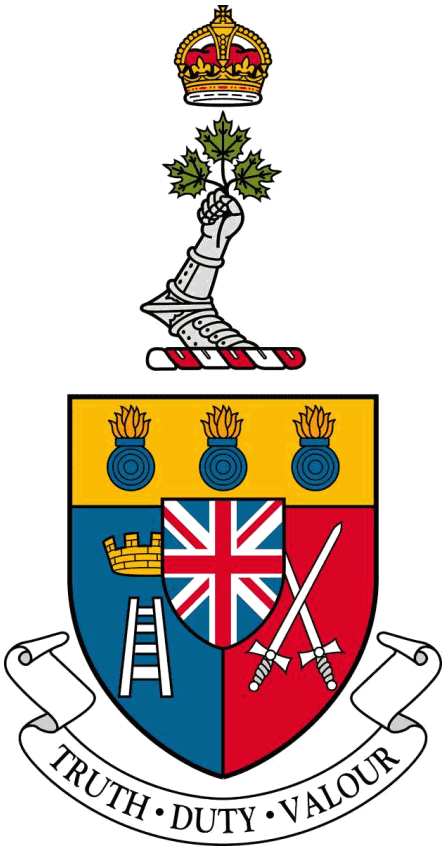
Competitive Reinforcement Learning for Autonomous Cyber Operations



A Thesis Submitted to the Royal Military College of Canada
Department of Electrical and Computer Engineering
by

Garrett McDonald

In Partial Fulfillment of the Requirement for the Degree of
Master of Applied Science in Computer Engineering

1 May 2023

Acknowledgments

I am deeply grateful for the help and support I received to make this thesis possible. I was very fortunate to have Dr. Ranwa Al Mallah as my committed supervisor, who provided guidance throughout this project. And I am especially grateful to Anna, for her constant support and understanding during the long hours that went into completing this thesis!

Abstract

Reinforcement learning (RL) has been responsible for some of the most impressive advances in the field of Artificial Intelligence (AI). RL has benefited substantially from the emergence of deep neural networks that enable learning agents to approximate optimal behavior in increasingly complex environments. In particular, research in competitive RL has shown that multiple agents competing in an adversarial environment can learn simultaneously to discover their optimal decision-making policies.

In recent years, competitive RL algorithms have been used to train performant AI for a variety of games and optimization problems. Understanding the fundamental algorithms that train these AI models is essential for using these tools to address real-world challenges. Cybersecurity is a domain where the emerging research in competitive RL is being considered for its real-world application.

In order to develop Automated Cyber Operations (ACO) tools using RL, various environments are available to simulate network security incidents. Many of these ACO environments have been made open-source in just the past three years. These new environments have facilitated promising research exploring the potential of AI for cybersecurity. The existing research in these environments is typically one-sided: a red or blue agent is trained to optimize their decision-making against a static opponent with a fixed policy.

By training against just one opponent, or a static set of opponents, the learning AI will not maintain high performance against every other possible opponent in the scenario. Competitive RL can be used to discover the optimal decision-making policies against any potential opponent in an adversarial environment. However, it has not been attempted in these emerging ACO simulations. The aim of this thesis is to train agents using competitive RL to approximate their game theory optimal policies in a simulated ACO environment.

Contents

List of Figures

List of Tables

Glossary

Agent – A decision-maker in an environment. A participant in a Markov Decision Process or Markov Game.

Batch – A collection of experiences gathered by an agent interacting with their environment. The sample experiences in a batch are used to update the agent's policy.

Best-Response – Describes an opponent policy that, in all cases, correctly selects the action that will minimize the return of an agent's policy. This is the worst-case opponent for an agent policy that is being examined.

Blue Agent – A decision-maker taking actions to maintain the confidentiality, integrity, and accessibility of network assets in an ACO environment. The defender during a network security incident.

Experience – An agent's encounter with an environment that can be used for learning. One experience includes an observation, the action taken by the agent, their new observation, and their reward received. An experience is also known as a "sample" for training. Each new experience is stored in an agent's memory buffer until it is used to update the policy.

Exploitability – The difference between a policy's expected return against a best-response opponent, and the expected return that is achieved by following an optimal minmax policy. This metric is used to measure how suboptimal a policy is, by considering the worst-case opponent.

Fictitious Play – A category of competitive RL algorithms, where new agent policies are trained by simulating experiences against the average of many potential opponents. These training experiences are curated so that new agent and opponent policies gradually converge towards optimal play. The implementation of fictitious play used in this thesis is used to approximate optimal Red and Blue agent policies for an ACO environment.

Generation – A complete process to train a new agent policy. This includes creating a new randomized policy, then undergoing many iterations of training updates until the policy is no longer improving. One loop within the

fictitious play algorithm creates a new generation of Red and Blue policies. Each saved policy is numbered so that it is uniquely identified by the generation it was produced.

Iteration – A complete training update for a reinforcement learning algorithm. This includes gathering experience from the training environment and storing that experience in a memory buffer. Once a complete batch of sample experiences have been collected, the learning agent's policy is updated.

Minmax – Describes a policy that will guarantee the most value for an agent, regardless of what action is selected by their opponent. In other words, this describes the policy that will achieve maximum value, when assuming that the opponent will correctly select actions to minimize the value of any policy.

Nash Equilibrium – A state during training where neither learning agent has any incentive to deviate from their current policy, because both agent's have arrived at policies that are the best-responses to each other.

Observation – The perceived state of the environment at a timestep. The environment's state from an agent's point of view.

Opponent Sampling – A technique for selecting the opponent that will be used during training, in order to generate new experience for the learning agent. This technique allows the learning agent to gather experience against a wide variety of potential opponents during training and develop a policy that maximizes its average score across the entire opponent pool that is being sampled.

Policy – The process used by an agent to select an action in their environment. A mapping between observations in an environment and the agent's behavior for each observation. To create this mapping, this thesis uses deep neural networks that are trained to produce action probabilities from agent observations.

Red Agent – A decision-maker taking actions to compromise assets on a target network in an ACO environment. The attacker during a network security incident.

Return – The total amount of reward an agent receives during a sequence.

Reward – A scalar measurement of how desirable a state-action was for a learning agent.

Reward Function – The system used to determine the reward that an agent receives for any state-action. This should be carefully crafted so that the agent will achieve a target behavior by learning to receive the maximum reward.

Sequence – A chain of states and actions that describe an agent's participation in an environment.

Static – Describes an agent that is following a single fixed policy. A non-learning agent or opponent.

Strategic Cycle – An open-ended learning problem that can occur if both agents in a two-player Markov Game attempt single-agent RL at the same time. Neither can converge on the optimal policy because the optimal policy is constantly changing. Often, they will appear to cycles through the same progression of policies that are the best responses to each other.

Target Behavior – The priorities that a trained autonomous agent should adhere to. The intended performance for a new AI.

Timestep – One abstract unit of time in a simulated environment that is using discrete time. Also called a "turn" in a Markov Game.

Training Score – The average return that was achieved by an agent during a single batch of training.

Value – The amount of reward that an agent is likely to accumulate from a certain state or state-action in a sequence, by following a given policy. Also called the "expected return".

1. Introduction

The rapid evolution of modern cyber threats has forced many organizations to spend significant resources protecting their digital infrastructure. It is a common practice to secure digital assets on private networks, and establish permanent Cybersecurity Operations Centers (CSOCs) to monitor network activity and mitigate the impact of cyber attacks. The exact risk of such an attack is unique to every organization and is often difficult to measure. The perceived value of digital assets can motivate cyber threat actors for a variety of reasons. Regardless of each organization's unique priorities and structure, the risk of a cyber attack continues to grow over time, driven by the number of malicious actors in the wild and the sophistication of their attack tools.

The cybersecurity analysts that operate a CSOC have to extract useful information from a tremendous amount of data, and rely on a variety of different sources, as they attempt to spot unusual (and possible malicious) network activity. Fortunately, sophisticated network security tools are available to assist the analyst with this responsibility. Firewalls, Intrusion Detection/Prevention Systems (IDS/IPS) and Quality of Service (QoS) protocols are all examples of systems that automate the inspection and control of network traffic.

Unfortunately, these tools are not always enough to make the analyst's goal reasonable. The sheer volume of network traffic that passes through an IDS means that there will almost always be too many false positive alerts for the CSOC to examine every alert as a potential threat [1]. For this reason, automated tools that can assist the human analyst with extracting information from data will be vital to future CSOC operations. Abstracting tasks away from the analyst will allow them to focus on higher-level problem solving.

It is still an open question how much of the analyst's current responsibilities can realistically be automated [1]. Network security operations can involve sophisticated sequential decision-making to protect against threat-actors. In this thesis, competitive Reinforcement Learning (RL) is explored as a potential tool for developing Artificial Intelligence (AI) that can automate certain network security actions during a cyber attack.

1.1. Motivation

During a targeted cyber attack, attackers may conduct long periods of reconnaissance, learning as much as possible about the target network and its users. This can be done incrementally throughout an attack: as the actor gains access

to new files and systems, they might choose to gather newly available information before committing to a crucial tactic such as building persistence or lateral movement [2] [3]. This cycle of observing and taking action allows an attacker to carefully choose the quietest or most time-efficient techniques to establish a presence throughout a target network until they can achieve their objective.

Advances in the field of Reinforcement Learning (RL) have shown that autonomous agents are capable of complex decision-making processes across a wide variety of domains [4]. RL is distinct from other popular machine learning paradigms, such as supervised and unsupervised learning, because RL problems uniquely describe an agent moving through a sequential decision-making process. The goal of RL is to train an agent to make optimal decisions at every observable state in order to reach the best possible outcome for a sequence. RL has benefited substantially from the emergence of powerful neural networks that can facilitate the learning process of an autonomous agent. These neural networks have been used to create RL models with superhuman performance in a variety of games and optimization problems. Game playing systems such as AlphaZero and DeepStack are capable of learning complex strategy in competitive games, and finding creative solutions to beat human experts [5] [6].

It is unclear what role these AI will have in the future of cyber security. Network defence could be an especially interesting application for decision making AI, partially because cyber incidents can involve two categories of agents with competing objectives in the same environment. First, an autonomous Red agent could be trained to conduct lateral movement and attempt to establish a foothold on a specific target network. Second, an autonomous Blue agent could monitor the same data used by analysts today, possibly finding useful insights more quickly, in order to take simple mitigating actions on the target network during a cyberattack.

On the one hand, defeating AI enabled cyber offense is paramount. On the other hand, autonomous cyber defence leveraging AI capability needs to be established. Research into the potential of RL for cybersecurity application is still emerging. However, there has already been notable success in the development of expert systems that are capable of Autonomous Cyber Operations (ACO). One such example is the development of Xandra: an autonomous cyber agent capable of conducting sophisticated decision-making for cyber offense [7]. In fact, Xandra was just one of seven systems developed for the Defense Advanced Research Projects Agency (DARPA) Cyber Grand Challenge, a capture-the-flag style competition held in 2016. Teams were each given vulnerable software, and autonomous cyber agents were expected to patch their own code while simulta-

neously exploiting any unpatched vulnerabilities in their opponents.

Today, the state-of-the-art for offensive ACO is CALDERA, a framework developed by the MITRE corporation for autonomous adversary emulation. CALDERA performs autonomous Red agent emulation for developers to find their own system vulnerabilities [8] [9]. CALDERA is another expert system, whose attack patterns can be customized and based on the MITRE ATT&CK Framework.

The current state-of-the-art tools for ACO are almost exclusively expert systems. The biggest challenge when bringing the success of RL to the cyber domain, is the need to repeatedly simulate many scenarios relatively quickly in order to train an agent. Simulating a cyber security incident requires a tremendous level of detail in the environment. Emulation environments virtualize hosts and network equipment using real-world host images and other software. Simulation environments are separate applications that attempt to model the relevant details of network. Simulations are used to examine the outcome of a network security incident in a fraction of the time it would take on an emulated network. Examining scenarios in real-time on an emulated network would be too slow to generate the experience required for RL.

Various RL environments have been developed to simulate and emulate cyber incident scenarios, and many of these environments have been made open source in just the past two years [8] [10] [11] [12]. The current research done in these environments is typically one-sided: a Red or Blue agent is trained to optimize their decision-making against a static opponent. Single-agent RL for ACO has shown promising results, and it is a crucial first step.

However, when either a Red or Blue agent is trained against a static opponent, their learned behavior could be flawed: there is no reason to think that the static opponent is selecting optimal actions, so the learning agent will almost certainly learn to rely on their opponent's sub-optimal decision making. By training against a single opponent, the learning agent might develop a policy that is ineffective against a wider range of possible opponents. Any learned policy for an ACO scenario will need to be effective against any adversary, or else the trained AI could be vulnerable to other potential opponent policies.

A possible solution to this is to train both the Red and Blue agents in these environments simultaneously. This is a common practice in RL when multiple agents have competing objectives within the same environment. Training both agents simultaneously can allow each agent to exploit weaknesses in their opponent's strategy, until they both arrive at an equilibrium that approximates optimal decision-making for that specific scenario. This sub-domain of RL is known as competitive RL.

Competitive RL has resulted in superhuman performance in many other applications and classical games, but it has not yet been attempted in the emerging ACO simulation environments [4] [13]. It is important to note that success in an ACO environment today will not result a trained agent that can be used on a real-world network. Many of these environments are still in early development, and choose to abstract away certain realistic details to accommodate RL.

Demonstrating the success of competitive learning agents in a simulated ACO scenario will be a small but crucial step in the evolution of these environments. If competitive RL can be used to optimize the performance of these agents, their optimal behavior will give insights into how future simulations should prioritize features that will add realism to the simulation. So that eventually, competitive RL might be used to train agents for an emulated or real-world target network, and truly evaluate their potential as a cybersecurity tool.

1.2. Statement of Deficiency

There is a demand for automated tools that can alleviate the unrealistic workload that is currently placed on cyber analysts in a CSOC. ML has created powerful tools for many other domains that require sophisticated data processing and decision making, but the potential and limitations of ACO are not well understood.

Existing Red ACO is dominated by expert systems, like CALDERA, and most attempts at Blue ACO are still simulated and experimental. There has been limited research into the use of RL to develop ACO agents because this will require realistic simulation environments where a cyber attack scenario can be easily repeated for training. Emerging ACO environments have been used to successfully train autonomous agents using RL, but existing research typically trains either a Red or a Blue agent in a static environment. There is a lack of research exploring Red and Blue that learn their game theory optimal policies in these environments. Red and Blue agents should be trained using competitive RL in an ACO environment to see if the learning agents can discover optimal behavior in these conditions.

Finding near-optimal agent policies will be a crucial step for the evolution of ACO simulations. Once Red and Blue agents are performing optimally in an environment, their behavior can be examined to see if their actions would be intuitive for a human analyst. The difference between these optimized agents, and the expected human behavior, will give insight into how an ACO environment might be lacking in realism and suggest areas for future development.

1.3. Thesis Aim

The aim of this research is to determine if Red and Blue agents can be trained using competitive RL to approximate their game theory optimal policies in a simulated ACO environment.

This attempts to bring the success of competitive RL from classical games into the cyber domain. To achieve this, two opponents are trained using a competitive RL algorithm in a simulated environment, where the Red attacker and the Blue defender take turns selecting abstract cybersecurity actions in order to alter the network environment towards their competing objectives.

After these agents have finished training, their learned behavior is evaluated to measure how closely their policies approximate optimal play. An existing ACO simulation environment is used for this training and validation. However, modifications are made to the environment to facilitate competitive training.

This exploratory research hopes to contribute to the development of more advanced ACO environments, so that these simulations might eventually have the realism necessary to develop AI that can operate in emulated environments, and begin experimenting with ACO agents that are trained for real-world systems.

1.4. Research Activities

The following activities were conducted in achieve this aim. A more detailed description of the actions and decisions taken during each of these phases will be included in Chapter 4, with the thesis methodology:

1. Select an ACO Environment. An open-source ACO simulation environment was selected for this experiment. This environment was modified to support competitive play.

2. Identify the Target Behavior. Suitable scenarios were considered for the learning Red and Blue agents in the chosen environment. A single scenario was selected for the evaluation. This scenario includes the specific network topology, the list of actions available to either agent, and the observable information that is available to either agent. The target behavior is defined by each agent's priorities in this scenario.

3. Define the Game Design. The scenario was modelled as a game in the simulation environment. This is done so that each agent learns its intended behavior by solving the game theory optimal policy for the scenario.

4. Implement Fictitious Play. A new competitive RL algorithm was designed for this ACO environment. This fictitious play algorithm is based on similar implementations that have been used to find game theory optimal policies for classical games. This design leverages available open-source RL resources, and will be discussed in greater detail in the methodology. This algorithm was used to produce competitive Red and Blue agent policies for the established game design.

5. Validate the Trained Policies. The competitive policies were validated by measuring how closely they approximated game theory optimal play. These results provide a proof-of-concept that competitive RL can be used to discover optimal decision-making policies in ACO environments.

The remainder of this document will show how these activities proved that a competitive RL algorithm can produce policies that converge towards optimal play in an ACO environment. The experiment included in this thesis produces optimal policies for a simple scenario in a popular ACO simulation. These policies are validated by examining their guaranteed performance against a worst-case opponent, and it is shown that the guaranteed performance of policies improves with competitive training. Sample games are played by the optimized agents to reveal how each agent achieves optimal performance in the target scenario. In these sample games, the trained agents demonstrate interesting strategic behaviour, which includes adopting stochastic policies at key decision points.

1.5. Document Structure

The next chapter will present the key concepts of competitive RL that will be relevant throughout the research. The theory and vocabulary in this background will be used frequently throughout the remainder of the document. Chapter 3 will continue with an overview related work in the field. This chapter will examine existing ACO research and compare ACO environments that were considered for this thesis. Chapter 3 also highlights what Competitive RL has achieved in other applications, which heavily influenced the algorithm used in this research.

Chapter 4 will outline the thesis methodology by providing detailed descriptions for each of the research activities. A significant portion of the methodology discusses the algorithm that was designed for this experiment. Chapter 5 evaluates the algorithm by applying it to the selected environment and scenario. This includes a discussion of the simulation environment, the training results, and the validation used to measure agent performance. Chapter 6 will conclude the thesis by reviewing contributions and considerations for future research.

2. Background

This chapter presents the key concepts for competitive RL, with an emphasis on the topics that will be used in this thesis. This background is intended to establish a vocabulary for discussing the methodology and evaluation included in Chapters 4 and 5, and for discussing the related literature presented in Chapter 3.

2.1. Reinforcement Learning

Reinforcement Learning (RL) is distinct from other machine learning paradigms, such as supervised and unsupervised learning, because RL problems uniquely describe an agent moving through a sequential decision-making process. The goal of RL is to train an agent to make the optimal decision at every observable state in an environment, in order to achieve some desirable behavior in that environment or sequence. An agent's "policy" is its process for selecting an action based on its observed state.

2.1.1 Reward Functions

In order to train a policy via machine learning, the first step is to define the target behavior for that agent in the environment. RL algorithms are used to learn the optimal policy, but they require a scalar measurement to compare different states. This scalar metric, called the "reward", represents how desirable an agent's state is based on all observable information [14].

For example, in many classical games a reward of 1 can be given for any state where the agent has won the game, with a reward of -1 if the agent has lost, and 0 for every other state. This can be enough for RL algorithms to learn which actions in that game will navigate towards winning states and avoid losing states. Reward functions can also be much more complex, with different features of the states contributing different amounts of reward, or features that interact with each other for a complex evaluation of the state. Rewards can also be negative to describe undesirable state, but the reward always needs to be measured using a single scalar value.

The reward function for an agent must be carefully crafted to describe the priorities of the desired behavior for that agent. Higher priorities should have a greater influence on the reward, and lower priorities can have less impact on the reward, so that when the agent learns to maximize the reward it will be learning the target behavior. RL trains autonomous agents to converge on an optimal

decision-making policy by experiencing a sequence many times and learning from the results. The optimal decision is the action that will maximize the expected total reward for that sequence.

