MCTS) based approach to mislead the classification of source code authorship. They define a set of actions (code transformation) for changing stylistic patterns. Then they create a Monte Carlo search tree, in which each node represents a variant of the code and each edge represents an action. Then the task of AE generation is converted to a path search problem. The goal is to find a path on the tree that leads to misclassification.

Genetic Programming. Demetrio et al. [17] propose a genetic programming-based approach to generate AEs of PE malware in a black-box attack manner. It formalizes the problem as a constrained minimization problem, to trade-off between the probability of evasion and injected payload size. The fitness function is defined as the sum of confidence scores and injected payload size. In each iteration, it selects variants with the lowest fitness score. Another paper from Xu et al. [56] also uses a genetic programming-based approach to generate adversarial PDF malware.

3.2 Our Insights

While these existing techniques have demonstrated their effectiveness more or less, we observe several key insights, which can motivate us to develop a better technique for AE generation.

Stateful vs. Stateless Modeling. Existing reinforcement learning techniques [4, 17, 42, 56] model the AE generation problem in a stateful way. They try to build a search tree, each node representing a state. The original sample is the root state. Each state has multiple actions to choose from. After applying one action, the sample enters the next state. Existing works attempt to learn a policy to choose the next action in different states. For different states, the action selection strategy is different. However, as the search tree grows, the total number of states increases exponentially. It makes the training of action selection strategies difficult or sometimes impossible.

Our first insight is that we don't need to learn a stateful model for the malware AE generation problem. A model or AV engine classifies a sample as malicious may simply because it matches a specific signature, such as a section name or byte sequence. Before we find the correct action to change that signature, no matter how many irrelevant actions are applied, the sample will be detected for the same reason. Therefore, we don't need to construct the search tree to model different states. It makes the learning task unnecessarily difficult. Instead, we should keep the sample in only one state: non-evasive. Then our job becomes much easier: how to choose the correct action to jump from the same non-evasive state to the desired evasive state. We call it stateless modeling because there is only one state before evasion.

We propose to utilize a classic reinforcement learning model, multi-armed bandit (MAB) [34] to solve the malware AE generation problem. The MAB problem is formally equivalent to a one-state Markov decision process. It has just one state. From that state, it selects candidate actions to apply. Through these selections, it gradually learns the reward probability of each action. It maximizes the total reward by finding the optimal tradeoff between exploration (learning the reward probability of unfamiliar actions) and exploitation (applying the action with the highest average reward).

Content Modeling. Many actions used for manipulating PE need to be associated with some contents. For instance, when adding a

new section, we need to specify what content to be filled in that section. When renaming a section, we need to provide a new section name. Our second insight is that these contents are as important as the actions. If content associated with one action has proved to be useful in generating one AE, the same action-content pair is likely to be useful for other samples.

Most existing works do not take contents into account. They only learn a decision-making policy to decide what action to take in the next step and take random content if required. For example, if the next action to take is "Section Add" according to the policy, they will fill the new section with random content. Our MAB-based framework treats an $< action, content >$ tuple as an integral unit (a slot machine in MAB) for modeling. If the new content is discovered to be useful to generate an adversarial example, it will be saved to be reused for other samples.

Precise Reward Assignment. Reward assignment is essential to all the existing AE generation techniques described above. When an AE is successfully generated, a positive reward is assigned to the corresponding sequence of actions. However, not all actions are essential to the generation of this AE. According to our evaluation in Section 5.4, in most cases, only one or two actions are essential. Therefore, assigning rewards to all the actions involved in an AE generation will lead to a less accurate reinforcement learning model. Hence, our third insight is that we should precisely assign rewards only to the essential actions.

4 METHODOLOGY

4.1 Adversarial Attack as a Multi-armed Bandit Problem

The Multi-Armed Bandit problem [34] is a classic reinforcement learning problem that embodies the exploration-exploitation trade-off dilemma. It is about how to maximize the total reward by allocating limited resources to multiple competing choices. The property of each choice is gradually learned in the process of resource allocation.

Slot machines in a typical casino are also called one-armed bandits [50] because the early machines have large mechanical levers attached to the sides, and they can empty the player's pockets like a thief. When the lever of the machine is pulled, its reward probability is θ. Therefore, a multi-armed bandit can be viewed as multiple slot machines with different reward probabilities. The reward probabilities are unknown to players. A player can observe each machine's reward probability by pulling it. However, the player has limited money. The goal is to maximize the sum of the rewards obtained through a series of lever pulls. The multi-armed bandit can be viewed as a tuple of $< \mathcal{M}, \mathcal{R} >$. $\mathcal{M}$ is a set of slot machines. $\mathcal{R}$ is a set of reward distributions $\{\theta_1, \ldots, \theta_K\}$, each distribution is associated a machine.

In the malware domain, we have many actions that can change the features of a PE binary without altering its functionality. Many actions require payload content to work. For example, adding a new section requires benign content as the content of the new section. Content plays a vital role in attacking machine learning models, because the added content can largely change the byte entropy of the original malware sample in a certain direction. Adversarial attacks can be viewed as a problem of how to choose a serial of action and content pairs to maximize the probability of generating adversarial examples.

We treat the tuple $< action, content >$ as a slot machine M. When M is selected, it will apply *action* to the target binary file using the payload *content*. In our framework, we have two kinds of machines: generic machines and specific machines.

Generic Machine. A generic machine $M_{<action,rand>}$ is a machine, when selected, applies *action* to the target malware sample, with a random content extracted from benign binary files. For example, the OA (overlay Append) generic machine $M_{<OA,rand>}$ extracts a random section content from a random benign binary and appends the content to the target malware sample as overlay data. The reason for creating a generic machines is that at the beginning of the attack, we do not know which content is effective. By choosing a generic machine, we can explore different benign content for a certain action.

Specific Machine. A specific machine $M_{<action,X>}$ is a machine, when selected, applies *action* to the target malware sample, with specific content X. After we generate an adversarial example x' by pulling a generic machine $M_{<action,rand>}$, if M machine is essential (see details in Section 4.3), we will create a specific machine $M_{<action,X>}$. The content X is the specific content used in generating x'. When $M_{<action,X>}$ is selected by other malware samples, the specific content X is exploited to generate more adversarial examples.

The workflow of our framework MAB-Malware is shown in Figure 1. It consists of two main modules: the Binary Rewriter and the Action Minimizer. The Binary Rewriter utilizes Thompson sampling to select machines from the machine set $\mathcal{M}$ and rewrites original malware sample x to generate adversarial example x'. The Action Minimizer removes *redundant machines* to generate adversarial samples x'_{min} with minimal feature changes. *Redundant machines* are machines selected by the Rewriter in the generation of adversarial example x', but later we find that without them, the rest machines can still generate an adversarial example. That is because, at the beginning of the attack, the property of each machine is unclear. Rewriter needs to select these redundant machines to infer their reward probability. The rest necessary machines are called *essential machines*.

Our problem can be viewed as a tuple of $< \mathcal{M}, \mathcal{R} >$. $\mathcal{M}$ is a set of machines (including generic machines and specific machines), each machine M_i refers to pulling one slot machine $< action_i, content_i >$. $\mathcal{R}$ is a set of reward distributions $\{\theta_1, \ldots, \theta_K\}$ (suppose we have K slot machines), each distribution is associated an action. The reward distribution θ_i of each machine is unknown. We have a limited number of attempts to pull these machines. The goal is to maximize the reward through a series of pulls.

