The Art of Digital Deception: Adversarial Evasion of an Autonomous Cyber Defence Agent

Melanie Meijer

Master of Science
Cyber Security, Privacy and Trust
School of Informatics
University of Edinburgh
2024

Abstract

The increase in frequency and complexity of recent cyber attacks has called for the need of more sophisticated defence systems. To this end, the field of autonomous cyber defence aims to train deep reinforcement learning (DRL) agents that are able to defend a network independently and more efficiently. However, such training is generally performed against predictable, logical attackers. Furthermore, DRL policies have been shown to be susceptible to adversarial perturbation attacks. This raises the question: is it possible to develop an adversarial policy that is able to use its own actions to influence the defender's behaviour and ultimately evade it? This project contributes a custom wrapper for the CybORG environment, and uses this to demonstrate the existence of such an adversarial policy for a trained defender. While the defender performs well against the environment's original hard-coded attackers, the adversarial agent is able to significantly diminish its performance with a decrease of over 40% in the defender's ability to protect the target operational server.

Research Ethics Approval

This project was planned in accordance with the Informatics Research Ethics policy. It did not involve any aspects that required approval from the Informatics Research Ethics committee.

Declaration

I declare that this thesis was composed by myself, that the work contained herein is my own except where explicitly stated otherwise in the text, and that this work has not been submitted for any other degree or professional qualification except as specified.

(*Melanie Meijer*)

Acknowledgements

I would like to sincerely thank my supervisor Dr. Marc Juarez for all of his support and guidance over the course of this project, and for providing me with the opportunity to work on this project in the first place.

I would also like to thank Dr. Vasilios Mavroudis for taking the time to support me in tackling this project, and providing me with incredibly helpful insights and feedback.

Table of Contents

Chapter 1

Introduction

1.1 Motivation

As our world becomes increasingly digital and interconnected, there has been a notable increase in both frequency and intricacy of cyber attacks. The UK Cyber Security Breaches Survey records that 50% of businesses and 32% of charities have reported a cyber attack in the last 12 months, which increases to 70% or more for medium and large businesses [10]. On top of this, emerging, more sophisticated cyber attacks have highlighted the need for more efficient countermeasures. As a result, the interdisciplinary field of Autonomous Cyber Defence (ACD) has gained traction, aiming to develop defender agents to autonomously combat cyber attacks.

However, the nature of the cyber security landscape is intrinsically unbalanced: "[Defenders] have to be right 100 percent of the time. Cyber criminals only have to be right once."[22]. As standard system vulnerabilities are better defended, a sophisticated attacker may shift their tactics from "traditional" cyber-attack techniques, to targeting the defender agent itself in order to achieve their objective. However, this attack vector has not yet been implemented within the field of ACD. This project aims to address this research gap by evaluating the security of the ACD agent itself in the face of an adversarial evasion attack.

The art of deception [25] refers to exploiting the human as the weakest element within the cyber security attack surface through *social engineering*. With the emergence of increasingly automated and autonomous defence systems, this project now focuses on the art of digital deception, where an adversary seeks to target and exploit the autonomous agent by using *adversarial evasion* to deceive the agent.

1.2 Objective

The goal of this project is to explore whether an adversarial agent can identify weaknesses in a defender's policy and mislead it to make it fail in defending the network in an existing environment. **The hypothesis here is that a defender agent trained against hard-coded, logical attackers is not robust against adversarial attacker policies at test time.** The key research questions that this work aims to answer are:

1: Can an adversarial policy attack be used as a technique to effectively evade a trained defender agent and compromise a simulated network?
2: How successful is such an adversarial agent in reducing the performance of the defender agent in comparison with the original, hard-coded attackers?

This experiment adopts a black-box threat model, where the adversary aims to use its own actions to indirectly influence the victim agent's behaviour by inducing adversarial observations. We improve over state-of-the-art DRL evasion attacks which generally employ white-box assumptions. We thus consider a stronger adversary that if successful can deceive the autonomous defender agent and thereby compromise the network. We thereby aim to demonstrate that autonomous defenders must become robust against such attacks in order to be effective in the real world, and not solely be trained and tested against hard-coded, logical attacker models.

This interdisciplinary work constitutes a novel contribution from multiple perspectives. Within RL research, at the time of writing, this is the first implementation of an adversarial policy within a simulation of a complex real-world task rather than of a game. From a cyber security perspective, it is the first research that addresses the arms race with respect to an adaptive adversarial attack targeting an ACD system.

1.3 Structure

The remainder of this document is structured as follows: Chapter 2 presents an exposition of relevant background and previous work for this project, providing context to the work. Chapter 3 will provide further detail on the environment and threat model employed for this project. Chapter 4 discusses the conceptual design and implementation of the work undertaken, exploring both the environment customisation and agent training. Chapter 5 presents an evaluation of the trained agents, along with an analysis of the results. Lastly, Chapter 6 concludes the project's findings and considers potential avenues for future work.

Chapter 2

Background and Related Work

2.1 (Deep) Reinforcement Learning

Deep reinforcement learning (DRL) is a field within machine learning (ML) that integrates reinforcement learning (RL) with deep neural networks, and has gained a lot of traction over the last few decades. This type of ML simulates tasks in an environment, which an agent can then learn to solve from scratch based on experience and feedback, thereby removing the need for large datasets required by supervised learning approaches. Many tasks have been successfully tackled by DRL agents, including Atari arcade games [27], board games such as chess and Go [35], as well as real-world tasks such as energy management [43] and autonomous driving [19]. In this project, a method to develop an adversarial policy is designed by extending an existing environment that models the task of defending a network against a cyber attack. This section will cover the fundamentals of both framing and solving an RL problem.

2.1.1 Modeling an RL Task

RL is rooted in the way humans and animals learn through experience. The two key components for an RL setup are the agent and the environment. The agent is the decision-making entity, performing actions to move from one state to the next. The environment models the agent's task and provd with feedback in the form of observations and rewards. This loop is depicted in Figure 2.1 and occurs at every timestep `t` within the environment. At no point is the agent instructed how to solve the modelled task. Instead, a trial and error approach is used in order to explore and learn which actions are optimal in terms of reward for a given state within the environment.

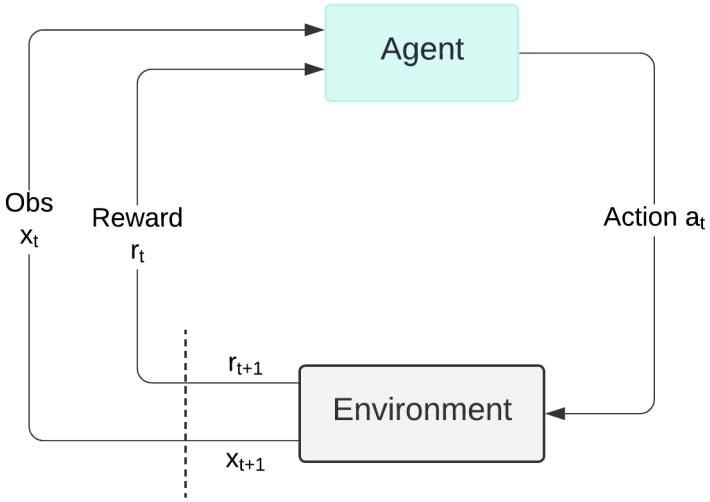

Figure 2.1: Reinforcement Learning Diagram

More specifically, Figure 2.1 presents the agent-environment interaction in a (Partially Observable) Markov Decision Process (MDP). An MDP is an RL framework where future environment states and rewards are independent of past states and actions, given the current state and action [38]. In general, an RL task can be modeled through the following set of MDP components: $\Omega = (\mathcal{S}, \mathcal{A}, \mathcal{R}, \mathcal{T}, \mathcal{X})$. These are defined below. Note that for this paper, notation will follow [38]: s_t and s_{t+1} for current and next state respectively, but other notation is also widely used, e.g. s and s'.

- Set of states $\mathcal{S}$ the environment can be in
- Agent action space $\mathcal{A}$ detailing the actions available to the agent (discrete or continuous)
- Reward function $\mathcal{R}$ determining the agent's reward at each timestep, based on the environment's current state, action and next state: $r_t = \mathcal{R}(s_t, a_t, s_{t+1})$
- Transition function $\mathcal{T}$ dictating the environment dynamics, i.e. the probability of a particular next state, based on the current state and action: $p(s_{t+1}|s_t, a_t)$
- Agent observation space $\mathcal{X}$ detailing the information available to the agent, created based on the environment state by an observation function: $x_t = O(s_t)$

Based on the above and re-iterating Figure 2.1, one discrete environment timestep can be denoted by the MDP transition tuple (s_t, x_t, a_t, s_{t+1}).

Finally, an MDP is partially observable (POMDP) when the agent's observation does not fully capture the underlying environment state, thereby creating a distinction between the state and observation, and adding uncertainty into the learning problem.

Our model adopts such an POMDP setup for all agents in its environment to create a more realistic simulation of the real-world task.

2.1.2 Learning an RL Task

For an episodic task, an episode within the RL environment is one attempt for the agent to solve the modeled task. An episode can end after a specified number of the timesteps, as for this project's environment, or when a particular environment state is triggered. Upon completing an episode during the agent's training, the environment is reset and the agent's policy is updated. At a high level, the agent's policy is a mapping from observations to the probability of performing particular actions for said observation, based on estimations of the expected reward. This is denoted as $\pi(a|s)$, and ultimately defines the agent's behaviour.

In order to develop an optimal policy for a given task, the reward function is essential. The reward signal is what is used by the agent to determine which actions are optimal for given observations, and therefore is crucial in influencing the agent's final behaviour [11]. It is important for the reward function to reflect the agent's goal, rather than how to achieve it, which can be difficult for complex tasks. A poorly designed reward function can lead not only to poor performance, but also unexpected and/or unsafe behaviour as the RL agent aims to optimise its reward. This is known as the alignment problem [9], and is an essential problem to consider when modeling an RL task.

The agent's learning is not only based on the immediate reward, but also on future reward, where the agent ultimately aims to maximise its expected cumulative reward over the course of an episode. The trade-off between immediate and future reward is controlled by a hyperparameter called the *discount factor* γ. This is used in the agent's *value function* $V_\pi(s)$, which signifies the expected reward value of a certain state based on all possible actions, next states, and expected rewards under the current policy [3]:

$$V_\pi(s_t) = \sum_a \pi(a|s_t) \sum_{s_{t+1},r} Pr(s_{t+1}, r|s_t, a)[r + \gamma\, v_\pi(s_{t+1})] \tag{2.1}$$

Fundamentally, the agent's learning is controlled by its algorithm's objective function (also known as the loss function), which encapsulates the agent's performance with respect to its goal of maximising cumulative reward. It does this by quantifying the value loss, which is the (average) difference between the agent's expected value for a given state and the actual observed state value during training. The particular loss function employed by this project's algorithm is defined in Equation 2.2 below.

Deep Neural Networks: In DRL, the agent's policy is modeled using a deep neural network (DNN). A DNN is comprised of layers of interconnected nodes (neurons), where each layer is made up of linear classifiers with non-linear activation functions. Within the DRL context, the environment observation is passed into the input layer of this network, where the output of each layer is computed and passed onto a node in the next layer. Ultimately, the output layer of the network generates the action to be taken by the agent. During the agent's training, the DNN's parameters θ (also known as its weights) are altered to update the agent's policy and learn to better predict the optimal action(s) for a given observation.

