Dynamic Programming-based Adversarial Windows Payload Generator

Francis Kingful
Kwame Nkrumah University of Science and Technology
Emmanuel Ahene (aheneemmanuel@knust.edu.gh)
Kwame Nkrumah University of Science and Technology
Benjamin Appiah
Ho Polytechnic
Bill K. Frimpong
Ho Polytechnic
Isaac Osei
Ho Polytechnic
Ebenezer N. A. Hammond
CSIR Building & Road Research Institute

Research Article

Keywords: Adversarial Payload , Antivirus Evasion, Dynamic Programming, Malware Transferability, Shellcode

Posted Date: June 29th, 2023

DOI: https://doi.org/10.21203/rs.3.rs-3100627/v1

Additional Declarations: No competing interests reported.

Dynamic Programming-based Adversarial Windows Payload Generator

Francis Kingful [a], Emmanuel Ahene [a,*], Benjamin Appiah [b], Bill K. Frimpong [b], Isaac Osei [b] and Ebenezer N. A. Hammond [c]
[a] *Dept. Of Computer Science, KNUST, Ghana*
E-mails: fkingful@stdmail.knust.edu.gh, aheneemmanuel@knust.edu.gh
[b] *Dept. Of Computer Science, Ho Technical University, Ghana*
E-mails: bappiah@htu.edu.gh, bfrimpong@htu.edu.gh, iosei@htu.edu.gh
[c] *Building & Road Research Institute, Ghana*
E-mail: eahammond@gmail.com

Abstract. This work presents a behavior preserving Adversarial payload framework against static Windows malware scanners. The framework uses Dynamic Programming to decide on the sequence of static code transformation actions to transform a Windows payload to its adversarial state. In an empirical evaluation with Windows payloads from Metasploit Framework in a black-box settings, static machine learning based and majority of commercial antivirus scanners can still be evaded by these transformations. The potency of these generated Adversarial payload capable of breaching commercial antivirus on users' devices was demonstrated. The experimental results show a generated Adversarial Backdoor Trojan evade static and also evade its offline dynamic detector and establish a backdoor on the users' device.

Keywords: Adversarial Payload , Antivirus Evasion, Dynamic Programming, Malware Transferability, Shellcode

1. Introduction

Adversarial attacks have been investigated in the area of image, audios, texts and recently in Windows executable files classification exercise and a number of successful attacks examples in image, audios and texts have been generated to cause misclassification [1–7]. The principal reason for the success in image, audios and texts is that their feature-space is comparatively fixed, an image or text can be formatted as a three-dimensional array of pixels with each pixel value as a three-dimensional RGB (red, green, blue) vector value ranged from 0 to 255, thus, is feasible to find an exact function that is differentiable, therefore, a feature-space attack built on gradients can instantly apply on text or images to create adversarial attack examples. When generating Adversarial payload in Windows environment, the circumstance is substituted by the more challenging requirement that perturbations do not imperil the payload behavior chased by the adversary. Identifying efficient means to perturbing payload files to invade detectors without compromising their behavior has become a challenging task. The feature-space of Windows payload files is not fixed and can take various forms of feature engineering. This characteristic almost makes the payload feature-space impracticable to discover an approximate or exact function that is differentiable [8–13].

*Corresponding author. E-mail: aheneemmanuel@knust.edu.gh.

Initial observations from literature [5, 8, 9, 12–16], point out that code transformation actions such as; appending semantic nop no instructions, insertion of jump instructions and replace existing instructions, when applied on a software or an execuatble file can obfuscate the file against pirating or lower the file's true positive rate. In this work, we enhanced these aforementioned code transformation actions with Dynamic Programming based search method-a reinforcement learning algorithm, to increase their evasive potency against static malware scanners whiles satisfying the behavior preserving criteria. The code transformation actions are actions that will manipulate the representation of a malicious file, utilizing the technicalities and redundancies of shellcode format, while not disrupting its behavior at run-time.

The experimentation results show that, the generated adversarial payload managed to invade two popular Machine learning based detectors: EMBER ([17]) and ClamAV, in addition to some static commercial antivirus systems on VirusTotal. The potency of the attack capable of breaching commercial antivirus detectors that employ static engine was tested. The test results show an adversarial Trojan payload generated using our attack framework maintained it behavior, evade static and its offline dynamic detector and establish a backdoor connection to the attackers machine. Our contribution is mainly reflected as follows:

- We present a adversarial payload samples generation framework that used Dynamic Programming search algorithm to find optimal code transformations techniques that will led to evasive payload.
- We show that these transformations successfully leads to a behavior-preserving adversarial attack that can evade static Windows based machine learning malware scanners and commercial static AV engines.
- The generated adversarial payload can invade both static and offline dynamic detectors on user's machine.

The rest of this paper is organized as follows: Section 2 introduces the related background of our approach and Section 3 presents the behavior preserving adversarial attack framework. Section 4, deals with experiments and results. Section 5 looks at summary, and conclusions.

2. Related Background

2.1. The Attack Framework

Given an input payload X, the payload classifier returns a 2-dimensional vector $H(X) = [H_1(X), H_{-1}(X)]$, where $H_1(X)$ represent the probability that the input sample is a benign, $H_{-1}(X)$ represent the probability that the input sample is a malware, and satisfies the constraint $H_1(X) + H_{-1}(X) = 1$ and H represent the objective function. We aim to apply a transformation on X to cause a misclassification by a classifier.