Thompson Sampling. In our task, we face a delayed feedback problem. When evaluating the static modules of commercial antivirus systems, we need to copy the generated sample to the virtual machine with antivirus and wait for the scanning result. This process takes seconds, even minutes for certain AVs. If we adopt a deterministic algorithm, such as upper confidence bounds, it will always select the one with the highest values before the result returns. It causes inefficient trials because of outdated information. To address this issue, we use Thompson sampling algorithm [55],

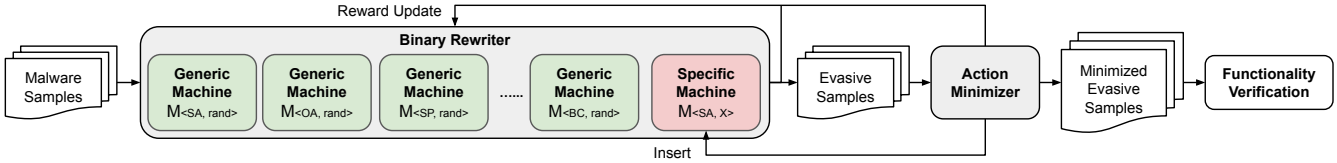


Figure 1: Workflow

which is more robust than deterministic algorithms in the delayed feedback environment [12].

We assume the reward follows a beta distribution [1] specific to that machine. The beta distribution is a continuous probability distribution parameterized by two positive parameters, denoted by α and β, i.e. $M \sim \text{Beta}(\alpha, \beta)$. α and β correspond to the counts of success or fail respectively.

At each action selection iteration, for each machine, we sample a value from its $Beta(\theta; \alpha, \beta)$ distribution and select the machine with the highest value as the next machine. When the α and β values of a machine are small, the uncertainty of M is high. Even if this average reward is lower than other machines, it still has a relatively high possibility to get a large value. In this way, new machines are more likely to be selected for exploration. After several trials, the α and β value of that machine becom large, and the uncertainty decreases. In this way, machines with high average rewards are selected for exploitation.

Reward Propagation. When a machine is created, we set α=1, β=1 for each machine. For every machine that is selected by Rewriter but fails to generate the adversarial example, we increase its β by 1. When an adversarial example is generated and minimized, for every essential machine, we increase the α by 1. If the machine is a generic machine, we also create new specific machines using its specific content (with α=1, β=1). If an essential machine is a specific machine, we also increase the α of its corresponding generic machine that it derives from, to encourage the exploration for certain types of actions.

4.2 Binary Rewriter

4.2.1 Action Set and Features. We implemented 13 actions in Table 1. Each action manipulates a set of features that a classifier may use to detect malware (shown in Table 2).

Macro-actions. We reimplemented actions proposed by Anderson et al. [4] using the pefile library and fix many corner cases that may break the functionality. We also adopt a code randomization action (CR) from Pappas et al. [39]. It is a defense method originally proposed to prevent Return Oriented Programming (ROP) attacks.

Micro-actions. If action a affects k features $\{f_1, f_2, \ldots f_k\}$ of the malware sample, then an action that affects only a subset of these features is the micro-action of a. We have implemented 5 micro-actions: OA1, SP1, SA1, SR1, and CP1. They are similar to the corresponding macro-actions but with minimized feature changes. Take SA1 as an example. Similar to SA, SA1 also adds a new section entry in the header, but it only adds a 1-byte section. SA adds a lot of benign content, and greatly changes the byte entropy of the original binary, while SA1 does not. Therefore, SA affects features $\{F_1, F_3, F_{10}\}$, SA1 only affect features $\{F_1, F_3\}$. By looking up Table 2, you can see that *SA*'s micro-actions also include *OA*1 and *OA*, affecting features $\{F_1\}$ and $\{F_1, F_10\}$ respectively. *OA* is also considered a macro-action during the attack. So, we can see that micro-action is a relative concept.

Algorithm 1 Adversarial Attack

Input: malware sample set X
Output: adversarial example set X_a

initialize($\mathcal{M}$)
$X_a \leftarrow []$
for all $x \in X$ **do**
  $list\_M \leftarrow []$
  for all $attempt\_idx \leftarrow 1$ to N **do**
    $M \leftarrow$ max(betaSampling($\mathcal{M}$))
    $x' \leftarrow$ apply(x, M)
    $list\_M$.add(M)
    if isEvasive(x') **then**
      $x'_{min}, list\_M_{min} \leftarrow$ minimize($x, list\_M$)
      X_a.add(x'_{min})
      for all $M' \in list\_M_{min}$ **do**
        incAlpha(M')
        if isGeneric(M') **then**
          $M'_s \leftarrow$ createSpecificMachine(M')
          $\mathcal{M}$.add(M'_s)
        else
          $M'_g \leftarrow$ getParentGeneric(M')
          incAlpha(M'_g)
        end if
      end for
      break
    else
      incBeta(M)
    end if
  end for
end for
return X_a

4.2.2 Workflow of Binary Rewriter. Algorithm 1 summarizes the workflow of Binary Rewriter. For a set of malware samples X, our goal is to generate a set of adversarial samples X_a. First, we initialize $\mathcal{M}$ by creating 8 generic machines, and each machine's α value and β value are set to 1. To select the next machine, for each machine, we sample a value from its β distribution and select the machine with the highest value. Then we apply the corresponding machine to x. We apply a serial of machines iteratively until we get an evasive

Table 1: Action Set.

Type	Abbr	Name	Description
Macro	OA	Overlay Append	Appends benign contents at the end of a binary
	SP	Section Append	Appends random bytes to the unused space between sections.
	SA	Section Add	Adds a new section with benign contents.
	SR	Section Rename	Change the section name to a name in benign binaries.
	RC	Remove Certificate	Zero out the signed certificate of a binary.
	RD	Remove Debug	Zero out the debug information in a binary.
	BC	Break Checksum	Zero out the checksum value in the optional header.
	CR	Code Randomization	Replace instruction sequence with semantically equivalent one
Micro	OA1	Overlay Append 1 Byte	Appends 1 byte at the end of a binary
	SP1	Section Append 1 Byte	Appends 1 byte to the unused space at the end of a section.
	SA1	Section Add 1 Byte	Adds a new section with 1 byte content.
	SR1	Section Rename 1 Byte	Change 1 byte of a section name.
	CP1	Code Section Append 1 Byte	Appends 1 byte to the unused space at the end of the code section.

Table 2: Affected Features by Actions.

		CR	OA	SP	SA	SR	RC	RD	BC	OA1	SP1	SA1	SR1	CP1
Hash-Based Signatures	F_1: File Hash	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓
	F_2: Section Hash	✓		✓				✓			✓			✓
Rule-based Signatures	F_3: Section Count				✓							✓		
	F_4: Section Name					✓							✓	
	F_5: Section Padding			✓										
	F_6: Debug Info							✓						
	F_7: Checksum								✓					
	F_8: Certificate						✓							
	F_9: Code Sequence	✓												
Byte Entropy	F_{10}: Byte Entropy		✓		✓									

sample or exceed the total number of attempts N. When an evasive sample is generated, we further use Action Minimizer to remove redundant machines. For the remaining machines $list\_M_{min}$, we first increase their α by 1. If it is a generic machine, we create a new specific machine M'_s. If it is a specific machine, we increase the α value of its parent generic machine, which has the same action type but with random content. For failed machine, we increase the value of β by 1.

4.3 Action Minimizer

The Action Minimizer removes redundant actions and uses micro-actions to replace macro-actions, to produce a "minimized" evasive sample that only changes minimal features.