Overall, DNN's have revolutionised the field of RL and enable agents to solve complex tasks. However, a common obstacle encountered with DNNs is that they are black-box systems with a complex non-linear structure. This makes them difficult to control and makes it hard to understand their exact reasoning, which is a challenge encountered not only in the field of ML security, but also in that of explainable AI [8]. This will be further covered in Section 2.2 below.

DRL Algorithms: Several different algorithms exist that use deep neural networks to represent the agent's policy, with different learning methods that make them suitable for different tasks. Such algorithms include DQN [28], A2C [26], DDPG [20], and PPO [33]. PPO is a state-of-the-art algorithm that is widely used within RL research, and is found to perform well in Multi-Agent RL (MARL) settings [42], which is one of the reasons we selected it for the agents developed in this project.

PPO is an on-policy algorithm, which means that the experience from which the agent learns is generated by the same policy that is being updated. It is also a type of actor-critic algorithm, meaning that it has two separate components: an actor which determines the actions performed, and a critic which evaluates the chosen actions. These properties are illustrated in the PPO policy update algorithm outlined in Algorithm 1.

Algorithm 1 PPO, Actor-Critic Style [33]

for $iteration = 1, 2, \ldots, M$ **do**
  for $actor = 1, 2, \ldots, N$ **do** ▷ *Actor-critic*
    Run policy $\pi_{\theta_{old}}$ in environment for T time steps ▷ *On-policy*
    Compute advantage estimates $\hat{A}_1, \ldots, \hat{A}_T$
  Optimize surrogate L w.r.t. θ, with K epochs and minibatch size $M \leq NT$
  $\theta_{old} \leftarrow \theta$

Specifically, this project makes use of the PPO-Clip variant [33]. The key idea behind this algorithm is to make the agent's policy updates stable and reliable by utilising a clipped surrogate loss. This surrogate learning objective ensures that the policy is updated in small, bounded modifications such that the new policy does not diverge too far from the old policy within one update. The clipped loss function is defined as follows for a given set of DNN parameters θ:

$$L^{CLIP}(\theta) = \mathbb{E}[\min(r_t(\theta)\hat{A}_t,\ clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)\hat{A}_t)] \tag{2.2}$$

This function can be broken down into the following components:

- Probability ratio $r_t(\theta)$ compares the probability of selecting a given action for a given state using the old policy versus the new policy.
- Advantage estimate $\hat{A}_t$ represents the relative quality of a given action in comparison to the average action for a given state.
- Clipping mechanism $clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)$ bounds the probability ratio to be within a specific range.
- Minimization $(\min)$ ensures the policy update only improves or maintains its current performance.

Not only does PPO prevent large policy updates and thereby stabilise training, it is also efficient and easy to implement which is why it is a popular choice for a wide variety of modern-day DRL applications. For the same reasons, PPO was selected for this project as its efficiency and stability helps the attacker agent in learning the complex task of finding effective attack sequences with sparse rewards. Note that no domain-specific modifications were made to the algorithm itself for this work. However, while the learning algorithm may be kept constant across many different learning problems, it is essential to tune the algorithm's hyperparameters to achieve good performance for a specific task. This project's hyperparameter tuning is further described in Chapter 4.

2.1.3 Multi-Agent RL

In Multi-Agent RL (MARL), multiple agents interact in a shared environment, either cooperatively or competitively. This field is increasingly more researched within RL, as real-world agents are likely to be deployed in an environment that is acted upon by other (human) agents. In this project, while there are multiple agents acting competitively upon the same, shared environment, the learning problem is not a true MARL setting

and is instead formulated as two individual single-agent tasks. This is because only one agent is actively learning and updating its policy at a given time, while its opponent's policy is fixed. This is further explained in Chapter 4 below.

2.2 Machine Learning Evasion Attacks

Following the recent widespread adoption of ML models, cyber criminals have adapted their attacks to target these new technologies including – but not limited to – privacy, poisoning and evasion attacks. In general, this notion is known as the cyber security arms race, where attackers and defenders are in a continuous loop of adapting to new opponent techniques, as depicted in Figure 2.2. The key lesson here is that within the field of cyber security, defenders must actively consider this arms race and assume they are up against a strategic adversary that is aware of possible defences and will adapt to them.

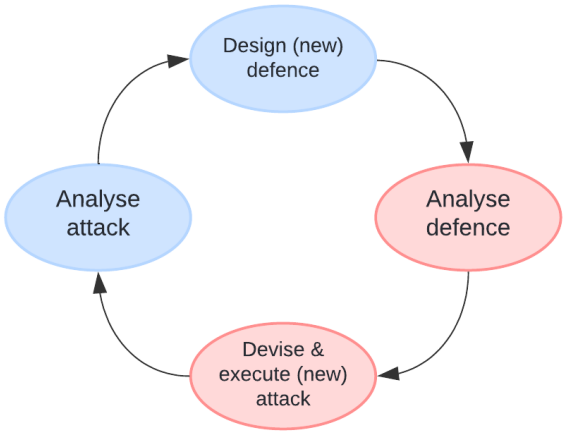

Figure 2.2: Cyber Security Arms Race

As ML models became more ubiquitous over the last few decades, threat actors developed the ML evasion attack, which was initially carried out against supervised classification models. This attack targets the integrity of the ML model through (near) undetectable adversarial examples. Such adversarial examples are generated by modifying an input test point with a small, imperceptible perturbation that causes the model to misclassify the input. Models that employ DNNs have been found to be especially vulnerable to such an attack, though the exact cause for this susceptibility is unclear due to their complex, black-box nature [7] as described in Section 2.1.

Adversarial attacks in DRL can be seen as a parallel to the adversarial examples in supervised ML, where the opponent aims to induce unexpected behaviour from the

victim policy through small perturbations to its input, and thereby deviating the victim's policy and/or decreasing its reward [31]. This is closely related to the concept of agent robustness and generalisation, where adversarial attacks demonstrate a lack of robustness in the victim policy to real-world perturbations. This in turn hinders systems from bridging the reality gap [6] between simulation and real-world deployment, highlighting that this is not merely a security-specific challenge.

2.2.1 Related Work - ML Evasion

Supervised Models: As described above, evasion attacks were initially developed to target supervised classification models. Szegedy et al. observed in their research that neural networks have blind spots and proposed the term "adversarial examples" to refer to the perturbed inputs used to evade the DNN models [39] . Their attack is untargeted, and aims to minimise the amount of perturbation required to cause a misclassification in an image-based classifier. Biggio et al. developed targeted evasion attacks against several classification models, with the aim to maximise the model's confidence in its misclassification of the perturbed test point [4]. Both of these attacks adopt a white-box threat model, where the adversary has knowledge of the model parameters as well as the training data. Goodfellow et al. proposed the Fast Gradient Sign Method (FGSM), which is a one-step attack instead of iterative like [39] and [4], and aims to maximise the victim model loss using the input gradient rather than the model parameters [14]. Hence, this attack was found to be surprisingly effective under a black-box setting where the model parameters and training data are unknown to the attacker.

RL Models: The majority of prior research into RL evasion attacks also assumes a white-box threat model, or at the very least direct access to the victim's environment components. Several strategies have been adopted to develop a DRL evasion attack, with a range of perturbation targets such as the agent's observation, selected action, state and transition function. Madry et al. used a white-box gradient-based attack that perturbs the victim's observation with the aim to deviate its policy [23]. Conversely, Russo et al. proposed a black-box attack that also modifies the victim's observations to minimise its overall reward. [18] instead perturbed the victim's action to minimise its reward within their white-box attack [30]. Lastly, Schott et al. targeted the environment state with their white-box perturbations with the objective of deviating the victim's policy [32]. While their findings are interesting for understanding the robustness of DRL models, they are not necessarily realistic for a potential real-world attack.

In contrast to these direct manipulation attacks, Gleave et al. argue that an indirect adversarial policy attack is a more realistic threat model [13]. This paper develops an adversarial policy in several two-player robot benchmark games, where the adversarial agent learns to indirectly induce natural, adversarial observations for its opponent, purely through its own action selection. While the behaviour of the adversarial robot agent appears random and uncoordinated, it achieves a high win rate through this adversarial evasion technique. Similarly, Wang et al. demonstrate that such adversarial policies exist for a state-of-the-art DRL agent within the two-player game of Go [41]. While the adversarial policy does not learn how to play the game well, it is able to learn a strategy that tricks its victim opponent into consistently losing. They argue that attaining a good average-case performance for an RL agent does not automatically lead to worst-case robustness, which is a particularly relevant notion for security critical tasks such as cyber defence.

As further detailed in Section 3.2, we follow the indirect evasion approach proposed by [13], under the assumption that direct perturbation of the victim's environment and/or observations is an overly powerful threat model within the realm of cyber security network defence. This work constitutes a novel contribution compared to the above described papers in ML/RL evasion in developing an indirect, black-box adversarial policy attack within a real-world application instead of benchmark games. Moreover, as described in the next section, this project provides a novel contribution within the field of Autonomous Cyber Defence (ACD) by incorporating adversarial attacks into an ACD environment.

2.3 Autonomous Cyber Defence

Over the last few decades, cyber attacks have become progressively prevalent and are executed with a range of motivations [5] and targets [36]. Not only are cyber attacks more frequent, but they are also increasingly more advanced, through modern-day techniques like adaptive malware, as well as AI-based attack tools such as DeepLocker[1] [16]. Apart from individuals and businesses, critical infrastructure (food, healthcare, electricity, etc.) and cyber warfare [12] have become notable targets within the field of cyber security. Consequences of successful cyber attacks include financial, reputational, societal and even physical harm [1].

[1] https://github.com/CyberWarefare/DeepLocker

Traditional defensive measures against cyber attacks include systems such as antivirus software, firewalls, intrusion detection and prevention systems in order to identify and attempt to prevent system breaches, as well as incident response plans that outline steps to contain any successful breaches. Although such traditional defences have been successful to a large extent, emerging cyber attacks call for more sophisticated defensive techniques. In particular, while these systems can be combined into an arsenal of tools that enable detection and notification of potential cyber attacks, they are limited by the need for human input in the actual mitigation of a breach.

To this end, ACD aims to use DRL in order to develop faster, more proactive defensive measures to counter emerging cyber attacks. The key idea within ACD is to develop autonomous defender agents that are able to defend a network from an active cyber attack without human input. ACD is part of the field of Autonomous Cyber Operations (ACO), which encompasses research into both RL-based cyber attacks and defences. Research within the field of ACO, however, largely focuses on either the attacker [40, 34], or the defender [2, 29]. This project aims to bridge this gap by customising an environment such that both agents may be trained against each other using reinforcement learning, while laying focus on the evasion of an RL-based defender through an RL-based adversarial attacker.

2.3.1 Related Work - Existing ACO Environments

Several training environments have been implemented to develop either autonomous defender or attacker agents.

CyberBattleSim (CBS) [40] is an ACD framework developed by Microsoft, with the aim to train offensive cyber agents within a flexible, highly abstract simulated network environment. While this environment enables efficient testing of new techniques and approaches, its non-realistic implementation prevents any trained agents to be transferred to a real-world setting.