2.1.2 Markov Decision Processes

RL algorithms all rely on certain assumptions about the agent's environment, and these assumptions must be true for RL to be practical. Specifically, a scenario is only suitable for RL if it can be described as a Markov Decision Process (MDP). An MDP is a type of Markov Chain. Markov Chains can be described with the tuple ⟨S,P⟩ [15]. S is the state-space, a set of all possible states for an agent in the sequence. P is the state-transition probability matrix which describes the probabilities that an agent in the current state s will transition to each state in the state space. Figure 1 presents an example of a simple Markov Chain, showing each state with transition probabilities. In this example, state E represents a terminal state where the state no longer changes.

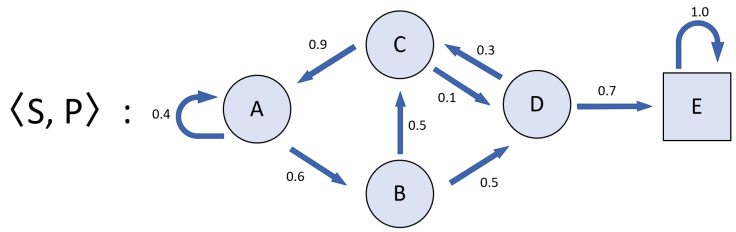


Figure 1: Simple Markov Chain.

Every state in this model must have the Markov Property: all relevant information for the state-transition probabilities must only depend on the current state s. This means that the history of past states for an agent in a Markov Chain has no effect on the transition probabilities for the current state. In other words, an agent's history in the sequence is irrelevant, given its current state. This relationship is shown in Equation 1.

$$\mathbb{P}[S_{t+1} \mid S_t] = \mathbb{P}[S_{t+1} \mid S_1, \ldots, S_t] \tag{1}$$

One possible extension to Markov Chains is the Markov Reward Process (MRP). An MRP is simply a Markov Chain that includes a reward function, as shown in Figure 2.

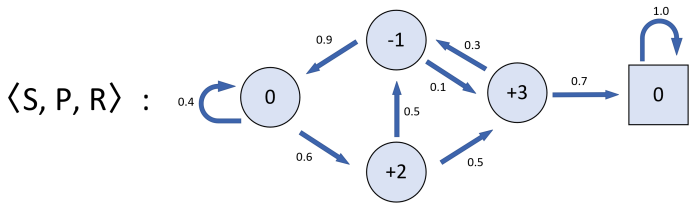


Figure 2: Simple Markov Reward Process.

S and P still define the state-space and state-transition probabilities. R defines the reward function, which measures how desirable a state is for an agent moving through an environment. The most desirable sequence for an agent moving through an MRP is the sequence that accumulates the maximum possible reward. This total accumulation of reward through all time steps in an MRP is called the "return" [14].

$$\mathbb{E}[G_t] = R_{t+1} + \gamma R_{t+2} + \cdots = \sum_{k=0}^{\infty} \gamma^k R_{t+k+1} \tag{2}$$

The expected return G at a timestep t is equal to the sum of all the rewards that will be received from all future timesteps. A discount factor γ can be used to represents the confidence that expected rewards will be available at future timesteps. The discount factor is a number between 0 and 1, and every future reward R is reduced by a power of γ to indicate a reduced certainty that the reward will still be available in the future. The relationship between the discount factor and the expected return is shown in Equation 2. This discount factor is just one example of a hyperparameter that can be used by an RL algorithm when evaluating states. These new parameters in the MRP can be used to calculate the value v of any state in the chain, as shown in Equation 3.

$$v(s) = \mathbb{E}[G_t \mid S_t = s] \tag{3}$$

Where the value of any state $s \in S$ is the expected return for an agent starting from that state. Finally, one more extension to the parameters of an MRP can describe an MDP. This extension is shown in Figure 3.

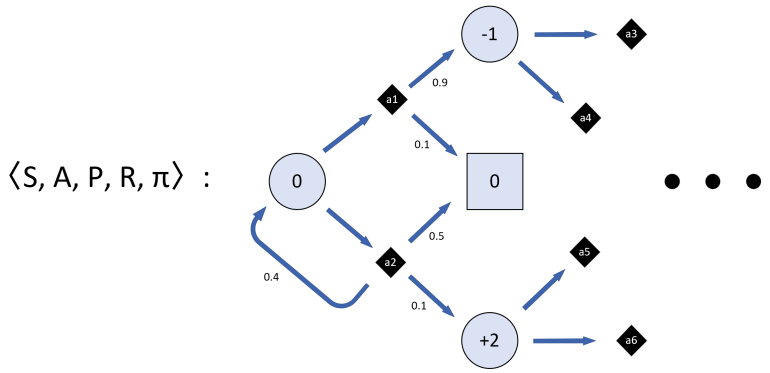


Figure 3: Simple Markov Decision Process.

An agent moving through an MDP can influence how it transitions between states. Agents have access to a set of actions A, and at each timestep they choose an action according to their policy π. In an MDP, the state-transition probabilities depend on the action selected by the agent. Therefore, the state-transition probability matrix P will also have a new dimension to describe the transition probabilities based on both the current state and selected action.

The agent's policy π determines what actions it will select in each state. In a simple MDP, such as in Figure 3, a reasonable policy would be to simply select actions greedily: always selecting the action with the highest value from the immediate future states.

But these values are not always obvious for real-world environments: the exact state-transition probabilities are often unknown, and it is not always obvious if a state is better or worse for long-term reward. For many applications, it is useful to have an AI agent that makes optimal decisions even when the transition probabilities and state-values are not obvious by examination. This is the objective of RL: to use an agent's past experiences in order to approximate the optimal policy of an MDP.

2.1.3 Tabular RL

Many of the first proposed RL algorithms find the optimal policy of an MDP by memorizing an approximate value for every possible state or state-action pair, by storing these values in a table. Algorithms that operate using these stored values are known as Tabular RL algorithms, and these traditional methods are the foundation for many modern algorithms [14].

For example, a tabular RL algorithm called Q-Learning is still used today for solving simple MDPs [16]. Like all RL algorithms, Q-Learning requires an agent to simulate the MDP many times. The value of every state-action pair, $q(s, a)$,

is stored in a table to track the expected return for every action in every state. In this model, the value of a state is equal to the highest q-value among available actions in that state. To update the q-table, each time the agent takes an action it compares the known q-value for $q(s, a)$ to the expected return in the new state. $q(s, a)$ is updated by a small step towards the expected return in the new state, using the q-values for available actions in that new state.

Since the q-values are updating using a mix of true rewards and estimated values, they gradually become more accurate approximations of the expected return for each action. If an agent can accurately approximate every q value, and select the action with the highest q value in every state, then it has converged on the optimal policy for that environment. This relationship is formalized in the Bellman optimality equation [14].

The Bellman equation, shown in Equation 4, is a recursive formula to demonstrate that selecting actions with the maximum expected return will maximize return over the entire sequence. Future states are shown as s' and the actions available in those states are shown as a'. Calculating the exact value of a state-action $q(s, a)$ requires the probabilities for every state transition and reward $p(s', r \mid s, a)$.

$$\begin{aligned} q(s, a) &= \mathbb{E}[R_{t+1} + \gamma \max_{a'} q(s', a')] \\ &= \sum_{s', r} p(s', r \mid s, a)[r + \gamma \max_{a'} q(s', a')] \end{aligned} \tag{4}$$

The Bellman equation also indicates the importance of exploring an environment during RL. In order to accurately determine which future action a' will provide the max q value, each state-action will need to be visited. The different possible paths that an agent can take through a scenario are commonly called the "game tree".

How often to select the best available action during training is an important decision that can impact how quickly an agent converges on the optimal policy, or if it converges at all. This trade-off is commonly known as "exploration versus exploitation": will the policy converge more quickly if the agent explores a different branch of the game tree, or if it collects more experience using the state-action that appears to be best?

If the highest value action was selected in every state during training, then a learning agent could get stuck in a local maximum where it is never exploring to discover an even higher q-value. This is why the Q-Learning algorithm maintains an exploration parameter ε for how often it should deviate from selecting the optimal action. If ε is 0.1, then there is a 10% chance during training that the

agent will select a different random action in each state.

Exploration should be tuned for each RL problem and environment, because the amount of exploration required will depend on the complexity of the game tree. When considering exploration requirements, it is common to organize scenarios into three types of game trees depending on their "horizon" [14]. Finite-horizon scenarios end after a fixed number of timesteps. Indefinite-horizon tasks can take variable amount of time, and visit a varied number of states, but must eventually terminate. Finally, infinite-horizon tasks describe environments where the scenario never terminates, and the game tree branches out infinitely.

Other tabular RL algorithms, such as SARSA and Monte-Carlo Control, use a similar process of storing q-values in a table [14]. Most of the differences between these traditional algorithms are in the nuance of how q-values are updated. Algorithms can be on-policy or off-policy, and they can apply varied degrees of Temporal-Difference (TD) learning.

Q-Learning is an off-policy algorithm. It considers the value of a state to be equal to the highest q-value among actions available at that state, regardless of which action was actually taken during training. This is different from on-policy algorithms like SARSA. On-policy algorithms maintain a probability for selecting each action in a state, and update prior q-values using the action that was actually selected, regardless of whether it had the maximum value.

Q-Learning is also a TD1 algorithm because it uses the estimated q-values just one state away to perform updates [14]. In other words, q-values are updated using the measured reward and the expected return from new states just one-step away. This is different from algorithms like Monte-Carlo Control, where the q-value for each state-action encountered during a sequence is updated based on the total return at the end of that sequence. Monte-Carlo Control does not rely on any other q-value approximations for updating.

Many of these tabular RL algorithms are excellent for small scale problems where it is reasonable to store a value for every possible state-action pair in a table. The drawback of these solutions is that many realistic applications, even simple applications, have many more state-action pairs than can reasonably be stored in a table. For example, a classical game like backgammon has 10^{20} possible states [14]. Many real-world problems can be modeled as MDPs, but cannot be optimized by traditional RL because of this constraint.

For the aim of this thesis, tabular methods will be insufficient because it is not feasible to store the value of every possible state-action in a realistic ACO environment. However, these tabular methods represent the earliest implementations of RL, and they are the foundation of many modern RL algorithms that

succeed by learning to approximate these tabular methods.

2.1.4 Deep Reinforcement Learning

Neural Networks can be used to apply the principles of RL without storing an infeasibly large table of state-action values for complex MDPs. Deep Neural Networks (DNNs) are exceptional tools for approximating complex functions. This can include training a DNN to predict the value of a state-action pair in an MDP. Training a DNN to approximate a value or policy for an RL agent is known as Deep Reinforcement Learning (DRL).

In DRL, an agent generates its own training data by interacting with the environment. Every decision in the MDP represents a state-action pair that can be used as a unique sample, and an agent can generate the samples required to train a DNN by simulating the MDP many times. The exact purpose of a DNN used for RL depends on the algorithm. The two most popular classes of DNNs used in RL are Value Prediction Networks and Policy Prediction Networks.

Value Prediction Networks use the agent's experience in an MDP to predict the value of a state or state-action pair. The output of this network is a single scalar number predicting the expected return from the MDP. The loss function $L_i(w_i)$ for a state-action Value Prediction Network can look reminiscent of the Bellman optimality equation, such as in Equation 5 where the network weights are represented by w.

$$L_i(w_i) = (r + \gamma \max_{a'} q(s', a'; w_i) - q(s, a; w_i))^2 \tag{5}$$

It is worth noting that this loss function itself relies on q value predictions. These values can be approximated using the same DNN that is being trained and the network will still converge towards an accurate approximation.

Policy Prediction Networks also use the state features as an input, but this input tensor does not include an action. These networks do not output a single scalar value. Instead, they use a softmax activation function on their output layer in order to generate a policy for available actions in the current state. The output vector represents the probability that each action should be selected from that state.

Policy-based learning can have high variance, especially when one sequence has many timesteps. This makes learning slow because the same policy can have dramatically different results depending on random factors in the environment

during each sequence. This is the same drawback as that of the tabular RL algorithm for Monte-Carlo Control, which does not use TD learning. However, policy networks do have certain advantages compared to value-based networks. Because they use an output vector, they are capable of learning stochastic policies, instead of deterministic policies that will greedily select the action with the greatest expected return [14]. As well, policy networks can be more performant in partially-observed environments, where the agent may not have access to all the state features.

One way to increase the learning efficiency for these policy networks is to introduce TD learning by using an actor-critic framework [14]. Actor-critic algorithms use both a policy-based DNN (actor) and a value-based DNN (critic). The actor-network is still ultimately responsible for producing a policy for each state. However, by training a critic-network simultaneously, the actor-network can now be updated after every timestep by using value-predictions, rather than waiting until the end of the sequence to train using the total accumulated reward. This framework can dramatically reduce the time it takes for the actor-network to approximate the optimal policy, while still maintaining the benefits of policy-based learning [14].

Algorithm 1 presents a generic example of an actor-critic framework [14]. At each timestep, the policy network π is used to select an action a. The critic-network q approximates the expected return for this action. In this algorithm, the weights of the policy network are represented with θ and the weights of the critic are represented as w. After transitioning to the new state s', the critic-network also approximates w, the expected return at the new state. The difference δ is measured between these two predictions. Since the new state includes a measured reward, and it is closer to the end of the sequence, the critic-network weights are trained to reduce the error of the prior prediction. Finally, the actor-network weights are also adjusted using δ in order to increase the probability of selecting actions that lead to greater expected return in the future.

One widely used class of actor-critic frameworks are Proximal Policy Optimization (PPO) algorithms, which were introduced by OpenAI in 2017 [17]. PPO algorithms operate by sampling experiences from the environment until a full batch of experiences has been collected. The size of one batch is a hyperparameter that is tuned for each application. The full batch is then split into minibatches, and several epochs of Stochastic Gradient Decent (SGD) are used to update the actor and critic networks for each minibatch. The new updated policy is used to gather the next batch of samples from the environment. This process repeats until the networks are no longer changing because the actor is accurately

Algorithm 1 Generic One-Step Actor-Critic.

Initialize policy-based actor network $\pi(s;\theta)$ and value-based critic network $q(s,a;w)$
while within computational budget **do**
  Initialize s (first state of the sequence)
  while s is not a terminal state **do**
    $a \leftarrow \pi(s;\theta)$
    $s', r \leftarrow$ take action a
    $\delta \leftarrow r + \gamma \max_{a'} q(s',a';w) - q(s,a;w)$
    $w \leftarrow$ adjust critic weights according to δ
    $\theta \leftarrow$ adjust actor weights according to δ
    $s \leftarrow s'$
  end while
end while
Return π

approximating an optimal policy, and the critic is accurately approximating the return from each state.

What distinguishes PPO algorithms from other actor-critic frameworks is the inclusion of clipped surrogate objectives [17]. Prior actor-critic frameworks had the possibility for excessively large policy updates if that update was heavily supported by a minibatch. These large policy updates might be supported by a critic network that is not yet accurate, and large policy updates can limit the ability for SGD to converge in certain environments.

Clipped surrogate objectives set bounds for the loss function used by SGD. This forces the actor network to update using small step sizes, because its loss function cannot recognize an extreme difference in advantage for a new policy. Instead, it will update the policy in the correct direction based on a clipped value, that still recognizes the advantage, but never in excess. This clipping parameter can improve the convergence of actor-critic frameworks in many environments.

Since PPO algorithms were introduced in 2017, many open-source resources for RL have included PPO implementations. In Chapter 4, one of these single-agent RL implementations will be used for the experiment included in this thesis.

2.2. Competitive Reinforcement Learning

The MDPs and algorithms discussed thus far have described a single learning agent in an environment. The mechanics of RL change when there are multiple agents operating in the same space. This is known as Multi-Agent Reinforcement Learning (MARL) [4]. Many single-agent algorithms are non-viable if multiple agents are influencing the state transition and reward function in the same envi-

ronment. MARL requires its own set of algorithms to overcome these challenges. MARL algorithms can be loosely grouped into three different settings: cooperative, competitive and mixed [4].

Coorperative MARL techniques are for agents working collaboratively, so that they can learn a desired behavior by sharing the reward. Competitive MARL is used when agents compete for a reward in the same environment, and a positive reward for one agent can mean a negative reward for another. Mixed settings require considerations from both Cooperative and Competitive MARL, such as teams of agents that are each trying to maximize the shared reward for their team in an environment. This thesis is focused specifically on a competitive MARL setting (simply called Competitive RL for the remainder of this document), with exactly two competing agents, so that is the only paradigm that will be discussed further.

2.2.1 Markov Games

Competitive RL is used for a specific type of MDP called a Markov Game [15]. A Markov Game is an MDP where multiple agents are selecting actions at the same time in each state, so the state transition depends on their combination of actions instead of just the one action taken by a single agent. Markov Games can be defined by the tuple (N, A, r) where N is a finite set of n players, $A = A_1 \times \cdots \times A_n$ where A_i is a finite set of actions available to player i, and $r = (r_1, \ldots, r_n)$ is a reward function for each player [15].

Competitive RL describes exactly two opposing agents, so in these applications $n = 2$. It is common in competitive Markov Games that these two agents have opposing objectives, and so their reward functions are exactly opposite. If every combination of actions $a \in A_1 \times A_2$ results in equal and opposite rewards for the players $r_1(a) + r_2(a) = 0$ then this is known as a Zero-Sum Game [15].

2.2.2 Non-Stationary Environments

Competitive games invalidate some of the assumptions used by traditional RL algorithms. First, single-agent RL assumes that every state has the Markov Property, which includes having stationary transition conditions. In other words, the state transition should depend solely on the learning agent's current state and action. This must be true in order for the agent to discover the value of each state-action and converge on optimal behavior.

This assumption is not true in a competitive setting because any state transitions will also depend on the actions of the opponent. When the state-transition probabilities are not constant, this is known as a non-stationary environment.

Recall that for single-agent RL, the optimal policy selects the state-actions that will accumulate the greatest total reward in a sequence. However, in these non-stationary environments, there may not be one single policy that guarantees maximum return against every opponent. The value of actions might be different depending on which action is selected simultaneously by the opponent.

Let's consider a scenario where learning could still be attempted by using single-agent RL algorithms in these non-stationary environments. For example, a Red and Blue agent could simply attempt to learn using PPO simultaneously. In this scenario, both agents would learn to take the actions that are most valuable against their opponent's current policy. However, in many environments, neither agent would ever converge on a single target policy, and would instead arrive at a "strategic cycle" [18]. A strategic cycle describes an open-ended learning problem, where the agents are constantly updating to exploit the current opponent, but in doing so are also making their own policy more exploitable to different potential opponents. Sun et al. summarizes this phenomena with an example:

> "There exist cases that independent RL is reported to be effective for MARL, but in many other applications it leads to poor results. In particular, it suffers [from] policy-forgetting during training when the policy space is rich and contains circulation. An example is the game Rock-Paper-Scissor. A naive independent RL will circulate over pure-rock, pure-paper, pure-scissor, [such] that the late policy (e.g., pure-scissor) forgets how to beat the early policy (e.g., pure-rock). From the perspective of Game Theory, the "gradient field" of independent RL rotates over (but never converges to) an optimal point."[19]

In many adversarial environments, attempting single-agent RL will cause the learning agent to cycle through the same sequence of strategies, based on whatever opponent it happens to be training against. In these cases, the optimal policy for single-agent RL becomes a moving target and the agents can never converge. A competitive algorithm is needed to ensure that agents are updating to become more effective against any possible opponent.

2.2.3 Nash Equilibrium

Competitive RL algorithms are specifically constructed to avoid these strategic cycles. Instead of learning to defeat the current opponent, most competitive RL algorithms target the optimal minmax policy [15]. A minmax policy means that the learning agent is selecting actions that will minimize their opponent's maximum return across every possible opponent action.

In a zero-sum game, this is the same as learning to select the action that will maximize the agent's own return, regardless of the opponent's action. In practice, this policy assumes that the opponent is going to use the optimal response to every action. By assuming the opponent's response for every state-action, the environment can be treated as stationary again. Therefore, RL can be attempted in the competitive environment when the optimal minmax policy is the target.