As shown in Figure 2, the original malware sample x resides in the malicious region of the feature space. We perform a sequence of single actions a_1, a_2 and a_3 until the generated sample x_{123} successfully reaches the benign region. x_{123} is an adversarial example. In the minimization phase, first, we remove useless actions. The action a_2 is essential, because by removing action a_2, the generated sample x_{13} is no longer evasive. The action a_1 is useless because by removing action a_1, the generated sample x_{23} has no effect in the classifier's decision. Then we disentangle these actions into micro ones (i.e., actions that cause smaller changes). a_2 can be replaced with micro-actions a'_2. Action a_3 can be replaced with micro-actions a'_3 or a''_3. We generate three samples $x_{2'3}$, $x_{2'3'}$ and $x_{2'3''}$. Finally, we have an adversarial sample $x_{2'3''}$ with a minimized action sequence

Algorithm 2 Minimize

Input: malware sample x, applied machines $list\_M$
Output: minimized AE x'_{min}, minimized machines $list\_M_{min}$

$list\_M_{min} \leftarrow list\_M$
for all $M \in List\_M$ **do**
  $list\_M' \leftarrow List\_M_{min} - M$
  $x' \leftarrow \text{apply}(x, list\_M')$
  if isEvasive(x') **then**
    $list\_M_{min} \leftarrow List\_M'$
    $x'_{min} \leftarrow x'$
  else
    $list\_micro \leftarrow$ get_micro_actions(M)
    for all $M_{mic} \in list\_micro$ **do**
      $list\_M' \leftarrow List\_M_{min} - M + M_{mic}$
      $x' \leftarrow \text{apply}(x, list\_M')$
      if isEvasive(x') **then**
        $list\_M_{min} \leftarrow List\_M'$
        $x'_{min} \leftarrow x'$
        break
      end if
    end for
  end if
end for
return $x'_{min}, list\_M_{min}$

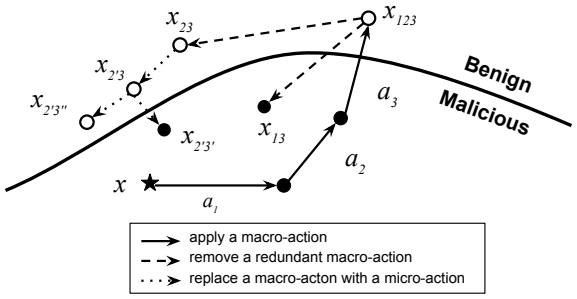


Figure 2: An example of action minimization.

(a_2', a_3''). So a positive reward can be precisely assigned to these essential actions a_2' and a_3''.

As shown in Algorithm 2, for each machine M in the applied machines $list\_M$, we try to remove it and apply the new sequence $list\_M'$ to the original sample x to generate x'. If x' is still evasive, it means that the machine M action is redundant. We can permanently delete M from $list\_M$. Otherwise, we will find that all micro-actions $list\_micro$ that only change the subset of features changed by M. Then we try to replace M with each micro-action M_{mic} in $list\_micro$ and apply the new sequence to generate x'. If x' is still evasive, then we use M_{mic} to permanently replace M. If we find that M cannot be removed or replaced, it means that machine M is essential. In this way, we can delete redundant feature changes and find the essential actions.

For example, to generate an evasive sample x' for x, Binary Rewriter has applied 5 actions: CR, OA, SP, BC, SA. Action Minimizer will check every machine to determine if it can be removed. It finds that the first 4 actions are redundant, only the last action SA cannot be removed. SA (add a new section) changes 3 features of the original binary file. It changes the file hash, creates a new entry in the section table, and adds a content block to the end of the file. Correspondingly, SA has 3 micro-actions: OA1, SA1, and OA. Each of them only changes one feature. If we replace SA with any micro-action, we will remove redundant feature changes. In this way, we can generate the minimized adversarial example x'_{min}.

From a defender's point of view, we also would like to understand how an evasion happens, where the weakest point of the classifiers is. The action minimization of evasive samples provides a good opportunity to infer that information. Figure 3 shows how we break macro-actions into micro-actions. Take the action Section Append (SP) as an example. First, by looking up Table 2, SP changes feature $F = \{F_1, F_2, F_5\}$ (File Hash, Section Hash and Section padding). The actions that only change a subset of F are OA1 that changes $\{F_1\}$ and SP1 that changes $\{F_1, F_2\}$. Starting from the minimum change, we try to replace SP with OA1 and check if the file is still evasive. If so, we can conclude that the evasion is caused by the change of file hash (F_1). If not, we continue to replace SP with SP1. If successful, the evasion is caused by the change of section hash (F_2). Otherwise, the evasion is caused by the change of signatures in section padding content (F_5).

If SP ← OA1, then feature = **File Hash**
Else If SP ← SP1, then feature = **Section Hash**
Else feature = **Section Padding**

(a) SP

if SA ← OA1, then feature = **File Hash**
Else If SA ← SA1, then feature = **Section Count**
Else If SA ← OA, then feature = **Byte Entropy**
Else feature = **Section Count** & **Byte Entropy**

(b) SA

If CR ← OA1, then feature = **File Hash**
Else If CR ← CP1, then feature = **Section Hash**
Else feature = **Code Sequence**

(c) CR

If RD ← OA1, then feature = **File Hash**
Else If RD ← CP1, then feature = **Section Hash**
Else feature = **Debug Information**

(d) RD

If SR ← OA1, then feature = **File Hash**
Else If SR ← SR1, then feature = **Section Name**
Else feature = Part of **Secton Name**

(e) SR

If OA ← OA1, then feature = **File Hash**
Else feature = **Byte Entropy**

(f) OA

If BC ← OA1, then feature = **File Hash**
Else feature = **Checksum**

(g) BC

If RC ← OA1, then feature = **File Hash**
Else feature = **Certificate**

(h) RC

Figure 3: Decision rules are used to map actions to feature space

5 EVALUATION

5.1 Experiment Setup

Dataset: In this paper, we generate adversarial examples for Windows PE binaries. To ensure the executability and functionality of the generated samples, the format and constraints of PE files must remain intact. To guarantee the quality of malware samples, we randomly select 5000 samples from VirusTotal that meet the following requirements: 1) more than 80% antivirus engines of VirusTotal label them malicious; and 2) the execution of those samples in a Cuckoo sandbox shows malicious behavior.

Setup: The experiments are performed on 20 virtual machines of the Microsoft Azure cloud platform. The configuration of each virtual machine is Standard D2s v3 (2 vcpus, 8 GiB memory). For all the antivirus software under testing, free versions and default settings are used. We choose three top commercial antivirus products for blackbox testing, which are anonymized as AV1, AV2, and AV3. Each antivirus is installed on an Azure virtual machine with Windows 7. To ensure the malware will not infect other machines in the network and the stability and reproducibility of our experiments, all network traffic is routed to an InetSim instance on the host machine to provide simulated network services.

We choose the following models as our target models:

- **EMBER** [5] is an open-source machine-learning-based classifier that uses a tree-based classifier model LightGBM to detect malware. It generates a 2350-dimensional feature vector for each sample consisting of two main types of features: raw features (e.g. ByteHistogram, ByteEntropyHistogram, Strings) and parsed features (e.g. GeneralFileInfo, HeaderFileInfo, SectionInfo, ImportsInfo, ExportsInfo). We use the model provided in MLSEC2019 (Machine Learning Security Evasion Competition) [38].
- **MalConv** [43] is a malware detection model that uses a convolutional neural network to learn knowledge directly on the raw bytes of malware samples. We also use the model provided in MLSEC2019 [38].
- **Commercial AVs.** We also test the static classifiers of 3 top commercial antivirus systems.