Yawning Titan (YT) [2] was developed around the same time as CBS, instead focusing on providing a framework in which defender agents can be trained. While this environment implements a larger action space for the defender agent, the agent actions do not necessarily map to realistic real-world actions. Similar to CBS, the key objective for the environment appears to be to efficiently test algorithms rather than develop transferable defensive systems.

Network Attack Simulator (NASim) [34] is an environment implemented for research

into autonomous penetration tester agents. Even though the attacker agent's action space is fairly restricted, this framework was built with generalization in mind. The authors trained agents in several scenarios of varying network complexity and sizes, evaluating the performance of agents trained in simpler networks within more complicated networks. However, as the agent's task here is penetration testing, this environment does not implement a defender.

FARLAND [29] is the only ACD environment that takes the above described arms-race (Figure 2.2) into account in their research. The highly flexible framework aims to address both generalisation and robustness of defender agents, but does not actually implement the adversarial attacker described in the paper. Moreover, the environment is closed-source, meaning the specific functionalities of the framework can not be analysed or used for further research.

CybORG [37] is an environment developed for a series of public research challenges into ACD named the CAGE challenges, aimed to serve as a benchmark for the field. While the challenges are aimed at comparing implementations of RL-based defensive agents within the environment, sophisticated hard-coded attacker agents and a genuine user agent are simulated in the environment as well. This highly configurable environment is considered to be the current state-of-the-art when it comes to ACD training environments, which is why it was selected for this project. More detail on the attack scenario modeled within the environment is provided below.

Chapter 3

Threat Scenario

3.1 Environment Scenario

As described above, the CybORG environment was selected for this project [17], which is an environment that was developed for multiple public research challenges into ACD named the CAGE challenges. Specifically, for this project, the second challenge environment has been employed as it is the most popular within the research community, and the best maintained version of the environment according to the developers.

This environment models a cyber attack on a commercial network of 13 hosts, and was originally designed to train an RL-based defender. The cyber attack that is modelled by the environment is based on the MITRE ATT&CK framework, and assumes that the attacker has gained an initial foothold into the network through a successful phishing attack. From this initial foothold, it is the attacker's goal to compromise and impact the target operational server within the network through lateral movement and privilege escalation. In this case, "impacting" the critical server signifies taking it offline, thereby severely affecting the availability of the organisation's operations. The environment's network layout is shown in Figure 3.1, where the attacker is assumed to have an initial foothold to one of the hosts in the user host subnet.

Environment Agents: Within this environment setup, three distinct agents are simulated: A defender (blue) agent that aims to protect the network from any malicious activity, an attacker (red) agent that aims to impact the operational server, and optionally a genuine user (green) agent which generates benign activity on the network that must be preserved by the defender as much as possible. For this project, the green agent is always included, both in the training and evaluation stages. This is because it is important to include genuine users within the network to create a more realistic simulation of a

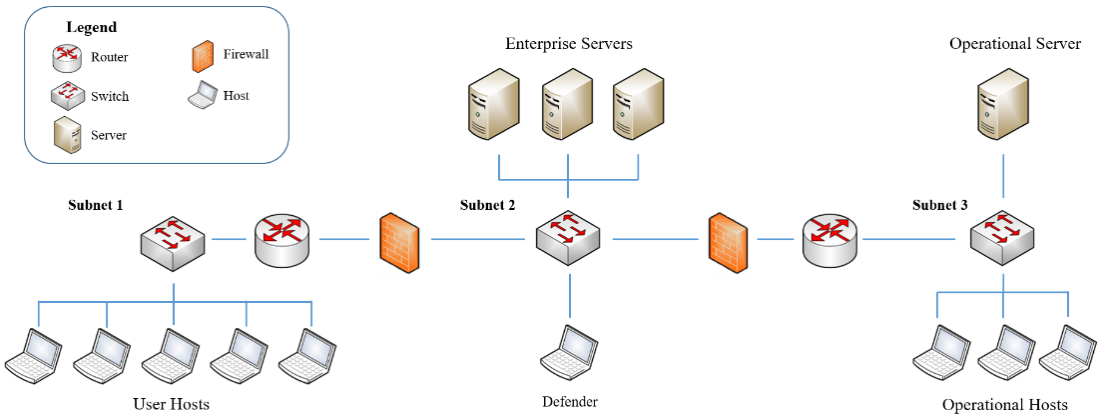

Figure 3.1: Network Layout Diagram [17]

real-world network. While multiple agents are modelled within the environment, only one agent is actively learning, reducing the multi-agent setup to a single-agent POMDP. These three agents perform their actions at each timestep within the environment in the following order: blue agent $\succ$ green agent $\succ$ red agent. For the remainder of this report, the term "adversarial agent" is used to refer to the specific red agent trained to evade the fixed blue agent.

The original challenge varies the simulated episode length between 30, 50 and 100 timesteps in their evaluation. For this project, however, each episode is assumed to last a maximum of 100 discrete timesteps in order to provide the adversarial attacker with sufficient time to attempt to evade the defender agent. The original environment also contains two hard-coded attacker agents, which select their actions according to explicit rules. These are developed to enable the training of an RL-based defender that can then be submitted to the challenge where it is evaluated against the same agents. While both these red agents are predictable in their behaviour, they have contrasting approaches.

The b-line agent is assumed to have prior knowledge of the network layout and attempts to take the most direct route to compromise the target server. The meander agent, on the other hand, is more explorative without any prior knowledge of the network, and attempts to map out all the different subnets before reaching the target server. The algorithms specifying these two agent strategies can be found in Appendix B. The aim of this project is to demonstrate that these two hard-coded attackers are insufficient in developing robust defender agents, and that adversarial policies can be found to evade the defender even when it is able to perform well against the original challenge attackers.

Action Spaces: The environment models the RL task of autonomous network defence using the following RL components explained in Section 2.1. The blue and red agents have their own distinct, discrete action spaces within the environment, shown in Tables 3.1 and 3.2 respectively. The blue action space is based on OpenC2 specification[1], while the red action space is based on MITRE ATT&CK techniques[2]. The green agent only has two available actions which it performs at random: the discovery action and the sleep action. The key idea for including the green agent is that it ensures the defender does not assume that all activity on the network is malicious.

Host States: Each host within the network can be in several states, which altogether determine the true state of the environment that is acted upon by the agents. Figure 3.2 shows the effects of the red and blue agents' actions on the host states. Each host, besides the initial foothold host, is initially unknown to the red agent. A host can become known to the red agent through either the `DiscoverRemoteSystems` actions, or by gaining privileged access to certain other hosts that contain the IP addresses of (unknown) hosts. Once known, the attacker is able to scan the host to discover active services, exploit one of these services and then escalate its privileges. At the same time, the defender can attempt to counter by removing the attacker's access or restoring the host. Note that removing an attackers access will only succeed if the attacker has `User` access to a host, and will automatically fail if the attacker has already escalated its privileges to `Privileged` access. The blue agent can also create different decoy services on a host, which both delay and locate the red agent. Any exploit action by the red agent that uses a decoy service automatically fails.

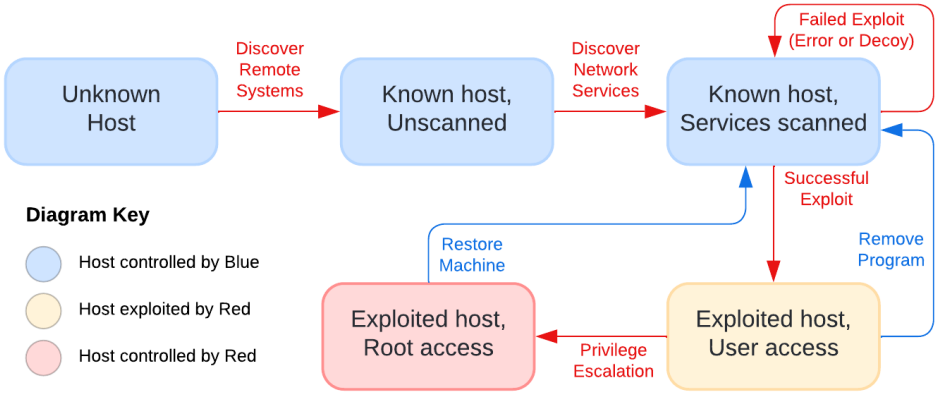

Figure 3.2: Effect of actions on host state [37]

[1]Open Command and Control (OpenC2), https://openc2.org/
[2]MITRE ATT&CK, https://attack.mitre.org/
[3]Note: This action occurs automatically if another action is chosen

Action	**Purpose**	**Parameters**	**Output**
Monitor	Collect information about flagged malicious activity on the system.[3]	None	Network connections and associated processes that are identified as malicious.
Analyse	Collect further information on a specific host to enable blue to better identify if red is present on the system.	Hostname	Information on files associated with recent alerts including signature and entropy.
Decoy: Apache, Femitter, HarakaSMPT, Smss, SSHD, Svchost or Tomcat	Setup of a decoy service (as specified by the action name) on a specified host. Green agents do not access these services, so any access is a clear example of red activity.	Hostname	An alert if the red agent accesses the new service.
Remove	Attempt to remove red from a host by destroying malicious processes, files and services.	Hostname	Success/Failure
Restore	Restoring a system to a known good state. This has significant consequences for system availability. This action punishes Blue by -1.	Hostname	Success/Failure

Table 3.1: Blue Action Space [37]

The above state transitions are not fully deterministic, and involve some probabilities to more accurately reflect the unpredictability of the real world. When there are multiple services (including decoy services) available to the attacker, there is a 75% chance that the attacker uses the highest ranked exploit. Furthermore, there is a small chance for the exploit action to result in privileged access directly, rather than user access. In this case, the red agent no longer has to perform the `PrivilegeEscalate` action on this host.

Action	**Purpose**	**Parameters**	**Output**
Discover Remote Systems	Discover new hosts/IP addresses in the network through active scanning using tools such as ping.	Subnet	IP addresses in the chosen subnet from hosts that respond to ping.
Discover Network Services	Discover responsive services on a selected host by initiating a connection with that host using tools such as nmap.	IP Address	Ports and service information.
Exploit Remote Service	Attempt to exploit a specified service on a remote system.	IP Address, Port	Success/Failure, Initial recon of host if successful.
Escalate	Escalate the agent's privileges on the specified host.	Hostname	Success/Failure, Internal information now available due to increased access to the host
Impact	Disrupt the performance of the network and fulfil red's objective of denying the operational service.	Hostname	Success/Failure
Sleep	Do nothing.	None	None

Table 3.2: Red Action Space [37]

Observation Spaces: Based on the environment's true state, the environment generates a partial observation for the agents at each timestep, where neither the red nor the blue agent's observation space reveals the full environment state. The red agent's observation space only ever includes information about the hosts that it is aware of. For each host known to the attacker, the environment keeps track of whether the host has been scanned through a `DiscoverNetworkService` action and what kind of access the red agent currently has on the host (None, User, Privileged). The red agent's observation also indicates whether its last action was successful, and then pads the rest of the observation vector with -1 values for unknown hosts to ensure all observations are the same length for the neural network input.

The blue agent receives a different view of the environment in its observation space. For each host in the network, the blue observation communicates the red agent's activity on the host, as well as its access. The activity value denotes whether there is no activity on the host, the red agent has scanned it, or the host has been exploited. The access value indicates whether the red agent's access to the host is none, user, privileged, or unknown to the blue agent. Therefore, in conclusion, the environment is partially observable for both agents, and models the ACD task as an asymmetric game where the agents have different action- and observation spaces.