Targeting the optimal minmax policy has important implications for the agent's learned behavior. Recall that a reward function should be constructed to represent an agent's priorities using a scalar metric. By choosing to converge on the minmax policy, the agent will learn very conservative behavior, choosing to prioritize a small guaranteed return over any larger return that would only work against certain opponents. The reward function for a competitive environment must be constructed so that this learned conservative behavior matches the target behavior of the AI agent.

By assuming the opponent will always choose their best-response, the issue of non-stationarity resolved. However, in order to assume the opponent selects the optimal response to every state-action, the opponent's optimal response must be known. For any novel application of competitive RL, the opponent's optimal policy will almost certainly be unknown. Therefore, in order to train an agent to their minmax policy, the opponent must be trained to their optimal policy as well. Competitive RL algorithms are processes for training both the agent and the opponent simultaneously in the environment.

Two competing learning agents in the same environment should learn to maximize their own reward by exploiting any sub-optimal behavior in their opponent's policy. In response, the opponent should learn that the behaviour was suboptimal because of the lower accumulated reward, and adjust their policy for future episodes. Unlike single-agent RL, these agents must update their own policies to improve the guaranteed return against any known opponent, instead learning against just one single arbitrary opponent.

This routine of exploiting and adapting will continue until neither agent has any incentive to change their policy any further, because any further changes could be exploitable by certain opponent responses, and learning will stop. This

occurs when both agents arrive at policies that are the best responses to each other [15]. When the policy of each agent in an environment is the best response to every other agent in that environment, this is known as a Nash Equilibrium.

Nash Equilibirum is defined by Equation 6, where v^i and π^i represent the value function and policy for an agent i, and π_* represents a policy at equilibrium, where for every state $s \in S$.

$$\begin{aligned} v^1(s, \pi^1_*, \pi^2_*) &\geq v^1(s, \pi^1, \pi^2_*) \\ v^2(s, \pi^1_*, \pi^2_*) &\geq v^2(s, \pi^1_*, \pi^2) \end{aligned} \tag{6}$$

2.2.4 Exploitability

In addition to revealing an optimal solution for the scenario, finding the Nash equilibrium also provides a useful metric for evaluating any alternative policies. The optimal minmax policy can be used as a baseline to evaluate all other policies. Any other sub-optimal policy can be measured by how far their expected return differs from the guaranteed expected return of the minmax policy.

The "exploitability" of any policy is the difference between the policy's expected return against a worst-case opponent, and the minmax policy's expected return against a worst-case opponent [20]. In other words, this is the difference in expected return that is guaranteed against any possible opponent, when compared to the minmax policy. This is shown in Equation 7, where π^i represents the policy used by agent i, π^i_* represents their minmax policy for the game, and π^{-i}_* represents the opponent's optimal response. A concrete example of exploitability is included in Chapter 4: it is used to validate the experiments included in this thesis, in order to confirm that learned policies are gradually converging towards a Nash Equilibrium.

$$expl = \mathbb{E}[G \mid \pi^i_*, \pi^{-i}_*] - \mathbb{E}[G \mid \pi^i, \pi^{-i}_*] \tag{7}$$

2.2.5 Competitive Algorithms

The optimal minmax policy will not always be the highest value action for every state. Competitive algorithms can use value-based or policy-based RL. value-based methods develop AI agents that select actions greedily (selecting the actions with the greatest expected value).

Greedy policies can be excellent in some settings, but in many Markov Games the optimal minmax policy will be stochastic, instead of deterministic, because a deterministic policy could be more easily exploited by an opponent [15] A more

optimal behavior might include a probability distribution across a set of actions for certain states, which would result in a less predictable behavior.

Many of the ACO environments considered by this thesis model Markov Games where the Red and Blue agents take actions simultaneously, and therefore these agents could be less exploitable with stochastic policies. Only policy-based competitive RL methods will be discussed in greater detail in this chapter. These algorithms are more suited to ACO environments, since it is possible that there will be no optimal deterministic policy. There are two dominant categories of competitive policy-based algorithms that can be used to reach a Nash equilibrium: Counterfactual Regret Minimization (CFR) and Fictitious Play [4].

CFR algorithms find the Nash Equilibrium of competitive games by learning to minimize the "regret" at every possible game state. The regret for a timestep is the difference between the value that was received and the value that could have been received (in hindsight) if the optimal action had been selected. This regret is calculated after an episode is complete, and the true state of the environment for the entire episode is known. Counterfactual Regret is a metric used in partially observed environments to measure the regret for an information state of the game [21].

The information state includes both the agent's own observation, as well as the information that has been communicated to their opponent. For example, modelling a scenario using information states can be crucial for partially observed games such as Poker [6]. In Poker, an agent and their opponent both rely on a mixture of private and public information. An agent takes actions to try and maximize its own reward, but every action taken reveals new information to their opponent. As a result, a learning agent must optimize to the most valuable information states. These are the states that maximize reward while not revealing enough information to the opponent such that the opponent could effectively interfere.

CFR algorithms are useful for environments where agents must rely on recursive reasoning: where each agent must consider the reasoning used by their opponent in order to construct an accurate approximation of their opponent's private information. The ACO environment examined in this thesis is partially observed, but the actions taken by agents do not reveal any new private information. As a result, recursive reasoning and CFR are not required. The other category of policy-based competitive algorithms, Fictitious Play, is more appropriate for this thesis and will be discussed in more detail.

2.2.6 Fictitious Play

The foundational algorithm for Fictitious Play was first proposed in 1951, and like many RL algorithms, this game theory proof for fictitious play existed for a long time before modern computing power made it feasible as a solution for problems [22]. A machine learning implementation of this algorithm called Fictitious Self-Play (FSP) was proposed in 2015, which was demonstrated using a tabular RL method called Fitted Q Iteration (a variant of Q-Learning) [23]. The following year, the same authors proposed Neural FSP (NFSP) as a framework to augment FSP using DRL [24].

During Fictitious Play, each learning agent generates samples by playing against the "average policy" of their opponent. This average policy is an approximation that combines all of the opponent's previous policies seen so-far. Experience is generated against the average policy, instead of the opponent's current policy, in order to avoid strategic cycles. During a single iteration of the algorithm, each player uses single-agent RL to discover the optimal policy that responds to the average opponent response. That new policy is then also used to update the player's own average policy, so that the opponent can learn.

Algorithm 2 Summarized Fictitious Play

$\Gamma \leftarrow$ initialize training environment
$\pi_0 \leftarrow$ set random initial average policies for generation 0
$g = 0$
while within computational budget **do**
  $g \leftarrow g + 1$
  $\pi_g \leftarrow \pi_{g-1}$
  for each player i **do**
    $\beta_g^i \leftarrow$ set random initial best-response policy for player i generation g
    while β_g^i is improving **do**
      $\beta_g^i \leftarrow$ update using RL in environment Γ with opponent π_g^{-i}
    end while
    $\pi_g^i \leftarrow$ update average policy using new best-response β_g^i
  end for
end while
Return π_g

A summarized NFSP algorithm is shown in Algorithm 2. Two policy networks are maintained for each player. The best-response network β is trained using RL to maximize return against the opponent, this network is constantly updating towards the optimal response to the opponent's policy. The average-response policy network π is trained using supervised learning to predict the player's own policy, based on past experiences. Every generation g, a new best-response is

calculated and the average-response is updated.

Over the course of training, each player's best-response policy β learns to optimize against their opponent's average-response policy π. The changes to β should generally become smaller each iteration, and π will gradually become more similar to β, as the average policies of either agent become gradually less exploitable. After enough iterations, their average strategy profiles π should converge to a Nash Equilibrium. At a perfect equilibrium, each agent's current response will match their best-response, and neither agent will have an incentive to deviate from their current policies, so learning is stopped.

If the agents have the same action and observation spaces, this describes a symmetric game. For symmetric games, only one agent policy needs to be trained. The best-response policy can be trained against the agent's own average policy, because the learning agent and the opponent are solving the same environment for the same optimal policy. This is not the case for ACO environments, where the Red and Blue agents have different observation and action spaces. This means ACO environments are asymmetric, and the new policies for either agent must be generated separately.

In the seminal paper on fictitious play, the average policy is approximated by using supervised learning [23]. This involves generating samples by mixing between the best-response policy and the average policy, and using this to update the new average policy. However, for many complex environments, developing one neural network to approximate the mixture of all past policies can be ineffective.

One alternative to approximating the opponent's average response with supervising learning, is to save every past opponent policy and train against the pool of opponents. This technique is called opponent sampling [19]. When using opponent sampling, each new best-response policy trains by selecting an opponent from the pool of previously trained opponent policies. When training is completed, the new best-response policy is added to its own pool and used to train future opponents. Figure 4 shows the relationships between the many systems used during fictitious play with opponent sampling.

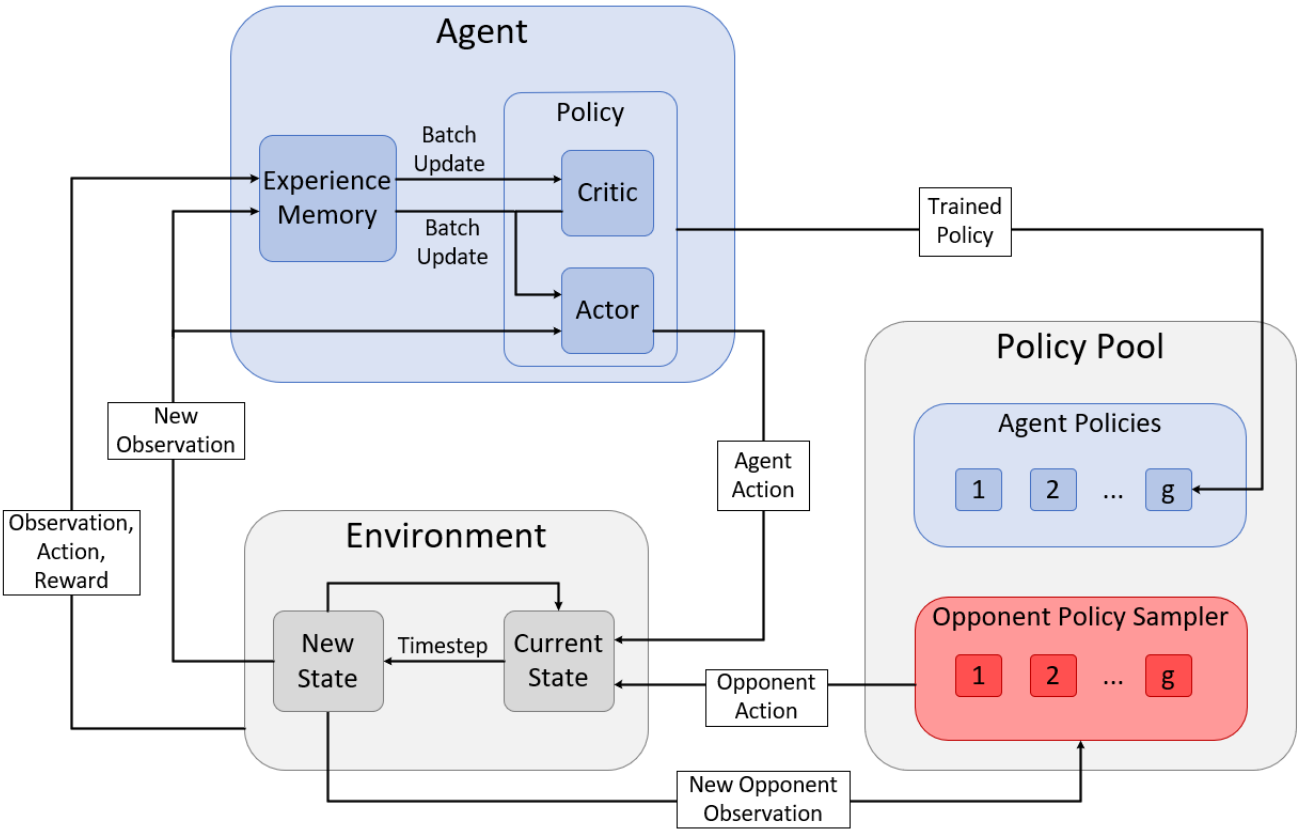


Figure 4: Overview of systems used during fictitious play, including an actor-critic framework and opponent sampling. Switch the agent and the opponent to train new policies for the opponent's policy pool.

To train a new policy, a single-agent RL algorithm such as PPO must be used. One iteration of PPO collects a batch of experiences by interacting with the agent's environment. Each experience includes the agent's observation, the selected action, the reward received, and the new observation. These sample experiences are stored in the agent's experience memory. Once a complete batch of experiences have been gathered, the actor and critic networks are updated to minimize loss on that batch of experiences. Finally, the experience memory is emptied so a new iteration can begin with the updated policy.

Generating new sample experiences requires an opponent that can select an action, in order to advance the environment to the next timestep. Opponent sampling is used so that games in the environment are played against a variety of different opponents. This way, the PPO algorithm is inherently optimizing the new policy for the best average performance against the opponent pool.

Once a policy is no longer improving (the average return seen in the experience batches has stagnated), the newly trained agent policy is saved to that

agent's policy pool. This trained policy includes the actor and the critic neural networks. These networks produce value predictions and action probabilities for an observation (note that only the actor network is actually required to produce action probabilities from a trained policy). Each generation of fictitious play trains a new agent policy and then a new opponent policy using this system. Both the agent and the opponent sample from each-other's policy pools during the fictitious play loop.

Various methods for sampling opponents from the opponent pool have been used successfully [19]. Uniform Sampling is the most-straightforward, where opponents are randomly selected from the existing pool. Uniform Sampling provides effectively the same result as training against an average policy. Each new generation is optimized against the entire existing pool of opponents, and then adds that response to their own policy pool. As the existing pools get larger, the emerging policies will become similar each generation, and the new generations will more closely resemble their own policies pools. When the new generations for all game playing agents are arriving at the sam policies each generation, this indicates that the agents have settled to a Nash Equilibrium, because neither agent is changing its policy any further as a response to the opponent.

One limitation of uniform sampling is that this could take an unreasonably long time for complex games. Each new generation contributes to smaller percent of its own policy pool with each generation. Therefore, it takes gradually longer for the new opponents to react to a flaw in emerging best-response policies. Whether or not uniform sampling can be used to optimize a Markov Game in a reasonable computation time will depend on the game complexity.

Other sampling techniques have been proposed to address this limitation [19]. Historic Sampling uses a higher probability for more recent opponent policies and a lower probability for sampling earlier policies. In some environments this can improve performance since the samples collected to train a new generation are skewed, such that the learning agent will collect more experience against more optimized opponents.

Quality-Based Sampling is similarly used to force the learning agent to sample from more optimized opponents [25]. However, instead of assuming that the more recent opponents are the most optimized, quality-based sampling maintains a quality rating for every policy in the pool. The measured quality of each policy in the pool is used to determine its sample rate. The quality rating of opponent's changes based on the opponents they win or lose against during training.

Sophisticated sampling solutions can be designed and tuned to optimize in specific environments. For example, OpenAI Five used a mix of historic and qual-

ity based sampling to optimize the game DOTA2 using fictitious play [26]. Their design used a mix of historic and quality-based sampling. 80% of the learning agent's games were played against the most recent opponent, and the remaining 20% were played against random past opponents that were sampled based on quality. Uniform sampling will be sufficient for the competitive scenario examined in this thesis, but an alternative sampling technique could be beneficial, or even essential, to implement fictitious play in more complex ACO environments.

3. Related Work

In order to meet the aim of this thesis, presented in Chapter 1, an experiment must demonstrate that a competitive RL algorithm can be designed for an ACO simulation to approximate the optimal policies in that environment. Chapter 2 discussed some of the most relevant vocabulary and concepts for constructing the experiment. However, it is still not obvious how to design competitive RL for an ACO environment from this background alone. This is partially because there are a variety of competitive RL algorithms that have been used for different applications, and because there are a variety of different simulation frameworks that model network security scenarios.

Any exploratory research that applies one of these algorithms to a novel environment must leverage existing research to select the most compelling algorithm and environment. For this experiment, an open-source simulation environment must be used, that is both relevant to ongoing research and suitable for competitive training. As well, a competitive RL algorithm must be designed for that environment based on successful implementations for similar game-designs. To make these design decisions, three distinct domains of research were reviewed while preparing the thesis methodology.

First, a review of existing ACO research reveals how RL has been already been applied to a variety of cybersecurity applications. This is done both to verify that competitive RL in a cyber incident simulation environment would be novel exploratory research, but also ensure that it would address an open question and contribute to the domain.

Second, a single environment must be selected to implement competitive RL. This necessitates a review of available simulation environments, and careful consideration about what would be a good candidate for this experiment. Every environment relies on trade-offs between realism and accessibility. An ideal candidate for this experiment should be accessible so that a new competitive RL algorithm can be built to interact with the environment, and the learned policies can be carefully examined. However, it should still be a realistic model of a real-world problem so that the research stays relevant to the domain.

Finally, a review of existing literature on successful competitive RL applications is needed. A significant portion of this thesis involves designing an algorithm for the chosen ACO environment, so it is vital that this algorithm is supported with evidence that it has succeeded in similar game designs.

3.1. ACO Research

"Cyber operations" can describe a wide breadth of activities related to network security. For this thesis, prior research that is related to automating the lateral movement of an attacker on a network, or the automated defence of that network during an attack is considered. Additionally, research that tries to prove the game theory optimal decisions in these scenarios is most relevant.

In the past, this type of ACO has been dominated by expert systems like Xandra and CALDERA, introduced in Chapter 1, but the aim of this thesis is to explore RL solutions. In 2021, a survey of existing research summarized how deep RL is being applied for various ACO applications across the domain [27]. They found that there are three main categories of research:

1. Automated Solutions for Cyber-Physical System Security
2. Automated Intrusion Detection Techniques
3. RL based Game Theory for Cyber Security

This thesis aims to contribute to the third category. Among the various RL applications in this space, just one paper from 2017 has also examined Red and Blue agents learning simultaneously in an adversarial environment through RL [28]. This research also aimed to explore RL as a tool for learning optimal policies in an adversarial environment, but their results do not meet the aim of this thesis for two key reasons.

The first reason, is that this research was severely limited by a lack of realistic simulation environments. A collection of open-source environments will be discussed next in Section 3.2, and these were all developed in just the past few years to support ACO research. In [28], to explore Red and Blue learning in an adversarial setting, they created their own simulation environment for the experiment. This "simulation" was much more abstract than the environments used in research today. It described four internet nodes where a Blue agent could allocate "defense value" while a Red agent allocated "attack value" to try and overpower the Blue defense and gain a foothold on a node.

This gameplay was entirely about economy of effort and how to prioritize "attack" and "defence" concepts. There is no modeling of exploits, vulnerabilities, host data, services, processes, or any other information that would help describe the actions taken during a cyber attack. Of course, this was simply beyond the scope of their research, and creating a detailed environment from scratch would have been an unrealistic body of work. Their aim was simply to evaluate different RL algorithms in their own adversarial environment. They succeeded in this aim,

but the simulation they describe is entirely different than the environments used for ACO research today.

The second reason this research is not sufficient, is that it does not actually attempt competitive RL. The Red and Blue policies were trained using single-agent RL, learning simultaneously, where both agents updated their policies after each game played based on the new experience:

> "Both agents needed to use the algorithms to learn a strategy to win as many games as possible, and due to the competition the environment became highly non-stationary. The results showed that the neural networks were not able to handle with this very well, but the Monte Carlo and Q-learning algorithms were able to adapt to the changes in behavior of the opponent." [28]

Learning was conducted in a non-stationary environment. This approach may have been sufficient to compare results in their own simulation, but simultaneous learning in a non-stationary environment can lead to a strategic cycle, as described in Chapter 2, and this becomes more likely in a complex environment. What this paper describes as "adversarial learning" did not employ the competitive RL algorithms that are explored in this thesis.