5.2 Adversarial Example Generation

Comparison with Other Off-the-Shelf Frameworks. We compare our MAB-Malware with other two off-the-shelf attack frameworks: SecML-Malware[5] and Gym-Malware[6]. SecML-Malware is a plugin for the SecML Python library. It contains many kinds of attacks, including black-box attacks with hard labels. We utilize its genetic programming-based black-box attack (GAMMA) in this experiment. Gym-Malware is a reinforcement learning-based malware manipulation environment using OpenAI's gym. Its agents learn how to manipulate PE files to bypass AV based on a reward provided by taking specific manipulation actions.

We measure the evasion rate for two machine learning-based models, MalConv and EMBER. Evasion Rate is defined as: $R_e = N_e/N_d$, where N_e is the total number of successful evasive samples, and N_d is the total number of original samples that can be detected by the target model. For a fair comparison, we use the same dataset (5000 samples from VirusTotal) and MalConv and EMBER models (from the Machine Learning Static Evasion Competition 2019 [38].) We run each experiment five times to calculate an average.

From Figure 4, we can see that MAB-Malware performs much better than the other approaches. It can generate AEs for 97.72% samples to evade MalConv, 74.4% samples to evade EMBER. The evasion rate of SecML-Malware (GAMMA-hard label) is 63.6% and 50.0% respectively. Gym-Malware has the lowest evasion rate (27.1% and 12.3%). The evasion rate is almost identical to random action selection using its own action set (28.8% and 12.1%). This indicates that this deep Q-learning model does not learn meaningful knowledge to guide the evasion. The reason is that the problem modeling creates an exponentially large search space. And without action minimization, the reward assignment is chaotic. Within 60 trials, it cannot explore enough in such a large space and learn meaningful policy to select the correct action and corresponding content.

Comparison with Other Algorithms. The action sets of these three frameworks are different. SecML-Malware only uses benign content injection and appending. Gym-Malware's operation set is similar to ours, but it also includes packing and unpacking. As a result, we cannot see the effectiveness of the MAB-based action selection algorithm.

So in this experiment, we only use our own action set and change the action selection algorithms. The baseline is random selection. Then we compare our method with the other reinforcement learning algorithm. In the experiment above, we have already shown that the Q-learning models cannot directly improve the evasion rate over random selection. In this experiment, we further implement another MCTS-based reinforcement learning algorithm. Quiring et al. [42] proposed an MCTS-based approach to mislead the classification of source code authorship. Because their code cannot be directly applied to malware classifiers, we borrowed their idea and reimplemented it for malware classification.

It can be seen from Figure 4 that in the same action set, our MAB algorithm greatly improves the evasion rate compared to random action selection, while the MCTS algorithm hardly provides any improvement. Existing frameworks model AE generation in a stateful way and try to find the best state path leading to escape. This makes it difficult to train in a large search space. In addition, the existing framework does not have a mechanism to effectively reuse the successful payload.

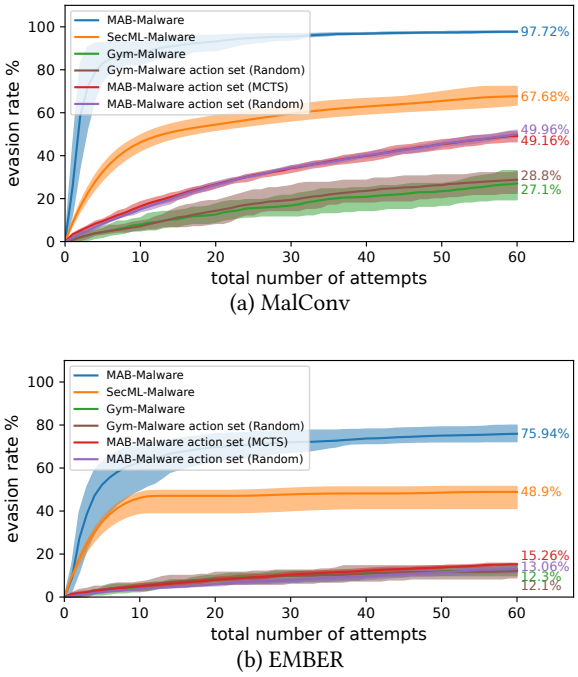


Figure 4: Evasion Results

Attacking Commercial Antivirus. We also test our framework on three commercial antivirus engines. The throughputs of commercial AV engines are much lower than machine learning classifiers. We need to copy a lot of generated samples into the virtual machine with a particular AV installed and wait for AV engines to scan them to get labels. It usually takes seconds or even minutes to get the result. As a result, we only use 1000 samples for this experiment.

[5]https://github.com/zangobot/secml_malware.git
[6]https://github.com/endgameinc/gym-malware.git

Table 3: Evasion Result on Antivirus.

Antivirus	Frameworks	
	SecML-Malware	MAB-Malware
AV1	5.61%	**31.99%**
AV2	11.40%	**46.2%**
AV3	12.75%	**48.3%**

Also, we do not compare Gym-malware since it cannot finish the experiment within a reasonable time frame.

As shown in Table 3, SecML-Malware only achieves 5% - 12% evasion rate for all AVs, while MAB-Malware achieves 31% - 48% evasion rate. It shows the advantages of MAB-Malware in generating adversarial examples in a pure blackbox setting.

Number of bytes changed. The Action Minimizer ensures that the minimized evasive samples only change minimal content to flip the classification label. So by checking how many bytes we need to change, we can infer the robustness of different malware classifiers. To measure the difference between the minimized evasive example and the original malware, we compute the total number of bytes appended or modified by our framework.

By positioning the samples in a line sorted by byte changes (Figure 5), we notice that:

- By only changing one byte of the original malware, we can generate 33 for AV1, 32 for AV2, 3 for AV3.
- Machine learning models are not vulnerable to small changes. However, it does not mean that ML models are more robust than commercial AVs. From the previous evasion rate results, we can see that using our framework, ML models are easier to evade than commercial AVs.

5.3 Testing Functionality Preservation

We found that the action set in Gym-Malware, which is implemented using LIEF [31] library, is not safe. According to our experiment result in Table 4, more than 60% of the generated binaries after a single action cannot be executed, or behave differently. To solve this problem, we carefully reimplement most actions using the pefile [9] library to avoid many corner cases that may lead to a broken binary. For example, before adding a new section, we check whether there is enough space between the last section header entry and the first section.

We implement our own action set using the pefile library whereas the Gym-Malware rewrites binaries using the LIEF library. We noticed that rewriting a binary with the LIEF library can cause unnecessary changes to the binary that can sometimes result in broken files, thus destroying the functionality of the original malware samples. To compare our actions with the actions from Gym-Malware, we randomly select 50 malware samples from our dataset, create adversarial samples by applying different actions, analyze all variants in the Cuckoo sandbox, and compare the behaviors with the original samples.

From Table 4 we can see that except for the Overlay Append action, most actions in the Gym-Malware framework cause 63.24% of the rewritten samples to lose functionality. In contrast, only less than 8% of the rewritten samples using our actions create broken binaries.