Reward Functions: The agents' reward functions were modified from the original CAGE challenge for the agent training in this project, and will be covered by the project's implementation (Chapter 4) below.

3.2 Threat Model

The following threat model is assumed for the adversarial attack implemented in this project. The key assumption is that a fixed, autonomous cyber defence system exists in the real world, which has been trained using deep reinforcement learning. For instance, such a trained defender system could be purchased by customers and put directly within their network. In this case, an attacker aiming to compromise a network protected by such a system is likely to adapt their attacks from "traditional" cyber attack methods to targeting the ACD agent itself, following the arms race depicted in Figure 2.2.

The goal for this adversarial attack is to deceive the defender opponent (blue agent) in order to make it fail its task of defending the network, thereby gaining an advantage as the attacking agent in achieving their ultimate objective of compromising the network target server. In terms of knowledge and capabilities, we adopt a black-box threat model, where the attacker has no knowledge of its opponent's model parameters, but has black box query access to the model. Unlike previous work into RL evasion, the attacker is unable to directly permute the defender's actions [18] or observations [23, 30]. This follows the argument proposed by [29], stating that the defender's observations are based on logs and traffic information, which would realistically be inaccessible to the adversary modeled within the environment (and whose integrity could be verified anyways).

Instead, the attacker is capable of performing actions within the shared environment which affect its opponent's observations and thereby indirectly influence its behaviour through an adversarial red agent policy. Moreover, the attacker is restricted to valid

actions within the environment. For instance, a host cannot be exploited if it has not been scanned yet. This constraint is not relevant for prior work into adversarial policies [13], as there were no invalid actions for the adversarial policy to take.

Since the deployed victim policy is a fixed defender agent, the MARL setup is reduced to a single-agent learning problem for the adversarial agent. The attacker's strategy within this threat model, then, is to identify weaknesses in the opponent's policy through RL training experience. It can then exploit these by identifying the actions that induce misleading observations for the defender, thereby compromising the integrity of the defender agent's behaviour. The implementation of this strategy is covered in more detail below.

Chapter 4

Design and Implementation

4.1 Environment Customisations

The implementation process for this project started by fully comprehending the original CybORG environment, particularly how its different layers work together in manipulating the environment state, input, and output. The original environment was set up to handle the actions of only one RL-based agent externally, while representing the remaining agent(s) internally through multiple wrappers built on top of the base CybORG environment class. Every timestep, the input and respective output is communicated between the external agent and the internal environment through each of the wrappers in the following order, where each abstraction layer uses the data to perform specific tasks:

1: CybORG ▷ *Base class, outlines abstract methods*
2: EnvironmentController ▷ *Base env controller, handles internal agent(s)*
3: SimulationController ▷ *Implements simulation-specific functionality*
4: BlueTableWrapper ▷ *Handles blue agent state & observation output*
5: EnumActionWrapper ▷ *Initialises and updates agents' action spaces*
6: OpenAIGymWrapper ▷ *Ensures env output is compatible with OpenAI gym*
7: ChallengeWrapper ▷ *Bridge between external agent and underlying env*

For the CAGE research challenge [17], this meant the environment outputs the blue agent's next observation and reward for each action supplied by the external RL-based defender, while sampling actions from the red and optionally green agents internally in the `EnvironmentController` wrapper without any external output.

Custom Wrapper: For this project, a custom wrapper that sits on top of the original environment was developed in order to enable two external RL models (blue and red)

to interact with the environment, and allow the optional internal representation of a green agent internally as in the original environment. The custom wrapper has been designed to maintain compatibility with the original environment as much as possible, such that any defenders trained in the original environment can be easily transferred to and used within the custom wrapper. Figure 4.1 depicts a high-level overview of the new environment structure.

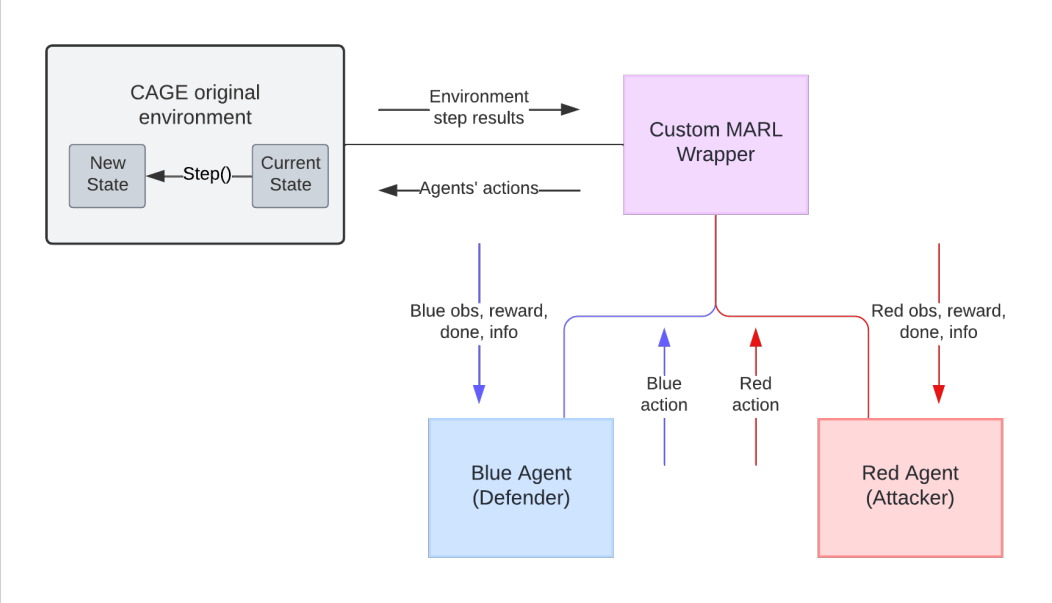

Figure 4.1: Custom Wrapper High-Level Diagram

In more detail, the custom wrapper assumes that either the red or blue agent's policy is fixed (i.e. not actively updating its model parameters) while the other is learning from the sample training episodes. This is achieved by passing a fixed model file into the wrapper, which is used to predict actions based on its individual observation at each timestep. The learning agent, on the other hand, is handled outside of the wrapper and provided with a reward along with its observations with which its policy can be updated.

As described in Section 3.2, the threat model for this project assumes that the defender's policy is fixed upon deployment. This assumption implies that for the purpose of this project, the fixed defender policy can be passed into the custom wrapper from which the adversarial policy can be developed through active learning within the new environment. Ultimately, the wrapper provides the underlying original environment with the actions of both the red and blue agents, which are used to step to the next environment state at each timestep. To enable the original environment to accommodate this new structure, the following key updates were made:

1. **Results.py**: The results class was updated such that the observations and action spaces for both the red and blue agent could be stored and communicated individually. The original single observation and action space for one external agent are initialized `None`, such that they remain available for use.

2. **CybORG.py**: The `step()` method in CybORG now accepts two external actions and returns a modified instance of the results with individual observations and action spaces. The original environment function is still available through `og_step()`.

3. **EnvironmentController.py**: Similar to CybORG, the `step()` method in EnvironmentController now accepts two external actions and returns a modified instance of the results with individual observations and action spaces. The `reset()` function also returns a modified results instance. The original environment functions remain accessible through `og_step()` and `og_reset()` respectively.

These changes have resulted in the following layer structure in comparison to the original CAGE wrapper layers above:

1: CybORG ▷ *Base class, can now process 2 external agents*
2: EnvironmentController ▷ *Base env controller, able to process 2 ext. agents*
3: SimulationController ▷ *Implements simulation-specific functions (unchanged)*
4: CustomWrapper ▷ *Combines and adapts functionality from original 4-7*

While the original environment separates different functionalities into various wrappers as a form of abstraction to enable customization, this project's wrapper is designed with clarity in mind. The custom wrapper includes the new functions necessary for the setup of this project, and adapts certain elements from the original layers to accommodate the new dual agent structure where applicable. As this wrapper is designed with a single purpose in mind, the choice to include all required methods in one central file was deemed more useful than building multiple layers, where the latter option would have made locating certain functions more difficult and time-consuming.

Action Inputs: As depicted in Figure 4.1 above, the wrapper provides the underlying environment with the actions of both the blue and red agents as input, in the form of an environment `Action` object instance. However, the RL-based agents contain a neural network that outputs an integer action, which maps to a particular action within the agent's action space. To enable the translation between these two formats, the wrapper implements an action space change function that translates the environment

action spaces into discrete action spaces compatible with OpenAI and therefore SB3 learning algorithms. The discrete action space is passed to the agent at every step in the environment to enable valid action selection. This design choice was made to avoid hard-coding each agent's action space into the wrapper, and thereby allow for easy updates to the agents' action spaces through the environment scenario YAML configuration files. This is based on the original environment's EnumActionWrapper [37].

Observation Outputs: Conversely, the observation outputs from the underlying environment are passed back to the agents through the wrapper. To enable the individual agent's observations to be used as input for their neural networks, the custom wrapper converts the dictionary observations received from the environment into a vector output by default. This follows the original environment's OpenAIGymWrapper to ensure compatibility with OpenAI gym as well as SB3 learning algorithms. Each agent's vector contains the information corresponding to their partial observation of the true environment state, as described in Section 3.1. Alternatively, the same observations produced by the environment can also be converted into a table that enables human readability and facilitates manual testing.

In conclusion, the design of this custom wrapper allows for a red adversarial agent to learn how to circumvent a particular fixed defender policy, as well as providing the functionality for (re-)training the blue agent to defend against an RL-based attacker instead of the hard-coded attackers used in the original environment.

4.1.1 Difficulties Faced

Overall, the task of understanding the environment inner workings was challenging as the environment setup is complex. However, this was essential in ensuring the same functionality is reflected in the project's custom wrapper for transferability between the original and custom environment, as well as compatibility with popular RL libraries such as OpenAI gym. This challenge was overcome by meticulous manual testing and maintaining clear comments in the project code to reflect function input and output expectations, as well as explicitly documenting any changes made to the original environment. When implementation bugs were inevitably encountered during the development of the custom wrapper, this allowed for faster identification of the root problem and thus easier debugging.

For instance, when first testing the new wrapper (once it successfully ran without errors) to train an adversarial agent against an RL-based defender, the actions supplied by both agents were consistently deemed invalid by the environment. This was resolved by manually stepping through the `test_valid_action()` function and identifying the action parameters that were failing the test. This revealed that there was a problem in the translation from the agents' selected integer actions into environment action instances, as well as a lack of update in the wrapper's IP address mapping which is changed more frequently in the underlying environment than expected. Once the encountered obstacles were identified and addressed, the custom wrapper was ready to be used for the next implementation step of this project: agent training.

4.2 Defender Training

To enable the development of an adversarial policy, an RL-based defender agent was trained first. For this task, the original CAGE challenge environment [17] was used, where the defender is trained against both of the original hard-coded attackers described in Section 3.1.

The Stable Baselines 3 (SB3) PPO algorithm implementation[1] is used for all agents trained in this project. This library was selected as it is known to be a reliable implementation of the learning algorithm [33] with good documentation, and allows for easy training, saving and subsequent loading of models which was essential for the setup of this project. For this particular agent, the algorithm's hyperparameters were set to those used in my Bachelor's thesis [24] as the initial defender's tasks were nearly identical. The defender used for this project was trained for a total of four million timesteps, with the original challenge's `Scenario2.yaml` scenario file.

