MPass: Bypassing Learning-based Static Malware Detectors

Jialai Wang*, Wenjie Qu†, Yi Rong*, Han Qiu*§, Qi Li*§, Zongpeng Li*♣✉, and Chao Zhang*§✉
*Tsinghua University, †Huazhong University of Science and Technology,
♣Quan Cheng Laboratory, §Zhongguancun Laboratory, ✉Corresponding author

***Abstract*—Machine learning (ML) based static malware detectors are widely deployed, but vulnerable to adversarial attacks. Unlike images or texts, tiny modifications to malware samples would significantly compromise their functionality. Consequently, existing attacks against images or texts will be significantly restricted when being deployed on malware detectors. In this work, we propose a hard-label black-box attack `MPass` against ML-based detectors. `MPass` employs a problem-space explainability method to locate critical positions of malware, applies adversarial modifications to such positions, and utilizes a runtime recovery technique to preserve the functionality. Experiments show `MPass` outperforms existing solutions and bypasses both state-of-the-art offline models and commercial ML-based antivirus products.**

I. Introduction

ML-based methods have been widely deployed in static malware detection tasks [1]–[5]. The main idea behind these detectors is to build ML models to classify whether the target sample (i.e., the target programs' on-disk bytes) is benign or malicious. Compared with traditional static malware detection methods such as signature-based approaches [6], ML-based malware detectors enjoy a proven higher detection rate and higher efficiency, require fewer human efforts, and generalize better to unseen malware.

Unfortunately, ML models are well-known to be vulnerable to adversarial attacks [7]–[10]. Recent studies show adding tiny perturbations to input samples could easily manipulate the inference of ML models. Such adversarial attacks have been widely studied in domains such as computer vision [11] and natural language processing [12], as well as malware detection [13]. In other words, ML-based malware detectors are vulnerable to adversarial attacks.

However, malware is confined to stringent formatting, with different sections such as file header, code section, data section, etc. Most of them are sensitive to modifications. Directly applying gradient-based methods to modify arbitrary bytes in malware will cause execution errors that compromise its functionality. Thus, the *challenge* of adversarial attacks on ML-based malware detectors is to generate perturbations that can bypass these ML models while preserving malware's functionality. The straightforward solution like existing attacks [13]–[16] generates perturbations without modifying sensitive sections (e.g. code or data sections). However, this will lead to the following issues. First, the importance of generating adversarial perturbations considering different malware sections remains unexplored. In other words, avoiding modifying sensitive sections like code or data sections may substantially limit the attack success rate since these sections not only represent a high ratio of the byte representation of the malware (often more than 60%) but also carry essential malicious features (e.g., access to sensitive data or invocations to sensitive APIs) that may be detected by ML models. Second, even if sensitive sections are proved to be vital for achieving a higher attack success rate, it is infeasible for existing attack methods to perform malicious modifications on those sensitive sections.

In this work, we address the above challenges by proposing `MPass`, a new hard-label black-box adversarial attack. `MPass` consists of three modules: (1) a *problem-space explainability method* to understand and locate the critical positions in malware for instructing the black-box attack, (2) a *runtime recovery technique* to modify sensitive sections (e.g. code or data) of malware while preserving its functionality, and (3) an *optimization-based perturbation method* to obtain function-preserving adversarial examples (AEs) to bypass ML-based static malware detectors in a black-box way. In summary, we make the following contributions:

- We propose a new hard-label black-box adversarial attack method `MPass`, which can preserve the functionality of malware samples and achieve a high attack success rate.
- We explain and quantitatively point out two critical malware sections considering adversarial attacks that are not explored by existing attacks.
- We conduct a thorough evaluation of four state-of-the-art (SOTA) offline ML-based static malware detectors and five commercial ML-based anti-viruses (AVs) products. Results reveal that `MPass` significantly outperforms SOTA black-box attacks with higher success rates, fewer queries, and a lower file appending rate.

II. Preliminaries

A. Problem Statement and Attack Requirements

Problem statement. The goal of this work is to design a *hard-label black-box adversarial attack* [17], [18] against ML-based static malware detectors. Dynamic malware detectors are out of our scope[1]. An *adversarial attack* is performed by manipulating malware (only binary, no source code available[2]) to generate function-preserving AEs to evade the ML-based static malware detection.

We consider the *black-box* scenario in which the attacker has no inside knowledge of target malware detectors including both the ML models and the feature extractors. Note AEs generated by us are in the problem-space (*i.e.*, malware samples) which are more practical than AEs in feature-space, as generating AEs in feature-space requires knowledge of the feature extractors. Besides, we assume only *hard-label* detection results (benign or malicious), from querying the target detectors with modified malware samples.

We consider attacking detectors for Windows-executable malware. We do not experiment with attacks on other detectors for Android malware [19] or PDF [20], but they can be evaded similarly.

