H4rm0ny: A Competitive Zero-Sum Two-Player Markov Game for Multi-Agent Learning on Evasive Malware Generation and Detection

Christopher Molloy*, Steven H. H. Ding*, Benjamin C. M. Fung†, and Philippe Charland‡
*School of Computing, Queen's University, Kingston, Canada. Emails: {chris.molloy, steven.ding}@queensu.ca
†School of Information Studies, McGill University, Montreal, Canada. Email: ben.fung@mcgill.ca
‡Mission Critical Cyber Security Section, Defence R&D Canada - Valcartier, Quebec, QC, Canada.
Email: philippe.charland@drdc-rddc.gc.ca

***Abstract*—To combat the increasingly versatile and mutable modern malware, Machine Learning (ML) is now a popular and effective complement to the existing signature-based techniques for malware triage and identification. However, ML is also a readily available tool for adversaries. Recent studies have shown that malware can be modified by deep Reinforcement Learning (RL) techniques to bypass AI-based and signature-based anti-virus systems without altering their original malicious functionalities. These studies only focus on generating evasive samples and assume a static detection system as the enemy.**

Malware detection and evasion essentially form a two-party cat-and-mouse game. Simulating the real-life scenarios, in this paper we present the first two-player competitive game for evasive malware detection and generation, following the zero-sum Multi-Agent Reinforcement Learning (MARL) paradigm. Our experiments on recent malware show that the produced malware detection agent is more robust against adversarial attacks. Also, the produced malware modification agent is able to generate more evasive samples fooling both AI-based and other anti-malware techniques.

***Keywords*: Adversarial learning, Malware analysis, Neural networks, Reinforcement learning, Markov decision process**

I. Introduction

Each day, more than 500,000 new and never-before-seen malware samples are recorded [1]. Development in recent years seen by malware researchers is the application of adversarial learning on malware. Adversarial learning is the domain of research focusing on the techniques for evading learning models. With the growing reliance on machine learning in malware detection, evasion techniques have become a popular tool used by malware developers. During 2019, the rate at which evasive malware was detected rose from 35% to over 66% by WatchGuard [2]. One example was the Cylance PROTECT system's evasive vulnerability. In 2019, it was shown that by appending benign strings from video games, a piece of malware could evade the AI detection system [3]. The company quickly resolved the problem and released patches to the endpoint systems. This brings us to think about the possibility of a more proactive and robust solution: designing a detection model already foreseeing this specific move by the adversary as well as other possible moves, so as to prevent similar vulnerabilities in the first place.

Malware development and detection have become a never-ending cat and mouse game. Adversarial learning on malware is still a relatively new research area, and many successful adversarial techniques that evade anti-malware engines are manually generated on a per-sample basis, such as in the previous Cylance example. Recent research has shown that by using reinforcement learning, systems can be created that modify malware to be evasive on a massive scale [4]. These reinforcement learning systems do not only work against signature-based methods, but also work well against machine learning malware detection systems [4], [5]. To address the challenges of adversarial malware attacks, typical solutions include recognizing file specific features within the dataset and training anti-malware engines with adversarial samples. These solutions can be seen in the Cylace PROTECT system and recent work proposed by Zhang *et al*[1] [6].

In contrast to static adversarial training, we propose a different direction simulating a real-life scenario: a dynamic and competitive game between two agents. One agent is tasked with producing functional adversarial samples of Microsoft Portable Executable (PE) malware. The other agent is an anti-malware agent, classifying malware generated by the first agent and learning the malware. In our game, one agent (modification agent) will be given a set of tools to modify malware and the other agent (detection agent) will be tasked with learning the modification methods. Our multi-agent learning system, H4rm0ny, consists of a convolution neural network and a Markov decision process reinforcement learning model. H4rm0ny builds on top of the models proposed by Raff *et al*. and Anderson *et al*. [4], [7]. The model proposed by Raff *et al*. is a gated convolution neural network that has been designed for Microsoft PE malware detection [7]. The Markov decision process proposed by Anderson *et al*. is a reinforcement learning application that is trained to modify malware to be miss-classified against a specified malware detection system [4]. These malware modifications occur in the problem space to keep the malware functional as the original.