Note that while the trained defender for this project effectively defends the network against the two hard-coded attackers, as shown in Section 5 below, it performs averagely in the original challenge evaluation where it scores close to that of the 12th-ranked submission. This is attributed to the fact the original challenge's evaluation does not include the green agent within its environment. While this was deemed an essential component to create a more realistic defender for this project, it also makes the blue agent's task more difficult. Moreover, this project's defender, as opposed to the challenge's winning submission[2], does not add any heuristics such as an action space reduction to

[1] https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html
[2] https://github.com/john-cardiff/-cyborg-cage-2

perform exceptionally well for the specific challenge evaluation. Instead, this project's defender aims to be more general, and does not alter to learning algorithm itself with domain-specific modifications. The hypothesis behind this choice is to avoid overfitting to the challenge's hard-coded attacker, and that a more general defender in turn would be more robust against an adversarial policy attack.

Defender Reward: One aspect that was slightly altered from the original challenge during training was the reward function for the defender. In particular, the reward received for performing a restore action was changed from -1.0 to -5.0 in order to develop a more realistic defender. As explained in Section 3.1, the restore action represents completely resetting a host, which the earlier iteration of the trained defender was found to perform nearly 50% of the time even if the attacker did not have privileged access to the hosts. In the real world, an autonomous defender would not be considered effective if it constantly takes key servers offline to reset them, thereby severely impacting the general availability of the entire network. Moreover, this also allows the defender agent to perform better in the original challenge evaluation as the negative reward for (unnecessary) restore actions builds up very quickly. Lastly, for the purpose of this project, this design choice also somewhat balances out the defender's advantage with the ability to create decoy services while the attacker agent does not have an action that enables it to identify said decoy services. In the real world, the defender usually does not have this advantage.

The defender's reward function has been summarised in Table 4.1 below, where the aim for the trained defender is to maximise the final episode reward and score as close to zero as possible. Besides the decreased restore reward, this reward function is identical to that of the original challenge environment, where the cost of the red agent's root access to a host is determined by its confidentiality value. For instance, the enterprise hosts within the network represent a higher value target for the attacker than the user hosts, which is reflected in a larger negative reward for the defender if the red agent obtains privileged access to them. Furthermore, the blue agent receives a large negative reward if the red agent achieves its goal of impacting the target operational server, which is seen as the ultimate compromise of the entire network as described in Section 3.1.

Subnet/Agent	Hosts	Action	Reward (per step)
Subnet 1	User hosts	Red root access	-0.1
Subnet 2	Enterprise hosts	Red root access	-1.0
Subnet 3	Target Server	Red root access	-1.0
Subnet 3	Operational Hosts	Red root access	-0.1
Red	Operational Server	Impact	-10
Blue	Any	Restore	**-5**

Table 4.1: Blue Agent Rewards [37]

4.3 Adversarial Attacker Training

Once a defender model had been successfully trained, the development of an RL-based adversarial attacker could commence. The first task in this process was tuning the PPO algorithm's hyperparameters for the attacker agent. Not only does the red agent have a different action space than that of the blue agent, the attacker's task requires learning valid sequences of actions with sparse rewards to successfully compromise the network. The defender, on the other hand, must learn to select the corresponding optimal defensive move for the given environment observation at each singular timestep. Therefore, the attacker's learning problem is a fundamentally different task than that of the defender and requires its own hyperparameters.

Hyperparameter Tuning: To find the optimal hyperparameters for the attacker agent, hyperparameter tuning was performed using Optuna[3]. Appendix A details the selected hyperparameters, along with their search ranges and the final results used to configure the training algorithm for the project's red agents. The Optuna tuning algorithm was run for 100 trials, where a set of hyperparameter values is selected for each trial using a Tree-structured Parzen Estimator (TPE) Sampler. Each hyperparameter combination is trained for 100,000 timesteps and then tested over 3 evaluation episodes. After the first 5 trials, a pruner is used to determine whether a given hyperparameter trial is promising using the median stopping rule. This states that a trial will be pruned "if the trial's best intermediate result is worse than median of intermediate results of previous trials at the same step"[4], and speeds up the tuning process significantly.

[3]https://optuna.readthedocs.io/en/stable/

[4]https://optuna.readthedocs.io/en/stable/reference/generated/optuna.pruners.MedianPruner.html

In addition to the hyperparameters selected by the tuning algorithm, two more factors were manually explored to influence the attacker's learning process. Firstly, a linear learning rate schedule was selected in order to reduce the size of the agent's policy updates as it progresses in its learning and further stabilize training. Secondly, the training episodes used as learning experience were generated across 16 parallel environments. The vectorised environments were also implemented to resolve instability that seemed to occur during agent training, where the agent got stuck trying a single action repeatedly without achieving anything midway through the training process.

Base Attacker: Once the attacker's hyperparameters had been configured for training in the custom wrapper, it quickly became clear that the adversarial agent struggled to learn which actions were valid for its provided observations against a strong trained defender that was countering its actions as it was trying to learn. To overcome this steep learning curve and figure out what valid action sequences it could use to compromise a network, a base red agent was first developed against a passive defender which only monitors the network without taking any defensive actions.

As shown in Figure 4.2 below, the base agent is able to learn a valid attack path to successfully impact the target operational server and overcome this initial learning curve. The base agent was trained for a total of 600,000 timesteps. The graph demonstrates that the agent starts to successfully impact the target server between 100,000 to 200,000 timesteps, where the average episode reward shows a steep incline. At the end of this initial training process, sample episodes show that the base agent is consistently able to successfully impact the target operational server within 25 timesteps.

This initial attacker agent was then used as a base model loaded into the custom training environment and fine-tuned to the fixed defender with further training. Further training against the fixed defender was required as the base RL-based attacker put directly against the trained defender agent was unable to gain access outside of the first subnet in the network. This is likely due to the defender's decoy actions, which the base agent had not encountered during its training against the passive defender and therefore had not yet learnt how to circumvent them.

Adversarial Reward: To fine-tune this base agent into an effective adversarial policy that takes advantage of the defender's weaknesses, a custom reward function was created. The design of the adversarial reward function, as discussed in Section 2.1, was integral to developing an effective adversarial policy. The key objective for the reward function design is that a higher total reward achieved by the agent during training corresponds to the agent moving closer to achieving its intended goal.

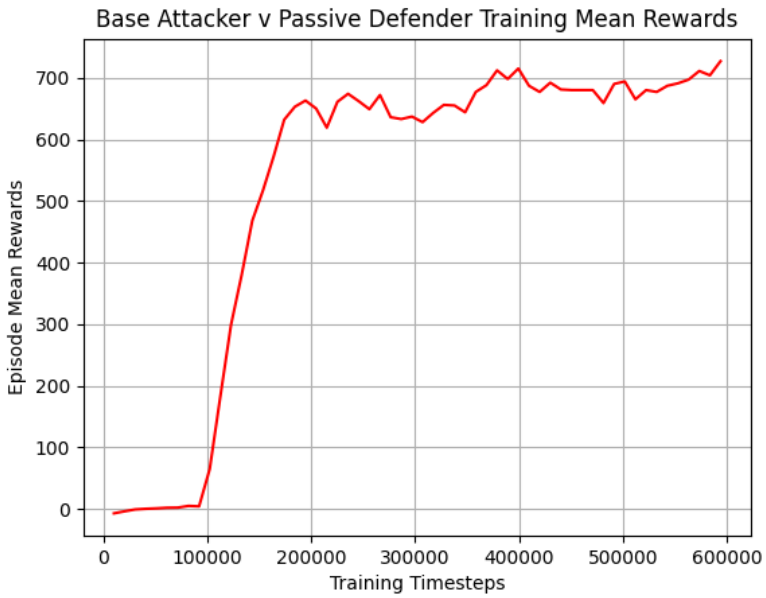

Figure 4.2: Attacker Initial Learning Curve

Prior research into adversarial policies generally uses a zero-sum reward setup [13, 41]. The concept of a zero-sum game comes from game theory, where one player's gain equals another player's loss. Within the scope of reinforcement learning, this occurs in a multi-agent environment where one agent receives the inverse reward of its opponent. For this project's asymmetrical setup, it became clear that this zero-sum reward setup is not effective for learning a successful adversarial policy, as the adversarial attacker is not merely aiming to reduce the defender's reward but also needs to learn how to effectively fulfill the original attacker's goal. The original environment's reward function (Table 4.1) is designed purely for the defender's task, where the attacker's task is fundamentally different and thus cannot be expressed using the same set of rewards. This is in contrast with prior adversarial work where the opposing agents have identical [41] or at least similar [13] abilities and goals.

To resolve this issue, several options were experimented with, including reward magnitudes, repetitions and bonuses. For context, the original defender's reward function is split into two child components: one that calculates the reward for the attacker's privileged access to hosts within the network, and one that calculates the reward for the attacker impacting (disrupting) services on the same hosts. The same functions were retained for the updated attacker's reward calculator, with the following changes: the red agent now only receives a positive reward for new privileged access to a host (where the defender receives a negative reward for each timestep the attacker

has privileged access to a host). In other words, if no new privileged sessions have been established by the attacker in relation to the previous timestep, this child reward function returns a reward of zero. This decision was made to motivate the attacker to exploit and compromise further hosts, and avoid the agent learning to do nothing / perform redundant actions while still receiving positive rewards.

Furthermore, the privilege reward calculator was updated to return `max(total - self.old_total, 0)` to avoid a situation where the red agent wrongfully receives a negative reward when it loses privileged access to a host due to the defender restoring or removing the host access. This way, the previously explained privileged access reward for the red agent is still implemented, without unexpected negative rewards that may accidentally teach the agent that its attempted action in the same timestep was ineffective or erroneous.

While experimenting with different reward setups for the adversarial agents, it was often observed that the attacker and defender got stuck in a loop where the attacker exploits a certain host and then escalates its privileges, which the defender subsequently restores. This leads to an accumulation of reward for the attacker for each "new" privileged access. In the context of a cyber security attack, this more closely resembles a Denial of Service attack on a particular host by forcing the defender to continuously restore it. While this is effective in terms of adversarial behaviour and reducing the defender's reward, it does not achieve the original attacker's end goal within the network. Thus, the reward function was updated to make it more appealing for the attacker to break out of this loop and move laterally to ultimately impact the operational server. To this end, the reward magnitude for impacting target operational server was increased from +10 to +25 in order to ensure this was the red agent's main objective.

Bonus Rewards: The sparse reward setup with only a large reward for a successful impact action on the target server did not enable the agent to learn the valid action sequence to move from an enterprise host to scanning, exploiting, escalating and impacting the target server. To aid with this, a bonus reward was added for the first time the red agent scans and exploits the target operational server in an episode (+2 and +5 respectively). This reward is then diminished for subsequent times the agent performs these actions (+0.1 and +0.5), such that the agent is aware that it is targeting the sensitive key operational server while making certain it does not learn to unnecessarily repeat these actions simply to increase its total reward. Nevertheless, the diminished reward is still positive to avoid discouraging the agent to make use of these actions for its adversarial behaviour if it finds that they are helpful in circumventing the defender.