Attack requirements. We expect the AEs generated by `MPass` to be effective and function-preserving. Particularly, `MPass` is expected to successfully generate AEs with *reasonable queries* to bypass ML-based static malware detectors. The AEs are expected to have *low file size increments*, as larger malware samples are hard to transmit and prone to be caught by AVs. The AEs are also expected to preserve malicious functions of the original malware. We use sandboxes like

[1]Due to the high runtime overhead of dynamic analysis, ML-based dynamic malware detectors are only deployed in lab environments for high-value suspicious sample analysis. Static detectors are widely deployed in practice and thus become adversaries' targets.

[2]Malware developers may reuse binary components and get caught by AVs. Thus, adversarial attacks should target binaries.

Cuckoo[3] to test AEs and confirm whether they could still trigger runtime malicious behaviors.

B. Existing Attacks and Their Limitations

Different from the adversarial attacks in computer vision, which can modify arbitrary pixel values in an image, most existing attacks on malware detectors can only modify limited sections in malware, to preserve the latter's functionality. For instance, rather than modifying code and data sections, they mainly focus on appending data to the tail or modifying PE headers or import/export tables. Some approaches then utilize Reinforcement learning (RL) [15], [21] or generative adversarial networks (GAN) [22], [23] to generate the perturbations and yield AEs. The approach [14] utilizes RNN to append perturbations at the tail. Some other approaches utilize genetic algorithms [13], [16] or directly inject benign bytes [24] to generate AEs. Furthermore, an explainable attack [25] is proposed, presuming most features used by target models are known.

Limitations of existing attacks. They have two main limitations as follows. First, without modifying code and data sections, a high attack success rate is hard to achieve. Code and data sections not only occupy a large portion of malware but also carry malicious features that may be detected.[4] Second, all existing attacks either lack explainability to attack or assume features used by target models are partially known, which is not practical.

III. Methodology

A. Overview

The workflow of MPass consists of three steps. First, MPass proposes a *problem-space explainability method* (PEM) to locate critical positions of malware based on a set of known models. The results can be used to instruct the critical sections for perturbation generation. Second, MPass designs a *runtime recovery technique* to support modifying code and data sections to preserve malware's functionality. Third, MPass addresses an *optimization-based perturbation generation* process to generate adversarial perturbations for the attack.

B. Problem-space Explainability Method

We propose a problem-space explainability method (PEM) to locate critical positions of malware. Inspired by SHAP [26], PEM can effectively explain which attributes of inputs are critical to the models' decisions. Considering the possible divergence between the known and unknown target models, the critical positions of malware could vary on different models. We apply PEM to an ensemble of known ML-based malware detectors and find their common critical positions, and then focus on these positions to attack black-box target models, as illustrated in Algorithm 1.

In the problem-space, without knowing the target model f's features, we treat a malware x's each section (such as code, data, resource section,) as an attribute, then compute the i-th section's Shapley value $\phi_{i,f,x}$ as in Eq. 1. The larger this value is, the more critical the section i is, for the given model f and malware x.

$$\phi_{i,f,x} = \sum_{\hat{s} \subseteq S \setminus \{i\}} \frac{|\hat{s}|!(|S| - |\hat{s}| - 1)!}{|S|!} (f(x_{\hat{s} \cup \{i\}}) - f(x_{\hat{s}})), \quad (1)$$

where S is the set of $x's$ all sections, and $f(x_{\hat{s}\cup\{i\}}) - f(x_{\hat{s}})$ represents the model f's prediction difference on sections $\hat{s}$ together with the presence or absence of the section i. To speed up the calculation, we only consider the top-30 most common sections' Shapley values rather than all sections'.

Algorithm 1: The Workflow of PEM

Input: C: N randomly sampled malware;
K: all M known models, ;
S_{all} : top-h common sections in C;
Output: $\widetilde{S}$: common critical sections on K;
for $f \in K$ **do**
  for $i \in S_{all}$ **do**
    for $x \in C$ **do**
      extract x's all sections as S;
      if $i \in S$ **then**
        compute the i-th section's Shapley value $\phi_{i,f,x}$ following Eq. 1;
      else
        $\phi_{i,f,x}$ = 0;
    compute the average $\phi_{i,f,x}$ of all N malware samples x in C, i.e., $E_f(\phi_i)$;
  $\widetilde{S_f}$: sort S_{all} by $E_f(\phi_i)$, and select the top-k sections;
$\widetilde{S} = \widetilde{S_1} \cap \widetilde{S_2}, ..., \cap \widetilde{S_M}$

To get critical sections for a specific model f, we normalize the effect of specific malware, by calculating the average Shapley values $\phi_{i,f,x}$ of all sampled malware x, denoted as $E_f(\phi_i)$. According to the ranking results of $E_f(\phi_i)$, we identify the top-k critical sections $\widetilde{S_f}$ on each model f. Lastly, we could obtain the final common critical sections $\widetilde{S} = \widetilde{S_1} \cap \widetilde{S_2}, ..., \cap \widetilde{S_M}$.