2.2. Code Transformation Actions

The code transformations are common actions deployed by malware developers to hide all or parts of their implementation by appending semantic nop no instructions, injecting benign files, insertion of jump instructions, replacing existing instructions, encryption, packing and unpacking, manipulating executable file section names, etc on the original malware file while avoiding rapid analysis and detection by conventional static and signature-based techniques. MAB-Malware[8], used micro and macro transformations that tries to modify the PE file and applied Genetic Algorithm search on these transformations. Similar to MAB-Malware, Secml-Malware[16] also generated the adversarial malware by respectively

transforming the DOS header, through an extending and shifting transformation of the first section and applied Genetic Algorithm to find the optimal transformation that will lead to an evasion. Another work by [18] used these transformations in conjunction with Genetic Algorithm. Gym-Malware[5] used code transformation techniques optimized by Reinforcement Learning to write agents that learn to transform PE files based on a reward provided by taking specific transformation actions. Their code transformation process first define a set of actions performed on Windows PE header such as insert overlay bytes, packing and unpacking. MERLIN[14] and Pesidious[15] used actions techniques optimized by Reinforcement Learning algorithms to write agents that learn to manipulate PE files based on a reward provided by taking specific manipulation actions. Their code manipulation process first define a set of actions performed on Windows portable executable(PE) header such as insert overlay bytes, packing and unpacking, etc.

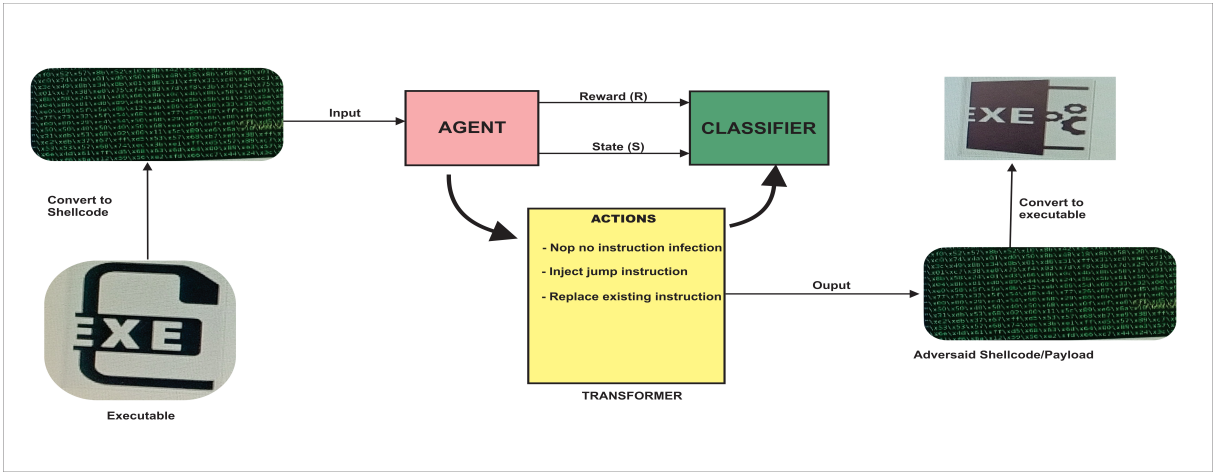


Fig. 1. Overview the attack framework.

3. Methodology

3.1. Overview

Figure 1, shows the overview of behavior preserving adversarial payload generation framework. The framework consist of an Action, Transformer and a Classifier. The agent's goal is to learn an optimal policy that maximizes the expected cumulative reward over time by taking the best possible action in each state. The Agent and the Transformer work together to mutate the payload sample based on a sequence of actions. These actions are code transformations that an agent can take in its environment. The mutated payload is sent to Classifier to retrieve a reward. The reward for each action is determined by the Classifier's score which is calculated as the difference in the score of the original payload and the mutated payload after every action. Details of the Classifier is presented in Sec 3.3. In Algorithm 1, given a well-trained Agent, when a payload is fed to the Agent, it constructs the best possible action sequence to maximize the cumulative reward. The action sequence is then followed by the Transformer (see Sect 3.4) to edit the payload and change it into a adversarial state.

Algorithm 1 Adversarial Windows Payload Generator

Require: Given X payload, π_x optimal policy for payload $x \in X$, alg the Agent's method for generating action sequences, x' the adversarial payload.

```
for x ∈ X do
    π_x ⟵ Agent(x, alg);
    x' ⟵ Transformer(x, π_x);
end for
```

3.2. Agent

Agent incorporates dynamic programming to search for an optimal code transformations when applied on x, can reduce the likelihood of the modified x being seen as malware and also preserve the original functionality of the input payload file with minimum number of bytes added.