Table 4: Functionality Preservation Rate of the Actions

Actions	Functional Rate	
	Gym-Malware Actions	MAB-Malware Actions
(OA) Overlay Append	45/48 (93.75%)	46/48 (95.83%)
(SP) Section Append	11/47 (23.40%)	42/43 (97.67%)
(SA) Section Add	11/47 (23.40%)	39/42 (92.86%)
(SR) Section Rename	11/47 (23.40%)	42/43 (97.67%)
(RC) Remove Certificate	1/3 (33.33%)	3/3 (100.00%)
(RD) Remove Debug	5/13 (38.46%)	13/13 (100.00%)
(BC) Break Checksum	9/48 (18.75%)	32/33 (96.97%)
Average	**93/253 (36.76%)**	**217/225 (96.44%)**

5.4 Explanation

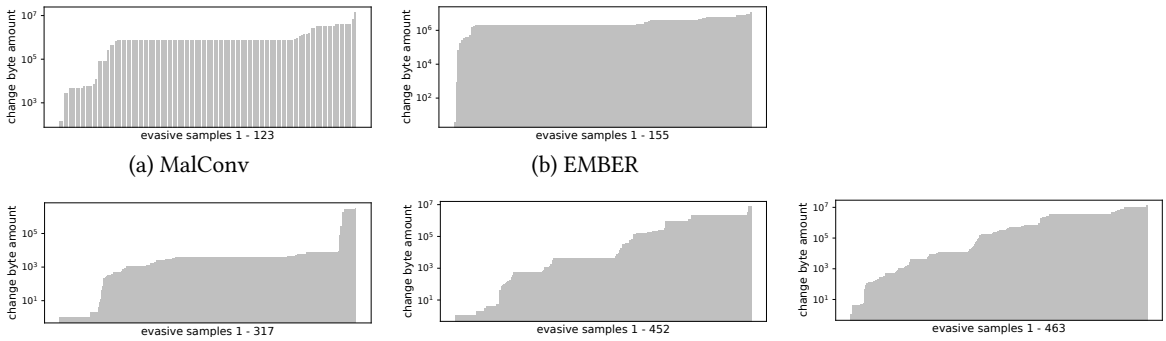
Understanding why an evasion happens can help improve the robustness of a classifier against adversarial attacks. For each evasive sample, the Action Minimizer first removes all redundant actions and uses micro-actions to replace the macro-actions. We summarize the most frequent action sequence combination is Figure 6. According to the rules in Figure 3, we can infer the root cause of each evasion, shown in Figure 7. We found that:

- For two machine learning-based classifiers, the most important action is Overlay Append (OA). Other actions that only change a few bytes have almost no effect on them. It shows that the change in byte entropy is the root cause of the evasions.
- The Section Add 1 Byte (SA1) action plays a significant role in evading all AVs. It indicates that all AVs utilize section count as an important feature for detecting malware.
- Comparing to AV2 and AV3, AV1 is also vulnerable to the Code Section Append 1 Byte (CP1) action. CP1 alters the hash of the code section. It indicates AV1 uses code section hash as an important feature for detecting malware.
- The Section Rename 1 Byte (SR1) action itself can generate many adversarial examples for AV2. SR1 changes one byte of one section name. It indicates that AV2 relies heavily on the section name for detecting malware.
- Comparing with AV2 and AV3, the Section Add (SA) action and the Overlay Append (OA) action have almost no effect on AV1. SA and OA greatly change the byte entropy of the original malware samples. It indicates that AV2 and AV3 integrate some machine learning models in static detection. And AV1 mainly uses the signature-based approach to detect malware.

5.5 Transferability

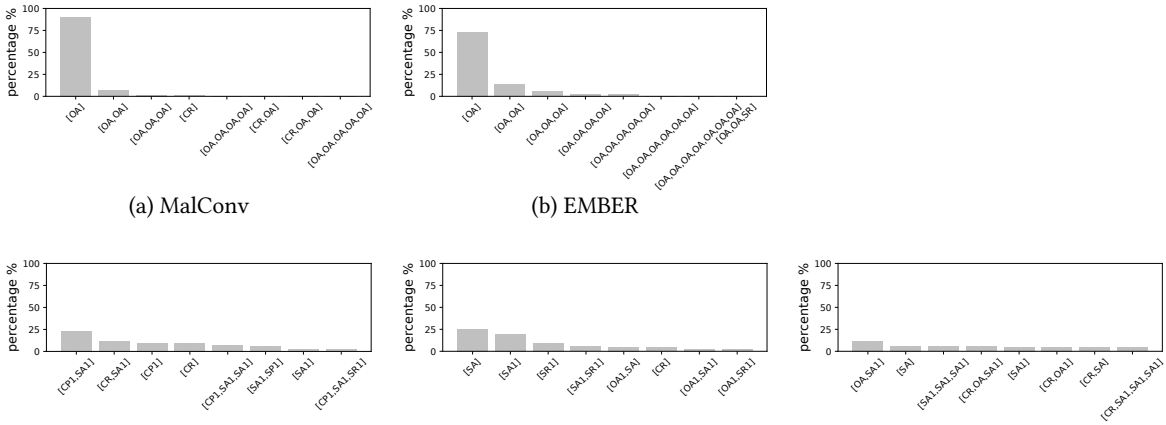
Transferability refers to the property that allows an adversarial sample that can evade one model can also evade other similar models. If the adversarial malware samples are transferable, then evading one malware detector would be enough to evade all malware detectors.

Figure 8 shows the percentage of evasive samples generated for one classifier that can also evade other classifiers. The number in the cell (model A, model B) shows the percentage of evasive samples generated for model A can also evade model B. We noticed:



(a) MalConv (b) EMBER (c) AV1 (d) AV2 (e) AV3

Figure 5: Number of Changed Bytes of Adversarial Examples.



(a) MalConv (b) EMBER (c) AV1 (d) AV2 (e) AV3

Figure 6: Action Sequences for Adversarial Examples

- The transferability between machine learning models is quite high, although EMBER and MalConv are trained on different features (2350-dimensional extracted features vs. raw bytes), and the architecture is different (decision tree vs. neural network).
- Both AV2 and AV3 utilize machine learning models and consider Section count as an important feature. So the transferability between AV2 and AV3 is relatively higher than others.

6 DISCUSSIONS

Triviality of Defense. The triviality of the defense depends on the type of attack. To defend the overlay append attack, the defender can ignore the overlay data when training models. To defend against the SA attack, the defender can lower the importance of benign features in models, and only consider malware features. To defend against RD, SR, and BC attacks, defenders should avoid using fragile patterns as malware features. However, completely ignoring the trivial features can reduce the accuracy of a malware detector. The code randomization (CR) attack is hard to defend because the defender cannot locate the small snippet of binary that is randomized.

Recommendation for Researchers. We demonstrate how adversarial examples can be used to explain a complex blackbox system. When training malware classifiers, researchers should use explanation techniques to understand the behavior of the classifiers and check if the learned features are fragile features that can be easily evaded or if they conflict with expert knowledge. We also argue that for security applications, demonstrating harm to real users is crucial to understanding the real ramification of an attack.

The generality of our evasive techniques. First, our proposed framework conducts blackbox attacks against classifiers. Unlike whitebox attacks, blackbox attacks do not require knowledge of the architecture and parameters of the target classifier. Theoretically, our approach can be used on any malware classifier, as long as the classifier returns a label for testing samples. Second, we attacked 5 representative malware detectors of diverse techniques, including a decision tree-based model (EMBER), a deep learning model (MalConv), and 3 commercial AV engines from top-level AV companies.

Figure 7: Feature changes that cause evasion.