We find the top-1 and top-2 are code and data sections, which have the highest average Shapely values on all known models, and nearly $1.3 \sim 6.0\times$ higher than the top-3. Such results also conform to the intuition in this domain. Code sections contain malware's executable instructions, which reflect the functionality and behavior of malware. Data sections contain initialized sensitive malware variables, *e.g.*, constants for encryption algorithms. Overall, PEM reveals that code and data sections are the critical sections for ML detectors' decisions.

C. Malware Modification

In addition to the critical positions, certain common *modification positions* are also modified by MPass to introduce *perturbations*. The challenge of modifying the critical code and data sections is how to successfully add adversarial perturbations while preserving malware's original functionality. To address this challenge, we design a runtime recovery technique for *functionality preservation*. The runtime recovery technique will also bring instruction patterns that may potentially be learned by adaptive ML models. To this end, we further apply a shuffle strategy to shuffle these instruction patterns.

The workflow is shown in Figure 1. This workflow will repeat until the yielded malware successfully bypasses target models or the maximum number of queries is reached.

Modification positions. Besides the code and data sections, MPass further modifies two other positions, which are commonly used by existing attack methods. (1) We create a new section at the tail of the original malware for adding perturbations, i.e., the blue region in Figure 2. For malware without sufficient space to create a new section, our method will append perturbations at the end of them (overlay appending), i.e., the purple region in Figure 2. (2) We also modify some header fields that do not affect malware's functionality, as RL-Attack [21] does, such as timestamps and section names (i.e., the

[3]https://cuckoosandbox.org/

[4]We note [21] pack malware for an attack but the packed malware cannot be modified or optimized anymore.

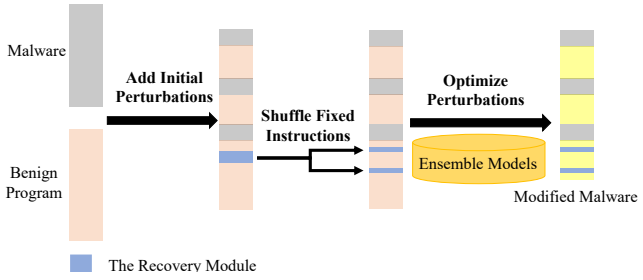


Fig. 1: Workflow of malware modification. (1) We modify malware with initial perturbations from a randomly selected benign program, together with a recovery module. (2) We shuffle fixed instructions in the recovery module. (3) We further optimize the perturbations based on an ensemble of known models, enhancing the effectiveness.

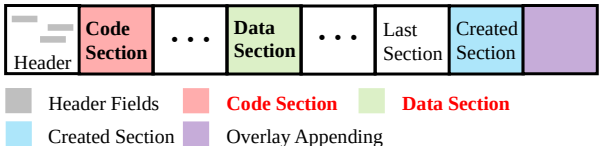


Fig. 2: Positions perturbed by `MPass`. These colored regions could get updated by `MPass`, and the two in red fonts (i.e., code and data sections) are first perturbed in our work.

grey region in Figure 2) [5]. For intuitive comparison, we illustrate the positions modified by `MPass` in Figure 2: the colored areas highlight positions where perturbations can be added by `MPass`.

Perturbations. `MPass` supports adding any perturbations we want to the modification positions. We first insert contexts from a randomly selected benign program into these positions, as an initial perturbation. We believe using benign programs as perturbations would increase the probability of bypassing malware detectors. Then, we design a loss function to help optimize these perturbations. Details of the optimization process are given in Section III-D.

Functionality-preserving. A delicate recovery module is implemented to enable malware modification while preserving functionality. `MPass` encodes the original code and data sections with some keys, creates a new section to insert the recovery module together with the decoding keys, and then switches the entry point of the malware to the address of this recovery module. When the modified malware starts execution, the recovery module will recover the original malware with decoding keys, restore contexts (*e.g.*, registers), and rerun to the original entry point to preserve its functionality.

For example, consider a malware with n bytes $x = \{x_1, x_2, ..., x_n\}$ and positions $i \sim j$ $(1 \leq i < j \leq n)$ to be modified. While we assign $x_{i\sim j}$ with benign contexts $b_{i\sim j}$, we save the corresponding keys $k_{i\sim j}$, where $k_{i\sim j} = b_{i\sim j} - x_{i\sim j}$. Then, when the modified malware runs, the recovery module restores $x_{i\sim j}$ with these keys: $x_{i\sim j} = b_{i\sim j} - k_{i\sim j}$. These modified positions are recovered and the functionality is preserved. In summary, based on this runtime recovery technique, `MPass` can modify code and data sections to any contexts we want, including but not limited to benign programs.

Shuffle strategy. The recovery module has fixed instructions that might be learned as a pattern adaptively by real-world ML AVs. To address this issue, we design a shuffle strategy to shuffle instructions for breaking the pattern. Given a fixed instruction sequence containing m instructions $P = \{p_1, p_2, ..., p_m\}$ corresponding to the recovery module, each element of P denotes a single instruction, *e.g.*, *add eax, ebx*.