3.2.1. Dynamic Programming

Dynamic programming-a class of the reinforcement learning algorithms, is used in this work to compute optimal policies given a perfect model of the environment as a Markov decision process (MDP). An MDP is a mathematical framework that models a sequential decision-making problem in a stochastic environment. In an MDP, an agent interacts with an environment by taking actions, receiving rewards, and transitioning from one state to another according to a set of transition probabilities. In the case of adversarial payload generation, the agent is a program that generates payload samples, and the environment is the antivirus classifier that the payload is trying to evade. The dynamic programming is used to find the optimal policy for the agent that maximizes the expected cumulative reward over time. The optimal policy is defined as the sequence of code transformations that leads to the highest expected cumulative reward, given the current state of the environment. Dynamic programming algorithm typically involve iteratively computing a value function for each state in the MDP. The value function represents the expected cumulative reward that can be obtained by following a particular policy from a given state. The value function is typically denoted as $V(s)$, where s is the current state. The specific steps from the DP algorithm employed in our work are as follows:

(1) Initialize the value function: The algorithm starts by initializing the value function $V(s)$ for all states $s \in S$ to 0. This is the initial guess for the value function. Here, s the payload and its adversarial variants.

(2) Iterate until convergence: The algorithm then iterates until the value function converges to an optimal solution. The convergence is guaranteed if a discount factor γ is less than one.

(3) Compute the new value function: For each state $s \in S$, the algorithm computes a new value function $V'(s)$ using the Bellman optimality equation. The Bellman optimality equation states that the optimal value function satisfies the following equation:

$$V(s) = max_a R(s,a) + gamma * sum'_s P(s,a,s') * V(s'),$$

where $R(s,a)$ is the expected reward for taking action a in state s, $P(s,a,s')$ is the probability of transitioning to state s' when taking action $a \in s$, γ is the discount factor that determines the importance of future rewards, and max_a denotes the maximum over all possible actions a. The reward strategy is motivated by [5].

(4) Compute the maximum difference: The algorithm then computes the maximum difference between the old value function $V(s)$ and the new value function $V'(s)$ for all states $s \in S$. This difference is denoted by ρ.
(5) Update the value function: If ρ is less than a certain threshold, the algorithm stops iterating and returns the converged value function $V(s)$. Otherwise, the algorithm updates the value function by setting $V(s) = V'(s)$ for all states $s \in S$ and repeats the iteration. In this work, the iterating ends once the score goes below a threshold of 80% as advised in Gym-Malware [5].
(6) Derive the optimal policy: Once the value function has converged, the algorithm derives the optimal policy by choosing the action with the maximum expected value for each state $s \in S$. The optimal policy is denoted by $\pi*(s)$ and is computed using the following equation:

$$\pi*(s) = argmax_a R(s,a) + \gamma * \sum_{s'} P(s,a,s') * V(s'),$$

where $argmax_a$ denotes the action that maximizes the expression inside the brackets. The optimal policy is the agent's action sequence that results in the final adversarial payload example.

3.3. Classifier

Creating adversarial payload in situations where access to a detection model's weights is restricted will always be necessary. In these circumstances, we evaluate our payload on MalConv [19], an end-to-end deep learning for malicious Windows portable executable file (PE) detection. The MalConv implementation and training in this work is based on the work [19]. We train MalConv on the Metasploit payloads. A successful transformation is achieved when MalConv classifier's an adversarial payload as benign.

3.4. Transformer

By using the strategies offered by Agent, Transformer controls the process of creating examples of adversarial payload. The Transformer iterates over actions in sequence and performs the corresponding action on the malware. After a series of editing, the original payload/malware is transformed into its adversarial version. In our context, these actions are: insert semantic nop no instructions, insert jump instruction and replace existing instructions inside the input payload file.

Nop no instructions insertion. Formally, this transformation is defined as $x^* = \{\delta_1, \delta_2, b_1, b_2, .., b_n, \delta_N\}$, b represent the bytes in the payload file and δ a perturbation in a form of nop no instructions. The transformation considers nop no instructions as it is the simplest dead code instruction available. The no-operation instruction is an assemble language instruction that does nothing but yet does not affect the program logic.

Insert Jump Instruction. This transformation changes the execution flow of the code using the concept of jump instruction insertion. For example, given an instruction in a shellcode, the jump instruction concept camouflages the instruction by two other normal instructions like *MOV, ADD, PUSH, etc*. The key factor is that the camouflaging instruction(s) should have size equal to or more than that of the jump instruction. This is to ensure that there are enough bytes in the program that can be modified to reconstruct the jump instruction during runtime. The number of additional jump instruction increases with the number of lines that are permuted, leading to different signatures and different memory mappings.

Replace Existing Instructions. This transformation recreates existing instructions in the payload file as different instructions with random and unique equivalents while preserving the original malicious

behavior and also making sure that no static data is left behind to trigger detection. For example, if a *MOV* instruction in a payload file with *0xB8* opcode is not changed the file can be detected just by looking for a *0xB8* opcodes. The transformation will convert each *MOV* instruction into several *ADD*, *SUB* or *XOR* instructions that will perform computation of the original immediate value. After replacement, the instruction is now completely different, but still after the last instruction is executed the original will still have its value as before.

4. Experiments

The experiment is conducted in a black-box setting and on a Kali GNU/Linux Rolling machine, version 2020.4, with Intel Core i5 and a 8GB of RAM. All scripts in our attack framework are implemented in Python. This experimentation answers three questions:

(1) Can a Dynamic programming search method based static codes transformation attack successfully leads to a behavior-preserving adversarial attack and evade static Windows based machine learning (ML) malware scanners?
(2) How transferable is the attack on different static commercial AV engines?
(3) Given that these commercial AVs installed on end users machines frequently use both offline dynamic and static detectors, how harmful are these attack to the end user?