Other research uses RL and game theory to discover the optimal policies for different adversarial cybersecurity settings. In Bland et al. competitive RL algorithms were used to find the optimal policies for game models of cross-site scripting and spear phising attacks [29]. This involved constructing game trees to represent possible states and realistic actions that would be available to the attacker and defender in example scenarios. Then, competitive RL algorithms were used to discover the optimal policies for either agent in these models. This is an interesting application of competitive RL to a cybersecurity setting, but the scenarios described in Bland et al. are entirely separate from the environments examined in this thesis, where a Red agent attempts lateral movement across a simulated target network.

He et al. describes another experiment where competitive RL was evaluated for a selection of cybersecurity applications [30]. This time, a jamming versus anti-jamming scenario, and cloud-resource allocation during an attack, were modelled as examples. Similar to Bland et al. this demonstrated competitive RL was viable for interesting cybersecurity problems, but uses simulations that are not reminiscent of the environments considered for this thesis.

These examples all attempt to explore game theory optimal behavior in a Red versus Blue scenario, but much more literature exists exploring either the Red or the Blue side of an application as a single-agent RL problem. Han et al. is a unique experiment that first trains a Blue agent to protect a simulated network, and afterwards trains a Red opponent to use adversarial training and learn the optimal ways to disrupt Blue learning [31]. This experiment was done using a network emulator called Mininet [32]. Single-agent RL was used, but because only one agent is learning at a time there is no issue of non-stationarity. First, the Blue agent learns to optimize its policy against a scripted Red opponent. Because the Red opponent follows a static policy, the Blue learner can employ a single-agent RL algorithm and treat the Red opponent as if it was part of the environment. After the Blue agent has converged on a policy, the "adversarial training" described in this research refers to a Red agent learning to tamper with the training samples used by the Blue agent. The Red agents learns to optimize attacks that disrupts these samples, such as flipping a fixed number of reward signals, in order to minimize Blue's ability to converge on an effective policy.

The first phase of this research, where a Blue defender is trained against a scripted Red attacker, appears often in the literature. Many of the environments that will be discussed in Section 3.2 include demonstrations of a Blue agent learning to defend against a set of static Red opponents in their publications. Similar research has evaluated single-agent RL for Red attackers. Maeda et al. attempts to train Red attackers for the post-exploitation phases of a cyberattack, which describes similar scenarios to those considered in this thesis [33]. However, this Red agent operates in the absence of a Blue defender, where their Red agent is simply learning to optimize its own actions in the environment.

All the literature considered so far has modeled adversarial scenarios using simultaneous turn games. One interesting alternative models a cybersecurity problem as a Stackelberg game. In Speicher et al. the Red and Blue agents influence the target network with an "attack budget" and "mitigation budget" [34]. Then, Stackelberg Planning, not RL, is used to solve for the game theory optimal policies.

The optimal Red policy here is the "critical attack path", meaning the deterministic policy that will maximize Red's chance of success, while a Blue agent converges on a dominant mitigation strategy. This game is meant to optimize the economy of effort for either agent, and uses abstract cybersecurity concepts as actions. Still, it represents a unique alternative for modeling a network security problem in a simulation environment, and then finding the optimal policies using game theory.

3.2. Simulated ACO Environments

Using RL to train an autonomous agent, for any application, will require an environment that can simulate scenarios in the environment many times while the agent is learning. Creating a sufficiently detailed simulation is perhaps the biggest challenge that limits the development of cybersecurity tools using RL. It is difficult to model the exact conditions of a target network during a cyber attack with a realistic level of detail. Most of the quality environments that do exist were published in just the last few years, and all of them still sacrifice certain elements of realism in order to accommodate RL.

RL is only appropriate for training agents to participate in a sequential decision-making process. A specific tactic of a cyber attack must be modeled as an MDP in order to train an AI to take actions using RL. For example, this thesis will focus on the lateral movement of an attacker expanding their foothold on a specific network, which can be modelled as an MDP. Ideally, a simulation would support both Red and Blue agents taking actions on a target network. A Red agent would represent attackers on the target network, and a Blue agent would represent an analyst working to prevent that attacker's progress. There has been some research into the application of single-agent RL for ACO, but as shown in Section 3.1 there is not yet published research exploring competitive RL in these environments.

One of the better known cyber gyms is Microsoft's CyberBattleSim (CBS), which was published in 2020 as an experimental research toolkit [10]. Like many of the environments in this review, CBS is built on the OpenAI Gym framework [35]. CBS is focused exclusively on training Red agents for the lateral movement phase of a cyber attack. Some constraints in this environment are more limiting than other alternatives: the scenarios are specifically for a windows enterprise environment. As well, the simulation does not model network traffic, and there are no public plans for this framework to be built into emulation.

The Framework for Advanced Reinforcement Learning for Autonomous Network Defense (FARLAND) was developed by the MITRE corporation in 2021. This simulation environment offers a detailed game, with both Red and Blue agent control. It is designed for a Software-Defined Network (SDN) scenario, so that Blue agent actions include network configuration options such as: migrating a service by deploying a new virtual host, isolate a host by changing packet processing rules, and creating honey networks [11]. FARLAND uses a detailed model of host and network activity, and it supports both simulation and emulation. However, FARLAND is designed for single-agent training of a Blue agent

only, using two static Red agent profiles. Only the Blue agent is RL-enabled.

CyGIL was another environment developed in 2021, at Defence Research and Development Canada [8]. CyGIL's first prototype uses an emulation environment, and only supports training a Red attacker. However, this environment was designed to eventually incorporate Red and Blue agents in competitive training. Although it is not suitable for competitive RL in its current state, because it does not support the simultaneous training of Red and Blue agents, CyGIL could be a potential environment for competitive simulation and emulation once these features are developed for a future version of the environment.

The Cyber Operations Research Gym (CybORG) was also published in 2021 by the Australian Defense Science and Technology Group (DSTG) [12]. This environment was made public as part of competition, called the Cyber Autonomy Gym for Experimentation (CAGE) Challenge, where teams competed to train the most effective Blue agent to defend a specific network during an attack [36]. CAGE Challenge 3 was completed in February 2023, and this was the first challenge to incorporate multi-agent play (multiple Blue defenders). Most importantly, this environment was also designed with the intent of supporting competitive play, and so the CAGE Challenge release can be easily modified to support training both Red and Blue agents simultaneously.

Finally, the Network Attack Simulator is another simulation built on the OpenAI Gym framework [37]. Unlike CybORG, this environment does model firewall rules and certain network traffic. However, this environment is exclusively used for training Red attackers, and does not include an adversarial component.

Many of these open-source environments were compared as part of a recent ACO review [13]. Published in February 2023, this paper acknowledges that there are very few ACO environments that are currently available for adversarial training. Interestingly, this review confirms that CybORG is the only available open-source environment to support multi-agent training, referring to the cooperative Blue agent training used in CAGE Challenge 3.

3.3. Other Competitive RL Applications

In order to facilitate RL, many of the simulation environments discussed in Section 3.2 operate as finite state machines, while using discrete time and using discrete action spaces. Organizing actions and network states into neat timesteps is far easier to simulate than a continuous time model, although this is less realistic. Fortunately, some of the most well document successes from competitive RL Applications have occurred in finite state environments with discrete time and

discrete action spaces.

Perhaps the most celebrated work in the competitive RL space is Google Deepmind's AlphaZero algorithm, which has been shown to reach superhuman level performance in a variety of classical games including Chess and Go [5]. There are two key reasons this fictitious play algorithm may not translate well to an ACO environment, without modification. First is the turn-based nature of these games: Chess and Go, like many classical games, require players to alternate their selection of actions. This is not the case in the most promising ACO environments, that model gameplay using simultaneous independent action selection from Red and Blue agents. The second reason, is that these classical games are played with perfect information, whereas ACO environments require both agents to operate with partial information. That is not to say that this algorithm cannot yield results in an imperfect information environment, but other examples of competitive RL in partially observed environments are discussed in more detail below.

Additionally, AlphaZero incorporates a general-purpose Tree-Search algorithm. Implementing a forward search is beyond the scope of this thesis, and this experiment will rely on a neural network to approximate the optimal policy using the current observation alone. However, Chapter 6 will discuss a forward search as a possibility to improve on this work in future research.

Some of the most notable research into RL with imperfect information takes place in competitive video games. Google Deepmind has also explored this space by training an AI to reach a Grandmaster rating in Starcraft II [38]. This time, using a dedicated algorithm called AlphaStar, that once again relies on fictitious play. Starcraft II is a much more complex environment that the ACO frameworks considered for this thesis, because the action and state spaces are large and continuous. But this is still a well documented example using fictitious play to approximate optimal policies in a partially observed environment where opponents act simultaneously.

Another example is OpenAI Five, where a similar fictitious play algorithm was used to master the game DOTA 2 [26]. This publication also describes how, in order to train more efficiently, they used quality-based opponent sampling. When a new agent policy was trained, they generated experience against higher quality opponents more often, and played against weaker opponents less frequently. Every trained policy in the policy pool maintained a quality rating that changed based on the quality of opponents they won or lost against, similar to the ELO rating system designed for chess.

These applications are relevant even though they are built for much more

complex environments. They meet that challenge using large, complex neural network architectures and more training time, in order to reach more difficult approximations. The process for training those neural networks (the algorithm itself) is equally viable for less complex environments. Scaled-down implementations of these network architectures have been proven in a number of OpenAI Gym environments with simple toy games [39] [40].

These applications relied on fictitious play algorithms, but as explained in Chapter 2, there is another important category of competitive algorithms to consider. CFR algorithms can be especially performant in imperfect information games. Perhaps the strongest example is an AI called Deepstack, which was trained at the University of Alberta to outperform professional poker players at Heads-up No-limit Texas Hold'em (HUNL) [6]. This impressive AI was built for a turn-based game and relies on a forward search for HUNL. Like many of the above examples, it is built for a game with a much more complex action and state space.

A CFR algorithm could be considered for an ACO simulation, depending on the scenario, if the performance of agents could be appropriately measured using regret. However, this application was valuable in poker specifically where agents must overcome recursive reasoning, because each action made by an opponent revealed information about the range of cards they might hold:

> "The correct decision at a particular moment depends upon the probability distribution over private information that the opponent holds, which is revealed through their past actions. However, how our opponent's actions reveal that information depends upon their knowledge of our private information and how our actions reveal it. This kind of recursive reasoning is why one cannot easily reason about game situations in isolation, which is at the heart of heuristic search methods for perfect information games." [6]

This kind of recursive reasoning is less of a factor in the ACO scenarios considered for this thesis, because the Red agent in particular has a relatively clear picture of the environment state, and the Blue agent actions betray no relevant new information. It is possible to construct scenarios where both agents are trying to withhold information, such as if the Blue agent has actions to establish decoy or honeypot devices, but these are beyond the scope of this thesis.

For the available ACO simulations, it is not obvious that CFR could be more

promising than fictitious play for approximating optimal policies. Although CFR could be an interesting topic for future research, especially as simulation environments become more complex and realistic, fictitious play has been used for more examples that train performant AI in similar environments, and so it is the priority to explore in this thesis.

Research into Deep RL for asymmetric games is less abundant. Goldwaser et al. provides one such example, with a game of 3-player Pacman where one player controls Pacman and the other players control the ghosts [41]. This research optimized performance in this game by extending the AlphaZero algorithm. Critically, this paper demonstrates how asymmetry between players is easily handled for fictitious play algorithms: by simply training separate neural networks to approximate the separate policy and value functions for each player. In fact, this research suggests that for many games the early layers of the neural network can be shared between agents since they will rely on similar features from the game state. However, this would require every agent has matching observability of the game (in order to have matching input tensors for the neural network), which is not the case in the considered ACO simulations.

3.4. Key Findings from Literature Review

This review of related works allowed for many key findings that influence the methodology of this experiment. First, it confirmed that a proof-of-concept for competitive RL in a simulated ACO environment would be a new contribution to the existing literature. The last time this research question was considered was before the emergence of the detailed environments discussed in this review [28]. Even then, the research only used competing single-agent RL without using a competitive algorithm like fictitious play. This thesis must use a proven competitive RL algorithm in a relevant ACO simulation environment.

Furthermore, existing environments are typically used for single-agent RL, where either a Blue or a Red agent is trained against static opponents, or a Red agent is trained against no opponent at all. A means to approximate optimal play in these environments could offer valuable insight into the realism of the simulations themselves.

Of the simulation environments examined, CybORG is the most appropriate for this thesis. It is an open-source option that can be modified relatively easily to support competitive play. The environment does forego certain elements of realism to accommodate RL, such as modeling scenarios using discrete time, but this is true of all the available research environments at this time. Many of

these environments, including CybORG, have ongoing development to gradually introduce more realism to the simulations. As well, the recent CAGE Challenges have made this environment especially relevant to ongoing research in the space.

In order to bring the success of competitive RL from classical games to ACO, a fictitious play algorithm is used for this thesis. This specific algorithm will be designed for the CybORG environment, but the experiment methodology described in Chapter 4 is structured to be repeatable in any similar simulation framework. Scenario details that are specific to CybORG will be introduced during the Evaluation in Chapter 5. The scenario network and agent action-spaces are easily configurable in CybORG. This will be important in order to simplify the scenario from the intricate CAGE Challenge network, so that the Competitive RL can be attempted in a network simulation with minimal complexity.

Finally, CybORG is built using the OpenAI Gym framework, and so it supports many existing RL tools for OpenAI Gym [35]. These existing tools will be leveraged as much as possible when implementing the RL portion of the fictitious play algorithm. However, separate actor and critic neural networks will be required for each agent since this game-design will be asymmetric. These design details will be explored more in the following chapter.

4. Methodology

This chapter presents the methodology which will describe the process of implementing a competitive algorithm, and verifying that the trained agents have approximated optimal play. This includes the evaluation process and design decisions for the experiment included in this thesis. This experiment will achieve aim of this research if it can demonstrate a competitive RL algorithm that is used to approximate optimal policies for Red and Blue agents in an ACO environment.

The experiment can be organized into five phases. Each of these phases aligns with a research activity listed in Chapter 1, and will be described in greater detail in the sections below.

1. Select an ACO Environment. Identify a simulated ACO environment that is suitable for competitive training.

2. Identify the Target Behaviour. Consider why a Red or Blue AI is being developed at all. This may be limited by the scope of the environment, including the simulation features that are modelled, and the features that may have been abstracted away. Within the constraints of this environment, identify all the priorities for the Red and Blue agents. These priorities will define the target behaviors of the optimized agents, and should reveal one or more eligible network topologies for the scenario.

3. Define the Game Design. Set the parameters for the Markov Game that will be used for training. This will include modelling the priorities of either agent as a reward function, and defining termination conditions for the scenario. This will also include identifying the state-space for the environment, the scenario topology, and the observation space of each agent.

4. Implement Fictitious Play. Use available RL tools to conduct fictitious play in the simulation environment. Monitor the training performance of either agent to observe their convergence to a Nash Equilibrium.

5. Validate the Trained Policies. Measure the exploitability of agents in the Red and Blue policy pools to confirm that exploitability decreases across generations, which demonstrates a convergence towards Nash Equilibrium. Examine sample games of the trained policies to ensure they meet the target behaviour.

This chapter will not discuss the specific implementation details for the CybORG environment that is used in this experiment. These will be included as

part of the evaluation in Chapter 5, because this methodology is meant to be repeatable for any eligible ACO environment.

4.1. Select an ACO Environment

Before attempting any sort of competitive game design, a simulation environment must be selected. The simulation environment will set limitations for the scenario and game design, because neither of these can violate the simulation constraints. Eligible simulation environments for competitive RL must accept actions and return observations for a Red and Blue agent operating simultaneously. Additionally, the environment must be able to simulate the scenario many times relatively quickly to accommodate RL. This is why a simulated environment is required for training instead of an emulated environment.

The process of selecting an environment was completed during the literature review in Chapter 3, by examining the available open-source environments. It was found that the CybORG environment could be modified to support competitive play. The environment was also publicly distributed during the CAGE Challenge, and as a result, it has a scenario and game design that have already been explored extensively with single-agent RL [36]. A simplified CAGE Challenge scenario is an ideal target for this thesis.

This environment does impose certain limitations, which will be discussed in more detail in Chapter 5. To operate in the CybORG environment, the fictitious play algorithm in this experiment will rely on the discrete timesteps, as well as discrete observation and action spaces. Although it might be possible to implement fictitious play in an ACO environment with continuous time or continuous action spaces, it will not be explored here.

4.2. Identify the Target Behavior

The target behavior describes what each agent should learn to do in the environment. Identifying the target behavior includes committing to a specific cyber incident scenario. This will be used to construct a game design where this behavior is the most rewarded. If RL was being used to develop a decision-making policy for some real-world application, the target behavior could be very nuanced. However, for the purpose of this thesis it is relatively straight-forward. The CybORG environment is used with a simplified CAGE Challenge scenario, and therefore the target behavior is inherited from the goals of the CAGE Challenge.

A Red and Blue agent will compete in a capture-the-flag (CTF) style game. In these games, a Red agent attempts to compromise a target server which represents

the "flag". It will also need to compromise other machines on the network in order to reach the flag. This style of cyber competition is a popular category for human experts, and so it is a sensible game to simulate in an ACO environment. The Blue agent must learn to take mitigation actions that will minimize Red's foothold on the network, while prioritizing the defence of the target server. The Red agent will attempt to expand its foothold on the network by compromising additional hosts, but impacting the target server is much more important than every other device. Intuitively, the desired behavior for an AI agent in an ACO environment would match the decisions a human-expert would take if given the same set of observations and actions.

The trained agents at this end of this experiment should prioritize their scenario objectives while selecting actions that are not obviously refuted by their opponent. The validation at the end of this experiment will confirm if the agents have achieved minmax policies, but observing sample games is still the only way to confirm that the optimized policies match the intended behavior. This is because it is possible that the agent has correctly optimized for the game design, but the game design did not adequately reward the desired behavior [42].

4.3. Define the Game Design

A competitive algorithm can be used to train autonomous agents in a suitable environment, but agents will only learn a desired behavior if the game design correctly rewards it. A Markov Game must be designed so that agents have a system to optimize. This game must be able to calculate either agents reward based on the true state of the simulation, in order to provide that reward to the learning agents. The observation and action spaces for the agents are defined by the environment and the scenario, but the reward function and termination conditions need additional consideration.

Agents need a reward signal in order to learn: their policies will update to maximize their return in the scenario. Therefore, the reward function must precisely reward the target behavior of the agents. The terminal conditions for the game during training are equally important, as intelligent agents might seek out or avoid terminal conditions if this effects their total accumulated reward.

4.3.1 Reward Function

If reaching a target server on the network is the objective of a Red agent, then defending this node would be the top priority of a Blue agent. But the Blue agent should not be impartial to the Red agent establishing a foothold elsewhere

on the network: a Blue agent that allows the Red attacker to reach everywhere except the operational server is not a very useful AI agent.

The intuitive target behavior for a Blue agent requires that it learns to address the different priorities, or objectives, within its environment. This includes minimizing Red's foothold on the network and avoiding downtime of services to network users. Often, the Blue agent will be required to choose between actions that both support different priorities. These conflicting priorities need to be quantified in the Blue reward function in order to shape the target behavior of an optimal Blue agent. Training an agent to resolve conflicting priorities using RL is known as Multi-Objective Reinforcement Learning (MORL) [43].

The seminal paper on MORL argues that when training an RL agent with multiple objectives, and it is impractical to scalarize their relative importance, it is often more effective to train multiple agents across a variety of reward functions for the different objectives [43]. Then afterwards, compare the behavior of these optimized agents and select the AI agent that most closely demonstrates the desired behavior.

This will not be necessary for this thesis because optimizing a single game design will be a sufficient proof of concept for fictitious play. However, if competitive RL was being used to achieve a more nuanced target behavior, it might be beneficial to attempt learning with a variety of sensible game designs and see which of the trained agents most closely resembled the desired AI agent.