[5]We do not modify slack space (space between sections, inserted by the compiler for alignments or other purposes) or import tables because these two parts are relatively small and their effect on attacks is negligible.

We randomize the order of P to break its fixed form and generates a new sequence $\hat{P} = \{\hat{p}_1, \hat{p}_2, ..., \hat{p}_m\}$ (e.g., $P = \{p_1, p_2, p_3\}$ and $\hat{P} = \{p_2, p_1, p_3\}$). We prevent these instructions from clustering together, by allocating spaces between instructions to insert benign data. In the new section created, we randomly allocates an unused space S and splits S into n parts, and then insert $\hat{P}$ to S to redefine the modified $\hat{P}$ as $\hat{P} = \{\hat{p}_1, s_1, \hat{p}_2, s_2..., \hat{p}_n, s_n\}$. Perturbations will be inserted into $\{s_1, s_2, ..., s_n\}$. To guarantee the recovery module's functionality, we insert `jump` instructions into $\hat{P}$ to maintain the original execution order of P.

Taking an original execution order of instructions $p_1 \rightarrow p_2 \rightarrow p_3$ as an example, if they are modified as $\hat{P} = \{p_2, s_1, p_1, s_2, p_3, s_3\}$, `MPass` needs to add corresponding `jump` instructions into $\hat{P}$, *i.e.*, $\hat{P} = \{jump\ p_1, p_2, jump\ p_3, s_1, p_1, jump\ p_2, s_2, p_3, s_3\}$. While $\hat{P}$ is executed, $jump\ p_1$ makes p_1 executed first, then $jump\ p_2$ will make p_2 executed, and $jump\ p_3$ will make p_3 executed. This means the actual instruction execution order of $\hat{P}$ is $jump\ p_1 \rightarrow p_1 \rightarrow jump\ p_2 \rightarrow p_2 \rightarrow jump\ p_3 \rightarrow p_3$, which is the same as P's semantically. Besides, as instruction positions are modified, we need to patch instructions that use relative addressing according to their new positions, in order to preserve the semantics of such instructions.

D. Optimization

The question left is how to generate perturbations against black-box models. We adopt the basic idea of transfer attacks [27], [28], which formulate the perturbation generation as an optimization problem on ensemble models. Different from traditional tasks, e.g., computer vision (CV), which could directly adopt ensemble loss functions, we need to make sure that the optimization is function-preserving. Therefore we develop a special loss function as follows.

By inserting benign contexts, the recovery module, and keys in a malware x, we generate $\hat{x}$ with length n and zero-pad x to length n. The initial perturbations $\delta = \hat{x} - x$ will be optimized. Specifically, we denote positions that can be optimized as I (§III-B), i.e., for $\forall i \in I$, δ_i can be optimized. While optimizing these bytes, their keys in the recovery module should be updated correspondingly to preserve functionality (§III-C). We use a tuple corpus J to record the mapping between optimized bytes and their keys: for $\forall (j,k) \in J$, $\hat{x}_j$'s corresponding key is $\hat{x}_k$. We further define a matrix M:

$$M_{ij} = \begin{cases} 1 \text{ if } i == j \text{ and } i \in I \text{ // byte i is optimized} \\ 1 \text{ else if } (j,i) \in J \quad \text{// byte i is a key for j} \\ 0 \text{ else} \end{cases} \tag{2}$$

such that $x + M * \delta$ is the function-preserving malware sample. Then, we formalize the optimization problem as follows:

$$\mathcal{L}_{opt} = \mathcal{L}(F(x + M * \delta), y), \tag{3}$$

where $\mathcal{L}$ is a cross-entropy loss function, y denotes a benign class and F is the local model. We hope to find δ that minimizes $\mathcal{L}_{opt}$, as a lower $\mathcal{L}_{opt}$ means a higher probability of the malware being misclassified as benign ones.

Note that the optimization only operates on floating values. Thus, the perturbations are first lifted to feature vectors using the embedding layer for optimization and get mapped back to discrete bytes after optimization [29], and finally used to yield the optimized malware.

IV. Experimental Results

We first evaluate the performance of `MPass` by comparing state-of-the-art black-box attacks on four state-of-the-art offline models and five real-world AVs. Furthermore, we evaluate the performance

of MPass considering adaptive defenses like adversarial training and AVs' learning. All experiments are conducted on a machine with an Intel Xeon Gold 6154 CPU and four NVIDIA Tesla V100 GPUs.

Datasets and baselines. We randomly select 2000 malware samples (PE) from two commonly used websites VirusTotal [30] and VirusShare [31]. To guarantee the quality of malware, we refer to MAB [15] and make sure they meet two requirements. (1) They are all initially correctly classified by target models. (2) The execution of them in a Cuckoo sandbox shows malicious behavior. We compare MPass with four highly related and state-of-the-art attack methods, i.e., RLA [21], MAB [15], GAMMA [16], and MalRNN [14]. We reproduce them by directly using their source code with recommended parameters for fair comparison.