4.1. Dataset

For this work, 100 staged Windows payload from Metasploit Framework are used. Due to Metasploit Framework popularity all if not majority of commercial antivirus systems and machine learning based detectors are well familiar with their payloads and so prepare their solutions accordingly with signatures that detect these payload families before they can cause damage.

4.2. Data representation, Transformation methods and Parameters

Nop no instructions insertion, *Insert Jump Instruction*, and *Replace Existing Instructions* are three transformation techniques used in this work. This, we take into account a restricted search space in our experimental investigation since a bigger search field can result in adversarial payloads that have poorer evasion rate as could signal malware scanners and search engines to recognize the created adversarial payload example or might leads to breakages [5, 8, 15, 18]. To that we also limit the length of transformation sequence to 3 passes, we noticed in the experimentation that beyond the 3rd transformation pass, the duration of the process increases dramatically.

We believe that in order to achieve a high evasion rate there should be a complete transformation of the payload file. The transformations in this work can achieve this obfuscation, however, not on the raw portable executable (PE) file, a practice adopted in state-of-the art attacks; Secml-Malware [16], MAB-Malware [8], Gym-Malware [5], etc. PE file structure is not flexible, therefore, we conduct our transformations in a raw shellcode environment. Shellcode format is malleable and can easily support complete transformation. Usually, Metasploit payloads are scripts, we convert these scripts into raw shellcode format. If any executable files were also used in our experimentation, they were also converted to raw shellcode format. A bit of disassembly was used to fingerprint several instruction types in the shellcode (i.e. particularly when performing the actions such as insert jump instruction and replace existing instructions inside the input payload file), for that purpose diStorm3 disassembler library was used.

	Evasive (E_r)	Insert nop no Instruction	Replace Existing Bytes	Insert Jump Instruction
ClamAV	94.73%	9/10	9/10	10/10
EMBER	88.98%	6/10	8/10	10/10

Table 1

The evasion rate results under each classifier on 100 staged Windows payloads and the number of evasive adversarial payload for each classifier under each actions on ten (10) randomly selected samples.

4.3. Evaluation Results

In this section, we report our evaluation results in terms of detection rate and evasive rate: the lower the detection rate and the higher the evasive rate, the better the proposed method performs because the adversarial payload produced can better evade detection.

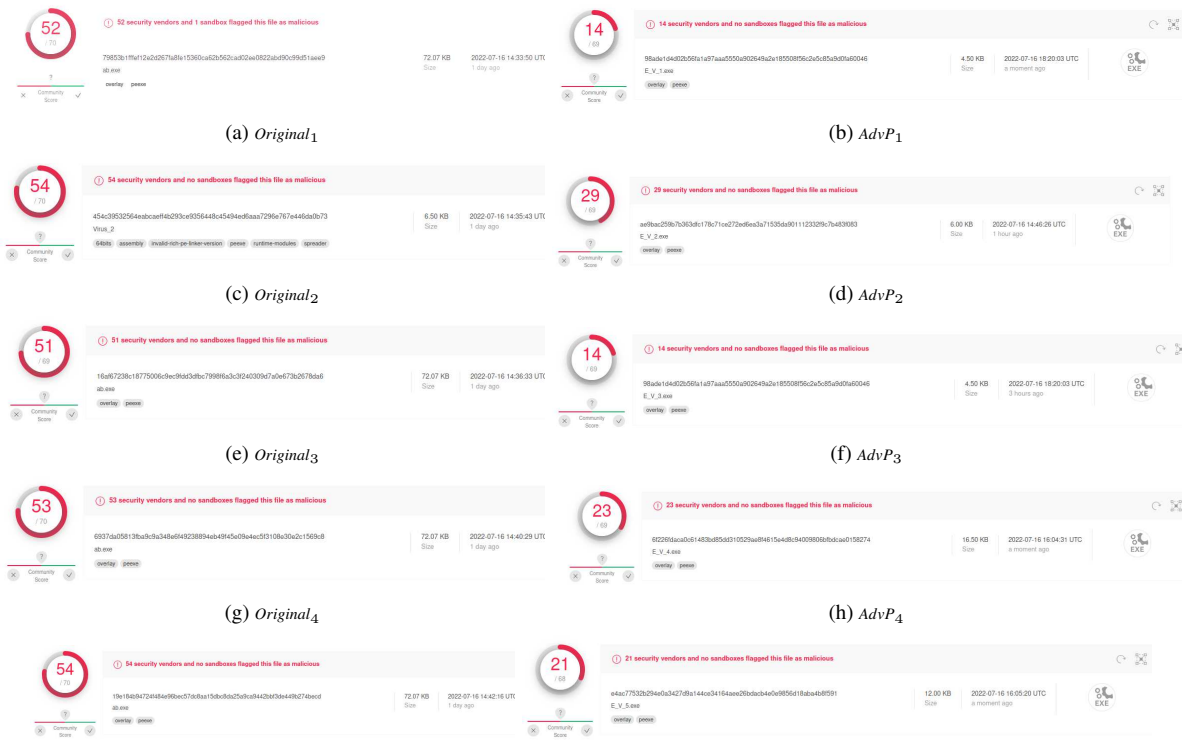
4.3.1. Transferability: evading other Static ML Classifiers

If the generated adversarial payload are transferable, then evading one classifier also imperils the other one. Here, this transformations are tested if they can pose a threat to other Machine learning based detectors; ClamAV and EMBER[17]. The performances on the EMBER is evaluated using the same Evasion Rate (E_r) presented in [9, 13]. ClamAV returns benign (0) or malicious (1). Table 1 shows the evasion strength of the adversarial example on the two popular AI-based classifiers. From the table it can be seen that, the evasion rate under ClamAV is the highest at 94.73%. The transformation performed on the payload file achieved high transferability rate on both classifiers.