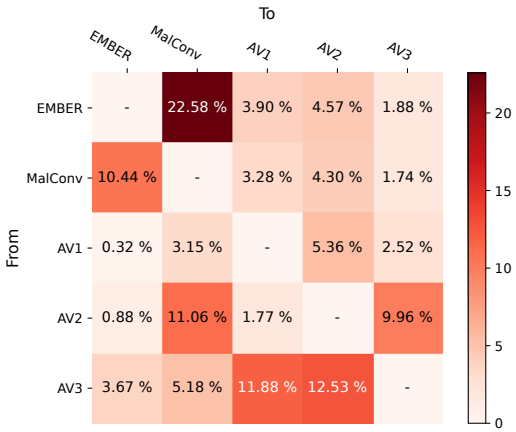


Figure 8: Transferability of Adversarial Samples

The significant evasion rate improvement of these detectors proves the generality of our method.

Mitigation using Dynamic Detection Our solution cannot bypass dynamic detectors, but we argue that dynamic evasion is another research topic. Static evasion itself is an important research direction because it provides a defense before users execute potentially dangerous programs. This is why ML-based static classifiers, such as EMBER and MalConv, increasingly attract attention in the security community.

7 RELATED WORK

Adversarial attacks on machine learning are a rapidly growing field. Since 2014, there have been more than 1400 papers on adversarial attacks and defense.[7] However, only about 42 papers focused on the malware domain, the rest focuses on the image domain. These works performed attacks on Android malware [18, 20, 41, 57], PDF malware [13, 19, 37, 56], Windows malware (PE files) [3, 4, 22–24, 28, 30, 46, 52], IoT malware [2] and Flash-based malware [36]. We compare ourselves with the papers on attacking Windows malware detectors (See Table 5). In this domain, only three papers performed blackbox attacks [4, 11, 21] where an adversary has only external access to the malware detector and also tested their approach against commercial AVs. Ceschin et al. [11] developed a white-box attack using open-source models and then submitted the examples to VirusTotal. We refrained from submitting adversarial samples to VirusTotal because many AVs use these samples to retrain their model and the adversarial samples can cause model poisoning. Fleshman et al. [21] performed blackbox random attacks against four commercial AVs. Anderson et al. [4] propose a reinforcement learning framework gym-malware to perform blackbox attacks against malware classifier EMBER. However, their method only shows about 15% improvement over random selection. Our attack demonstrates a much higher evasion rate against commercial AVs. We also perform an in-depth analysis of the AVs to understand why evasion was successful. Ashkenazy et al. [6] performed a targeted attack against Cylance, a machine learning-based malware detector, which cannot be generalized to other AVs.

Only two papers verified the functionalities of the adversarial malware samples using the Cuckoo sandbox [11, 47]. However, none of them used adversarial examples to interpret how anti-virus systems work. Our work is the first to generate minimized adversarial examples that can be used for blackbox interpretation of anti-virus systems.

Many papers propose blackbox attacks to generate AEs for malware classifiers. However, most of them still assume that they can get the confidence score from the target classifier. For example, Lucas et al. [33] propose an attack that adopts in-place randomization and code displacement to transform malware. It follows a general hill-climbing approach and applies a serial of transformations. However, it still relies on the confidence score to decide whether to keep the transformations.

8 CONCLUSION

We design a reinforcement learning guided framework MAB-Malware to perform adversarial attacks on state-of-the-art machine learning models for malware classification and top commercial antivirus static classifiers. We model the action selection problem as a multi-armed bandit problem. During the attack, MAB-Malware infers the property of actions and dynamically adding new machines with unseen successful content. It finds an optimal balance between exploitation and exploration to maximize the evasion rate within limited trials. The Action Minimization module of MAB-Malware filters out the actions that are ineffective for adversarial sample generation and only change minimal features, so our framework can also be used to explain why evasion occurs. For each commercial antivirus system, we compute the effectiveness of each action and the key features that cause evasions. Our results show that MAB-Malware largely improves the evasion rate over other reinforcement learning frameworks and that some of the adversarial attacks are transferable between different antivirus systems that are similar to one another.

[7] https://nicholas.carlini.com/writing/2019/all-adversarial-example-papers.html

ACKNOWLEDGMENTS

We appreciate the anonymous reviewers for the valuable comments. The research work is partly supported by National Science Foundation under grant No. 1719175 and a gift fund from Avast Inc. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation and Avast Inc.