Hyperparameters. For adding initial perturbations, we randomly collect 50,000 benign programs from the local Microsoft Windows system and GitHub. For all attack methods, we set the maximum number of queries on target ML detectors for each sample to 100.

A. Attacking Offline Models

Offline models. We compare MPass with baselines on four state-of-the-art offline ML detectors, i.e., MalConv [1], NonNeg [32], LightGBM (implemented by Ember) [2], and MalGCG [3]. For the first three models, we use their pre-trained models provided by Machine Learning Security Evasion Competition 2019 [33] as previous works do [14], [15]. We use the pre-trained model for MalGCG. For MPass, an ensemble of known models is required. Thus, while attacking a target model, we treat the remaining models as known models[6] and adopt the same ensemble method proposed in [27] to attack multiple models simultaneously, where the attack loss is defined as the sum of the individual model loss. We use the Adam optimizer, set learning rate $\eta = 0.01$ and number of iterations $\gamma = 50$.

Metrics. We use the widely used comparison metrics in previous works. (1) Attack success rate (ASR). ASR measures the percentage of original malware samples that can derive AEs to bypass detectors. (2) Average queries (AVQ). It measures the average query number required for generating one AE and is defined as AVQ= $\frac{Q_{all}}{N}$, where Q_{all} is the sum of queries for all samples and N is the number of samples tested. Higher AVQ will bring larger costs while lower AVQ means attack efficiency. (3) Average appending rate (APR). It measures the file size increment ratio and is defined as: APR= $\frac{\texttt{size}(x_{adv})-\texttt{size}(x)}{\texttt{size}(x)}$, where x and x_{adv} are the original malware and its AE, $\texttt{size}(x)$ and $\texttt{size}(x_{adv})$ represent their size respectively. AEs with high APR are liable to be suspected, reducing their usability [16].

Results. Table I, Table II, and Table III show the ASR, AVQ, and APR, respectively. Overall, our method MPass significantly outperforms baselines on all metrics, showing great effectiveness. For example, on the new model MalGCG, the ASR of MPass is 12.2% higher than the relatively powerful attack method, i.e., MAB, while other baselines only achieve ASR below 80%. On this model, our MPass only needs 1.6 AVQ for generating one AE, while other methods need at least 12.4 AVQ (MalRNN). Besides, MPass only needs 82.6% APR for crafting AEs which is less than baselines.

TABLE I: ASR of attacking offline models.

Models	ASR (%)				
	MPass	**RLA**	**MAB**	**GAMMA**	**MalRNN**
MalConv	**98.6**	33.7	94.2	81.8	94.3
NonNeg	**99.2**	35.4	93.6	90.2	97.0
LightGBM	**98.3**	20.3	91.8	84.8	28.2
MalGCG	**99.6**	68.7	87.4	61.4	76.8

[6]LightGBM is not used as a known model since it cannot be backpropagated.

TABLE II: AVQ of attack methods on offline models.

Models	AVQ				
	MPass	**RLA**	**MAB**	**GAMMA**	**MalRNN**
MalConv	**2.6**	92.3	7.6	83.9	9.3
NonNeg	**2.2**	79.5	10.5	15.8	5.7
LightGBM	**2.8**	94.2	11.7	18.0	70.8
MalGCG	**1.6**	61.4	17.0	63.1	12.4

TABLE III: APR of attack methods on offline models.

Models	APR (%)				
	MPass	**RLA**	**MAB**	**GAMMA**	**MalRNN**
MalConv	**108.6**	613.5	430.3	4013.5	402.8
NonNeg	**68.4**	657.4	300.3	3721.4	362.4
LightGBM	**182.5**	432.8	475.0	3613.2	506.3
MalGCG	**82.6**	389.6	959.2	4214.3	324.5

Analysis of improvements. We argue the main reason we perform better than baselines is that MPass is able to modify code and data sections which baselines cannot. For example, the relatively powerful baseline MAB mainly focuses on adding perturbations to the newly created sections or the tail of malware, without modifying code or data sections. RLA manually specifies features to construct the state for reinforcement learning. However, features perturbed by those baselines are not within the same scope of the code and data sections which are explored by target detectors especially the commercial ML AVs. Thus, their ASR will be significantly limited.

Verifying functionality-preserving. For all AEs acquired, we use the Cuckoo sandbox to track the runtime behaviors (API call sequences) of modified and original malware and verify their functionality as other works do [13], [15]. We only find that 23% AEs generated by RLA suffer from this problem, while other methods could preserve the malware functionality in their AEs.

B. Attacking Commercial ML AVs

Commercial ML-based AVs. We also evaluate MPass on five representative real-world commercial static ML AVs, namely, MAX, CrowdStrike, Acronis, SentinelOne, Cylance. Here, we represent them as AV_1, AV_2, AV_3, AV_4 and AV_5. We refer to the introduction of each AV in VirusTotal and figure out these ML AVs. Then, we check these AVs' official websites and white papers to make sure they are ML-based. For performing attacks, we use the VirusTotal which offers APIs for uploading AEs and querying the target AVs for detection results. While attacking these target ML AVs, we use the same attacking settings and AEs as in Section IV-A.