To reveal which particular action is responsible for majority of the evasion, it is needed to break the transformation process into several micro-actions: insert nop no instruction, replace existing bytes, and insert jump instruction. For ClamAV, a simple action like injecting random bytes into the payload file is an effective action for fooling the classifier. Ember classifier recorded the highest detection rate compared to ClamAV, detecting 4 out of the 10 malicious samples as shown in Table 1. According to our experiments, inserting jump instruction and replacing existing bytes, resulted in majority of evasive samples. These results demonstrate that using these two is sufficient for generating effective adversarial examples. These classifiers are mostly created to work on specific problem sets, under the presumption that their training and test data are created from the same statistical distribution. Nevertheless, this presumption is often dangerously contravene in practical high-stake applications, where our transformation techniques violates these statistical assumption leading to the evasion rate recorded from these detectors.

4.3.2. Transferability: evading other static commercial scanners

Here, we test the evading capabilities of our attack framework on other commercial Antivirus scanners from VirusTotal.com, a popular online interface for many threat scanners. It is expected that most of the commercial Antivirus scanners will not be affected by such attacks. A test using five (5) random malicious samples were performed. We used limited samples for testing since analysis of a file on VirusTotal API takes time which slowed down the process and the analysis must reflect real-world scenario as possible. We assume that if our attack framework can successfully obfuscate few samples then the possibility of it obfuscating many should be high. Figure 2, shows the results of this experiment. From the table, figures in circle represents the number of antivirus these adversarial examples were able to bypass. It can be seen that the adversarial example $AdvP_1$ and $AdvP_4$ have the highest evasion rate to other classifiers with only 14 out of 69 antivirus were able to detect them, even the least $AdvP_2$ scores 29 out of 69 antivirus. VirusTotal does not show the full detection platform for vendors in VirusTotal, nevertheless, these scanners manifest a weakness against transformations used for this investigation.



(a) $Original_1$ (b) $AdvP_1$

(c) $Original_2$ (d) $AdvP_2$

(e) $Original_3$ (f) $AdvP_3$

(g) $Original_4$ (h) $AdvP_4$

(i) $Original_5$ (j) $AdvP_5$

Fig. 2. VirusTotal scanned results for five (5) original payloads and their adversarial counterparts.

4.3.3. Functionality Verification

The Cuckoo sandbox is used to verify the behavior of the adversarial example. Cuckoo sandbox gathers sample behavior from the payload and transforms it into understandable descriptive signatures. Each signature is a text string that encapsulates a particular sample behavior. The behaviors of the transformed Trojan are compared with the original one, if a transformed Trojan has the same behavior as the original one, it is considered as an evasive example. The behavior similarity between two example is defined as the two example sharing the same behavioral functions. Otherwise, it is considered that the transformation has changed the behaviors of the original Trojan making it non-functional. In Figure 3, each signature in (a) is checked and search to see if the same signature exists in (b). It can be seen that the adversarial Trojan shares the same behavioral function (i.e NtAllocateMemory) as the original. Nevertheless, even with carefully replacing existing instructions with alternative ones, making sure that no static data is left behind that may aid in creation of detection signatures, still this work don't deny the possibility that some generated adversarial payload might become nonfunctional after the transformation.

4.3.4. Adversarial Payload Harm to End Users

We determine, how easily these transformation attack bypass commercial antivirus and infect users is analyzed. It is assumed that the offline behavioral and dynamic malware classifiers of the antivirus will recognize and stop the adversarial example bypassing the the static-only classifiers when they are executed, thus, posing no real harm to the end users. To answer this research question, an adversarial Trojan