This could include training multiple agents that each use different reward functions, while changing the relative importance of rewards in the environment. For example, it could be worthwhile to train separate agents for the same scenario where the target server is worth 2, 5, and 10 points in their respective reward functions, then examine the optimized agents to see which policy best matches the desired behavior.

4.3.2 Terminal Conditions

In many MDPs, terminal conditions for a sequence can be just as important as the reward function. If an agent can continue to accumulate reward by delaying end of a game, it might learn to do so even if this is not the desired behavior. This is especially important during a zero-sum game, where a reasonable agent might attempt to end the game rather than lose reward.

The ideal terminal conditions for a game design will depend heavily on the scenario and the desired behavior. For example, in order to encourage agents to achieve their goal as soon as possible, a game design might penalize the agents more heavily if it takes them longer to reach a terminal state. This is espe-

cially important for infinite-horizon and indefinite-horizon scenarios, but even in environments with a finite-horizon the termination time must be chosen carefully.

The scenario used in this research has a finite-horizon, and uses discrete timesteps instead of continuous time. Every scenario will run for a fixed amount of timesteps, with no opportunity for agents to end the sequence prematurely. Their ability to accumulate reward will depend entirely on their decisions prior to the final termination time. During the game design in Chapter 5, a horizon of 12 timesteps is used to create a game that is less complex than the CAGE Challenge, where the optimal policies are still non-trivial.

4.4. Implement Fictitious Play

Although fictitious play has been used in many other adversarial environments, this experiment is the first implementation in an ACO environment. Algorithm 3 presents the details for each component of fictitious play with opponent sampling.

Algorithm 3 Fictitious Play for ACO Environments.

$\Gamma \leftarrow$ initialize training environment
$\pi_0 \leftarrow$ set random initial policies for generation 0 $(\pi_0^{blue}, \pi_0^{red})$
$g = 0$
while within computational budget **do**
  $g \leftarrow g + 1$
  for each player i in $[blue, red]$ **do**
    $\pi_g^i \leftarrow$ set random initial policy for player i generation g
    while π_g^i is improving **do**
      $M^i \leftarrow$ clear memory buffer to store new batch of samples
      while memory buffer M^i is not full **do**
        $\pi^{-i} \leftarrow$ select opponent policy from the pool Π^{-i}
        $\Gamma \leftarrow$ reset the training environment for a new game
        $M^i \leftarrow$ store samples $(u_t^i, a_t^i, r_{t+1}^i, u_{t+1}^i)$ for every timestep t in Γ
      end while
      $\pi_g^i \leftarrow$ update policy using PPO for batch of samples M^i
    end while
    $\Pi^i \leftarrow$ add new policy π_g^i to pool
  end for
end while
Return $(\pi_g^{blue}, \pi_g^{red})$

This algorithm requires an environment Γ as input and produces approximations of the optimal Red and Blue policy (π^{blue}, π^{red}) for that scenario as an output. The accuracy of this approximation will depend on the complexity of the environment and the computational budget. For this reason, the experiment

shown in Chapter 5 uses a relatively simple scenario. A simple scenario should allow for a better approximation of the optimal policies, to demonstrate a proof of concept for this algorithm.

The first step is to initialize the training environment. A new OpenAI Gym environment was constructed to act as the training environment for this experiment [35]. The OpenAI Gym framework is a well-established resource for building RL environments. The training environment is used by the algorithm to generate samples for RL. It must be able to accept a Blue and Red action (a_t^{blue}, a_t^{red}), advance the environment one timestep, and return each agent's new observation u_{t+1}^i and reward r_{t+1}^i. To do this, the training environment used in this experiment maintained a CybORG simulation as an attribute, and used that simulation to determine new observations and rewards.

Many open-source RL solutions are built to train specifically in OpenAI Gym environments. For this experiment, resources from the open-source library RLLib are used. RLLib is part of the Ray framework [44]. Ray provides a means for parallel processing Python applications. RLLib is used here because the included RL algorithms integrate easily with OpenAI Gyms environments, but also because collecting samples in parallel substantially improves the speed that new policies can be produced. Using many parallel training environments means that more generations of policies can be produced within the computation budget, allowing the algorithm to output a better approximation of the optimal policy.

Once the training environment is set, the initial policies (generation 0) must be created and saved. Each generation g represents one loop of the algorithm where a new set of policies will be saved. Policies in this experiment are saved using RLLib's checkpoint system. The learning algorithm is selected when a new RLLib policy object is created. For this experiment, PPO is used for the RL component of fictitious play.

PPO was selected for this experiment because it has been demonstrated to be performant in the CybORG environment. It was used by many teams during the CAGE Challenge to train performant Blue agent policies, including by Mindrake which won the first CAGE Challenge [36] [45]. Moreover, PPO has been proven for other applications in environments that are similarly partially-observed, asymmetric, or competitive.

For generation 0, two RLLib PPO objects are created and saved as the initial Red and Blue agents. The observation space and action space of the agents are declared when they are initialized. A multi-binary observation space is used for both agents. This means that the PPO agents must receive their observations as a vector of 0 and 1 bits.

Observation vectors are a one-hot encoding of the categorical features from the agent's most recent observation in the environment. They are used as the input tensor for the actor and critic neural networks in order to output a policy or value. The Red and Blue agents may have different sized observation vectors, since this is an asymmetric game, and the encodings used by each agent are different.

The observation space will also be different for every environment and scenario. For an ACO environment such as CybORG, one-hot encoding is used to describe the status of each machine on the network, as seen by either agent (such as using five bits per machine to represent five observed boolean features on each host). Examples of these encoded observations are included as part of the evaluation in Chapter 5.

This highlights an important limitation of this algorithm: a policy trained for one network topography will not generalize to any other networks. A trained policy cannot produce actions for a different network, because the encoded observation of that network will be unusable as an input to the actor. The multi-binary observation vector must exactly match the expected input of the neural network. Options for generalizing trained policies is another topic of ongoing research in ACO environments.

Both agents also use a discrete action space. When selecting an action, an agent's entire action space is encoded so that each action is represented by a unique integer. The output layer of the actor network has one unit per action in the agent's action space. A softmax activation function is used for the output layer, so that every index of the output vector holds a probability for selecting the corresponding action. This way, the actor network does not just produce an action but also a probability. The output vector is a stochastic policy that approximates the probabilities that should be used for selecting each action, based on the observation.

The observation and action spaces determine the input and output layers of the networks. For the other components of the network architecture, design decisions must be made to try and maximize performance and lower training time. The number of hidden layers, the units in each layer, and their activation functions will all depend on the complexity of the environment. The architectures that were used for this experiment are shown in Figure 5.

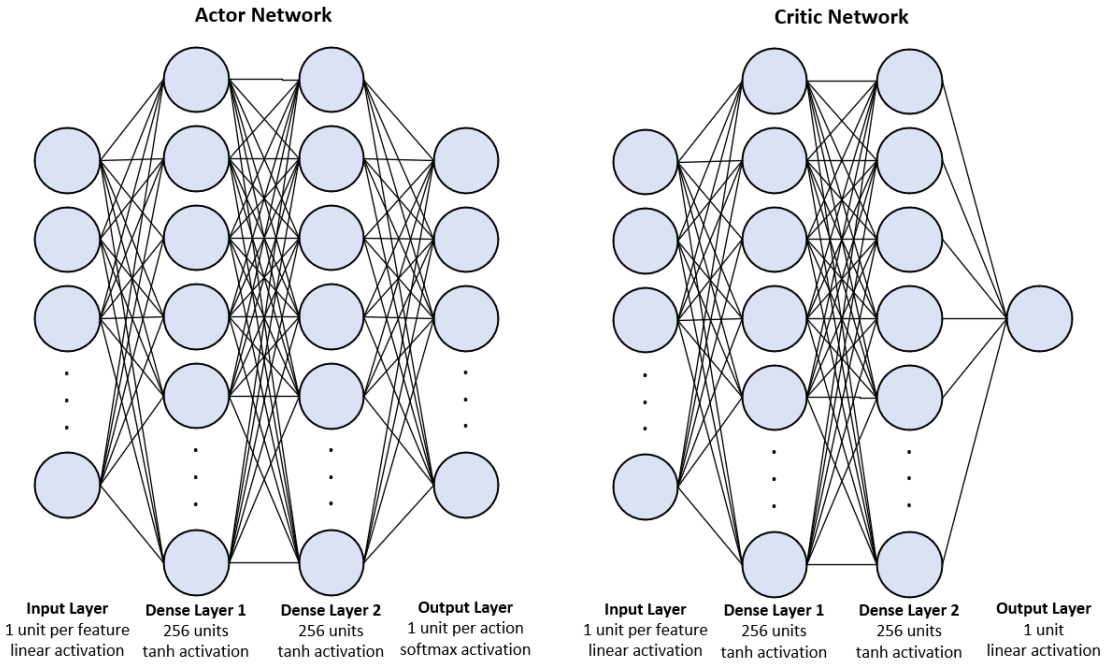


Figure 5: Actor and Critic Neural Network Architectures.

The CybORG environment and scenario used in this thesis are relatively simple (less than 50 encoded bits in the observation vector, and less than 20 discrete actions for both agents). A recent study investigated the effectiveness of different PPO network architectures with various activation functions [46]. This study, as well as some local tuning, were used to contruct PPO networks that were suitable for the scenario environment. The actor and critic networks for the Red and Blue agents each use two hidden layers of 256 units. These are fully-connected layers using a *tanh* activation function.

The critic network uses the same multi-binary input, but the output for this network is a single unit. The critic network is not trying to produce a probability distribution, it is used to produce a value prediction from an observation. An entirely separate network was used for the critic for this experiment, although this is not necessarily required.

RLLib includes an option to have the actor and critic networks share hidden layers, so that only a separate output layer is used. This can allow the agents to learn more quickly in many environments because the actor and critic often rely on the same environmental features. However, the networks were kept separate for this experiment because Andrychowicz et al. found that separate networks led to more performant policies in most environments, at the cost of slower training [46].

After the generation 0 agents have been saved, the first generation of training starts. The OpenAI Gym environment is split into seperate "RedTrainer" and "BlueTrainer" environments for the experiment. The environments are identical, except that the RedTrainer environment only required Red actions as an input to step the environment, and the BlueTrainer environment only required Blue actions as input.

The generation 1 Blue agent trains first, learning to optimize against the only available Red opponent. Since the Red opponent is taking random actions, it is unlikely that the Red opponent will accumulate any significant reward during these initial games. As a result, the generation 1 Blue agent should lower the score as close to 0 as possible, and adopt a policy to simply avoid any actions that would incur a reward penalty.

During each generation of training, the agents are trained until their score is no longer improving. The average return from each batch of samples is recorded. The batch size is a very important hyperparameter that should be tuned for each environment and scenario. The batch size determines how many samples will be added to the memory buffer M before updating the networks.

If the batch size is too small, then the learning agent might converge on a policy before thoroughly exploring the game tree. In environments with delayed rewards, it is crucial that the batch size is large enough so that the learning agent will encounter critical rewards in the game tree before updating. Otherwise, the policy is likely to get stuck in a local minima, capitalising on one source of reward while never exploring to find an even better source elsewhere in the game tree. However, larger batch sizes can substantially increase training time, since more samples are required to perform one update step.

For this experiment, a large batch size was required for the Red agent to find crucial rewards that were delayed in game tree. Often the Red agent will need to select the correct combination of three actions in sequence before receiving a reward. Consider that this Red agent which has 19 actions will start learning by exploring randomly, and will only select the correct combination of actions one in 19^3 attempts. To ensure sufficient exploration, batches of 61440 samples were used. This was a deviation from the recommendations in Andrychowicz et al. but a crucial design choice for the CybORG environment.

A generation of training is stopped when the average return from a batch is no longer improving. This is controlled using a variable called "tolerance". The tolerance sets how many batches in a row an agent can go without improvement before training is stopped.

During a typical generation of training, the agent should show continuous

improvement every single batch until it is nearing the optimal policy for that generation. At that point, the average score for any individual batch might vary based on random elements the environment. Small optimizations may impact the overall score less than this random chance, and tolerance is an important tool to control how agents proceed in this state.

If the tolerance is too low, then an episode of training can be stopped prematurely before an agent has approximated the optimal policy. If the tolerance is too high, then every single generation of training can take significantly longer as agents dwell on nearly negligible improvements (even if these policies are more optimal, this time could be better spent training additional generations). In this experiment, a tolerance of 3 batches was used, and this allows the agents to reach accurate policies in reasonable time.

In rare cases, a policy might suffer from forgetting if it hovers at the near optimal policy for too long. This occurs when the network is updated in such a way that it has an unintended effect. In most cases, the learning agent can recover to its near optimal state (for example, see the training performance for dedicated agents in Chapter 5, Figure 18 where the performance occasionally drops temporarily).

Fortunately, in the rare cases that a sub-optimal policy reaches the end of its tolerance and joins the policy pool, this never has enough of an impact to prevent the overall convergence of later generations (consider that every early generation adds a sub-optimal policy to the pool). However, this is still an important factor for training efficiency. Training parameters should be tuned to ensure that learning is stable and forgetting happens as little as possible.

The first Blue agent is trained until it has experienced 3 batches of training with no improvement. This trained policy is saved as an RLLib checkpoint and added to the pool of Blue policies. Next, the generation 1 Red policy is trained against the generation 1 Blue opponent. Since the Blue agent has not learned how to intelligently defend the network (because it was trained against the initial random opponent), this first Red generation should learn to accumulate a near maximum possible score in the environment when the policy finishes training.

The algorithm continues by training the second and third generations of agents. However, all generations past the first are trained against more than one opponent. This is the most important component of the algorithm, because it ensures that subsequent generations converge towards an optimal minmax policy. Instead of optimizing against a single opponent, every new generation collect samples by simulating scenarios against various opponents selected from the existing opponent pool.

The opponent sampling procedure depends on which minmax policy is being solved by fictitious play. Because this is an asymmetric environment, where the Red and Blue agents have separate actions and observation spaces, only one optimal policy is solved at a time.

The Blue minmax policy is solved first. Each generation, a Blue agent is trained against the entire Red opponent pool equally with uniform sampling. Then, the Red policy learns the best-response to the most recent Blue agent. Figure 6 shows this cycle of training and saving new policies to the pool.

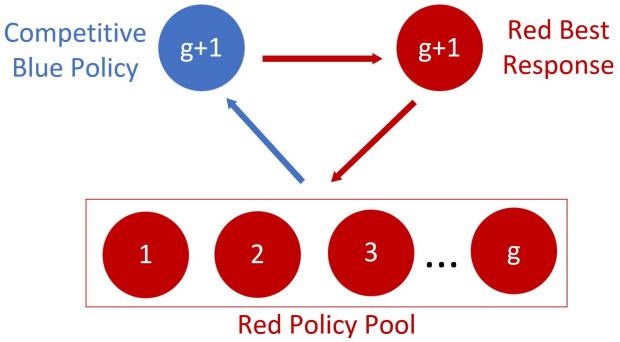


Figure 6: Opponent sampling technique used during fictitious play to find the Blue minmax policy. Each new generation $(g+1)$ is trained against the existing pool of Red policies. Then, the new Red best-response is trained and added to the pool.

When the Red agent is learning a new best-response policy, 90% of the Red agent's games are played against the most recent Blue policy and 10% are played against random opponents from the existing Blue pool. This ensures that the new Red policy is still capable of operating in game tree branches that are never reached by the most recent Blue agent.

This cycle of solving the optimal policy against the opponent pool, and then adding the opponent's new best-response, ensures that any exploitable behavior in the Blue policy becomes less likely each generation. Because with each new generation, the learning Blue agent is more likely to encounter an opponent that will discourage that exploitable behavior. Through this cycle, the emerging Blue agent policies are gradually steered towards game theory optimal play.

The RedTrainer and BlueTrainer environments maintain the opponent's sampled policy as an attribute. Every timestep, the training environment uses the current opponent's policy and observation to determine an opponent action and step the environment. Every time the reset method is called, the environment updates the opponent for the next game by selecting a new opponent policy from the pool. By using these separated RedTrainer and BlueTrainer environments, an agent can train a new policy using just RLLib's PPO algorithm, and the custom

environment controls the opponent sampling.

Training against the entire pool of past opponents has the same effect as training against the average opponent policy, as proposed in [23]. Each new generation is not specifically targeting an optimal minmax policy. Instead, PPO is used to optimize expected return against the entire pool without any consideration for guaranteed return against individual opponents. However, by adding each newly trained policy to the pool, this does ensure a gradual convergence towards the optimal minmax policy.

Consider a newly trained Blue policy that has optimized against the existing Red pool, but this is an early generation and it does not represent a true minmax policy. If this new generation is not minmax optimal, then it will be exploitable in some way. This means that the Red best-response will converge towards a policy that exploits this fault. Then, when future Blue generations are trained, they will be less likely to inherit the same exploitable behavior, because it will now be less effective against the pool of Red opponents with every subsequent generation.

As each new generation of Blue agent learns to maximize score against the entire existing opponent pool, newer generations are forced to avoid any non-optimal behaviour, because any exploitable behaviour will be less viable with every passing generation. Generations should start to demonstrate near-optimal behavior once a minmax policy will offer the best return against the varied pool of opponents. This is how fictitious play converges towards the optimal minmax policies for an environment.

After the Blue agent training is completed, fictitious play is used one more time to discover the optimal Red agent policy. The process is exactly the same, except that new Red policies are trained against the entire existing Blue pool, and Blue best-responses are calculated, as shown in Figure 7.

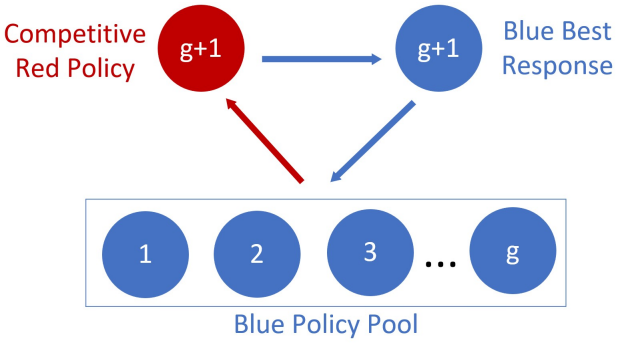


Figure 7: Opponent sampling technique used during fictitious play to find the Red minmax policy.

The training scores for the Blue and Red agents should converge during both instances of fictitious play. As the training scores for both agents flatten to single-values, this indicates that the algorithm has reached a Nash Equilibrium where neither agent is being encouraged to deviate their policy further. In this state, opponent training scores should be held near the minmax value each generation. The true minmax score of the competitive policies is confirmed in the validation.

4.5. Validate the Trained Policies

With the competitive policies trained, the last step of the experiment is to confirm that they approximate minmax optimal policies. This is done using two tests. First, every trained policy from the entire pool will be evaluated for its exploitability. This should demonstrate a gradual convergence towards more optimal policies throughout training. Second, new dedicated opponents will be trained against the competitive policies in order to measure the worst possible return for the competitive policies. Their true score against a worst-case opponent, that has been optimized exclusively against the competitive agents, will reveal how accurately the policies were able to approximate optimal play.

To evaluate the exploitability of every policy in the pool, every Blue competitive policy simulates 50 games against every Red opponent policy. This determines the expected return for every possible combination of these Blue and Red agents. The maximum expected return for every Blue policy is saved. If the policy pool has enough variance, then these scores should approximate the guaranteed return for every agent against a worst-case opponent. Then, the exploitability is calculated for the Red competitive policies in the same way. This time by measuring the minimum expected return for every Red policy against any Blue opponent.

Recall that exploitability measures the difference between a policy's guaranteed return and the guaranteed return of the optimal policy. Plotting the guaranteed return for every Red policy should indicate a gradual increase in scores across generations. In the later generations, these guaranteed scores should converge towards the asymptote that represents the Red minmax score: the highest expected return that an optimal policy can guarantee against any opponent.