978-1-6654-9952-1/22/$31.00 ©2022 IEEE

[1] Adversarial ML: How Artificial Intelligence is Enabling Cyber Resilience

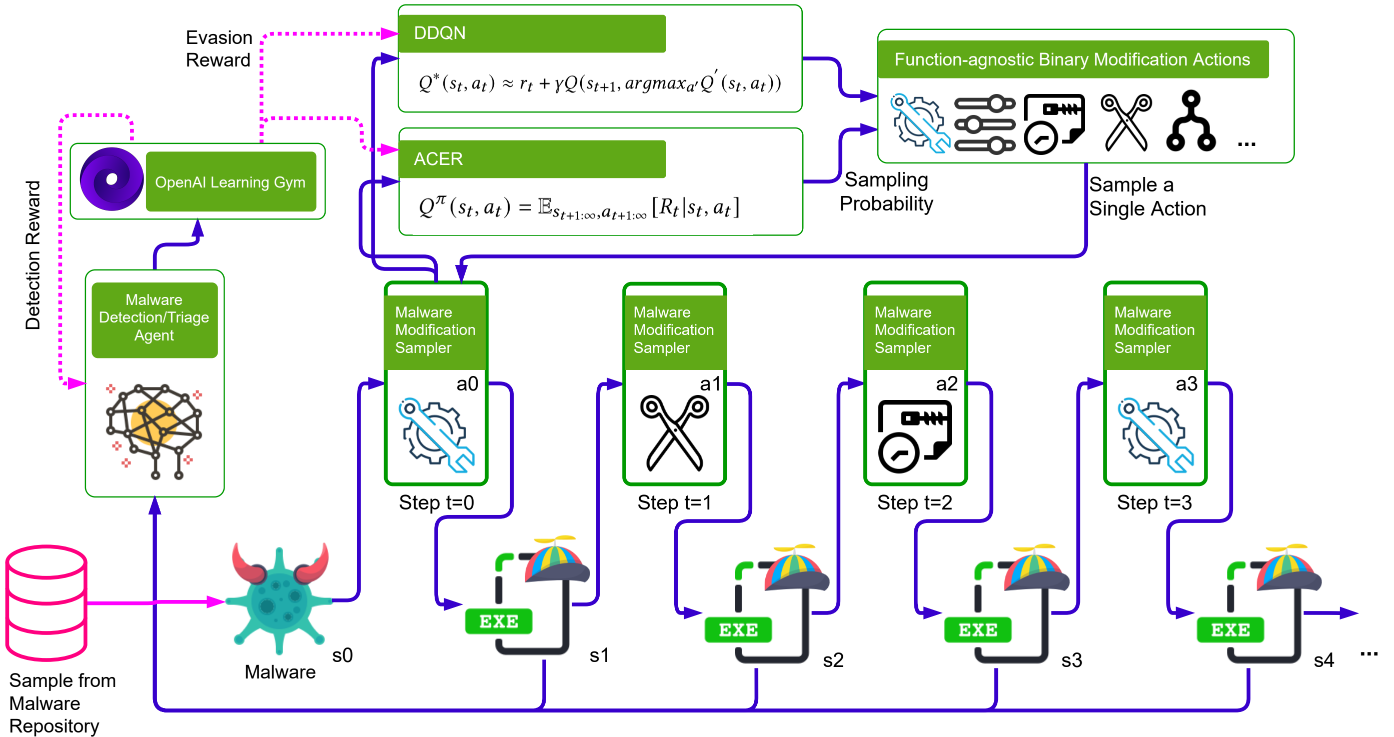

Fig. 1: The H4rm0ny training process for a single episode. Results from all system configurations are also shown. A sample piece of malware is chosen from our dataset. It is then sent through a process of modifications. If any modification produces an evasive sample of the malware, that sample is trained into the detection agent. Once the sample has been up-trained into the detection agent, the modification agent policy is updated with the set of actions performed on the malware sample and the state of the detection agent.

The modifications that we are using have been shown to keep the functionality of the malware [4]. The main contributions of this paper are as follows:

- This is the first work that designs and implements a two-party competitive gameplay environment for evasive malware generation and detection.
- We proposed a custom reward loss function with a special up-training function to balance the gameplay for the agents to learn from both benign and malicious samples. Integrated with Actor-Critic with Experience Replay (ACER) and Double Deep Q-Network (DDQN) for the malware generation agent [8], [9], and MalConv for continuous action space malware detection agent, the gameplay becomes stable and starts converging.
- We evaluate different agent designs on recent malware samples and show that the malware generation agent can generate more evasive samples, and the detection agent is now more robust against adversarial samples.

II. Methodology

The H4rm0ny system consists of a malware modification agent and a malware detection agent. These two agents compete with each other in a game of modifying and detecting malware. This game is played in a gym environment, where the modification agent is given a toolbox of functionality-preserving modifications that it can make to the incoming malware sample. In each turn of the game, the modification agent has multiple attempts to obfuscate a single malware sample to evade the detection agent. If the malware sample evades the detection agent, the detection agent will be up-trained with the sample, the modification agent will be trained with the information of the turn, and the turn will end. If the maximum number of attempts is reached, the detection agent is up-trained with the sample and the modification agent will then be trained with the turn information, and the turn will end. This format of gameplay is in line with a turn-based zero sum game, this will be further elaborated on in the next subsection. A turn-based zero-sum game was chosen to give both agents an equal impact in the outcome of the game. At the end of the game, the modification agent will create a training set of malware which will be trained into the detection agent. The resulting models that are produced from this game are better at their designed task than models trained statically.

A. A Turn-based Zero-Sum Game

In game theory, a zero-sum game is when one player's gain is equivalent to another player's loss. We define our zero-sum game as a tuple (X, Y, f), where X and Y represent the respective action space to be taken by Player X and Player Y [10]. We set Player X as the malware modification agent that aims at generating evasive malware samples and Player Y, as the malware detection agent that classifies a given sample.

Therefore, X is the set of possible modifications that Player X can make to a sample without changing its original behaviors and Y is a set of discrete values $\{0, 1\}$ indicating if the given sample is benign or malicious. Function $f : X \times Y \rightarrow \mathbb{R}$ represents the mapping from any combination of actions taken to a real-valued reward for Player X at a certain time step. In this game, each of the learning agents is always confronted with an opponent learning agent that is of comparable strength, and the goal of the learning system is to find an equilibrium state of the min-max game. The convergence of the learning system, finding the optimal equilibrium state, is not guaranteed. Instead, the learning system in practices may reach an approximate equilibrium state with a probability [10].