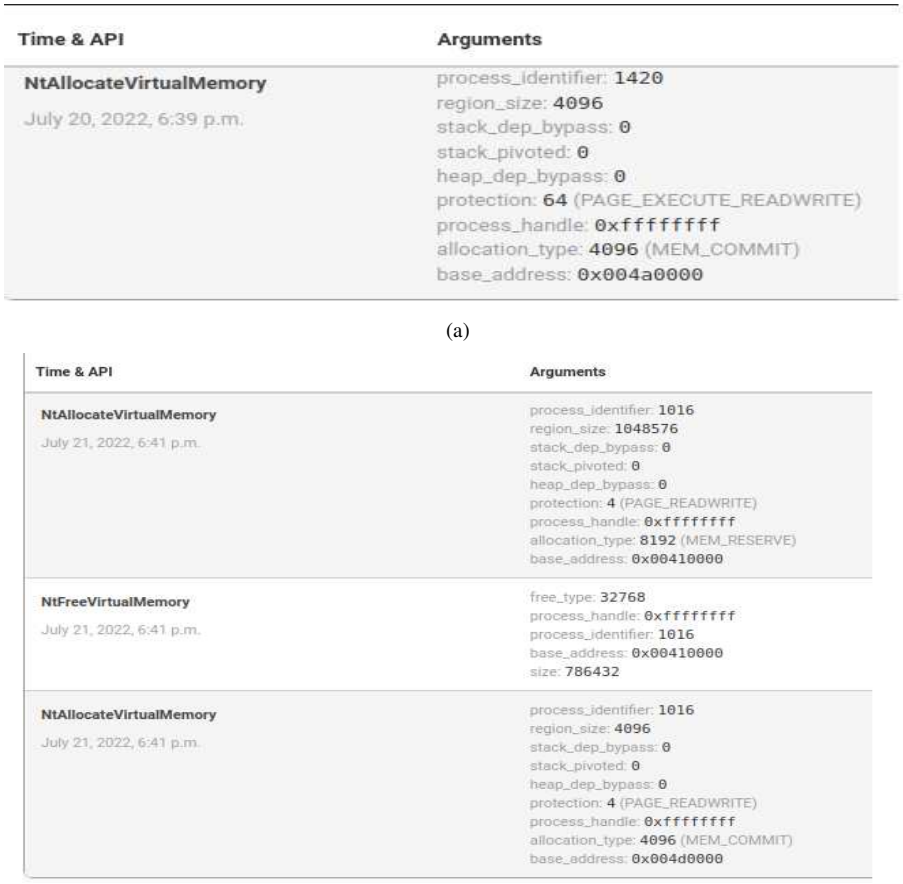


Fig. 3. Screenshot of Cuckoo Sandbox analysis showing a transformed Trojan payload (b) with same behavior as the original (a).

is generated using the proposed transformation attack and test whether the transformed Trojan that evade static classifier can actually infect users' machine and create a backdoor to an attacker. The adversarial Trojan selected was evaluated by the Cuckoo sandbox signature to be functional. In Figure 4 (b), the Windows 10 antivirus detected and deleted the original Trojan sample (i.e. "*virus2.exe contained a virus and was deleted*"). However, adversarial Trojan is the exception. It evade the behavior detector of the Windows 10 machine and establish a meterpreter connection with the attackers' machine. In Figure 4(c), the Windows 10 machine only shows "Unknown Publisher" warning from Windows SmartScreen under adversarial Trojan attack but does not delete nor stop the meterpreter connection. Windows SmartScreen is a useful feature built into Windows 8 and 10 to block the download of programs or files from malicious websites or from an Unknown publisher. However, adversarial example could be injected into legitimate programs with known publisher which will be convincing enough to run. This represents an attack surface for adversarial example and a recommendation for future antivirus systems. Anti-virus systems rely on both static and dynamic detection to detect malicious file. However, since static detectors are easy to evade, Antivirus need a robust offline dynamic detection to protect their clients from

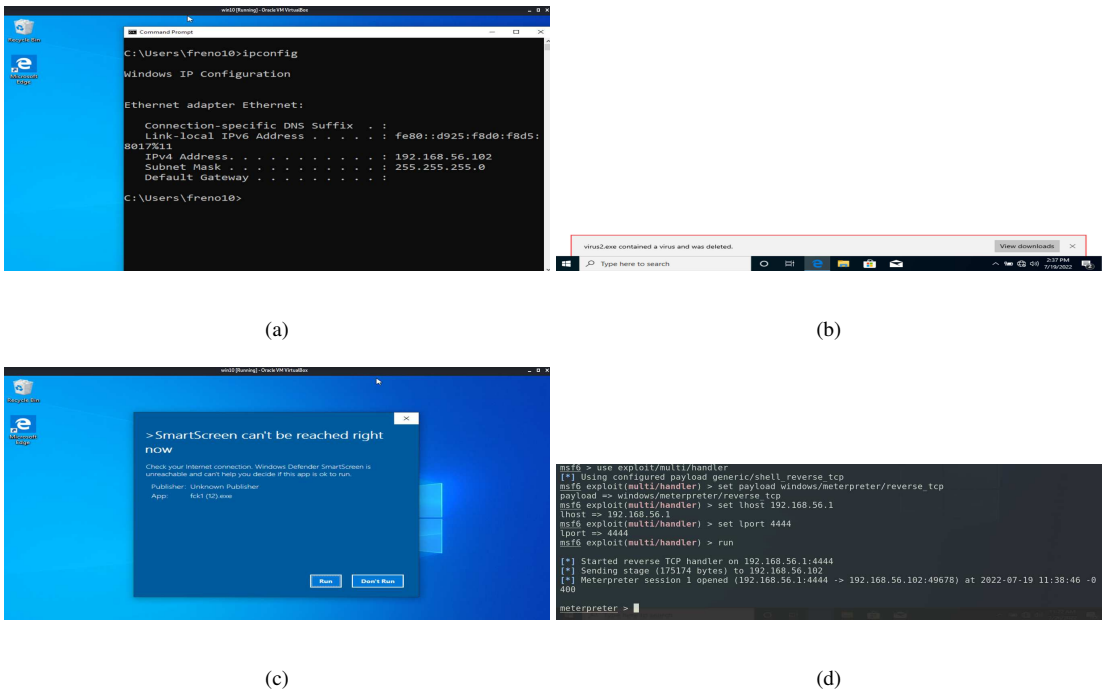


Fig. 4. Screenshots of the extent to which adversarial example evades antivirus and infect users. (a) The Windows 10 IP Address of 192.69.56.102. (b) The Windows 10 successfully detects and deletes the original payload. (c) The Windows 10 failed to detect our transformed payload as a malware/virus. (d) A meterpreter connection from the Windows 10 client system to the attacker.

malicious programs. A popular Windows inbuilt antivirus scanner is tested and it lucks a robust offline dynamic detection to detect the adversarial example when the example is executed.