Blue agent scores should reveal a similar but opposite trend. Early Blue policies will likely be more exploitable and allow high expected return. Later Blue generations should lower the expected scores as they approach a guaranteed maximum: the Blue minmax score. By validating that agent exploitability decreases across generations, it is proven that the policies are converging towards optimal

play.

Once it is observed that policies are gradually less exploitable, the last step in the experiment is to verify how accurately the competitive agents have approximated optimal play. To do this, new dedicated opponents are trained using PPO, in the exact same RedTrainer and BlueTrainer environments, except without opponent sampling. Their only opponents are the individual competitive policies.

If the competitive policies are in any way exploitable, it will be trivial for dedicated opponents to discover these vulnerabilities and lower their score. The first evaluation provides the expected minmax score using the fictitious play opponent pool. The dedicated opponents will show the true guaranteed score for either agent.

If the dedicated opponents match the expected minmax scores from the evaluation, then this confirms the agents have perfect optimal policies, because the competitive agents are able to hold even a dedicated opponent to the expected return limit. If the dedicated opponents are able to reduce the performance of the competitive agents, this will reveal exactly how accurately (or inaccurately) the competitive agents were able to approximate optimal play.

4.6. Experiment Summary

This experiment details a process for identifying an eligible environment and target behavior, developing optimal policies through fictitious play, and validating those optimal policies. If this process can be followed and the learned policies are confirmed to be optimal, this achieves the aim of this thesis. The evaluation in Chapter 5 will document the specific implementation details for this algorithm in the CybORG environment, and the results obtained via this methodology.

5. Evaluation

This chapter presents an implementation of a fictitious play algorithm in a cyber incident simulation environment, and evaluates the performance of a Red and Blue agent pair that have been trained to Nash Equilibrium. The exploitability of these trained agents will be measured to determine their minmax performance and validate that the agents have approximated their optimal policies. This chapter will also discuss in detail the specific environment, the algorithm implementation, and the parameters that were used when obtaining these results.

5.1. CybORG Environment

The methodology was designed such that it would be repeatable in other simulation environments. This was done to support the continued development of ACO environments for RL, by providing an experiment methodology that could applied to any eligible environment. However, in order to observe the algorithm for this evaluation, a specific environment and scenario are selected.

The Cyber Operations Research Gym (CybORG) was selected as the simulation environment for this research. As discussed in Chapter 3, this environment was initially made public as part of a competition called the CAGE Challenge, where teams competed to train the most effective Blue agent to defend a target network during an attack [36].

The simulation models a cyber security incident as a turn-based game between a Red and Blue agent that takes place over a fixed number of timesteps. Each timestep, both Red and Blue independently select one action simultaneously. The environment resolves their actions to update the network, and provides each player with a new observation for the next timestep. This is an asymmetric game, where Red and Blue are using different sets of actions. Red actions model an attacker trying to conduct lateral movement on the network. Blue actions model the options of a SOC analyst defending the network during a cyber attack. The environment is also only partially observable for either agent. Neither player has access to the complete state of the environment, both are limited to their own observations during each timestep when selecting actions.

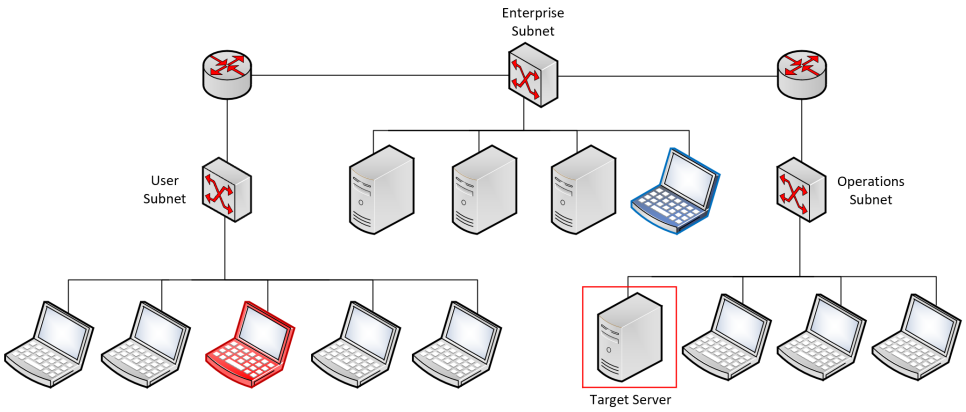


Figure 8: Network Topology for the CAGE Challenge. This network will be simplified for the evaluation.

The network topology is easily configurable in CybORG. Figure 8 shows the topology used for the CAGE Challenge. The environment includes a variety of host operating system images that all have some form of vulnerability. These devices are simulated, not emulated, so these are not real or virtual machines. These hosts are simply python objects that are generated at the start of each scenario to track the status of each device on the network. As a result, many elements of realism are missing from the environment, but the simulation attempts to track the most important details for a cyber incident scenario. Host details include the operating system, what processes are running, what ports are open, what user accounts exist on the host, what are the privileges of each user, and various other details that would be relevant during a cyber attack.

CybORG was an ideal environment for this research for a few key reasons. First, it is built using the OpenAI gym framework, and so it supports many existing RL tools for OpenAI gyms [35]. The scenarios are easily configurable, and different game designs have been publicly tested during past iterations of the CAGE Challenge. Most importantly, CybORG can be modified relatively easily to support competitive play.

The CAGE Challenge environment was only intended to train Blue agents, so it only accepts actions from one agent and simulates the choices of other agents within the environment. However, the Fictitious Play algorithm requires the environment to accept actions from both the Red and Blue agents, and it must return each of their rewards and observations as they transition to the next timestep. This is needed to train both agents simultaneously. In order to accommodate competitive play, two key changes were made to the source code:

1. Results.py. The Results object class was modified to store both a red_observation

and a blue_observation as attributes, rather than just a single observation. These separate dictionaries ensure each agent only receives their own partial observation of the environment state.

2. EnvironmentController.py. The Step method in the EnvironmentController class was modified to accept both a Red action and a Blue action as parameters, and to return a Results object that contained the separate Red and Blue observations. Previously, the Step method only accepted one action as a parameter, and simulated the actions of other agents within the environment. The Reset method was also modified to return an updated Results object.

Committing to this framework introduces some important considerations for the experiment. Like all other suitable environments, CybORG abstracts away certain elements of realism in order to facilitate RL. Although the framework attempts to model all the relevant information for each host, it is unreasonable to assume that a decision-making agent in this simulation environment would maintain its performance in an emulation environment. The most egregious abstraction is the use of discrete time to model the cyber incident. This alone means that any trained policy in this simulation would not translate to a real world or emulated scenario, where the actions of a Red and Blue actor would not be organized neatly into timesteps that resolve actions simultaneously.

Of course, the design philosophy of the environment is iterative, and more elements of realism (including continuous time) are planned as future features by the developers of the framework. Regardless, these current limitations are still relevant to this experiment, as both the algorithm and the learned policies abuse this discrete time model. The game design that is optimized here would be significantly different in a continuous time model. A similar fictitious play algorithm could still be implemented in such an environment, despite substantially different observation and action spaces, but that discussion is beyond the scope of this thesis.

A final key limitation to acknowledge is that this environment is still under development, and thus it is not bug free. This was most noticeable when a Blue agent uses the remove action in the environment, which appears to have a much lower success rate than intended. Some actions in the CybORG environment have less than a 100% chance of success, and the low success rate for the remove action can be dismissed as part of the game design for this scenario. The learning agents simply optimize around these new success rates instead, and the Blue agent is forced to consider other, more reliable, choices. As long as the agents still learn

to optimize their policies for this game design, the results are equally viable. It is possible that this behaviour was caused by the modifications to the Environment Controller that were required to support competitive play. Although this does influence the learned policies in this environment, the low success of the remove action does not impact the experiment.

Additional details about the CybORG environment and its design decisions can be found in the research paper, and the original source code can be found at the CAGE Challenge Github repository [12] [36].

5.2. Scenario and Game Design

This experiment will simulate a smaller network than was used during the CAGE Challenge. This is done to focus the evaluation on the fictitious play algorithm and ensure that any performance issues cannot be dismissed as the consequence of a complex scenario. A simple network has been chosen because the goal of this thesis is to provide a proof of concept for fictitious play, not to explore the limits of the parameters and hardware that were used in this experiment.

The network topology used for the evaluation is shown in Figure 9. The scenario used in this evaluation contains two sub-networks: a User subnet, and an Operations subnet. The Red agent starts with Root privileges on a compromised host in the User subnet. The goal of the Red agent is to achieve Root access and impact a target server on the Operations subnet.

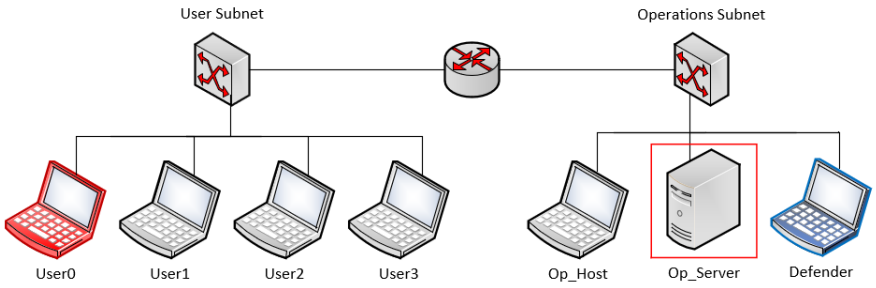


Figure 9: Network Topology for the Evaluation.

Red receives a reward for expanding its foothold on the target network. A single scalar reward is required for RL, and it is the same reward function that is used in the CAGE Challenge. Each parameter of the reward function is assigned a value based on how critical these events are to the Blue defender. As the Blue agent learns to minimize the expected reward through its actions, it should prioritize protecting the assets that have been assigned a higher value. The components of this reward function are:

1. Impacting the Op Server. If Red uses the impact action while it has Root privileges on the Op Server, Red scores 10 points.

2. Foothold on an Ops Device. Red scores 1 point for each Ops device it has Root privileges on at the end of a timestep.

3. Foothold on a User Host. Red scores 0.1 points for each User host it has Root privileges on at the end of a timestep.

4. Blue Defender Restores a Device. Red scores 1 point every time Blue restores a device to a clean image.

The final scoring parameter represents a Blue penalty if it is required to restore any machine to a fresh image. This penalty is included in the game design because this action is undesirable for Blue, as restoring a potentially compromised host or server would impact regular network users.

The Red agent has 12 timesteps to collect as much reward as possible before the scenario terminates. The goal of the Blue agent is to minimize this score, making this scenario a zero-sum game. During the CAGE Challenge, policies were evaluated using three different termination conditions: agents were evaluated for average performance in scenarios that lasted 30, 50, and 100 timesteps. The termination condition used in our evaluation was simplified for the same reason as the network topology. 12 timesteps is long enough for the Red agent to reach and impact the target server, with some extra timesteps to spare in order to allow for strategic play (specifically, the Red agent can Impact the Op Server on turn 8 at the earliest). This keeps the scenario as simple as possible to observe the fictitious play algorithm, while keeping the game complex enough that the optimal policies are not obvious.

Finally, it is worth noting that Red does not score any points for its initial compromised machine, nor can Blue restore this initial foothold to remove Red from the network entirely. Red must maintain control of at least one User machine to have any agency in the game.

5.3. Environment Wrapper

In order to facilitate the training of RL agents, a wrapper is put around the environment to act as the interface between the agents and the simulation.

Although the simulation tracks the specific vulnerabilities and exploits used during a scenario, these details are abstracted away from a decision-making agent during training. The wrapper provides this layer of abstraction between the agent

and the environment. For example, the wrapper accepts an abstract action choice from an agent that only requires an action-type and a target, such as "Exploit User1", and then it will construct an exploit with appropriate parameters based on the agent's intention and the collected information about User1. If the agent does not have the required information about the target to perform that action, then the action will be unsuccessful. To perform this task, the observation and action spaces for the agents are explicitly defined in the wrapper.

The CAGE Challenge required teams to use a standard wrapper, called the Challenge Wrapper, for the competition. However, since the CAGE Challenge involved training Blue agents against scripted Red opponents, the Challenge Wrapper is not suitable for training both agents in fictitious play. To overcome this, a new Competitive Wrapper was designed for this experiment. This Competitive Wrapper defined the observation and action spaces for each agent for this evaluation.

5.3.1 Action Space

Table 1 provides a summary of the action space for the Red and Blue agents. The Red attacker has a list of discrete actions that are based on tools and exploits found in the Metasploit framework. They can scan subnets to find the IP addresses of reachable hosts, and scan individual devices for open ports and services. The information gathered is typically limited to what the Red agent could collect using scanning tools such as NMap.

Using this information, the attacker can take actions to exploit a target host. These exploits are only successful if the Red agent has identified a suitable vulnerability and can match the correct parameters for the exploit. A successful exploit will give the Red agent either User or Root credentials on the target. Red can also escalate privileges on a target in order to transition from User to Root credentials.

Red Action Space	Blue Action Space
Discover Remote Systems. Discovers hosts/IP addresses in a subnet through active scanning tools.	Analyze Host. Collect information on a specific host to identify if Red is present on the systems. Receive information on any malicious processes, files, and services to identify if Red has Root or User privileges.
Discover Network Services. Discover responsive services on a selected host by initiating a connection with that host. Receive ports and service information.	Remove. Removes Red from a host by destroying any known malicious processes, files and services. This action attempts to stop all processes identified as malicious.
Exploit Network Services. Exploit a known vulnerable service on a target system. Requires the IP and service port. Establishes user privileges on the target system.	Restore. Restores a system to a known good state. This has consequences for system availability.
Escalate. Escalate to admin privileges on a compromised machine.	
Impact. This action can only be used once Red has admin privileges on the target server. This action indicates that Red has completed their lateral movement and is ready to achieve their objective on the target.	

Table 1: Action Space Summary.

The Blue agent actions are limited to simple actions that a network analyst could take to protect a network during a security incident. This includes Analyzing hosts to identify a Red foothold, removing users from hosts if they appear to be compromised, and restoring devices to a clean image to ensure Red is off the machine. The restore action is undesirable for Blue because restoring a machine would impact regular network users. A penalty for this action was included in the reward function to measure exactly how important this priority is for the Blue agent.

Both Red and Blue actions have varying probabilities of success. For many actions, the likelihood of success depends on the host, because the host image determines what vulnerabilities can be exploited and how easily Red can establish a foothold on the device. For example, the remove action has a very small chance of success on the Op Server, forcing Blue to rely on the restore action to disrupt Red on the server, but removing has a high chance of success on the Op Host. As well, when Red exploits a User host it will very often (but not always) gain Root access immediately, without needing to escalate privileges, but this almost never occurs on Ops devices.

Finally, Blue does receive certain information passively without the need to analyze a host. Blue receives an alert whenever Red scans a host for open ports, or when Red exploits a vulnerable service. The information that Blue receives

passively is meant to model the alerts that would be received by an endpoint monitoring tool, such as Velociraptor. Any information gathered passively or with the analyze action is encoded in Blue's observation space.

5.3.2 Observation Space

The Competitive Wrapper converts the raw observation data into an observation vector that is usable by game playing agents. The raw observation from the environment is a python dictionary describing all the known information about every known host, and the observation vector is a series of bits that represent the key information for that observation state. This process is crucial to the successful convergence of the learning agents, because this observation vector will also serve as the input tensor for the actor neural network that will determine the agent's policy. The Red and Blue observation vectors in the Competitive Wrapper are specific to the scenario used in this evaluation, and these vectors would need to be modified to correctly represent a different game design.

The Blue observation vector includes 5 bits to represent the state of each host (excluding the User0 and Defender hosts, since these are out of play). Each bit is a flag :

1. This device is not compromised.
2. This device was exploited last timestep.
3. The status of this device is unknown (the device has not been analyzed or restored).
4. Red has at least User privileges on this device.
5. Red has Root privileges on this device (device was analyzed).

Finally, the Blue observation vector also includes a 13 bit sequence to indicate how many timesteps are remaining in the scenario (from 12 to 0). An example of a complete Blue observation vector is shown in Figure 10. This is an observation returned by the Competitive Wrapper for turn 9. In this scenario, Red has previously established Root privileges on the User2 and User3 hosts, and has just exploited the Op Server. Blue has not analyzed either User host, so it is not known if Red has escalated beyond User privileges on these devices:

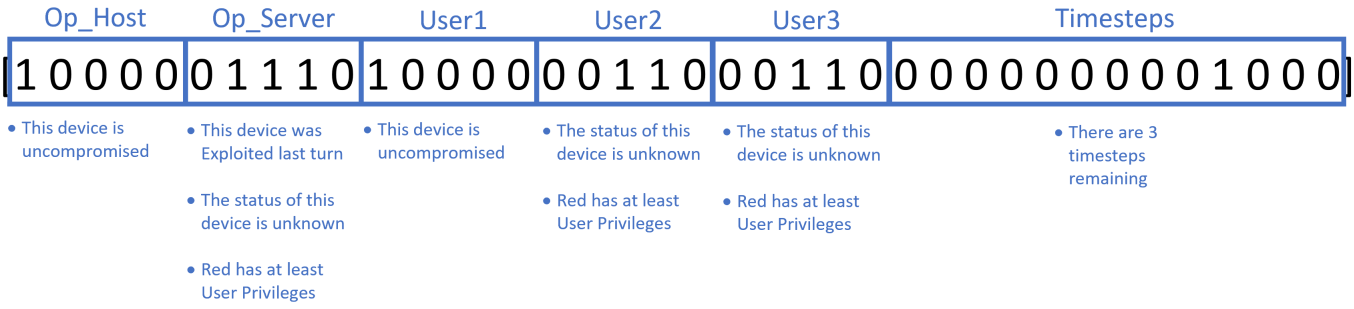


Figure 10: Labeled Example of a Blue Observation Vector.

Note that this observation vector does not provide a complete picture of the true state of the environment. It only encodes that information that the Blue agent has gathered. Blue receives some information passively each timestep, but detailed information about the state of a host requires the analyze action. For example, the Blue agent is alerted whenever a Red agent exploits a device based on the activity that is seen on the open port, but after this, Blue cannot passively observe what Red does on that device.

The only way Blue can be sure of the status of a compromised host, is to either analyze the host to learn Red's privileges, or restore the host to be sure that it is clean. Either of these actions requires a valuable timestep, during which Red will continue to attack the network.

The Red observation space is a little more complex. First, Red uses five bits to track which devices it has scanned. A device can only be scanned once Red knows its IP address. Red learns the User IP addresses by scanning the User Subnet, and as part of the scenario, Red discovers the Operations IP addresses once it has Root access to any User device.

Next, Red uses three bits to represent its privileges on any device. Either it has User, Root, or No privileges. Red uses 2 bits to track its progress scanning each subnet: if the IPs in a subnet are known, and if any device has been scanned for vulnerable ports. Red also uses 2 bits per subnet to track its highest level of access in that subnet: a bit for User access and another for Root access. Red uses a single bit to track whether it is currently impacting the Op Server. Finally, Red has 13 bits to track how many timesteps are remaining (from 12 to 0).

An example of a Red observation is shown Figure 11. This is extracted from the exact same simulation as the Blue vector shown in Figure 10 after the ninth timestep where Red has just exploited the Op Server.

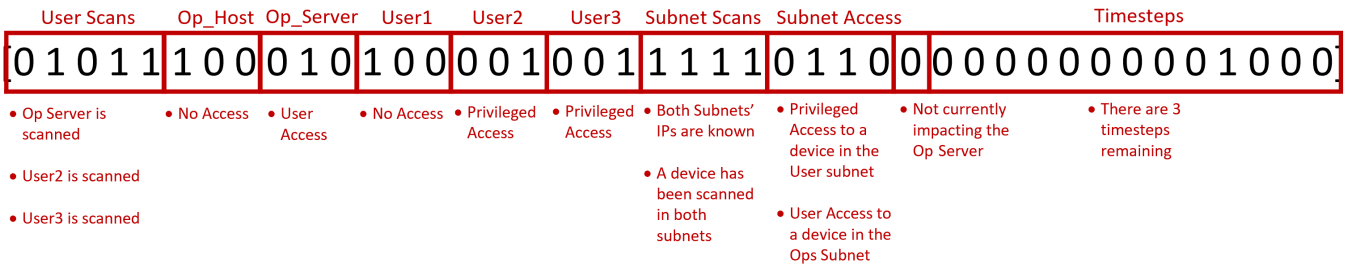


Figure 11: Labeled Example of a Red Observation Vector.