In the context of the H4rm0ny system, the mapping f will go to the reward function of the malware modification agent R_t at time t. The game will play in episodes. In each episode, the game is given a random malware sample. The modification agent tries to make it evasive and the detection tries to detect it as malicious. Time t represents the time at which a specific turn has occurred in an episode. A single turn is defined as the modification agent making a specified amount of modifications to the sample based on the output of the detection agent at each modification step. In our game, there are two possible values for our reward function R_t. The values are -10 and 10. Our modification agent will receive a reward of -1 multiplied by the reward given to the detection agent. If our modification agent is given a reward of 10, then our detection agent will be given a reward of -10. The reward is given to our detection agent by a scalar multiplied by the classification of the malware sample and multiplied by the loss of the malware detection agent during the up-training process of the malware sample. As described above, the convergence of H4rm0ny to an equilibrium state is not guaranteed. We have ensured the highest probability of equilibrium through empirical testing and defining the game rules as shown above. At equilibrium, the generation agent is able to produce malware more evasive than statically generated evasive sample, and the detection agent is able to detect adversarial samples better than the same detection agent that has not played the game.

B. Markov Decision Process and Q-Learning

Our malware modification agent is a reinforcement learning framework that builds on the architecture proposed by Andersen *et al.* [4]. Our reinforcement learning agent has a set of functionality-preserving operations that it may perform on the PE file. Our malware modification agent is trained by attempting to generate malware that can evade our malware detection agent. The learning framework used by Anderson *et al.* is a Markov decision process. The Markov decision processes consist of the tuple $(S, A, \gamma, R(S, A))$. S: a set of environment states. Our environment state is the maliciousness prediction that our sample PE is given by the malware detection agent. Our state set is the set of continuous values between 0 and 1. A: a set of actions. Our H4rm0ny system has an action space which is a set of functionality-preserving actions that can be used by the malware modification agent to modify a piece of malware. γ: the discounted factor $\gamma \in [0, 1)$. The discount factor is used to control the importance of immediate and future rewards [8]. $R(S, A)$: the set of rewards to give to the modification agent depending on the state of the environment and actions used. $s_t \in S$ is the returned state at time t, and $a_t \in A$ is the action at time t. In the Markov decision process, the chosen agent sends an action to obfuscate the malware before it is sent through the anti-malware engine. The action a_t and a vector describing the state of environment s_t are then sent back to the agent to be trained. This is a process of sequentially training the network directly based on the previous result. The agent is trained through a specified policy algorithm. The policy algorithm is the method chosen for updating the probability distribution of the action set A. When our modification agent is given the output of a training step, $R(S, A)$, the policy algorithm will update the probability distribution of the action set A. This distribution is the probability that each action $a \in A$ has of being chosen by the modification agent to modify the current sample PE.

The objective of our malware modification agent is as follows. Through sequentially training and updating our policy algorithm, a model will be produced that achieves the highest expected reward $r_{t+1} = R(s_t, a_t)$ given our current policy algorithm π_θ. This will hold for all time steps t up until the terminating time $T - 1$. The objective function is formulated in (1). Here, θ are the parameters of our policy algorithm.

$$J(\theta) = \mathbb{E}[\sum_{t=0}^{T-1} r_{t+1} | \pi_\theta] \tag{1}$$

We update our policy algorithm parameters by taking the gradient ascent of our objective function.

$$\theta \leftarrow \theta + \frac{\partial}{\partial\theta} J(\theta) \tag{2}$$

From this, our policy gradient can be derived as follows.

$$\bigtriangledown_\theta J(\theta) = \mathbb{E}_\tau[\sum_{t=0}^{T-1} \bigtriangledown_\theta log \pi_\theta(a_t|s_t])G_t] \tag{3}$$

In (3), G_t is the learning agent specific function and τ is used to represent our given trajectory from s_t and a_t. Once optimized by (3), the parameters that form our objective function are then updated into a neural network, which is representing the policy algorithm.

The objectives of the H4rm0ny system are as follows. First, create a malware modification agent that modifies malware to maximize the false-negative rate of malware against an anti-malware engine. The second objective is to create a malware detection agent such that the classification error against adversarially obfuscated malware is minimized.

C. Evasive Malware Generation Agent

The purpose of the evasion malware generation agent in our game is to learn how to make evasive malware. For each turn in the game, our generation agent is given a number of attempts to modify the malware to evade the detection agent. Depending on the outcome of the turn, the generation agent will be given a reward. If the agent can produce a piece of evasive malware, it will be given a larger reward than if it cannot. Our generation agent is trained to maximize the reward. It is important to note that we do not change the reward based on how many episodes are needed to produce the evasive sample. When the game is over, our modification agent can be used to modify unseen malware with a specified number of modifications.