The above updates to the attacker's reward function are valid within the project's threat model, as the reward is only based on the information available to the attacker within that timestep. In other words, the reward function only looks at the privileged access on already known / compromised hosts, not including any unknown hosts. Therefore, it does not use any knowledge that would not be available to the attacker at any point in time. Finally, following the original environment's reward function, any invalid action chosen by the red agent still receives a small negative reward as per the original environment reward function. Table 4.2 below summarises the above-described reward function for the red agent, where the aim for the agent is to maximise the cumulative episode reward.

Hosts	**Action**	**Reward (per step)**	**Repeated reward**
User hosts	Red root access	+0.1	n/a
Enterprise hosts	Red root access	+1.0	n/a
Target Server	Red scan	+2.0	+0.1
Target Server	Red exploit	+3.0	+0.5
Target Hosts	Red root access	+5.0	n/a
Target Server	Impact	+25	n/a
Any	Invalid action	-0.1	n/a

Table 4.2: Custom Red Agent Rewards

Using the above hyperparameters and reward function, the base attacker model was further trained for 2 million tinesteps against the fixed trained defender policy. Interestingly, when analysing the adversarial attacker's behaviour in sample episodes over the course of the adversarial attacker's training, it proved difficult to see whether it is performing seemingly illogical or redundant actions as a form of deceptive behaviour or because it has not learnt sufficiently to perform optimally. Furthermore, based on manual testing, some of the exploit actions need to be tried quite a few times by the attacker before they are successful due to the stochastic nature of the environment as well as the different decoy services set up by the defender, which can be difficult to learn for the agent as it does not receive any reward for unsuccessful actions. Nevertheless, as the next section will demonstrate, the adversarial agent managed to learn a policy that is able to circumvent the decoy services and ultimately evade the defender.

Environment Issues: Over the course of the implementation stage of this project, a few issues were identified in the original environment which are mainly attributed to the fact that the environment was designed only for training a defender and thus lacked a few key functionalities implemented for training an attacker RL-based agent. As per the original challenge description [37], there is a small chance that an attacker's successful exploit leads directly to privileged access rather than user access. While the environment implements this functionality from the defender's point of view, it fails to implement the relevant updates to the attacker's state to process this scenario. Instead, while the red agent's observation shows it has gained privileged access, its internal state has not been updated and thus acting upon this access is judged to be invalid by the environment, creating a confusing and incorrect learning problem for the red agent. To rectify this, and to ensure the attacker does not still have to perform an escalate action, the underlying environment was edited such that this escalate action is automatically performed where applicable, and the project's custom wrapper updates the red agent's observation with the relevant information as described in Section 3.1.

Furthermore, manual debugging of the environment revealed that the red agent's access does not get updated in its observation when the blue agent removes or restores a host, likely because this is not necessary for the challenge's original hard-coded attackers. For the training of an RL-based red agent, the project's wrapper ensures that the red agent's observation is appropriately updated when the blue agent performs a remove or restore action on a host if the host was already known to the red agent (i.e. if the host was already in the red agent's observation information). Finally, a few sanity checks were implemented in the environment to avoid a crash due to edge cases not encountered by hard-coded red agents but that are faced by the RL-based agent as it explores both valid and invalid actions.

Chapter 5

Evaluation and Analysis

5.1 Agent Evaluation

5.1.1 Evaluation Methodology

The evaluation method used in this project includes several distinct attackers to assess the performance of the defender. The first attacker is a passive red agent which only performs the sleep action, to evaluate whether the defender is able to identify a lack of malicious activity on the network and act accordingly. Both the meander and b-line hard-coded red agents used in the original challenge have also been included in this evaluation, as well as the adversarial attacker developed over the course of this project.

Following the original CAGE challenge evaluation, the defender's performance is assessed in terms of its average final reward across 1000 evaluation episodes of 100 timesteps. To ensure comparability and reproducibility between the results from different attackers, these evaluation episodes have been seeded. During these evaluation episodes, as opposed to the training process, both the blue and red agent's policies are fixed without any learning updates. In contrast to the original challenge evaluation, the green agent has been used in the evaluation environment. As explained in Section 3.1, this agent represents the actions of genuine users and is essential for evaluating whether the trained defender can distinguish malicious activity from benign user behaviour.

Furthermore, to assess the defender's robustness with respect to the attacker's objective, two more evaluation metrics have been designed for this project. The compromise rate denotes the percentage of the 1000 evaluation episodes in which the red agent was able to gain privileged access to the target operational server. Therefore, a larger compromise rate indicates a worse performance by the defender. Conversely, the win

rate signifies the percentage of evaluation episodes where the blue agent was able to successfully prevent the red agent from impacting the target operational server. Here, a higher win rate indicates better performance for the defender.

5.1.2 Evaluation Results

The below table summarises the results of the above-described evaluation:

Blue	Red	Blue reward	Compromise rate	Blue win rate
Trained RL	Passive (sleep)	0.0	0.0%	100.0%
Trained RL	Meander [37]	-27.5	0.0%	100.0%
Trained RL	B-line [37]	-23.4	0.5%	99.7%
Trained RL	Adversarial RL	**-119.3**	**99.7%**	**59.4%**

Table 5.1: Evaluation Results (1000 seeded episodes, 100 timesteps each)

Win/Compromise Rates: To re-iterate Section 1.2, the objective for this project is to develop an adversarial policy that is able to effectively decrease the trained defender's performance and successfully impact the target operational server within the scenario network environment. The results table shows that the defender is near-perfect in defending the target operational server from the hard-coded attackers, with a 100% and 99.7% win rate against the meander and b-line agent respectively. However, the trained defender is only able to successfully prevent the adversarial attacker from impacting the target server in 59.4% of the evaluation episodes. Moreover, while the hard-coded attackers are not able to compromise the target server at all in nearly all evaluation episodes, the adversarial attacker manages to gain privileged access to the server 99.7% of the time. In other words, the adversarial attacker is mainly hindered from reaching its final objective by a last-minute countermove where the defender restores the server in the same timestep that the attacker attempts to perform the impact action.

Mean Rewards: These results are also reflected in the mean final reward of the defender. The reward obtained by the defender against the adversarial attacker sees a significant reduction from that obtained against the hard-coded attackers, with a decrease of more than 300%. This larger negative reward is due to the successful impact actions performed by the adversarial attacker. Furthermore, the adversarial attacker is able to force the defender to perform more restore actions, thereby also decreasing its final reward.

Intriguingly, the b-lines agent's average final reward is higher than that of the meander agent even though it performs better according to the compromise and win rate. This is likely because the trained defender has a slightly different focus for the different attacker strategies. For the meander agent, which spreads its privileged access more widely across the network, the defender is more skilled at preventing the attacker from gaining privileged access to the target operational server but struggles to completely remove privileged access to all other hosts in the process, thereby gaining relatively more negative rewards at each timestep. For the b-line agent, however, the defender is better at removing its privileged access from user and enterprise hosts as the attacker does not spread its access as much. Nevertheless, as the table shows, this results in a very small chance that the attacker slips through the blue agent's defences. This in turn demonstrates that the original challenge evaluation may not be fully comprehensive in judging the success of the defender.

Another interesting observation that was made during the evaluation process is that the win rate for the defender further decreases to 49.4% against the adversarial attacker when the episode length is increased to 150 timesteps. This suggests that the attacker agent may not always have enough time to successfully evade the defender within the original challenge's evaluation setup. However, in most real-world settings, the attacker would not be constrained to a maximum time within which the attack must be successfully completed, and the defender should be robust against such an attack regardless of an episode time limit.

All in all, the evaluation results show that the trained defender developed in this project is able to effectively defend the network from the original hard-coded attackers. However, by using reinforcement learning to develop an adversarial attacker that is able to find weaknesses in the fixed defender policy, the defender's performance was drastically decreased. Not only is the adversarial policy able to compromise the target server 99.7% of the time, but the defender is only able to prevent the server from being impacted in less than 60% of the evaluation episodes. A key consideration to be made here connects back to the asymmetrical nature of cyber security described in Chapter 1, where the attacker only needs to succeed once to achieve their attack objective while the defender must be able to successfully defend against all attack attempts 100% of the time. Thus, a failure rate of over 40% would be considered severely inadequate for any model to be deployed and adopted in the real world. An essential question, then, is how the adversarial attacker managed to deceive and evade the seemingly effective defender policy?

5.2 Policy Analysis

To analyse the behaviour of the different attackers evaluated for this project against the same trained defender, radar charts have been used to highlight the difference in agent policies. To create the plots shown in Figure 5.1, the actions performed by each red agent during the above-mentioned 1000 evaluation episodes were logged to a `JSON` file. This file was then further processed to calculate the number of times each type of attacker action (Table 3.2) was performed by the red agents, expressed as percentages in the below charts.

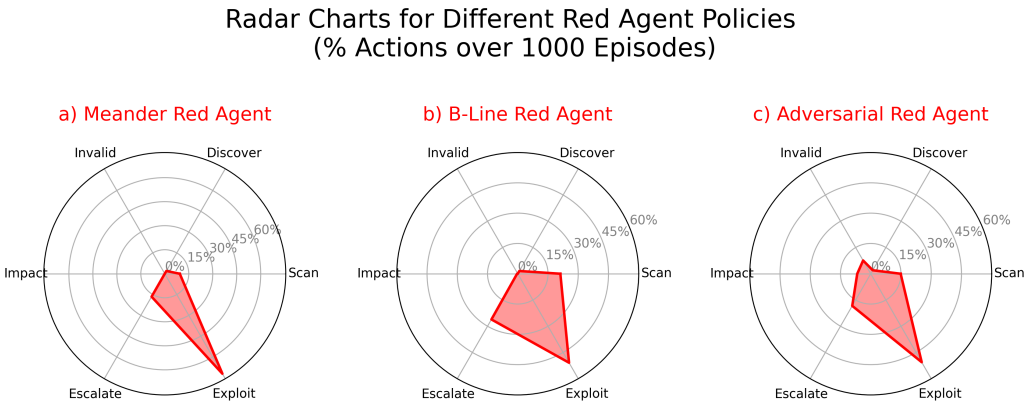

Figure 5.1: Red Agent Policies Comparison

The chart shows a noticeable increase in the amount of impact actions for the adversarial attacker in comparison with both the meander and b-line agents. Moreover, the plot also shows that while the adversarial policy does successfully use its impact actions, it still spends the majority of its actions exploiting hosts on the network similar to the hard-coded attackers. However, the adversarial agent's policy is visually in-between the two hard-coded agents, which is surprising as both the hard-coded agents are ineffective against the same defender. In general, these results show that the adversarial attacker has been able to find a policy that defeats the defender by surprising/confusing it with behaviour it has not encountered before in the hard-coded "logical" attackers while using the same actions.

Adversarial Strategy: Based on example episode analysis, this adversarial strategy includes distracting the defender by seemingly moving backwards at certain points to stretch its attention. For example, the defender generally responds to the attacker escalating privileges on a user host by setting up a decoy service to an enterprise host and/or restoring the user host. However, the adversarial agent has learnt that if it already

has access to said enterprise host and wants to retain this access to connect to the target operational server, it can prevent the defender from restoring its access by re-escalating a previous user host. This tricks the defender into focusing on the wrong subnet, and provides the attacker with an additional timestep to expand its access across the network and move a step closer to achieving its objective. The adversarial policy has also learnt that once new IP addresses have been obtained through privileged access to certain hosts, they do not have to be exploited from the same host that contained the info. This is in direct contrast to the hard-coded agents' strategies. For instance, once the IP for the target operational server has been discovered through privileged access to Enterprise 2, privileged access to any of the Enterprise hosts can be used to connect to the target server. This allows the adversarial agent to diversify its attack paths compared to the hard-coded agents.