The Red agent has a much clearer picture than Blue as to the true state of the environment. There is very little Blue can do that Red cannot observe, because Red will see if it loses any privileges. However, Red does not know the IP addresses or vulnerable services of any device at the start of a scenario. Red must learn these by scanning devices on the network. This observation vector encodes whether that information is known or not, in order to determine what action Red should take next.

5.4. Fictitious Play Algorithm Implementation

Agents were trained through fictitious play to compete in this scenario. First, to train a minmax Blue policy, and then to train a minmax Red policy. In both cases, the Red and Blue agents were initialized with random policies (generation 0). Then, each iteration of the algorithm produced a new generation of policies.

Each generation, the Blue agent would use opponent sampling to optimize against the existing pool of Red policies, and the resulting Blue policy would be added to the Blue agent pool. Then, a new Red agent would be created and trained against the existing Blue agent pool. The distributions that were used for opponent sampling depended on which minmax policy was being solved, and these distributions were explained in greater detail in Chapter 4.

Each new generation produces a policy that optimizes its average score against the entire existing pool of opponents. Then, the opponent's best-response to that policy is trained and added to the pool. Fictitious play should continue for enough generations that the opponent's best-response scores stagnate near an asymptote. This indicates that the learning agent is producing near-optimal policies each generation, since new best-responses are being held near the minmax score. This occurs because each best-response policy that exploits the latest generation is used to train future generations towards a minmax policy.

For example, early in training, if no Red agent has targeted the Op Host yet, then no existing Blue policy will have learned to defend the Op Host. A new

Red best-response might discover that the best way to exploit the latest Blue policy is to target the Op Host. However, every subsequent generation of Blue agents will now learn from this Red policy and learn that the Op Host is trivial to protect with a remove action. Future Red agents will find that targeting the Op Host is less viable with each generation, because now the emerging Blue agents are trained to protect it, so these future Red agents optimize by finding other exploitable behavior.

This process must occur for every combination of actions that would exploit the existing agent pool. A diverse opponent pool ensures that later generations will experience every relevant branch of the game tree, and will never end up in a game state that was not seen during training and could exploit their policy.

For a relatively simple game such as this, the fictitious play was continued for 100 generations. By observing the policies discovered by these agents during training, it was clear that by 100 generations, all the most recent agents were producing similar stochastic policies. There were exceptions where the learning agent or opponent got stuck in a suboptimal local minima during training, but the majority of later generations followed the same priorities of eligible actions.

Each new generation was trained using a PPO algorithm from RLLib, an open-source reinforcement learning library from the Ray framework [44]. RLLib allows parallel sample collection during training. Using the Ray framework, it creates multiple "workers" that each play sample games in the environment. During a single iteration of training, workers repeatedly play games and collect samples until their total number of samples is enough for one complete batch. That batch is used to update the current policy via PPO, and the new policy is pushed to the workers. For this experiment, 40 Red workers and 40 Blue workers were used.

Two new OpenAI gym environments were designed to be compatible with the PPO algorithm included in RLLib: a BlueTrainer and a RedTrainer. Each training environment contains its own CybORG simulation as an attribute, which uses the competitive wrapper. Because Red and Blue each used 40 workers in parallel, this meant that 80 CybORG simulations were used simultaneously in this experiment. Increasing the number of workers any further caused the program to hang when it was run. 40 was the maximum number of parallel workers for the hardware used in this experiment.

RLLib require that the BlueTrainer and RedTrainer environments included a Step method and a Reset method, which will be called by the PPO algorithm during training. This Reset method is where opponent sampling occurs. At the start of each new game, the Reset method is called within the training envi-

ronment to generate a fresh scenario. Then, the environment checks how many policies currently exist in the opponent pool. It randomly selects one of these opponents, and stores that policy in the training environment as an attribute.

During the Step method, this sampled opponent policy will be used to determine what action is taken by the opponent. The learning agent is never aware of which opponent policy it is currently playing against, and so the PPO algorithm optimizes for performance against the entire opponent pool.

A generation of training ends when the learning agent is no longer improving: it has reached the best possible performance against the existing opponent pool. When the learning agent has finished converging, the new policy is saved as an RLLib checkpoint in a new directory for that policy. A pool_size file is also updated to indicate that a new policy has been added to the pool. When a worker starts a new game, they read the pool_size file for the opponents pool and then restores the saved checkpoint for the sampled opponent. The complete file structure used for this fictitious play implementation is shown in Figure 12.

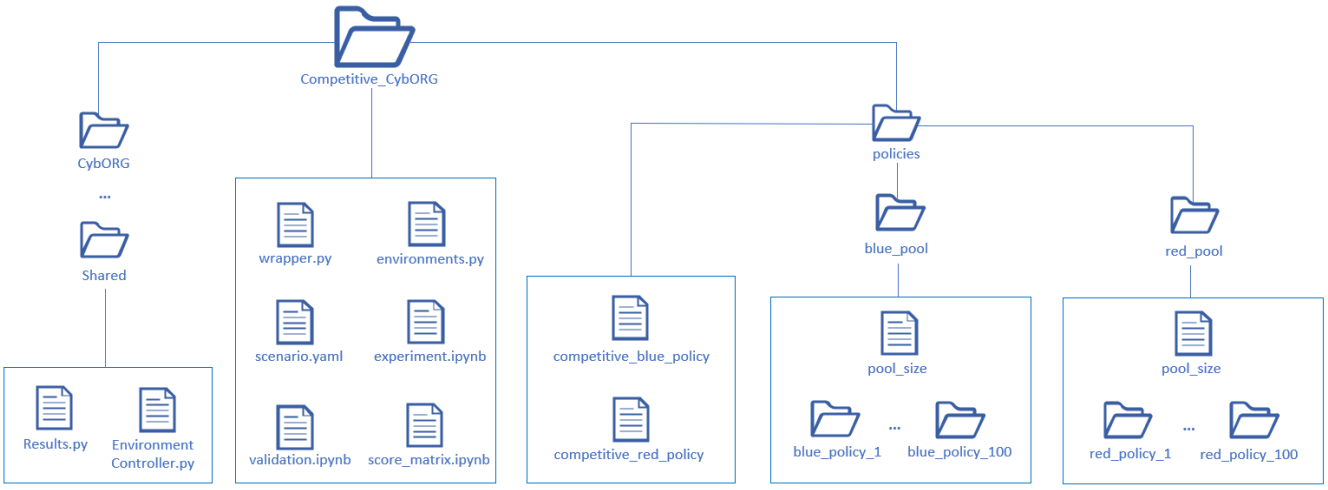


Figure 12: File Structure for Fictitious Play Implementation for CybORG.

So far, this chapter has described the key concepts used in this fictitious play implementation, and shown an overview of the project structure. In addition to the small changes made to the EnvironmentController.py and Results.py files, many brand-new processes were designed to facilitate this experiment. The environment wrapper was built by using the Blue Table Wrapper (included with the CybORG environment) as a template. This new competitive wrapper incorporates the new observation vectors for both agents, and includes a new method to resolve their discrete action choices.

The BlueTrainer and RedTrainer environments, included in environments.py,

were designed to train new policies for this scenario by using the RLLib PPO algorithm and interacting with the competitive wrapper. In addition to these Trainer environments, there are also the BlueOpponent and RedOpponent environments, which are used to create best-response opponents during the fictitious play loop. Finally, the DedicatedBlue and DedicatedRed environments are used to produce the dedicated opponent policies used during the validation. The only differences between the Trainer, Opponent, and Dedicated environments are the opponent sampling pools and distributions that are used during training.

The opponent sampling framework that is used by these environments was also designed from scratch for this experiment. Each generation, the newly trained actor and critic networks are saved in a directory that represents that agent's policy pool. Then, the pool_size file is updated so that any other environment that samples opponents from this pool can check the number of policies that it currently contains.

To conduct the experiment, the experiment.ipynb notebook trains the competitive agents and the best-response opponents, and then calculates the exploitability for every competitive policy relative to the other policies in their pool. The validation.ipynb notebook trains four dedicated opponents to play against each of the top performing competitive policies, in order to complete the validation. Finally, score_matrix.ipynb displays the expected scores between competitive, dedicated, and random agents by having each combination of agents play 1000 games to determine their average outcome.

Throughout the experiment, sample games between agents are used to examine the behavior of newly trained policies. The function to print a sample game is included in environments.py. This function can reveal each agent's action-probabilities for different observed states, and is crucial to examine why the optimal policies are able to achieve better scores in this scenario.

Table 2 shows the values for some parameters that were used during training. This table is limited to the key parameters that have been discussed in this chapter and in Chapter 4. Additional design details, experiment parameters, and more can be found in the source code for this project at https://github.com/RMC-AIvsAI/CybORG-Competitive. The remainder of this chapter will discuss the outcome of this evaluation.

Fictitious Play Parameters	
Generations	100
Tolerance	3
Best-Response Mixing Parameter	0.9
Minmax Evaluation Games	50
PPO Parameters	
Hidden Layers	2
Activation Function	tanh
Units per Hidden Layer	256
Batch Size	61440 (5120 Games)
Learning Rate	1e-3
Discount Factor	0.99
Parallel Workers	40

Table 2: Key parameter settings that were discussed in this thesis. Additional parameters and configuration settings that were used in this experiment can be examined with the source code at https://github.com/RMC-AIvsAI/CybORG-Competitive.

5.5. Results and Analysis

Training was run on a single machine with an NVIDIA Quadro P2000 GPU. One generation typically took about 30 minutes of training to produce a new pair of Blue and Red policies. Training a new Blue competitive policy took between 22 and 62 iterations of training, whereas training a new Red competitive policy ranged between 27 and 60 iterations.

One single iteration updates the policy using a batch of 61440 samples. At 12 timesteps per game, 5120 games were played each iteration. Therefore, we can observe that a new Blue policy required at least 113k games to converge, and a new Red policy required at least 138k games.

Policies used separate actor and critic neural networks. Each of these consisted of two fully-connected hidden layer, with 256 units and *tanh* activation functions. Both used multi-discrete input tensors to match the observation vectors used by each agent, and the actor networks used a softmax output across each agents action space. This architecture is shown in Figure 5.

5.5.1 Training Scores

The Blue minmax policy was trained first. 100 generations of policies were trained, and the average score during the final batch of training was recorded for every new generation. These training scores are shown in Figure 13. Examining these training scores gives a first indication that the agent policies have converged on meaningful scores.

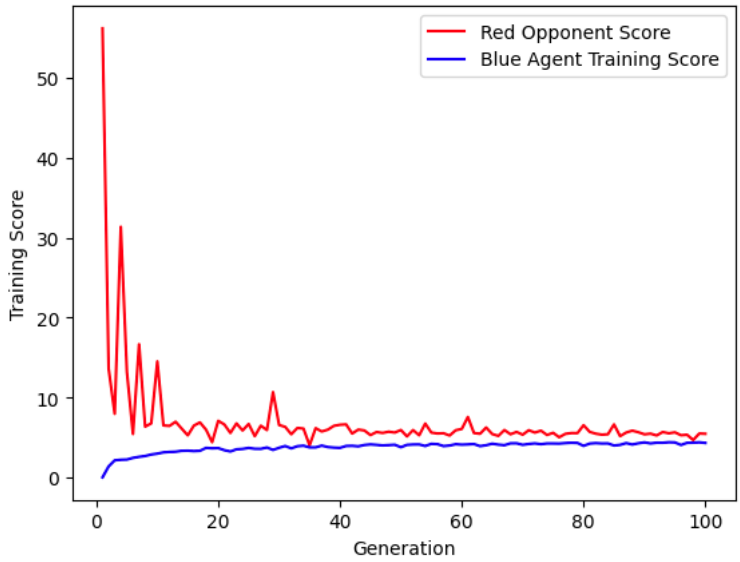


Figure 13: Training scores for fictitious play, solving for the Blue minmax policy. Average scores are taken for each generation when that policy finished training.

The first generation for both Red and Blue was optimized against an untrained opponent, so the scores from this generation are outliers compared to the rest of the data. The first Blue generation achieved an expected score of 0.02, and the first Red generation achieved an expected score of 56.20. These results are approaching the minimum and maximum possible scores for the game.

The minimum score occurs because the Blue agent has learned to never use a restore action, and the untrained Red agent is almost never able to exploit a User host because it is selecting random actions. The maximum score occurs because the Red agent has learned to reach the Op Server and impact as quickly as possible, and the only trained Blue policy has not learned to take any mitigating actions.

Both agents improved substantially in the first few generations of training, as these agents converged on obvious policies that exploited their opponents by exploring new branches of the game tree. After this, improvements across generations became much slower and less consistent.

The same general progression of scores can be seen when fictitious play is used to solve the Red minmax policy, as seen in Figure 14. A key difference is that the agents here have converged to different values. The Blue opponents here should be held at or near the Red minmax score. Whereas, when the Blue competitive policy was trained, the Red opponents in Figure 13 were held at or near the Blue minmax score.

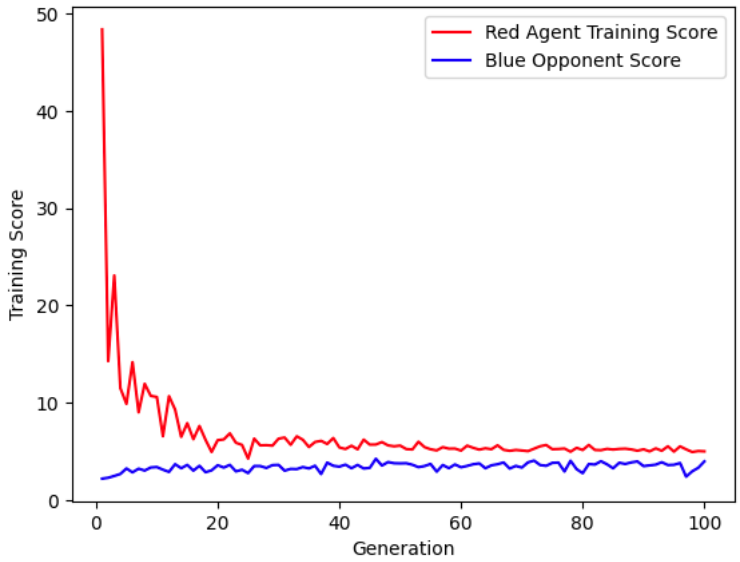


Figure 14: Training scores for fictitious play, solving for the Red minmax policy. Average scores are taken for each generation when that policy finished training.

The Red agent scores gradually decrease during training as the quality of agents in the Blue pool increased. Conversely, Blue scores gradually increased as new Blue policies were forced to compete with more performant Red agents. This is true for both instances of fictitious play, regardless of which minmax policy is being solved, which is why Figures 13 and 14 have similar shapes.

As training continues, new policies emerge with less and less variance between them. Many policies in the later generations achieve similar scores using very similar policies. This training state, where new agents are converging on the same policies without an incentive to deviate, indicates a Nash Equilibrium.

The rate at which these scores converge decreases over time. This is partially because there is gradually less room for improvement, but also because each new policy added to the pool contributes less to new samples with each passing generation. For example, at generation 10 a new Red opponent will contribute to 10% of the samples used for training the next Blue Agent, but at Generation 50 a new Red policy will only be used for 2% of the samples. So even though newer policies are better over time, they influence the existing pool less and less.

This is a limitation of uniform sampling, and possible alternatives to this will be suggested for future work in Chapter 6.

This is also the most likely reason that the opponent scores appear to be more volatile in both cases. Scores for the learning agent become relatively stable, as each generation is trained against an opponent pool that is mostly the same. However, the opponent scores continue to vary since each best-response is trained against a specific policy. Even minor changes in a policy, such as the exact distribution of probability between two state-actions, can have an effect on the opponents best-response.

5.5.2 Discussion

Sample games were used to observe the learned policy of the competitive agents. For brevity, these sample games are placed in Appendix A of this document, but they will be referenced throughout the results.

The first interesting behavior observed in these games, is that the Blue agent never prioritized the analyze action, and only ever used the remove action to defend the Op Host. This is a great example showing how the optimal policies can indicate problems with a scenario or environment. The information provided by the analyze action is never useful to the Blue agent in this scenario. The information that the competitive Blue agent receives passively is enough to make informed decisions, without ever wasting a timestep on an analyze action.

Similarly, this indicates the issue with the remove action in this environment. It can be seen during training that the remove action is selected any time the Red Agent has exploited the Op Host, because it is effective at removing the compromised User before Red can escalate to Root privileges on the machine. However, the remove action had a very low probability of success when it was used on any User host or the Op Server, which is why it is not a significant tool used by the optimized policy.

Of the 16 discrete actions in the Blue action space, only 7 options were significant to trained policies: the restore actions for the 5 devices, the remove action for the Op Host, or selecting a non-restore action. Since the Op Host was easily defended, the trained Red agents learn to avoid it. The Blue agent policies seen in sample games were typically distributions across all the non-restore actions, except in the cases where a restore action was viable. For this reason, in Appendix A, the Blue policy is only shown if there was a non-zero probability of selecting a restore action. Otherwise, the Blue policy was simply to never restore a device for that state, and only the Red agent's action had an effect.

The training scores for both instances of fictitious play suggest that the com-

petitive Red and Blue agents maintain different minmax scores, as expected. Red opponents typically scored above 5 points against the optimized Blue agents, but Blue opponents were typically able to hold optimized Red agents below 4 points. The exact exploitability of either agent will be confirmed in the validation.

The expected score when two competitive agents play against each other should land somewhere within this range. A dedicated opponent can reach the minmax score, but only because doing so would be risky against other policies. To explore this idea further, consider the sample game shown in Appendix A: Sample Game 1. This game is between a Red and Blue policy that were each trained through fictitious play.

For many states, Red will select a single viable action with over 99% certainty, and Blue will maintain a less than 0.1% chance of selecting any restore action. However, these agents will instead use stochastic policies at crucial decision points during the game, in order to be less predictable and increase their average performance against the entire opponent pool.

The most obvious occurrence of this, that appears in almost every sample game, is just after Red has exploited the Op Server. In Sample Game 1, Red exploits the Op Server on turn 8, and both agents reach a key decision point. The Red agent could gain two points by escalating privileges on the Server, which will score a point now and another point next turn when Blue is forced to restore the machine (to avoid a Red impact action).

The precise rules of the simulation become important here, because both Blue and Red need to make their selection simultaneously each timestep, without knowing the other's action. This has an important implication, because the Blue action resolves first every turn, and then the Red action resolves. This means that, if on turn 9, Red tries to escalate privileges but Blue restores the Op Server, Blue's action will resolve first and Red will lose its foothold on the Server, causing the privilege escalation to fail, and Red will have wasted a turn.

Early in training, Red best-responses do adopt this strategy of always trying to escalate privileges in this situation. But as the Red opponent pool fills with policies that always escalate the Op Server, new generations of Blue policies learn to minimize the score by restoring the server as soon as its exploited. Red wastes their privilege escalation action, because it no longer controls the server.

Blue is equally exploitable with this deterministic policy, because new Red policies stop trying to escalate privileges, and defeat the new Blue policies by exploiting the server every turn, forcing Blue to restore every single turn. These deterministic policies form a strategic cycle, shown in Figure 15, where any deterministic policy can be exploited by certain opponents.