Figure 3 shows ASR of different attack methods on five AVs, respectively. Overall, MPass significantly outperforms all baseline attacks on attacking the commercial ML AVs. For instance, MPass achieves 61.2% ASR and 58.8% ASR on AV_3 and AV_4 respectively, while baseline attacks could only achieve 23.2% and 6.7% at most. Such a significant ASR improvement can prove the effectiveness of our method. Besides, we also verify these AVs are not hash-based and show details in Section V.

TABLE IV: Comparison with obfuscation techniques on ASR of attacking commercial AVs.

Models \ ASR (%)	**AV1**	**AV2**	**AV3**	**AV4**	**AV5**
UPX	17.1	19.8	11.5	14.8	7.6
PESpin	12.2	16.4	4.0	11.8	5.5
ASPack	17.6	4.2	9.6	12.6	9.3
MPass	**42.3**	**35.8**	**61.2**	**58.8**	**29.2**

Comparison with obfuscation/permutation. Obfuscation techniques could also bypass malware detectors. We therefore compare MPass

with three commonly used obfuscators, i.e., UPX[7] , PESpin[8], ASPack[9].

Table IV shows that MPass outperforms these obfuscators. Such obfuscation techniques are not specific to ML-based detectors, and thus lack guidelines to generate perturbations towards attacking ML-based static malware detectors. MPass considers the explainability and formulates an optimization problem to guide perturbation generation, and therefore has better performance.

C. Evaluation on Commercial ML AVs' Learning

We further attack these five commercial ML AVs considering the potential learning mechanisms they have. It is well-known that commercial ML AVs can constantly learn from abundant samples submitted and keep learning to improve their detection ability. Note that the learning procedure of these commercial ML AVs is not known to us which makes this evaluation challenging. Therefore, we propose to evaluate if the AEs generated by MPass can be detected after these learning-based ML AVs update their detection models.

We select all the success AEs generated by MPass and baselines in Section IV-B as the initial malware sample sets. We submit these AEs to the five target ML AVs for detection every 7 days to see if the update of these ML AVs can have better detection rates. Here, we use the bypass rate to present the attack results and use AEs that are successful at the first time detection. The decrease of bypass rate along with the time means more successful AEs are detected by updated ML AVs. This can directly reflect the effectiveness of one attack method against ML AVs' improvement by their learning.

Figure 4 shows the bypass rate of MPass and baselines on the five commercial ML AVs, respectively. For the first time, all bypass rates for MPass and baseline attacks are 100% since we select only those successful AEs. Later on, after another four evaluation (time period is $4 \times 7 = 28$ days), we can observe that the target ML AVs had been updated such that the bypass rate for AEs generated by all baselines are significantly decreased. This indicates that the commercial ML AVs can keep learning new patterns of malware to improve their detection rate even on sophisticated AEs. However, the AEs generated by MPass keep a 100% bypass rate on all five ML AVs all the time which means the patterns of our AEs are not learned by these ML AVs' updating policy. This proves the effectiveness of MPass even considering the continuous learning ability of commercial ML AVs.

V. Ablation Study

Changing modification positions. We verify the effectiveness of modifying code and data sections in bypassing malware detectors. In particular, we compare MPass with the setting modifying any other sections without code and data sections. We denote this setting as Other-sec. In Other-sec, for each malware, we randomly select other sections to modify. For fair comparisons, we make sure all other attack settings are the same as MPass (e.g., the same APR). Tables V shows that MPass outperforms Other-sec on all ML AVs. Therefore, we conclude that modifying code and data sections enables MPass to effectively bypass malware detectors.

TABLE V: Impact of changing modification positions on commercial ML AVs.

Models	ASR (%)				
	AV1	AV2	AV3	AV4	AV5
Other-sec	2.3	4.8	3.2	2.4	5.2
MPass	**42.3**	**35.8**	**61.2**	**58.8**	**29.2**

[7]https://upx.github.io/

[8]https://www.start64.com/index.php/64-bit-software/64bit-development/3052-pespin

[9]http://www.aspack.com/

A question left is whether these ML AVs are hash-based, and modifying code and data sections might change the corresponding hash values which induce bypassing malware detectors. To this end, we consult with experts in AV companies and confirm they are not hash-based. To further demonstrate this point, we add random data to the same modifications positions as MPass does. We then compare malware modified by random data with AEs generated by MPass on ASR. Results are shown in Table VI. We find that modified malware with random data gets a much lower ASR on AV_{1-5}, proving naive hash changes cannot efficiently evade such AV tools. We conclude these ML AVs are not hash-based.

TABLE VI: ASR of modified malware with random data, and AEs generated by MPass on commercial ML AVs.