Our action space is a set of modifications that can be chosen by our modification agent when modifying a malware sample. These modifications have been chosen for our action space, because they do not change the form or function of the sample they are being acted on [4]. The number of total modifications has been kept low, because the larger the action space in a reinforcement learning system, the more time an agent will take to fully converge in training [11]. Additionally, it was known that similar actions are sufficient to generate evasive malware against anti-virus engines [4]. First, our modification agent will parse through the incoming malware file. It will then begin testing the modifications against the detection agent. The modifications are described as follows: **Rename section**: A random section from the PE sample is chosen and replaced with a random section from a list of common benign section names. **Add bytes to section cave**: If any byte cave exist in the PE sample, fill the cave with a random byte value. **Modify machine type**: The machine type of the PE sample is changed to a random machine type. **Modify timestamp**: The time date stamp in the PE header is changed to a value from a random selection. **Pad overlay**: A random byte value is appended at the end of the malware sample 100,000 times. **Append benign data overlay**: The `.text` section of a randomly chosen benign PE file is appended at the end of the PE sample. **Append benign binary overlay**: A random benign PE file is read as bytes and appended at the end of the PE sample. **Add section benign data**: An unused section is created in our PE sample and the contents of a random benign file .text are added to the new section. **Add section strings**: A string is selected from a random set of benign strings and set as the section contents of a newly generated section in our PE sample. **Add string overlay**: A string is randomly selected from a set of benign strings and appended at the end of the PE sample. **Add imports**: A random library is chosen from a set of libraries commonly used by benign files and added to the PE sample if it does not already exist. **Remove debug**: If a data directory debug exists in the PE sample, it is deleted. **Modify optional header**: A random value for a random optional header is chosen and added to the optional header list of the PE sample. **Break optional header checksum**: The optional header checksum of the PE sample is set to 0. **UPX unpack**: The bytes of the PE sample are unpacked using UPX. **UPX pack**: The bytes of the PE sample are packed using UPX.

There are many deep reinforcement learning agents that we could have applied for this task. We adopted two recent works: Actor critic with experience replay (ACER) and the Double Deep Q-Network (DDQN). ACER is a deep reinforcement learning agent that has been shown to perform remarkably well in challenging environments [8]. ACER was the agent used by Anderson *et al.* in their work showing that reinforcement learning could be used for adversarial sample generation against static networks [4]. We chose ACER due to the algorithm already being shown to succeed at creating adversarial samples in the work proposed by Anderson *et al.* [4]. DDQN is a Q-learning algorithm similar to ACER, but has been specifically adapted to combat overestimating action values that Q-learning algorithms are known to cause [9]. This overestimation is reduced by changing the max operation in the target to action selection and action evaluation [9]. The DDQN algorithm has higher sample efficiency over ACER at the cost of a longer convergence time. These two algorithms where chosen to compare if higher sample efficiency can beat the current state-of-the-art within the same training time.

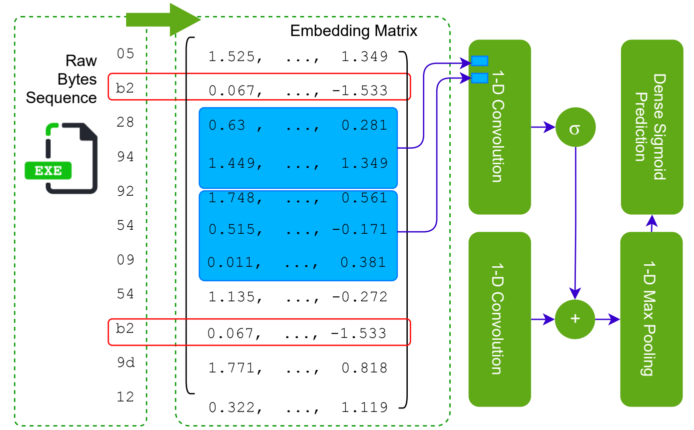

Fig. 2: Malware Detection Agent Architecture. The input of the detection agent is the raw modified malware sample and the output is the real-valued prediction by the detection agent.

Both the ACER and the DDQN networks use experience replay [8], [9]. Experience replay is the strategy of periodically reminding the reinforcement learning algorithm of its past experiences [12]. When complicated tasks are involved, reinforcement learning is a delicate game of trial and error. Experience replay strengthens reinforcement learning systems by uniformly sampling past observations made by the network. This sampling has shown to greatly raise the performance of reinforcement learning systems and reduce sample correlation [12], [9]. Most malware are designed to be unique from other malware, as an attempt to evade static anti-malware engines. Because of this, there may be malware samples that poison the training of the reinforcement learning system. With experience replay, the chance of network poisoning is greatly reduced, due to the reintroduction of past experiences.

D. Malware Detection Agent

As discussed above, the state-of-the-art MalConv network is the foundation for the malware detection agent in our game. MalConv is a gated Convolution neural network (CNN) that analyses raw files for malware detection [7]. Its ability to scan a PE sample without any feature extraction greatly reduces the computation time for the H4rm0ny system.

Figure 2 shows the architecture of our CNN model. The input of the MalConv is the raw bytes extracted from the first megabyte of the PE sample. Our MalConv network that has an input of a single vector of length 1048576 and an output of a single real value between 0 and 1. This number denotes the probability for the detection agent to take an action of flagging the sample as malicious. Then, we sample the final action for the detection agent in $\{0, 1\}$ following this probability. It should be noted that we do not use the threshold to derive the final decision, since this number is interpreted as the action probability for getting a certain reward.