The policy analysis also shows that the adversarial attacker still performs invalid actions for roughly 7% of its total actions across the 1000 evaluation episodes. This can in part be justified by the fact that the red agent may attempt to perform an action that is made invalid by the blue agent's action in the same timestep. Additionally, it is likely that the adversarial agent still makes mistakes from time to time, which is seemingly difficult to avoid for such a complex learning problem. While this indicates that there is still room for improvement for the adversarial policy, this does not prevent it from achieving a significant win rate against the trained defender. Ultimately, the adversarial agent does not have to learn to perform only valid actions, as long as it is able to evade the defender.

Network Access: The difference in red agent behaviours and their resulting network access can also be shown visually on the scenario network diagram. Figure 5.2 shows the maximum access that the b-line attacker is able to obtain in a sample episode of 100 timesteps. The b-line agent successfully compromises another user host (`User4`) in the first subnet and proceeds to attempt to gain root access to an enterprise host for the remainder of the episode. However, the defender prevents it from escalating its privileges and thereby prevents it from moving laterally towards the target server.

The maximum network access of the adversarial attacker is shown in Figure 5.3. This figure shows that the adversarial policy described above successfully gains root access to the target operational server and is then able to perform the impact action to fulfill the attacker's objective. While this particular snapshot demonstrates that the attacker is able to compromise all user hosts, the remainder of this episode also shows that it is able to use its actions to deceive the defender into restoring several of these user hosts

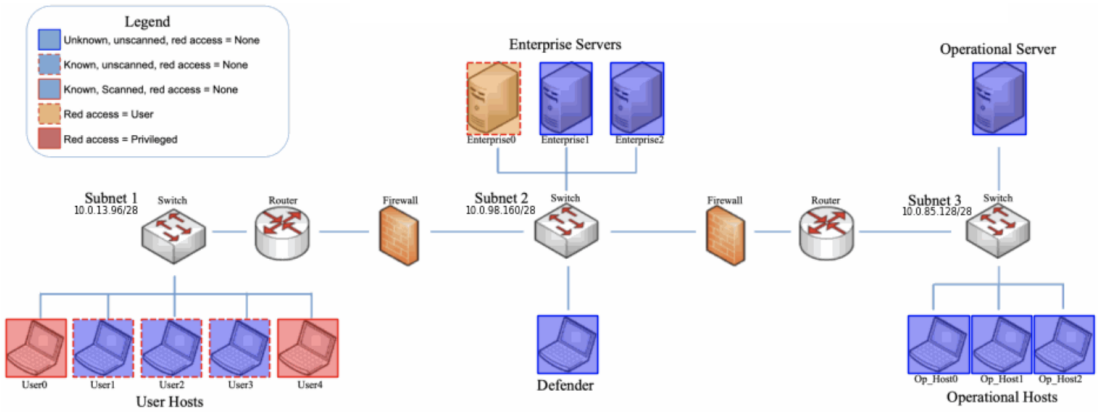

Figure 5.2: Attacker Access Trained Defender v B-Line Attacker

while privileged access to the other subnets has already been achieved. This provides the red agent with an extra timestep to reach its impact goal. Furthermore, as previously described, the figure shows that the attacker is able to exploit the operational server from multiple enterprise servers thereby further stretching the defensive capabilities of the blue agent. A final observation from the episode visualisation is that the hosts in the 3rd subnet are not very relevant and are generally skipped altogether by the adversarial attacker. GIF versions of these episodes have been submitted within the project's source code along with this report to enable visualisations of an entire episode.

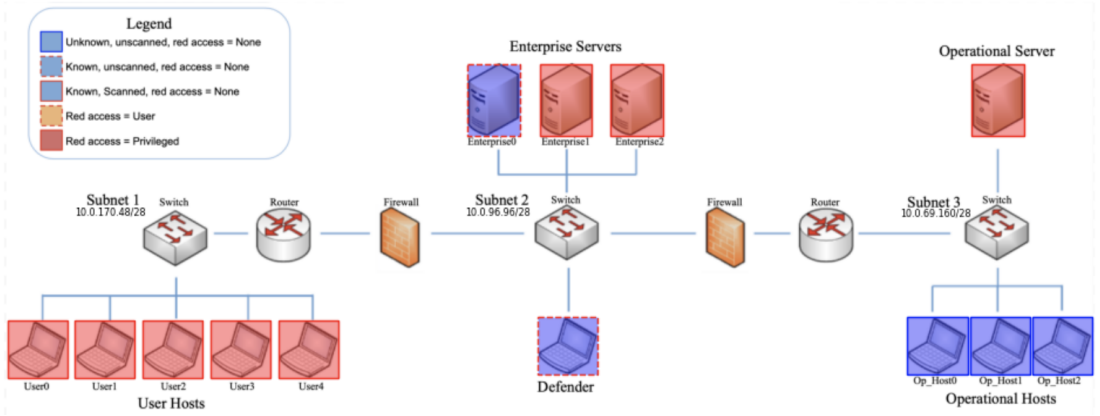

Figure 5.3: Attacker Access Trained Defender v Adversarial Attacker

Chapter 6

Conclusions

6.1 Summary

The focus of this project was the customisation of the CybORG ACD environment, to enable the training and evaluation of an adversarial policy against a fixed, trained defender. This project proved to be a difficult task due to the imbalance between the attacker and defender in the original environment. The resulting impact rate on the target server was lower than hoped for. Nevertheless, as outlined in Chapter 1, the adversary only needs their attack to succeed once, while the defender should succeed in their defence 100% of the time.

To answer the research questions outlined in Chapter 1, the adversarial policy was able to consistently evade the defender and gain root access to the target server, decrease the defender's reward by over 300%, and decrease its win rate by over 40%. It achieved these results by adopting a strategy that is not always logical. For instance, by re-attacking already compromised nodes and finding different attack paths, thereby diverting the defender's attention.

All in all, the project's aims were successfully fulfilled, and the evaluation results demonstrate a clear lack of robustness in the trained defender policy. This is a significant finding in the field of ACD, and DRL in general, and highlights the importance of considering such an attack in future research. The following sections will provide more detail on potential avenues for future work as well as the work's wider implications.

6.2 Future Work

The primary direction for future work is developing potential countermeasures to this project's attack. Adversarial training has been found to be an effective defence to adversarial examples in supervised learning. Gleave et al. demonstrated that while this strategy can be used to make DRL agents more robust against a specific adversarial attack, it does not solve the underlying arms race as the attack can simply be repeated to find a new adversarial policy [13].

Instead, Guo et al. proposed a training method to find the Nash equilibrium point for provable worst-case performance of both agents [15]. Liu et al. extended the original adversarial training with timescale separation to avoid overfitting and achieve robustness against a range of adversarial policies [21]. It would be of interest to test these proposed countermeasures for this project's particular attack.

As outlined in Chapter 4, the defender has an (unrealistic) advantage where it is able to set up decoy services to block the red agent, while the attacker is unable to identify decoy services. This was addressed in the latest CAGE challenge, where the red agent can perform `DiscoverDeception`, and it would be interesting to test whether the availability of such an action would further improve the adversarial agent's performance.

Lastly, the transferability and generalisation of the attack could be explored within future work. For instance, testing whether this adversarial policy could evade the original challenge's winning submissions. As these agents are more specific and perform better in the challenge, are they more vulnerable to the attack compared to this project's more general agent? Furthermore, the network size and/or complexity could be experimented with to evaluate the generalisability of both attacker and defender, as well as evaluating the generalisation of the adversarial policy from a simulated to an emulated system.

6.3 Discussion

This work highlights to the ACD research community that adversarial attacks must be considered and countered within future systems if autonomous defenders are to be deployed in the real world. To this end, this project may serve as a novel benchmark for future research to test their defenders in general, and their countermeasures against adversarial policy attacks in particular. Ultimately, security professionals and clients may benefit from this progress within the field of ACD, as it moves a step closer to real-world deployment of autonomous defence systems.

Furthermore, robustness against such evasion attacks is not only applicable within the domain of cyber security. Schott et al. argue that security, robustness and generalisation go hand-in-hand [31], making this research not only applicable to ACD, but also to the wider field of DRL. Securing agents against adversarial attacks is a crucial step for overcoming the reality gap [6] between simulation and unpredictable real-world deployment.

Bibliography

[1] Ioannis Agrafiotis, Jason RC Nurse, Michael Goldsmith, Sadie Creese, and David Upton. A taxonomy of cyber-harms: Defining the impacts of cyber-attacks and understanding how they propagate. *Journal of Cybersecurity*, 4(1):tyy006, 2018.

[2] Alex Andrew, Sam Spillard, Joshua Collyer, and Neil Dhir. Developing optimal causal cyber-defence agents via cyber security simulation. *arXiv preprint arXiv:2207.12355*, 2022.

[3] Richard Bellman. The theory of dynamic programming. *Bulletin of the American Mathematical Society*, 60(6):503–515, 1954.

[4] Battista Biggio, Igino Corona, Davide Maiorca, Blaine Nelson, Nedim Šrndić, Pavel Laskov, Giorgio Giacinto, and Fabio Roli. Evasion attacks against machine learning at test time. In *Machine Learning and Knowledge Discovery in Databases: European Conference, ECML PKDD 2013, Prague, Czech Republic, September 23-27, 2013, Proceedings, Part III 13*, pages 387–402. Springer, 2013.

[5] Samuel Chng, Han Yu Lu, Ayush Kumar, and David Yau. Hacker types, motivations and strategies: A comprehensive framework. *Computers in Human Behavior Reports*, 5:100167, 2022.

[6] Jack Collins, David Howard, and Jurgen Leitner. Quantifying the reality gap in robotic manipulation tasks. In *2019 International Conference on Robotics and Automation (ICRA)*, pages 6706–6712. IEEE, 2019.

[7] Ciprian A Corneanu, Meysam Madadi, Sergio Escalera, and Aleix M Martinez. What does it mean to learn in deep networks? and, how does one detect adversarial attacks? In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pages 4757–4766, 2019.

[8] Arun Das and Paul Rad. Opportunities and challenges in explainable artificial intelligence (xai): A survey. *arXiv preprint arXiv:2006.11371*, 2020.

[9] Leonard Dung. Current cases of ai misalignment and their implications for future risks. *Synthese*, 202(5):138, 2023.

[10] Maddy Ell and Saman Rizvi. Cyber security breaches survey 2024, Apr 2024.

[11] Jonas Eschmann. Reward function design in reinforcement learning. *Reinforcement Learning Algorithms: Analysis and Applications*, pages 25–33, 2021.

[12] Robin Gandhi, Anup Sharma, William Mahoney, William Sousan, Qiuming Zhu, and Phillip Laplante. Dimensions of cyber-attacks: Cultural, social, economic, and political. *IEEE Technology and Society Magazine*, 30(1):28–38, 2011.

[13] Adam Gleave, Michael Dennis, Cody Wild, Neel Kant, Sergey Levine, and Stuart Russell. Adversarial policies: Attacking deep reinforcement learning. *arXiv preprint arXiv:1905.10615*, 2019.

[14] Ian J Goodfellow, Jonathon Shlens, and Christian Szegedy. Explaining and harnessing adversarial examples. *arXiv preprint arXiv:1412.6572*, 2014.