	$AdvP_1$	$AdvP_2$	$AdvP_3$	$AdvP_4$	$AdvP_5$
Secml_Malware	47 / 71	49 / 71	49 / 71	51 / 71	48 / 70
MAB-Malware	45 / 71	52 / 71	50 / 71	51 / 68	51 / 70

Table 2
VirusTotal detection results for five (5) adversarial payloads from Secml_Malware and MAB-Malware.

4.3.5. Compare with Other Works

To illustrate the advantage of the proposed transformation, it is compare to two available state-of-the-art black-box Adversarial attacks against PE payload detection framework: Secml-Malware [16] and MAB-Malware [8]. The Secml-Malware and MAB-Malware generators are evaluated against commercial antivirus on the same five (5) random payload samples used in in Figure 2. Evaluating on five samples should not be classified as cherry picking since these models are purposely designed to generate adversarial examples, we assume that if they were successful in generating many examples in their

original works then the possibility of them being successful on few should be higher. Secondly, another reason of using few payload samples is due to significantly low response times from VirusTotal API.

To eliminate any future argument of using the wrong settings, this experiment strictly follow the Adversarial payload generation settings presented in the manuals and Github pages of Secml-Malware and MAB-Malware. Secml-Malware is a plugin for the SecML Python library. It contains many kinds of attacks, however, in this experiment, only its genetic programming-based black-box attack is utilized. In the selection step, it supports both confidence score-based selection and hard label-based selection. It is assumed that the attackers cannot get the confidence score in the threat model generation. So only the hard label-based attack is evaluated. Secml-Malware only utilizes transformation techniques such as benign content injection, shifting and appending. Genetic Algorithm is used to find an optimal transformations that will lead to evasion. MAB-Malware is a Reinforcement Learning-based payload manipulation environment. The payload generation problem is modeled as a classic multi-armed bandit (MAB) problem, by treating each transformation pair as an independent slot machine. Each machine's reward is modeled as a Beta distribution and use Thompson sampling to select the next transformation, striking a balance between exploitation and exploration. An action is devised to minimization process, which minimizes an Adversarial by removing redundant actions and further reducing essential actions into even smaller actions. Rewards are then assigned only to these essential micro-transformations. This minimization process also helps interpret the root cause of evasions. Readers are referred to the Secml-Malware and MAB-Malware Python libraries presented in their respective works for full understanding of implementation. Table 2, presents the screenshots detailing Secml-Malware and MAB-Malware evasion results from antivirus scanners.

Comparing the VirusTotal scanned results in Figure 2 and Table 2, it can be seen that the optimized transformation greatly improves the evasion rate whiles the Secml-Malware and MAB-Malware generators barely provides any improvement. The evasion results from Table 2 indicate that the search strategies used in Secml-Malware and MAB-Malware respectively, does not learn meaningful knowledge to guide the evasion. These evasive learning process miss other sections in the payload PE file containing static data that can be easily identified with signature detection. Unlike Secml-Malware and MAB-Malware, the framework convert the PE file into standard shellcode format and perform the transformation on it. The effect of the transformation is evasive, since the whole structure of the malicious file is modified leaving less static data available which might be known by detection engines, hence evading many static scanners.

4.4. Discussion

The results presented in the previous sections show the worrying nature of adversarial attack. As the above results confirm that payload classifiers are not efficient in detection these attack strains as consumers are made to believe. By using a black-box technique, the proposed attack ensures that the adversarial example preserve it malicious behavior and effective against static payload classifiers.

These classifiers are mostly designed to work on specific problem sets, under the assumption that their training and test data are generated from the same statistical distribution. However, this assumption is often dangerously violated in practical high-stake applications, where the complete transformation violates these statistical assumption leading to the high evasion rate recorded from these classifiers. The results of the harm to end users (Figure 3) can be considered in many cases, catastrophic. Malware detectors, needs a strong offline dynamic detection to safeguard their clients from malicious programs because static detectors are simple to evade. During testing, a well-known Windows-based antivirus

program failed to detect the adversarial payload when it was actually being executed. Nevertheless, the generated payload samples can also be used to train classifiers and detectors because both dynamic and static features can be extracted from these payload strains.

5. Conclusion

This study provides an argument to use Dynamic Programming to optimize code transformations techniques to generate adversarial payload. These transformations were evaluated on staged Windows payload samples from Metasploit Framework. The transformed payload samples were tested on static ML classifiers such as EMBER, ClamAV, and static commercial antivirus scanners on VirusTotal. The adversarial payload achieved high evasion rate on all classifiers and commercial antivirus systems. They were able to bypass many popular antivirus scanners whiles preserving their behavior. The future works intent to increase the sample size, and also include non-staged payloads samples with larger file size, dynamic analysis and seek for stealthy examples.

Declarations

- **Funding:** This work is partially supported by the Department of Computer Science, Kwame Nkrumah UNiversity of Science and Technology of Ghana.
- **Conflict of interest:** The authors declare that they have no conflicts of interest to report regarding the present study.
- **Ethics approval:** All experimental data in this paper are real and valid, and there is no case of multiple submissions for one manuscript in this article.
- **Availability of data and materials:** This article uses samples dataset from github.
- **Code availability:** Due to the sensitivity of this work, codes are only released upon strict formal request.
- **Authors' contributions:** All authors played equal role in the development of this work.