Algorithm 1: H4rm0ny training algorithm

Input: malware samples S, benign samples B, malware classifier f, modification policy function p, modification set M, max actions on single sample $max_episodes$, total samples to train on max_steps, empty set A, empty set S'

Output: evasive samples S', malware classifier f'

```
1  for s in randomSelect(S, max_steps) do
2      s' = s
3      A = []
4      for _ in {1, ..., max_episodes} do
5          action = p.selectAction(s', M)
6          A.add(action)
7          s' = p.applyAction(s', action)
8          r = −10
9          if f.Score(s') = 0 then
10             r = 10
11             S'.addSample(s')
12             break
13         end
14     end
15     p.updatePolicy(s, action, r)
16     b = randomSelect(B, 1)
17     f = f.up-trainSamples(s', f.Score(s'), b)
18 end
19 f' = f
20 return S', f'
```

E. Gameplay Environment

As Algorithm 1 shows, the malware detection agent and the malware modification agent have been combined into the zero-sum game H4rm0ny. The inputs to H4rm0ny are the malware samples that will be used as the training set S, a malware classifier f, a modification policy function p, a set of benign samples for up-training B, and the set of modifications M. Parameters for the algorithm are the maximum actions the modification agent can make for each sample PE, the number of total samples to train on, the empty set of actions that are performed on a malware sample, and the empty set that will return the modified samples. The outputs to H4rm0ny are the new malware detection agent and the set of modified samples. For each step in H4rm0ny, a single sample is chosen and for each of the given number of modification attempts, H4rm0ny runs as follows. An action is chosen by the modification agent based on the sample (line 5) and that action is added to our list of actions (line 6). The chosen action is applied to the sample (line 7). The reward is set to the default value of -10 (line 8). At line 9, the score of the malware sample is chosen from the set of 0 and 1, based on the probability of maliciousness given by the malware detection agent. If the modified sample is evasive, the reward is set to 10 and the evasive sample is added to our evasive set (lines 9 and 10). The training process is then terminated and the training step of the game begins (line 12). The policy of the modification agent is trained based on the evasive sample, the set of actions performed on the sample, and the reward for the modification agent (line 15). The detection agent is then up-trained with the malware sample, the score of the given sample, as well as a randomly selected benign sample (line 17). The up-training of the detection agent ends with with both a malicious and benign sample, to ensure our detection agent is not biased towards predicting all incoming samples as malicious. If the sample cannot be made evasive within the given episodes, the policy is then updated to the modifications made to the sample, and the modification agent moves on to train on the next sample. Once the training of the modification agent has terminated, the evasive set and the new detection agent are returned (line 20).

We will describe in detail the algorithm for up-training samples into our detection agent. Our detection agent is first up-trained with a benign sample. As described above, up-training with a benign sample reduces any bias that the detection agent will gain from training with only one classification as data. This up-training is not a part of the game, so the traditional loss algorithm for MalConv, binary cross entropy, is used. For the malware sample, we have derived the maximum likelihood malware loss function for the game. The maximum likelihood malware loss function is a product of the agent reward and the maximum likelihood loss function.

$$r = \begin{cases} -10 & s = 0 \\ 10 & s = 1 \end{cases} \tag{4}$$

The reward for the detection agent is derived in Equation 4. As per the definition of a zero-sum game, the reward for the detection agent is the opposite of the reward for the modification agent, thus the sum of the rewards is 0.

$$p = \begin{cases} \hat{y} & s = 0 \\ 1 - \hat{y} & s = 1 \end{cases} \tag{5}$$

The probability of the sample classification is then calculated in 5, where $\hat{y}$ is the predicted probability of maliciousness from the detection agent training, and s is the score of the malware sample from gameplay. Our classification is set up like this, due to the use of the predicted score being used as the correct classification of the malware sample during training. For example, if our sample is successfully evasive, it will then be up-trained into the detection agent with the true classification as 0, so the probability that the sample is malicious based on the training is $1 - \hat{y}$.

$$\text{maximum_likelihood}(m) = -log(p) * r \tag{6}$$

Maximum likelihood malware loss is finally derived in 6.

F. Building the Final Detection Agent

While the evasive agent and the detection agent constantly improve each other throughout the gameplay, there is no guarantee who will be the final winner. Despite the fact that the evasive agent may be stronger by the end of the game, our ultimate goal is to have a robust detection agent. To achieve this, we can combine both agents by up-training.

The final winner is the malware detection agent that has been up-trained with a dataset generated by the malware detection agent, to combine the knowledge learned by both agents. The final winner is built once we have completed the training cycle of the H4rm0ny system. Once the detection agent training has been terminated, a separate dataset will run through the modification agent, and the modification agent will be allowed a specific number of modifications it can make to each malware sample. Here, the modification agent is not given

TABLE I: Results for evaluating the evasive agents. ⇑ denotes the higher the better and ⇓ denotes the lower the better. It should noted that **ClamAV is not included in the learning process and is used for evaluation purpose only**. #FN denotes the number of false positives which implies the number of successful evasions. Since a method can have a lower AUC by simply increasing #FP (such as the random method below), #FN is a more applicable metric for agent evaluation.

	MalConv Trained on EMBER (>1M samples) [13]				ClamAV Scanning Result	
Testing set modified by different agents	AUC ⇓	Accuracy ⇓	# FN ⇑	% Rise in FN ⇑	AUC ⇓	Accuracy ⇓
Original	0.8434	0.7720	342	-	0.5751	0.5714
Randomly modified data	0.7995	0.6653	502	46%	0.5657	0.5619
Agent (tzm-acer)	**0.7927**	**0.6460**	**531**	**55%**	**0.5174**	**0.5140**
Agent (spg-acer)	0.7957	0.6547	518	51%	0.5420	0.5380
Agent (tzm-ddqn)	0.8236	0.7273	409	20%	0.5664	0.5628
Agent (spg-ddqn)	**0.8009**	**0.6693**	496	**45%**	**0.5630**	**0.5595**

any result from the detection agent for each modification. The modification agent determines the modifications based on the sample PE alone. Once the dataset has been generated by the modification agent, it is split into a training and testing set. The training set is then completely up-trained into the malware detection agent.