[15] Wenbo Guo, Xian Wu, Lun Wang, Xinyu Xing, and Dawn Song. {PATROL}: Provable defense against adversarial policy in two-player games. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 3943–3960, 2023.

[16] Nektaria Kaloudi and Jingyue Li. The ai-based cyber threat landscape: A survey. *ACM Comput. Surv.*, 53(1), feb 2020.

[17] Mitchell Kiely, David Bowman, Maxwell Standen, and Christopher Moir. On autonomous agents in a cyber defence environment. *arXiv preprint arXiv:2309.07388*, 2023.

[18] Xian Yeow Lee, Sambit Ghadai, Kai Liang Tan, Chinmay Hegde, and Soumik Sarkar. Spatiotemporally constrained action space attacks on deep reinforcement learning agents. In *Proceedings of the AAAI conference on artificial intelligence*, volume 34, pages 4577–4584, 2020.

[19] Guofa Li, Yifan Yang, Shen Li, Xingda Qu, Nengchao Lyu, and Shengbo Eben Li. Decision making of autonomous vehicles in lane change scenarios: Deep

reinforcement learning approaches with risk awareness. *Transportation research part C: emerging technologies*, 134:103452, 2022.

[20] Timothy P Lillicrap, Jonathan J Hunt, Alexander Pritzel, Nicolas Heess, Tom Erez, Yuval Tassa, David Silver, and Daan Wierstra. Continuous control with deep reinforcement learning. *arXiv preprint arXiv:1509.02971*, 2015.

[21] Xiangyu Liu, Souradip Chakraborty, Yanchao Sun, and Furong Huang. Rethinking adversarial policies: A generalized attack formulation and provable defense in rl. *arXiv preprint arXiv:2305.17342*, 2023.

[22] Hal Lonas. Cybersecurity: An asymmetrical game of war, August 2017.

[23] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083*, 2017.

[24] Melanie Meijer. Automated cyber defence by deep reinforcement learning. Bachelor's thesis, Cardiff University, 2023.

[25] Kevin D Mitnick and William L Simon. *The art of deception: Controlling the human element of security*. John Wiley & Sons, 2003.

[26] Volodymyr Mnih, Adria Puigdomenech Badia, Mehdi Mirza, Alex Graves, Timothy Lillicrap, Tim Harley, David Silver, and Koray Kavukcuoglu. Asynchronous methods for deep reinforcement learning. In *International conference on machine learning*, pages 1928–1937. PMLR, 2016.

[27] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[28] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Alex Graves, Ioannis Antonoglou, Daan Wierstra, and Martin Riedmiller. Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*, 2013.

[29] Andres Molina-Markham, Cory Miniter, Becky Powell, and Ahmad Ridley. Network environment design for autonomous cyberdefense. *arXiv preprint arXiv:2103.07583*, 2021.

[30] Alessio Russo and Alexandre Proutiere. Towards optimal attacks on reinforcement learning policies. In *2021 American Control Conference (ACC)*, pages 4561–4567. IEEE, 2021.

[31] Lucas Schott, Josephine Delas, Hatem Hajri, Elies Gherbi, Reda Yaich, Nora Boulahia-Cuppens, Frederic Cuppens, and Sylvain Lamprier. Robust deep reinforcement learning through adversarial attacks and training: A survey. *arXiv preprint arXiv:2403.00420*, 2024.

[32] Lucas Schott, Hatem Hajri, and Sylvain Lamprier. Improving robustness of deep reinforcement learning agents: Environment attack based on the critic network. In *2022 International Joint Conference on Neural Networks (IJCNN)*, pages 1–8. IEEE, 2022.

[33] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. Proximal policy optimization algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[34] Jonathon Schwartz. Autonomous penetration testing using reinforcement learning, 2018. URL for simulator documentation: https://networkattacksimulator.readthedocs.io/en/latest/.

[35] David Silver, Thomas Hubert, Julian Schrittwieser, Ioannis Antonoglou, Matthew Lai, Arthur Guez, Marc Lanctot, Laurent Sifre, Dharshan Kumaran, Thore Graepel, et al. A general reinforcement learning algorithm that masters chess, shogi, and go through self-play. *Science*, 362(6419):1140–1144, 2018.

[36] Chris Simmons, Charles Ellis, Sajjan Shiva, Dipankar Dasgupta, and Qishi Wu. Avoidit: A cyber attack taxonomy. *University of Memphis, Technical Report CS-09-003*, 2009.

[37] Maxwell Standen, Martin Lucas, David Bowman, Toby J Richer, Junae Kim, and Damian Marriott. Cyborg: A gym for the development of autonomous cyber agents. *arXiv preprint arXiv:2108.09118*, 2021.

[38] Richard S Sutton and Andrew G Barto. *Reinforcement learning: An introduction*. MIT press, 2018.

[39] Christian Szegedy, Wojciech Zaremba, Ilya Sutskever, Joan Bruna, Dumitru Erhan, Ian Goodfellow, and Rob Fergus. Intriguing properties of neural networks. *arXiv preprint arXiv:1312.6199*, 2013.

[40] Microsoft Defender Research Team. Cyberbattlesim. https://github.com/microsoft/cyberbattlesim, 2021. Created by Christian Seifert, Michael Betser, William Blum, James Bono, Kate Farris, Emily Goren, Justin Grana, Kristian Holsheimer, Brandon Marken, Joshua Neil, Nicole Nichols, Jugal Parikh, Haoran Wei.

[41] Tony Tong Wang, Adam Gleave, Tom Tseng, Kellin Pelrine, Nora Belrose, Joseph Miller, Michael D Dennis, Yawen Duan, Viktor Pogrebniak, Sergey Levine, et al. Adversarial policies beat superhuman go ais. In *International Conference on Machine Learning*, pages 35655–35739. PMLR, 2023.

[42] Chao Yu, Akash Velu, Eugene Vinitsky, Jiaxuan Gao, Yu Wang, Alexandre Bayen, and Yi Wu. The surprising effectiveness of ppo in cooperative multi-agent games. *Advances in Neural Information Processing Systems*, 35:24611–24624, 2022.

[43] Liang Yu, Weiwei Xie, Di Xie, Yulong Zou, Dengyin Zhang, Zhixin Sun, Linghua Zhang, Yue Zhang, and Tao Jiang. Deep reinforcement learning for smart home energy management. *IEEE Internet of Things Journal*, 7(4):2751–2762, 2019.

Appendix A

Hyperparameter Tuning

Parameter	**Value**	**Search Range**	**Distribution**
Total timesteps	2M	[600k, 1M, 2M, 4M]	Manual
Batch Size	512	[32, 64, 128, 256, 512]	Categorical
Number environments	16	[0,8,16]	Manual
Number steps	2048	[256, 512, 1024, 2048, 4096, 8192, 16384]	Categorical
Discount factor	0.9	[0.9, 0.95, 0.98, 0.99, 0.995, 0.999]	Categorical
Learning rate	0.00151	[1e-5, 0.01]	Log Uniform
Learning rate schedule	Linear	[None, Linear]	Manually
Entropy coefficient	0.00371	[0.0000001, 0.01]	Log Uniform
Clipping range	0.1	[0.1, 0.2, 0.3, 0.4]	Categorical
Epochs per update	20	[1, 5, 10, 20]	Categorical
GAE lambda	1.0	[0.8, 0.9, 0.92, 0.95, 0.98, 0.99, 1.0]	Categorical
Max gradient norm	0.7	[0.3, 0.5, 0.6, 0.7, 0.8, 0.9, 1, 2, 5]	Categorical
Value function coefficient	0.23666	[0, 1]	Uniform
Network architecture	medium	[tiny, small, medium][1]	Categorical
Orthogonal initialization	True	[True, False]	Categorical
Activation function	tanh	[tanh, relu]	Categorical

Table A.1: Hyperparameters for SB3 Proximal Policy Optimization[2] Adversarial Agent

A.1 Hardware Specifications

The following details describe the hardware and software versions used for the agent tuning, training and evaluation over the course of this project:

- Operating System: Microsoft Windows 10 Pro
- Processor: AMD Ryzen 5 1600 Six-Core Processor, 3200 Mhz, 6 Core(s), 12 Logical Processor(s)
- RAM: 32.0 GB
- Python version: 3.12.0
- IDE: Visual Studio Code (version 1.92.0)

[1] tiny: dict(pi=[64], vf=[64]), small: dict(pi=[64, 64], vf=[64, 64]), medium: dict(pi=[256, 256], vf=[256, 256])

[2] https://stable-baselines3.readthedocs.io/en/master/modules/ppo.html

Appendix B

CAGE Attackers

The following sections illustrate the general tactics of both hard-coded attackers included in the original challenge.

B.1 Meander Agent

The meander agent is described as more explorative without any prior knowledge of the network [17], and attempts to map out all the different subnets before reaching the target server. Based on the original challenge code, the meander agent selects its actions as outlined in Algorithm 2.

B.2 B-line Agent

The b-line agent is assumed to have prior knowledge of the network layout and has a hard-coded sequence of actions that it attempts to take, representing the most direct route (15 actions in total) to compromise the target server [17]. Based on the original challenge code, the meander agent selects its actions as outlined in Algorithm 3.

Algorithm 2 Meander agent get_action()

```
if Privileged access Op_Server0 then                    ▷ Always impact if able
    return action ← Impact Op_Server0
for subnet in known subnets do                          ▷ Start by scanning
    if subnet not in scanned subnets then
        add subnet to scanned subnets
        return action ← DiscoverRemoteSystems subnet
for address in known addresses do                       ▷ Act on IP addresses discovered
    if address not in scanned addresses then
        add address to scanned addresses
        return action ← DiscoverNetworkServices address
for host in known hosts do                              ▷ Escalate privileges on owned hosts
    if host is exploited and not already escalated then
        add host to escalated hosts
        return action ← PrivilegeEscalate host
for address in known addresses do                       ▷ Access unexploited hosts
    if address not in exploited addresses then
        add address to exploited addresses
        return action ← ExploitRemoteService address
```

Algorithm 3 B-line agent get_action()

```
action_count ← 0
jumps ← [0,1,2,2,2,2,5,5,5,5,9,9,9,12,13]
if last action was successful then
    if action_count < 14 then
        action_count += 1                ▷ Move to next action if previous succeeded
else
    action ← jumps[action]               ▷ Jump to relevant action if previous failed
action 0 : DiscoverRemoteSystems on user subnet
action 1 : DiscoverNetworkServices on random user host                ▷ 1,2,3 or 4
action 2 : ExploitRemoteService on same user host
action 3 : PrivilegeEscalate on same user host
action 4 : DiscoverNetworkServices on newly discovered enterprise host    ▷ 0 or 1
action 5 : ExploitRemoteService on same enterprise host
action 6 : PrivilegeEscalate on same enterprise host
action 7 : DiscoverRemoteSystems on newly discovered subnet
action 8 : DiscoverNetworkServices on enterprise 2 host
action 9 : ExploitRemoteService on enterprise 2 host
action 10 : PrivilegeEscalate on enterprise 2 host
action 11 : DiscoverNetworkServices on discovered target Op_Server0 host
action 12 : ExploitRemoteService on target Op_Server0 host
action 13 : PrivilegeEscalate on target Op_Server0 host
action 14 : Impact Op_Server0
return action corresponding to action_count
```