REFERENCES

[1] [n. d.]. Beta distribution. https://en.wikipedia.org/wiki/Beta_distribution.
[2] Ahmed Abusnaina, Aminollah Khormali, Hisham Alasmary, Jeman Park, Afsah Anwar, Ulku Meteriz, and Aziz Mohaisen. 2019. Examining Adversarial Learning against Graph-based IoT Malware Detection Systems. *arXiv preprint arXiv:1902.04416* (2019).
[3] Abdullah Al-Dujaili, Alex Huang, Erik Hemberg, and Una-May O'Reilly. 2018. Adversarial deep learning for robust detection of binary encoded malware. In *2018 IEEE Security and Privacy Workshops (SPW)*. IEEE, 76–82.
[4] Hyrum S Anderson, Anant Kharkar, Bobby Filar, David Evans, and Phil Roth. 2018. Learning to evade static PE machine learning malware models via reinforcement learning. *arXiv preprint arXiv:1801.08917* (2018).
[5] Hyrum S Anderson and Phil Roth. 2018. Ember: an open dataset for training static PE malware machine learning models. *arXiv preprint arXiv:1804.04637* (2018).
[6] Adi Ashkenazy and Shahar Zini. 2019. Cylance, I Kill You! https://skylightcyber.com/2019/07/18/cylance-i-kill-you/.
[7] Avast 2018. AI & Machine Learning. https://www.avast.com/en-us/technology/ai-and-machine-learning.
[8] Nicholas Carlini, Anish Athalye, Nicolas Papernot, Wieland Brendel, Jonas Rauber, Dimitris Tsipras, Ian Goodfellow, and Aleksander Madry. 2019. On evaluating adversarial robustness. *arXiv preprint arXiv:1902.06705* (2019).
[9] Ero Carrera. 2016. pefile. https://github.com/erocarrera/pefile.
[10] Raphael Labaca Castro, Corinna Schmitt, and Gabi Dreo. 2019. AIMED: Evolving Malware with Genetic Programming to Evade Detection. In *2019 18th IEEE International Conference On Trust, Security And Privacy In Computing And Communications/13th IEEE International Conference On Big Data Science And Engineering (TrustCom/BigDataSE)*. IEEE, 240–247.
[11] Fabrício Ceschin, Marcus Botacin, Heitor Murilo Gomes, Luiz S Oliveira, and André Grégio. 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*. 1–9.
[12] Olivier Chapelle and Lihong Li. 2011. An empirical evaluation of thompson sampling. In *Advances in neural information processing systems*. 2249–2257.
[13] Lingwei Chen, Yanfang Ye, and Thirimachos Bourlai. 2017. Adversarial machine learning in malware detection: Arms race between evasion attack and defense. In *2017 European Intelligence and Security Informatics Conference (EISIC)*. IEEE, 99–106.
[14] George E Dahl, Jack W Stokes, Li Deng, and Dong Yu. 2013. Large-scale malware classification using random projections and neural networks. In *2013 IEEE International Conference on Acoustics, Speech and Signal Processing*. IEEE, 3422–3426.
[15] Park Daniel, Khan Haidar, and Yener Bülent. 2019. Generation & Evaluation of Adversarial Examples for Malware Obfuscation. *arXiv preprint arXiv:1904.04802* (2019).
[16] Luca Demetrio, Battista Biggio, Giovanni Lagorio, Fabio Roli, and Alessandro Armando. 2019. Explaining Vulnerabilities of Deep Learning to Adversarial Malware Binaries. *arXiv preprint arXiv:1901.03583* (2019).
[17] Luca Demetrio, B. Biggio, Giovanni Lagorio, F. Roli, and A. Armando. 2020. Functionality-preserving Black-box Optimization of Adversarial Windows Malware. *arXiv: Cryptography and Security* (2020).
[18] Ambra Demontis, Marco Melis, Maura Pintor, Matthew Jagielski, Battista Biggio, Alina Oprea, Cristina Nita-Rotaru, and Fabio Roli. 2019. Why do adversarial attacks transfer? explaining transferability of evasion and poisoning attacks. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 321–338.
[19] Saeed Ehteshamifar, Antonio Barresi, Thomas R Gross, and Michael Pradel. 2019. Easy to Fool? Testing the Anti-evasion Capabilities of PDF Malware Scanners. *arXiv preprint arXiv:1901.05674* (2019).
[20] Aurore Fass, Michael Backes, and Ben Stock. 2019. Hidenoseek: Camouflaging malicious javascript in benign asts. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. 1899–1913.
[21] William Fleshman, Edward Raff, Richard Zak, Mark McLean, and Charles Nicholas. 2018. Static malware detection & subterfuge: Quantifying the robustness of machine learning and current anti-virus. In *2018 13th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 1–10.
[22] Weiwei Hu and Ying Tan. 2017. Generating adversarial malware examples for black-box attacks based on GAN. *arXiv preprint arXiv:1702.05983* (2017).
[23] Weiwei Hu and Ying Tan. 2018. Black-box attacks against RNN based malware detection algorithms. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.
[24] Alex Huang, Abdullah Al-Dujaili, Erik Hemberg, and Una-May O'Reilly. 2018. On visual hallmarks of robustness to adversarial malware. *arXiv preprint arXiv:1805.03553* (2018).
[25] Yonghong Huang, Utkarsh Verma, Celeste Fralick, Gabriel Infantec-Lopez, Brajesh Kumar, and Carl Woodward. 2019. Malware Evasion Attack and Defense. *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks Workshops (DSN-W)* (Jun 2019). https://doi.org/10.1109/dsn-w.2019.00014
[26] Kyriakos K. Ispoglou and Mathias Payer. 2016. malWASH: Washing Malware to Evade Dynamic Analysis. In *10th USENIX Workshop on Offensive Technologies (WOOT 16)*. USENIX Association, Austin, TX. https://www.usenix.org/conference/woot16/workshop-program/presentation/ispoglou
[27] Aminollah Khormali, Ahmed Abusnaina, Songqing Chen, DaeHun Nyang, and Aziz Mohaisen. 2019. COPYCAT: Practical Adversarial Attacks on Visualization-Based Malware Detection. *arXiv preprint arXiv:1909.09735* (2019).
[28] Bojan Kolosnjaji, Ambra Demontis, Battista Biggio, Davide Maiorca, Giorgio Giacinto, Claudia Eckert, and Fabio Roli. 2018. Adversarial malware binaries: Evading deep learning for malware detection in executables. In *2018 26th European Signal Processing Conference (EUSIPCO)*. IEEE, 533–537.
[29] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Adversarial examples on discrete sequences for beating whole-binary malware detection. *arXiv preprint arXiv:1802.04528* (2018).
[30] Felix Kreuk, Assi Barak, Shir Aviv-Reuven, Moran Baruch, Benny Pinkas, and Joseph Keshet. 2018. Deceiving end-to-end deep learning malware detectors using adversarial examples. *arXiv preprint arXiv:1802.04528* (2018).
[31] Lief [n. d.]. LIEF. https://github.com/lief-project/LIEF.
[32] Xinbo Liu, Jiliang Zhang, Yaping Lin, and He Li. 2019. Atmpa: Attacking machine learning-based malware visualization detection methods via adversarial examples. In *2019 IEEE/ACM 27th International Symposium on Quality of Service (IWQoS)*. IEEE, 1–10.
[33] Keane Lucas, Mahmood Sharif, Lujo Bauer, Michael K Reiter, and Saurabh Shintre. 2021. Malware Makeover: Breaking ML-based Static Analysis by Modifying Executable Bytes. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security*. 744–758.
[34] MAB [n. d.]. Multi-armed bandit. https://en.wikipedia.org/wiki/Multi-armed_bandit.
[35] Davide Maiorca, Davide Ariu, Igino Corona, Marco Aresu, and Giorgio Giacinto. 2015. Stealth attacks: An extended insight into the obfuscation effects on android malware. *Computers & Security* 51 (2015), 16–31.
[36] Davide Maiorca, Battista Biggio, Maria Elena Chiappe, and Giorgio Giacinto. 2017. Adversarial Detection of Flash Malware: Limitations and Open Issues. *CoRR* abs/1710.10225 (2017).
[37] Davide Maiorca, Battista Biggio, and Giorgio Giacinto. 2018. Towards Robust Detection of Adversarial Infection Vectors: Lessons Learned in PDF Malware. *arXiv preprint arXiv:1811.00830* (2018).
[38] MLSEC2019 [n. d.]. Machine Learning Static Evasion Competition 2019. https://github.com/endgameinc/malware_evasion_competition.
[39] Vasilis Pappas, Michalis Polychronakis, and Angelos D Keromytis. 2012. Smashing the gadgets: Hindering return-oriented programming using in-place code randomization. In *2012 IEEE Symposium on Security and Privacy*. IEEE, 601–615.
[40] Jithin Pavithran, Milan Patnaik, and Chester Rebeiro. 2019. D-TIME: Distributed Threadless Independent Malware Execution for Runtime Obfuscation. In *13th USENIX Workshop on Offensive Technologies (WOOT 19)*. USENIX Association, Santa Clara, CA. https://www.usenix.org/conference/woot19/presentation/pavithran
[41] Fabio Pierazzi, Feargus Pendlebury, Jacopo Cortellazzi, and Lorenzo Cavallaro. 2020. Intriguing Properties of Adversarial ML Attacks in the Problem Space. *2020 IEEE Security and Privacy* (2020).
[42] Erwin Quiring, Alwin Maier, and Konrad Rieck. 2019. Misleading authorship attribution of source code using adversarial learning. In *28th {USENIX} Security Symposium ({USENIX} Security 19)*. 479–496.
[43] Edward Raff, Jon Barker, Jared Sylvester, Robert Brandon, Bryan Catanzaro, and Charles K Nicholas. 2018. Malware detection by eating a whole exe. In *Workshops at the Thirty-Second AAAI Conference on Artificial Intelligence*.
[44] Konrad Rieck, Philipp Trinius, Carsten Willems, and Thorsten Holz. 2011. Automatic analysis of malware behavior using machine learning. *Journal of Computer Security* 19, 4 (2011), 639–668.
[45] Ishai Rosenberg, Asaf Shabtai, Yuval Elovici, and Lior Rokach. 2018. Query-Efficient GAN Based Black-Box Attack Against Sequence Based Machine and Deep Learning Classifiers. *arXiv preprint arXiv:1804.08778* (2018).
[46] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 490–510.
[47] Ishai Rosenberg, Asaf Shabtai, Lior Rokach, and Yuval Elovici. 2018. Generic black-box end-to-end attack against state of the art API call based malware