Models	ASR (%)				
	AV1	AV2	AV3	AV4	AV5
Random data	8.3	4.1	5.9	7.2	6.6
MPass	**42.3**	**35.8**	**61.2**	**58.8**	**29.2**

VI. Discussion and Future Work

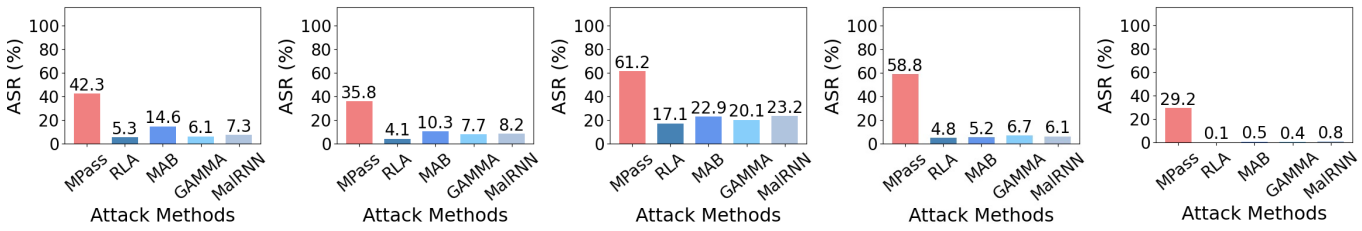
Bypassing dynamic analysis. In this paper, we focus on attacking various learning-based static malware detectors. We admit that MPass can hardly bypass malware detectors using dynamic analysis which is designed based on different perspectives. Usually, dynamic analysis requires a lot of effort such as human expertise, and is very slow on large and complex programs. Therefore, compared with deploying static malware detectors, dynamic analysis is inefficient and less deployed in practice. We plan to list designing adversarial attacks to bypass the dynamic analysis method as our first future work.

Adversarial training. It is well known that adversarial training [8] is effective to mitigate the adversarial attacks by increasing the robustness of models. However, directly deploying existing gradient-based adversarial training methods in our scenario will not work due to two reasons. First, famous adversarial training methods like PGD-AT [8] will generate AEs during training in each epoch to improve the robustness of models. This cannot be adopted to the malware detection tasks since these AEs generated by existing gradient-based methods just added uniform perturbations which does not consider the function-preserving for malwares. In other words, they are not in the same distribution with AEs generated by MPass such that they cannot help increase the robustness. Second, we also use the effective AEs generated by MPass to mix with clean malware samples (AE/clean both 50%) to train the model which follows the classic adversarial training method [7]. This will suppress the ASR of MPass by less than 10% which is not useful. We analysis the reason as that the AEs of malware has a large space than samples in computer vision such that including more AEs of malware to train has little contribution to improve model robustness. Therefore, we list our second future direction as to explore the robustness of malware detectors.

Advanced defense schemes. We do note that there are other well-developed defense approaches against adversarial attacks in computer vision domain. However, due to the different data nature between images and malware, it is hard to naively adopt them to MPass for evaluation. We aim to explore other defense shcemes that detect or mitigate the AEs generated by MPass as our third future direction.

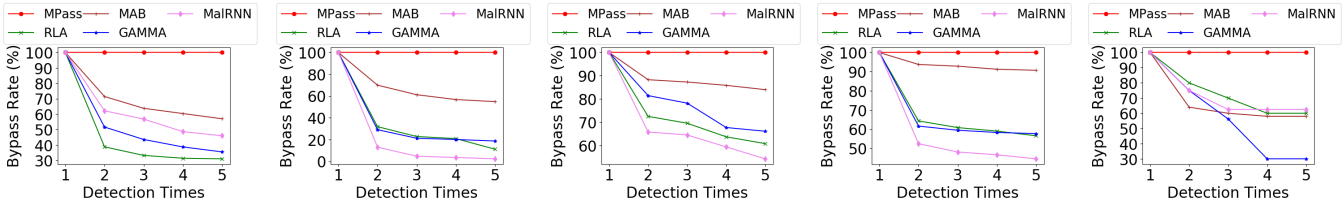
VII. Conclusion

We propose an effective black-box adversarial attack MPass against ML-based static malware detectors. We first use a problem-space explainability method to indicate malware's critical sections. Then,



a) Attack AV_1 b) Attack AV_2 c) Attack AV_3 d) Attack AV_4 e) Attack AV_5

Fig. 3: ASR of MPass compared with baseline attack methods for commercial ML AVs.



a) Attack AV_1 b) Attack AV_2 c) Attack AV_3 d) Attack AV_4 e) Attack AV_5

Fig. 4: Evaluation MPass and baseline attacks considering commercial ML AVs' learning on bypass rate.

we design a runtime recovery technique to preserve malware's functionality for modifying these critical sections. Finally, we formalize an optimization problem to modify malware to generate effective AEs. We conduct a thorough evaluation of four offline models and five commercial ML AVs by comparing MPass with several state-of-the-art attacks. The evaluation results show that AEs generated by MPass can effectively bypass the four offline models and five commercial ML AVs, and also significantly outperform baseline attacks.