III. Experiments

There are two experiments that we conducted in order to validate the performance of H4rm0ny. The dataset used in the experiments consisted of approximately 100,000 samples of Microsoft PE malware and 100,000 samples of benign Microsoft PE software. The first experiment evaluated the modification agent and the second, the detection agent. We evaluated the modification agent by training the H4rm0ny system with a set of different modification agents. These modification agents then generated training and testing sets that were run against the original MalConv network. Whichever modification agent had the highest evasion rate (highest total number of false positives) was the best modification agent. For benchmarking, we also evaluate our results on the unmodified data, as well as randomly modified data.

Evaluating the Evasive Agents. For the ACER and DDQN policy algorithms, two malware modification agents are trained. The first agent is trained by playing a single-player-game, and the second agent is trained in a turn-based zero-sum Markov game. We use spg to indicate single player game, and we use tzm to indicate turn-based zero-sum Markov game. This gives us a total of four malware modification agents: Agent (spg-acer), Agent (tzm-acer), Agent (spg-ddqn), and Agent (tzm-ddqn). For each of these configurations, the modification agent and results datasets are the same. However, they may be differences in the benign data chosen for modification, while training the modification agent. Along with these four configurations, the dataset of original data, as well as the dataset of randomly modified data, are compared. Along with testing the strength of our different datasets against the MalConv model, we also scanned the testing sets with the open source anti-malware engine ClamAV. ClamAV is a widely used anti-malware engine developed by Cisco for commercial and private internet security.[2] We conducted our experiments with ClamAV version 0.103.3 and the daily database version 26211.

[2]https://www.clamav.net/about

The results of this evaluation are described above in I. As can be seen, the agent that produced the testing set with the highest number of false negatives is the turn-based zero-sum Markov game ACER agent. The inverse of this can be seen with the different DDQN agents, where the single-player game produced more false negatives than the turn-based zero-sum Markov game. This indicates that the detection agent was the winner of the turn-based zero-sum Markov game. This shows that the turn-based zero-sum Markov game does not only improve modification agents within the game, but it also improves modification agents against state-of-the-art consumer anti-malware systems. Figure 3 shows the distribution of steps needed in a game to successfully generate an evasive sample. For both the DDQN and ACER agents, the game type does not cause a significant difference in the number of turns needed to produce an evasive sample. but the mean steps for DDQN is significantly higher than the mean steps for ACER. This continues to support the observation that ACER is a better policy algorithm for the modification agent. We found that the most frequent modification made to successfully evasive samples were benign data overlay followed by benign binary overlay.

Evaluating the Detection Agents. The malware detection agent used for our experiments was trained on the EMBER dataset. The EMBER dataset consists of 1.1 million benign and malicious Microsoft PE files. [13]. This initial training on the Microsoft PEs gives our malware detection agent a very strong foundation for malware detection. An explanation of the models is as follows. **MalConv.** The MalConv model is the original malware detection agent that has not been trained on any of the newly generated evasive malware. **MalConv (tzm-acer).** The tzm-acer detection model is the detection model that was up-trained with all evasive samples generated by the Agent (tzm-acer) modification model. **MalConv (tzm-ddqn).** Similar to the tzm-acer detection model, the tzm-ddqn model is the malware detection agent that was up-trained with the evasive samples generated from the tzm-ddqn modification agent. Along with these three detection agents, these agents have been up-trained with specific training datasets for our experiments. An explanation of these up-trained datasets is as follows. **MalConv (up-trained with original).** This detection agent is the original malware detection agent that has been trained on the unmodified training set. **MalConv (up-trained with random).** This detection agent is the original malware detection agent that has been trained on a randomly modified

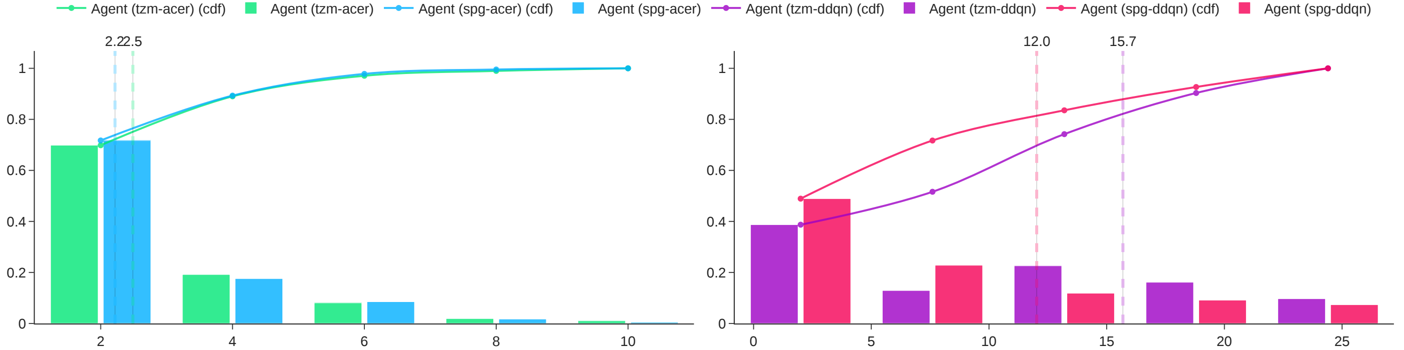

Fig. 3: The number of modifications needed in order to generate an evasive sample. Left: histogram and CDF for the modification agent sgp-acer and tzm-acer. Right: histogram and CDF for agent spg-ddqn and tzm-ddqn. The X axis shows the number of modifications, and the Y axis shows the CDF.