classifiers. In *International Symposium on Research in Attacks, Intrusions, and Defenses*. Springer, 490–510.
[48] Joshua Saxe and Konstantin Berlin. 2015. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*. IEEE, 11–20.
[49] Matthew G Schultz, Eleazar Eskin, F Zadok, and Salvatore J Stolfo. 2000. Data mining methods for detection of new malicious executables. In *Proceedings 2001 IEEE Symposium on Security and Privacy. S&P 2001*. IEEE, 38–49.
[50] slot-machine [n. d.]. Slot machine. https://en.wikipedia.org/wiki/Slot_machine.
[51] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. 2017. Attack and defense of dynamic analysis-based, adversarial neural malware classification models. *arXiv preprint arXiv:1712.05919* (2017).
[52] Jack W Stokes, De Wang, Mady Marinescu, Marc Marino, and Brian Bussone. 2018. Attack and Defense of Dynamic Analysis-Based, Adversarial Neural Malware Detection Models. In *MILCOM 2018-2018 IEEE Military Communications Conference (MILCOM)*. IEEE, 1–8.
[53] Octavian Suciu, Scott E Coull, and Jeffrey Johns. 2019. Exploring adversarial examples in malware detection. In *2019 IEEE Security and Privacy Workshops (SPW)*. IEEE, 8–14.
[54] Microsoft Defender ATP Research Team. 2019. New machine learning model sifts through the good to unearth the bad in evasive malware. https://www.microsoft.com/security/blog/2019/07/25/new-machine-learning-model-sifts-through-the-good-to-unearth-the-bad-in-evasive-malware/.
[55] TS [n. d.]. Thompson Sampling. https://en.wikipedia.org/wiki/Thompson_sampling.
[56] Weilin Xu, Yanjun Qi, and David Evans. 2016. Automatically evading classifiers. In *Proceedings of the 2016 network and distributed systems symposium*. 21–24.
[57] Wei Yang, Deguang Kong, Tao Xie, and Carl A Gunter. 2017. Malware detection in adversarial settings: Exploiting feature evolutions and confusions in android apps. In *Proceedings of the 33rd Annual Computer Security Applications Conference*. 288–302.

A BROKEN MALWARE EXAMPLES

Case 1: Implementation errors in instruction replacement. As shown in Figure 9, the original implementation of code randomization only supports 8-bit and 32-bit, not 16-bit instruction. It tries to replace a 16-bit add/sub instruction in a wrong way (assuming 32-bit format). It treats the last four bytes as the second operand, but actually, only the last two bytes are the second operand. This would break the CFG of the program.

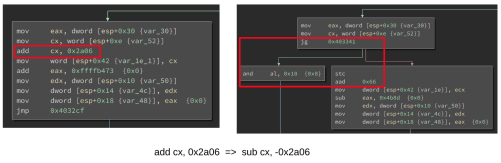


Figure 9: Implementation Errors in CR

Case 2: Overwriting overlay. As shown in Figure 10, when adding a new section at the end of the last section, if the sample has overlay data, the added new data may affect the overlay data extraction of the malware.

B MORE DISCUSSIONS

Why do commercial AVs rely on simple features? Looking at the result, it seems surprising that trivial changes to malware can evade professionally developed commercial anti-virus systems used by millions of users. Why have not commercial AVs fixed these problems already? One hypothesis could be that adversarial examples are a result of the trade-off between true positive and false-positive rates. The commercial systems need to provide a fast

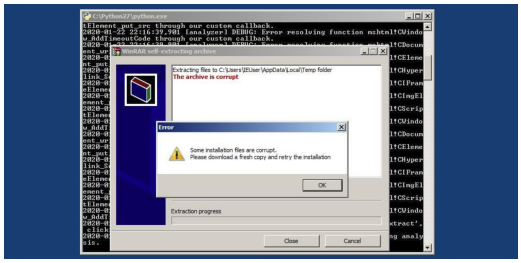


Figure 10: Errors after Overlay Data Append

decision while maintaining a low false-positive rate. The relatively simple features, such as file hash or white-listing benign strings can help gain a high accuracy with a low false-positive rate. For example, in their attack against Cylance [6], researchers noticed that to reduce their false positive rate the Cylance team whitelisted some families of executables, one of them was an online game. So, the whitelisting used to reduce the false-positive rate was enough to create an adversarial attack. Another hypothesis is that anti-virus systems need to protect against the malware of today, instead of focusing on new attacks that are not currently happening. Real adversaries probably use techniques different from the ones used to create ML-based adversarial attacks. A third hypothesis is that anti-virus systems do not rely only on static detection, but also on dynamic and behavioral detection. If all the adversarial malware samples get detected when they are executed, the AVs are not concerned with the static-only adversarial attacks as these attacks cannot infect real users.

Are adversarial attacks harmful to users? We perform a preliminary test to see the extent to which static-only adversarial examples evade the full AV pipeline and infect users. We create adversarial samples by modifying 30 ransomware samples and test whether the samples that evade static classifiers can infect users' machines. We hypothesize that the dynamic and behavioral classifiers of the AVs will detect and stop the static adversarial examples when they are executed, thus, posing no real harm to the users. Except for AV2, all the other AVs blocked the execution of adversarial ransomware samples. All of the 30 adversarial ransomware samples evade the behavior detector of AV2; files are encrypted and blackmail messages are shown on the screen. However, the online version of AV2 can detect all the samples as AV2 relies heavily on cloud techniques. This represents a potentially new attack surface to investigate in the future, where static-only evasion can sometimes evade the entire AV pipeline and infect users due to the design decision of an AV.

Recommendation for Antivirus Systems. Our attacks demonstrate that static classifiers are easy to evade. So for full protection, commercial AV systems need to rely more on dynamic and behavior-based detection. Even though some papers already demonstrated that dynamic classifiers can be evaded by splitting a malware sample into many different pieces [26, 40], currently no one demonstrated a scalable and generalized attack against dynamic classifiers.

Table 5: Adversarial attacks on Windows malware detectors

Year	Paper	Target Model	Threat model	Problem space attack	Verification
2020	Demetrio et al. [17]	**AV**, MalConv, GBDT	blackbox	Yes	Yes
2019	Ashkenazy et al. [6]	**AV (Cylance)**	greybox	Yes	No
2019	Ceschin et al. [11]	**MalConv**, EMBER, **AVs**	**blackbox**	Yes	**Yes**
2019	Demetrio et al. [16]	**MalConv**	whitebox	Yes	No
2019	Huang et al. [25]	DNN (API existence)	grey&whitebox	**No**	No
2019	Khormali et al. [27]	CNN (Visualization)	whitebox	Yes	No
2019	Liu et al. [32]	CNN (Visualization), RF, SVM	whitebox	Yes	No
2019	Park et al. [15]	CNN (Visualization), **MalConv**	whitebox	Yes	No
2019	Suciu et al. [53]	**MalConv**, **EMBER**	whitebox	Yes	No
2018	Al-Dujaili et al. [3]	DNN	whitebox	**No**	No
2018	Fleshman et al. [21]	**MalConv**, **AVs**	**blackbox**	Yes	No
2018	Hu et al. [23]	RNN (API call sequence)	greybox	**No**	No
2018	Kolosnjaji et al. [28]	**MalConv**	whitebox	Yes	No
2018	Kreuk et al. [29]	**MalConv**	whitebox	Yes	No
2018	Rosenberg et al. [47]	RNN (API call sequence)	greybox	Yes	**Yes**
2018	Rosenberg et al. [45]	RNN (API call sequence)	greybox	Yes	No
2017	Anderson et al. [4]	**EMBER**	**blackbox**	Yes	No
2017	Chen et al. [13]	DNN (API existence), **AVs**	whitebox	Yes	No
2017	Hu et al. [22]	DNN (API existence)	greybox	**No**	No
2017	Stokes et al. [51]	DNN	whitebox	**No**	No