Acknowledgments

This work was supported in part by the National Key Research and Development Program of China (2021YFB2701000), National Natural Science Foundation of China (61972224), China Telecom, and NSFOCUS (2022671026).

References

[1] E. Raff *et al.*, "Malware detection by eating a whole EXE," in *AAAI Workshops*, 2018.
[2] H. S. Anderson and P. Roth, "EMBER: an open dataset for training static PE malware machine learning models," *CoRR*, 2018.
[3] E. Raff *et al.*, "Classifying sequences of extreme length with constant memory applied to malware detection," in *AAAI*, 2021.
[4] S. Shukla *et al.*, "On-device malware detection using performance-aware and robust collaborative learning," in *DAC*, 2021.
[5] H. Kumar *et al.*, "Towards improving the trustworthiness of hardware based malware detector using online uncertainty estimation," in *58th ACM/IEEE Design Automation Conference, DAC*. IEEE, 2021.
[6] A. Souri and R. Hosseini, "A state-of-the-art survey of malware detection approaches using data mining techniques," *Hum. centric Comput. Inf. Sci.*, 2018.
[7] C. Szegedy *et al.*, "Intriguing properties of neural networks," in *ICLR*, 2014.
[8] A. Madry *et al.*, "Towards deep learning models resistant to adversarial attacks," in *ICLR*, 2018.
[9] K. Yang *et al.*, "3d-adv: Black-box adversarial attacks against deep learning models through 3d sensors," in *DAC*, 2021.
[10] G. Zizzo *et al.*, "Adversarial machine learning beyond the image domain," in *DAC*, 2019.
[11] N. Carlini and D. Wagner, "Towards evaluating the robustness of neural networks," in *IEEE SP*, 2017.
[12] W. E. Zhang *et al.*, "Adversarial attacks on deep-learning models in natural language processing: A survey," *ACM TIST*, 2020.
[13] I. Rosenberg *et al.*, "Generic black-box end-to-end attack against state of the art api call based malware classifiers," in *RAID*, 2018.
[14] M. Ebrahimi *et al.*, "Binary black-box evasion attacks against deep learning-based static malware detectors with adversarial byte-level language model," *CoRR*, 2020.
[15] W. Song *et al.*, "Mab-malware: A reinforcement learning framework for blackbox generation of adversarial malware," in *ASIA CCS*, 2022.
[16] L. Demetrio *et al.*, "Functionality-preserving black-box optimization of adversarial windows malware," *IEEE TIFS*, 2021.
[17] M. Cheng *et al.*, "Query-efficient hard-label black-box attack: An optimization-based approach," in *ICLR*, 2019.
[18] C. Tu *et al.*, "Autozoom: Autoencoder-based zeroth order optimization method for attacking black-box neural networks," in *AAAI*, 2019.
[19] F. Pierazzi *et al.*, "Intriguing properties of adversarial ML attacks in the problem space," in *IEEE SP*, 2020.
[20] W. Xu *et al.*, "Automatically evading classifiers: A case study on PDF malware classifiers," in *NDSS*, 2016.
[21] H. S. Anderson *et al.*, "Evading machine learning malware detection," *Black Hat*, 2017.
[22] L. Yu *et al.*, "Seqgan: Sequence generative adversarial nets with policy gradient," in *AAAI*, 2017.
[23] I. Rosenberg *et al.*, "Query-efficient black-box attack against sequence-based malware classifiers," in *ACSAC*, 2020.
[24] H. Aghakhani *et al.*, "When malware is packin' heat; limits of machine learning classifiers based on static analysis features," in *NDSS*, 2020.
[25] I. Rosenberg *et al.*, "Generating end-to-end adversarial examples for malware classifiers using explainability," in *IJCNN*, 2020.
[26] S. M. Lundberg and S. Lee, "A unified approach to interpreting model predictions," in *NIPS*, 2017.
[27] Y. Liu *et al.*, "Delving into transferable adversarial examples and black-box attacks," in *ICLR*, 2017.
[28] F. Tramèr *et al.*, "Ensemble adversarial training: Attacks and defenses," in *ICLR*, 2018.
[29] N. Papernot *et al.*, "Crafting adversarial input sequences for recurrent neural networks," in *IEEE MILCOM*, J. Brand, M. C. Valenti, A. Akinpelu, B. T. Doshi, and B. L. Gorsic, Eds., 2016.
[30] G. Sood, "Virustotal," https://www.virustotal.com/gui/home/upload, 2021.
[31] J.-M. Roberts, "Virusshare," https://virusshare.com/, 2021.
[32] W. Fleshman *et al.*, "Non-negative networks against adversarial attacks," *arXiv preprint arXiv:1806.06108*, 2018.
[33] H. S. Anderson, "Machine learning static evasion competition," https://github.com/endgameinc/malware_evasion_competition, 2019.