TABLE II: Results of detection agents on more evasive datasets. ⇑ denotes the higher the better and ⇓ denotes the lower the better.

		Agent tzm-acer		Agent tzm-ddqn		Random	
	Models	AUC ⇑	FN ⇓	AUC ⇑	FN ⇓	AUC ⇑	FN ⇓
Evaluate on more evasive datasets	MalConv	0.7927	531	0.8236	409	0.7995	502
	MalConv (up-trained with original)	0.8075	276	0.8326	148	0.7462	250
	MalConv (up-trained with random)	0.8434	318	0.8479	227	0.8134	294
	MalConv (tzm-acer)	0.6988	3	0.8467	57	0.7513	7
	MalConv (tzm-acer, up-trained)	0.8570	77	0.8270	465	0.8268	164
	MalConv (tzm-ddqn)	0.6239	4	0.6964	7	0.6184	4
	MalConv (tzm-ddqn, up-trained)	**0.8668**	40	**0.8983**	45	**0.8512**	38
		original		Agent spg-acer		Agent spg-ddqn	
	Models	AUC ⇑	FN ⇓	AUC ⇑	FN ⇓	AUC ⇑	FN ⇓
Evaluate on Less evasive datasets	MalConv	0.8434	342	0.7957	518	0.8009	496
	MalConv (up-trained with original)	0.9466	108	0.8341	258	0.8239	251
	MalConv (up-trained with random)	0.9419	181	0.8666	323	0.8624	298
	MalConv (tzm-acer)	0.9141	100	0.7124	5	0.7610	16
	MalConv (tzm-acer, up-trained)	0.8731	522	0.8764	91	0.8405	179
	MalConv (tzm-ddqn)	0.8950	20	0.6449	7	0.6781	7
	MalConv (tzm-ddqn, up-trained)	**0.9531**	54	**0.8764**	52	**0.8907**	52

training set. **MalConv (tzm-acer, up-trained).** This detection agent is the resulting detection agent from the turn-based zero-sum Markov game with the ACER policy algorithm. It has been up-trained on the training set, which has been modified by the agent (tzm-acer). **MalConv (tzm-ddqn, up-trained).** This detection agent is the same as MalConv (tzm-acer, up-trained), but from the turn-based zero-sum Markov game involving the DDQN policy algorithm.

As can be seen in Table II, the strongest detection agent is MalConv (tzm-ddqn, up-trained). MalConv (tzm-ddqn) + tzm-ddqn training has the highest AUC on all the datasets that have been generated for experimentation. Although MalConv (tzm-ddqn, up-trained) does not have a lower number of false negatives than MalConv (tzm-acer) and MalConv (tzm-ddqn), the lower AUC of those two models against the training data indicates a lower total performance. This result is expected due to malicious content having more weight than benign content in gameplay. Although we cannot claim that our system generates a malware detection agent that is robust against all current and future adversarial learning techniques, we have shown that multi-agent learning can be used to further strengthen malware detection systems against reproducible adversarial attacks. The most evasive testing set against the MalConv detection agent, Agent (tzm-acer) continued to have a high false positive rate against the detection agents. Although it was the most evasive testing set against MalConv, it was not the most evasive against MalConv (tzm-ddqn, up-trained). The agents with the highest false-negative rates against the MalConv (tzm-ddqn, up-trained) detection agent are the original, spg-acer, and spg-ddqn agents.

IV. Related Work

Generating Adversarial Malware. One of the biggest vulnerabilities of neural networks is their statistical bias towards

their training set. In the context of malware detection, this can lead to many problems, such as the bias that the networks have towards certain features indicating if a piece of software is benign. Adversaries take advantage of this fact by modifying their malware with benign features to evade detection [14], [4]. Other works proposing an RL-based framework for adversarial malware detection include work by Bai *et al* [15]. In the work proposed by Bai *et al.* adversarial samples are generated against a static surrogate of a detection system [15]. The anti-malware engines used for their experiments are based on recurrent neural networks [15]. The framework proposed by Bai *et al.* is similar to the one proposed by Anderson *et al.* due to both training frameworks having generating adversarial samples against a static malware classifier [15], [4]. Unlike these two works H4rm0ny does not train against a static malware detection system, but a dynamic system that is learning how to detect the adversarial samples that are generated during the generation agent training time.**Neural Networks for Malware Detection.** Our review on networks is limited to Microsoft PE malware. One method for malware detection using neural networks is to extract feature vectors from the file being analyzed. For example, this is done by Anderson *et al*. in their work proposing a gradient-boosted decision tree model using LightGBM for malware detection [16]. The use of feature extraction for malware detection is a popular and very successful method for malware detection [17], [16], [18]. In these networks, before the networks can analyze a file, it is run through a strenuous prepossessing pipeline that extracts features from the file for analysis. Some related works include: a gradient-boosted decision tree model using LightGBM for PE malware detection [16], a Multi-Naïve Bayes classifier based on features extracted from the PE libBFD [18], and a malware classifier that enforces monotonicity on the extracted feature vector [17]. **Multi-Agent Learning.** Similar to our work, Hu *et al*. propose an adversarial sample generation system based on a multi-agent learning system [19]. Their work uses a generative adversarial network (GAN) for creating feature space adversarial samples for a network that models a black-box malware detector [19]. Unlike their system ours generates real adversarial malware samples instead of just feature vectors.