References

[1] K. Rieck, P. Trinius, C. Willems and T. Holz, Automatic analysis of malware behavior using machine learning, *J. Comput. Secur.* **19**(4) (2011), 639–668. doi:10.3233/JCS-2010-0410.

[2] I.J. Goodfellow, J. Shlens and C. Szegedy, Explaining and Harnessing Adversarial Examples, in: *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, Y. Bengio and Y. LeCun, eds, 2015.

[3] N. Papernot, P.D. McDaniel, S. Jha, M. Fredrikson, Z.B. Celik and A. Swami, The Limitations of Deep Learning in Adversarial Settings, in: *IEEE European Symposium on Security and Privacy, EuroS&P 2016, Saarbrücken, Germany, March 21-24, 2016*, IEEE, 2016, pp. 372–387. doi:10.1109/EuroSP.2016.36.

[4] C. Szegedy, W. Zaremba, I. Sutskever, J. Bruna, D. Erhan, I.J. Goodfellow and R. Fergus, Intriguing properties of neural networks, in: *2nd International Conference on Learning Representations, ICLR 2014, Banff, AB, Canada, April 14-16, 2014, Conference Track Proceedings*, Y. Bengio and Y. LeCun, eds, 2014.

[5] B.F. Hyrum. Anderson Anant Kharkar, Evading Machine Learning Malware Detection (2017).

[6] B. Appiah, Z. Qin, A.M. Abra and A.J.A. Kanpogninge, Decision tree pairwise metric learning against adversarial attacks, *Computers & Security* **106** (2021), 102268.

[7] B. Appiah, E.Y. Baagyere, K. Owusu-Agyemang, Z. Qin and M.A. Abdullah, Multi-Class Triplet Loss With Gaussian Noise for Adversarial Robustness, *IEEE Access* **8** (2020), 171664–171671. doi:10.1109/ACCESS.2020.3024244.

[8] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov and H. Yin, MAB-Malware: A Reinforcement Learning Framework for Blackbox Generation of Adversarial Malware, in: *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, Association for Computing Machinery, New York, NY, USA, 2022, pp. 990–1003–. ISBN 9781450391405. doi:10.1145/3488932.3497768.

[9] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov and H. Yin, Automatic Generation of Adversarial Examples for Interpreting Malware Classifiers, *CoRR* **abs/2003.03100** (2020).

[10] A. Al-Dujaili, A. Huang, E. Hemberg and U. O'Reilly, Adversarial Deep Learning for Robust Detection of Binary Encoded Malware, in: *2018 IEEE Security and Privacy Workshops, SP Workshops 2018, San Francisco, CA, USA, May 24, 2018*, IEEE Computer Society, 2018, pp. 76–82. doi:10.1109/SPW.2018.00020.

[11] L. Chen, Y. Ye and T. Bourlai, Adversarial Machine Learning in Malware Detection: Arms Race between Evasion Attack and Defense, in: *European Intelligence and Security Informatics Conference, EISIC 2017, Athens, Greece, September 11-13, 2017*, J. Brynielsson, ed., IEEE Computer Society, 2017, pp. 99–106. doi:10.1109/EISIC.2017.21.

[12] W. Hu and Y. Tan, Generating Adversarial Malware Examples for Black-Box Attacks Based on GAN, *ArXiv* **abs/1702.05983** (2017).

[13] H.S. Anderson, A. Kharkar, B. Filar, D. Evans and P. Roth, Learning to Evade Static PE Machine Learning Malware Models via Reinforcement Learning, *CoRR* **abs/1801.08917** (2018).

[14] T. Quertier, B. Marais, S. Morucci and B. Fournel, MERLIN - Malware Evasion with Reinforcement LearnINg, *CoRR* **abs/2203.12980** (2022). doi:10.48550/arXiv.2203.12980.

[15] C. Vaya and B. Sen, 2020.

[16] L. Demetrio, S.E. Coull, B. Biggio, G. Lagorio, A. Armando and F. Roli, Adversarial EXEmples: A Survey and Experimental Evaluation of Practical Attacks on Machine Learning for Windows Malware Detection, *ACM Trans. Priv. Secur.* **24**(4) (2021), 27:1–27:31. doi:10.1145/3473039.

[17] H.S. Anderson and P. Roth, EMBER: An Open Dataset for Training Static PE Malware Machine Learning Models, *CoRR* **abs/1804.04637** (2018).

[18] L. Demetrio, B. Biggio, G. Lagorio, F. Roli and A. Armando, Efficient Black-box Optimization of Adversarial Windows Malware with Constrained Manipulations, *CoRR* **abs/2003.13526** (2020).

[19] E. Raff, J. Barker, J. Sylvester, R. Brandon, B. Catanzaro and C.K. Nicholas, Malware Detection by Eating a Whole EXE, in: *The Workshops of the The Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, Louisiana, USA, February 2-7, 2018*, AAAI Technical Report, Vol. WS-18, AAAI Press, 2018, pp. 268–276. https://aaai.org/ocs/index.php/WS/AAAIW18/paper/view/16422.