V. Conclusion

In this study, we studied the problem of malware detection networks being vulnerable to adversarial attacks. We mitigated this problem by designing and implementing the first multi-agent learning environment for evasive malware generation and detection. We then showed that through self-play policy optimization, the resulting generation agent is able to produce highly evasive but still functional malware, and the detection agent is more robust against adversarial samples.

VI. Acknowledgements

This research is supported by Defence Research and Development Canada (contract no. W7701-176483).

References

[1] B. Jovanović. (2021) A not-so-common cold: Malware statistics in 2021. [Online]. Available: https://dataprot.net/statistics/malware-statistics/#:~:text=Every%20day%2C%20there%20are%20at,they%20stay%20installed%20long%20enough.

[2] H. N. Security. (2020) Evasive malware increasing, evading signature-based antivirus solutions. [Online]. Available: https://www.helpnetsecurity.com/2020/03/26/evasive-malware-increasing/

[3] (2019) Cylance, i kill you! [Online]. Available: https://skylightcyber.com/2019/07/18/cylance-i-kill-you/

[4] 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*, vol. abs/1801.08917, 2018.

[5] F. Pierazzi, F. Pendlebury, J. Cortellazzi, and L. Cavallaro, "Intriguing properties of adversarial ML attacks in the problem space," in *2020 IEEE Symposium on Security and Privacy, SP 2020, San Francisco, CA, USA, May 18-21, 2020*. IEEE, 2020.

[6] J. Zhang, X. Xu, B. Han, G. Niu, L. Cui, M. Sugiyama, and M. Kankanhalli, "Attacks which do not kill training make adversarial learning stronger," in *Proceedings of the 37th International Conference on Machine Learning*, 2020.

[7] 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*, ser. AAAI Workshops, vol. WS-18. AAAI Press, 2018.

[8] V. R. Konda and J. N. Tsitsiklis, "Actor-critic algorithms," in *Advances in Neural Information Processing Systems 12, [NIPS Conference, Denver, Colorado, USA, November 29 - December 4, 1999]*, S. A. Solla, T. K. Leen, and K. Müller, Eds. The MIT Press, 1999.

[9] H. van Hasselt, A. Guez, and D. Silver, "Deep reinforcement learning with double q-learning," in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, D. Schuurmans and M. P. Wellman, Eds. AAAI Press, 2016.

[10] Y. Zhong, Y. Zhou, and J. Peng, "Efficient competitive self-play policy optimization," 2020.

[11] G. Dulac-Arnold, R. Evans, P. Sunehag, and B. Coppin, "Reinforcement learning in large discrete action spaces," *CoRR*, vol. abs/1512.07679, 2015. [Online]. Available: http://arxiv.org/abs/1512.07679

[12] L. J. Lin, "Self-improving reactive agents based on reinforcement learning, planning and teaching," *Mach. Learn.*, vol. 8, pp. 293–321, 1992. [Online]. Available: https://doi.org/10.1007/BF00992699

[13] H. S. Anderson and P. Roth, "EMBER: an open dataset for training static PE malware machine learning models," *CoRR*, vol. abs/1804.04637, 2018.

[14] W. Song, X. Li, S. Afroz, D. Garg, D. Kuznetsov, and H. Yin, "Automatic generation of adversarial examples for interpreting malware classifiers," *CoRR*, vol. abs/2003.03100, 2020.

[15] S. Bai, J. Z. Kolter, and V. Koltun, "Trellis networks for sequence modeling," in *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019.

[16] G. Ke, Q. Meng, T. Finley, T. Wang, W. Chen, W. Ma, Q. Ye, and T. Liu, "Lightgbm: A highly efficient gradient boosting decision tree," in *Advances in Neural Information Processing Systems 30: Annual Conference on Neural Information Processing Systems 2017, December 4-9, 2017, Long Beach, CA, USA*, I. Guyon, U. von Luxburg, S. Bengio, H. M. Wallach, R. Fergus, S. V. N. Vishwanathan, and R. Garnett, Eds., 2017.

[17] I. Incer, M. Theodorides, S. Afroz, and D. A. Wagner, "Adversarially robust malware detection using monotonic classification," in *Proceedings of the Fourth ACM International Workshop on Security and Privacy Analytics, IWSPA@CODASPY 2018, Tempe, AZ, USA, March 19-21, 2018*, R. M. Verma and M. Kantarcioglu, Eds. ACM, 2018, pp. 54–63. [Online]. Available: https://doi.org/10.1145/3180445.3180449

[18] M. G. Schultz, E. Eskin, E. Zadok, and S. J. Stolfo, "Data mining methods for detection of new malicious executables," in *2001 IEEE Symposium on Security and Privacy, Oakland, California, USA May 14-16, 2001*. IEEE Computer Society, 2001.

[19] W. Hu and Y. Tan, "Generating adversarial malware examples for black-box attacks based on gan."