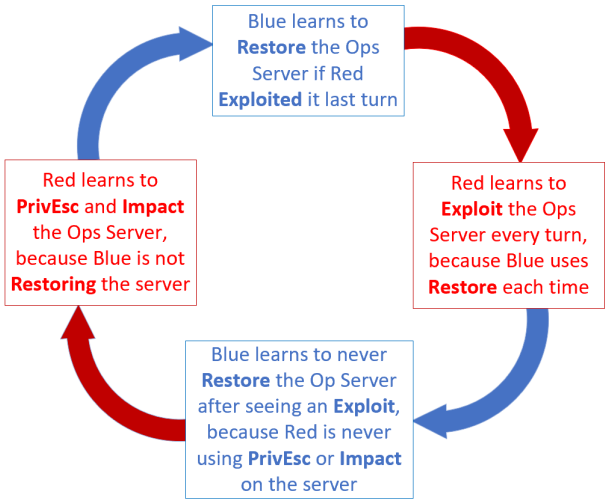


Figure 15: The open ended learning problem for this scenario. Any deterministic policy can be exploited. Only an optimal stochastic policy is non-exploitable.

Fictitious play, with opponent sampling, addresses this problem and forces new policies to become less exploitable across training. The later generations converge on policies that mix these strategies stochastically, they no longer converge on any one deterministic policy. Once the opponent pool has enough variety, new agents converge towards multiple viable strategies proportional to how effective those strategies are against the entire opponent pool.

This section has discussed a key decision point after Red exploits the Op Server, but this is just one example from this scenario. A similar strategic cycle occurs on a User host after Red has exploited it. Even though the User hosts are only worth 0.1 point each, Blue can delay Red reaching the Op Server by guarding these hosts. As a result, Blue will usually have some probability of restoring a User host after Red has exploited it. This forces the competitive Red agent to mix between escalating privileges (required to learn the IP address of the Op Server) or scanning or exploiting another User host, because either choice would be predictable on its own.

The final decision point occurs when Red is selecting which User host to scan and exploit. This probability distribution varied widely across different policies, but in later generations Red typically maintained a non-zero probability of selecting each host. Red's decision here does not effect the exploitability of either agent. The difference between User hosts is arbitrary, because whichever host Red chooses to scan and exploit, Blue is alerted and can take actions to protect the target host. Most likely, Red converges to a random distribution across which hosts to target, because all options are equally viable.

The actions of the Red and Blue agents at these three decision points typically shape the entire game tree for a single scenario. For all other observed states, the agents are consistent with just one optimal action. By observing the learned behaviour of these optimized agents, through sample games, it is clear why there was a substantial initial improvement as agents converged on strong deterministic policies, and then a much longer gradual improvement as agents became less exploitable by stochastically mixing between viable strategies.

It is also important to acknowledge that the scores did not improve consistently every generation. Although the learning agent's scores tended to become smaller over time, this was not the case in every generation. This is a reminder that these policies are approximations, and each new policy is relying on other approximations to train. The improvement across generations might have been more consistent with even larger batch sizes, more tolerance, and perhaps other combinations of neural network hyperparameters, all of which would have resulted in longer training time. Other possibilities for quality control features, such as forward searches and non-uniform opponent sampling, will be discussed in the final chapter of this thesis.

Even if they were not always consistent, the convergence in Figures 13 and 14 shows that the agents are objectively improving against the opponent pool. This is the expected behavior for agents converging towards optimal performance. However, the gradual improvement against the opponent pool does not prove that they have converged on optimal policies. The research question still needs to be answered: did fictitious play produce policies that accurately approximate optimal play?

In the next section, the validation will determine if the exploitability of newer generations improved over time, to see if the agents were objectively improving their guaranteed scores against any opponent. This will confirm that the agents are not converging towards an arbitrary score, but are indeed achieving the goal of increasing minmax performance across generations.

5.6. Validation

5.6.1 Measuring Exploitability

The first component of validation involves confirming that newer generations are gradually less exploitable. This is perhaps the most important component of the experiment, because it will show that the improvement in performance against the opponent pool is not arbitrary, and that the agents are improving their minmax performance throughout training.

To measure a policy's exploitability, the worst-case performance for every policy must be found first. The worst-case performance is the maximum score that a Blue policy allows, and the minimum score that a Red policy achieves. Recall from Equation 7 that this is required to calculate exploitability, since exploitability measures the difference between this score and what the agent could have achieved by following an optimal policy. Equation 8 shows the process for finding this worst-case score. Each competitive policy played 50 sample games against every policy in the opponent pool. This determined its expected score against every available opponent. The worst-case score for that generation is equal to the worst expected score against any opponent. In other words, a Blue agent's maximum expected score across all possible opponents, or a Red agent's minimum expected score across all possible opponents. This is the individual minmax score for any policy.

$$\begin{aligned} \mathbb{E}[G \mid \pi^{blue}, \pi_*^{red}] &= \max_{\pi^{red}} \frac{\sum_{k=0}^{50} G(\pi^{blue}, \pi^{red})}{50} \\ \mathbb{E}[G \mid \pi_*^{blue}, \pi^{red}] &= \min_{\pi^{blue}} \frac{\sum_{k=0}^{50} G(\pi^{blue}, \pi^{red})}{50} \end{aligned} \tag{8}$$

Once the every Blue agent's maximum expected score has been found, and every Red agent's minimum expected score has been found, the relative exploitability is calculated using Equation 9. Since the true game theory optimal policy is unknown, the Blue and Red agents with the best minmax scores are used as optimal (benchmark of 0 exploitability) and the exploitability of every other agent it calculated relative to these scores.

$$\begin{aligned} expl(\pi^{blue}) &= \mathbb{E}[G \mid \pi^{blue}, \pi_*^{red}] - \mathbb{E}[G \mid \pi_*^{blue}, \pi_*^{red}] \\ expl(\pi^{red}) &= \mathbb{E}[G \mid \pi_*^{blue}, \pi^{red}] - \mathbb{E}[G \mid \pi_*^{blue}, \pi_*^{red}] \end{aligned} \tag{9}$$

Once the relative exploitability of every competitive policy is known, their exploitability across training is plotted in order to observe if there was any trend. The agents are converging towards optimal play if the Blue agents are gradually learning to lower the maximum Red reward, and Red agents are learning to increase their minimum guaranteed reward. This can be confirmed using a regression line, in both cases, the exploitability should trend towards 0. The relative exploitability for Blue agent policies for each generation is shown in Figure 16.

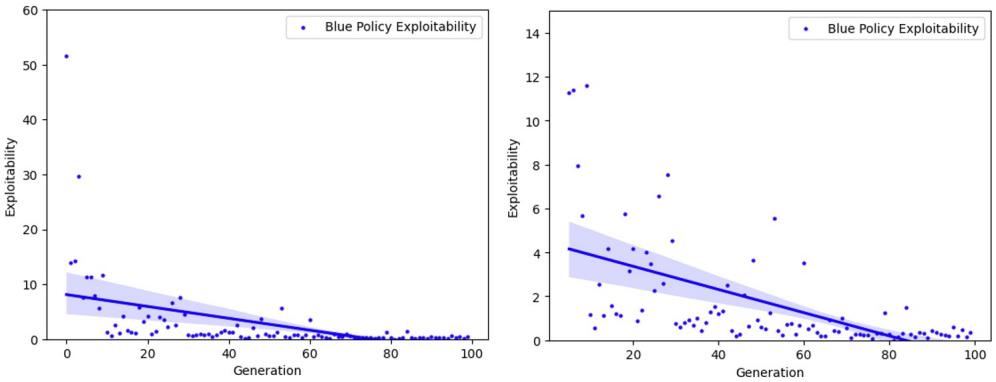


Figure 16: Each Blue generation's relative exploitability (left). This data is shown again with the first five generations dropped (right) to adjust the scale and show the progression of the later generations more clearly.

Plotting the relative exploitability of Blue policies reveals that there was a dramatic improvement in the first few generations, then a much more gradual improvement into the later generations. By generation 80, most new policies seem to be approaching 0 exploitability. It would appear that this is where a Nash Equilibrium occurred, where each new Blue agent is producing similar policies and the Red best-response can only approximate the minmax score each generation. The top performing Blue agent actually occurs at generation 82, with a minmax score of 5.39.

This proves that the Blue agents are indeed converging towards optimal policies through fictitious play, as their minmax score improves and they become less exploitable across generations. The relative exploitability of Red agent policies is shown in is shown in Figure 17.

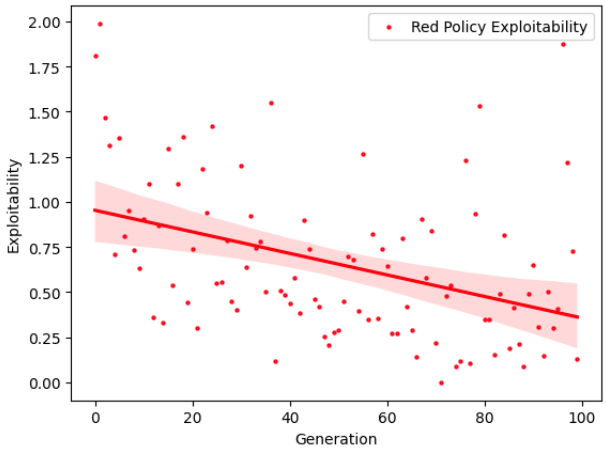


Figure 17: Each Red generation's relative exploitability.

Red agent performance is much less consistent across training, although the regression line does still confirm a general downward trend. The scale on this graph, when compared to the Blue agent, is crucial to understanding why this occurs. Although the exploitability of Red agents appears to be far more volatile across generations, consider that every policy falls within a range of just two points. By observing sample games from the early generations, it is clear that a Red agent can score very well by simply selecting actions to move directly to the Op Server and exploiting any device that is restored by Blue.

Like the Blue agent, the Red agent can reduce its exploitability by adopting non-predictable behavior and using a stochastic strategy at key decision points, but unlike the Blue agent, this only results in a marginal improvement for Red. The Blue and Red agents were trained using the same PPO network architectures and hyperparameters. Both would occasionally converge on local minima that did not represent their optimal policy during an individual generation. Based on some of the (relatively) high exploitability that still occurs in late Red generations, it would appear that the Red agent is more likely to converge on a local minima in this game design.

Regardless, and most importantly, the Red agent does also demonstrate a gradual decrease in exploitability across training, as shown by the regression line. The top performing Red agent occurs at generation 72, with a minimum expected score of 3.97. It is possible that additional generations would have gathered more consistently at the minmax score with additional training. Although these results are less dramatic than the improvement shown by the Blue agents, the Red agents are also converging towards the optimal minmax policy.

5.6.2 Minmax Evaluation

The next step in validation is to confirm exactly how accurately each agent has approximated a non-exploitable policy. To do this, the most performant Red and Blue agent policies are selected, based on their scores in the minmax evaluation. These are Blue agent 82, whose highest expected score across every Red opponent was 5.39, and Red agent 72, whose lowest expected score across every Blue opponent was 3.97. These two policies will be known as the competitive policies for the rest of the validation, because they represent the closest approximations of optimal policies that were produced by fictitious play.

These optimal policies will be evaluated against a dedicated opponent to measure their true minmax score. To do this, new dedicated opponents will be trained against each competitive policy using single agent RL. If the competitive agents are precisely optimal, then a dedicated opponent should only be able to

match their minmax scores. But if the agents are non-optimal then it should be trivial for a dedicated opponent to lower the Red agents score, or score more points against the Blue agent. This is because these dedicated agents are not trained against a pool of opponents, they are optimizing in a static environment where their only opponent uses a competitive policy.

Each dedicated opponent was trained for 100 iterations, using the same PPO algorithm and hyperparameters that were used for each generation of fictitious play. Recall that each point on the training graphs in Figures 13 and 14 represented the score of a policy that had finished converging, and that each of these individual policies converged in no more than 62 iterations of training. 100 iterations were used here to show that the dedicated opponents could not improve any further. Figure 18 shows the training scores for four separate dedicated agents that were trained in this way.

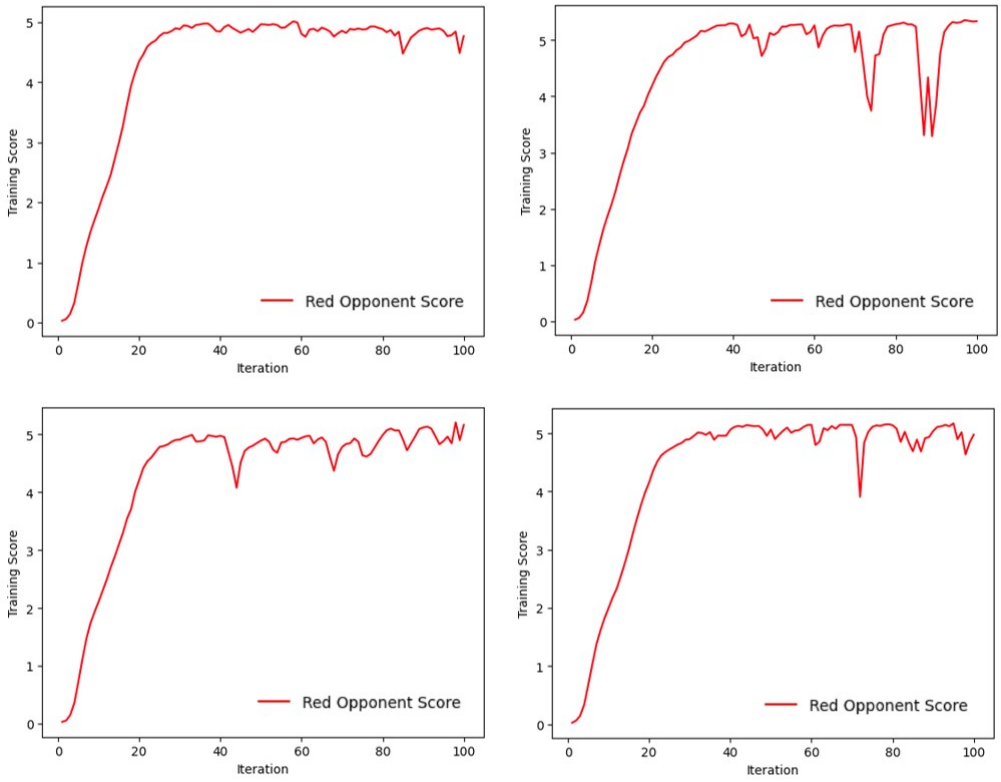


Figure 18: Dedicated Red opponent's average score each iteration, when training against the Competitive Blue agent, across four separate attempts.

The competitive Blue agent is able to hold the dedicated Red opponent to a score of 5.36, the policy that achieved this score is used as the dedicated Red opponent for the remainder of the evaluation. The true exploitability of the competitive Blue agent is only 0.03 more than the exploitability measured against

the opponent pool. Note that the training scores of the Red agents occasionally dip when they stagnate at the minmax score. This demonstrates the forgetting problem described in Chapter 4, and it is likely a consequence of hyperparameters that force the network to learn too quickly. A slower learning rate with a reduced clipping parameter would likely address this problem, but this would also drastically slow down training. Regardless, the learning agent recovers from any forgetting incidents in every attempt.

The behavior of the dedicated Red agent can be examined through sample games to see how it achieved this score. An example game between the competitive Blue agent and the dedicated Red opponent can be found in Appendix A - Sample Game 2. The most notable difference between this Red opponent and the competitive Red agent, is that this dedicated opponent has a deterministic policy in almost every state.

The dedicated opponent was trained using single-agent RL to defeat a single policy, and therefore, it no longer benefits from being unpredictable at key decision points. Instead, it has converged on the actions that are the ideal responses to the most likely Blue actions in every state. This makes the dedicated Red opponent a worst-case opponent for the competitive Blue policy. However, this also means the dedicated Red opponent would also be much more exploitable than the competitive Red policy.

A similar outcome is observed as the dedicated Blue opponents are trained against the competitive Red agent in Figure 19.

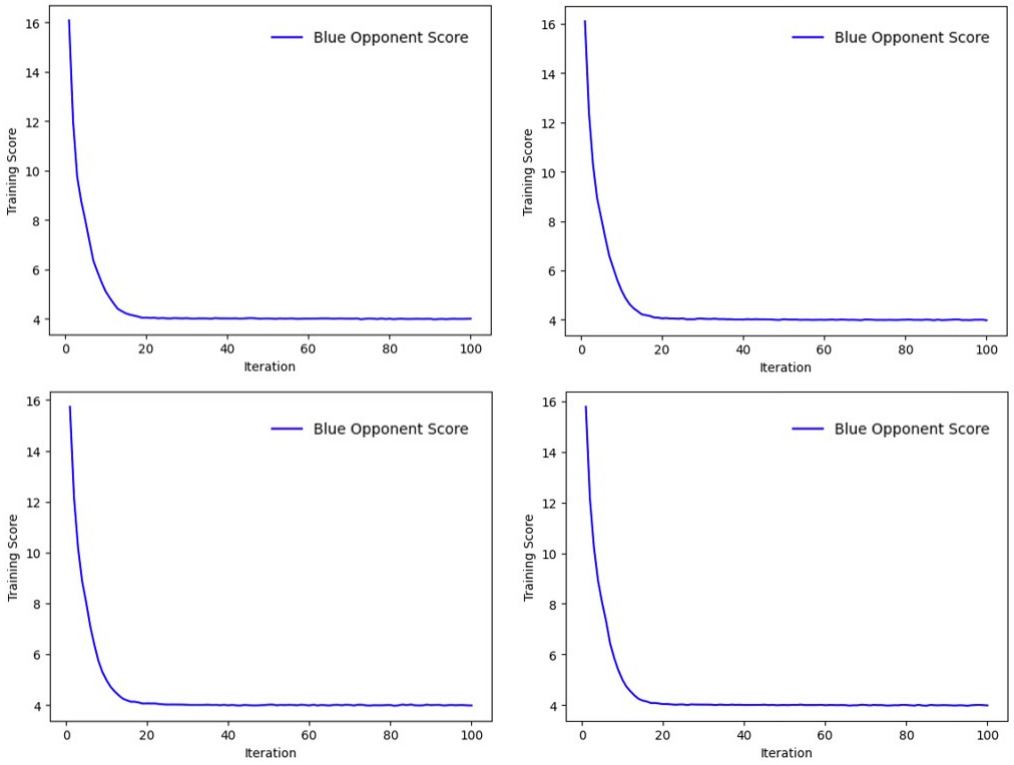


Figure 19: Dedicated Blue opponent's average score each iteration, when training against the Competetive Red agent, across four separate attempts.

The dedicated Blue opponent is able to hold the competitive Red agent to a maximum expected score of 3.97. This score exactly matches the expected exploitability from the evaluation. It relies on many of the same strategies as the competitive Blue agent, but no longer benefits from any unpredictable behavior, and so it learns to hold the competitive Red agent to the minmax score. An example game between the competitive Red agent and the dedicated Blue opponent is found in Appendix A - Sample Game 3.

The behavior of these two dedicated opponents confirm that the competitive agents have correctly identified the correct set of actions for every state they encounter. They correctly distribute their action-probabilities at the key decision points where it is ideal to adopt a stochastic strategy, instead of a deterministic one. The competitive agents have correctly identified the viable set of actions for every state, with relatively low margin of error in their expected exploitability, confirming that they are accurate approximations of an optimal policy.

5.6.3 Expected Reward Comparison

For the final phase of validation, the expected score for every pair of competitive, dedicated, and random agents were compared. The expected scores were gathered by having each pair of agents play 1000 games and taking their average score. These scores are shown in Table 3, and confirm that the scores seen during training were accurate. They also show each agent's utility against an opponent selecting random actions.

	Competitive Red	Dedicated Red	Random Red
Competitive Blue	4.79	5.38	0.03
Dedicated Blue	4.02	4.98	0.03
Random Blue	15.43	12.24	4.06

Table 3: Expected Scores between agents used in this experiment.

The expected score between a Blue and Red agent selecting random actions is about 4.06 each game. It is very unlikely for the random Red agent to score any points at all against a Blue agent. A Red agent in this scenario has 19 discrete actions to select from. To compromise a User host and score just 0.1 point it needs to: scan the User subnet, scan a User host, exploit that User host, and possibly escalate privileges, in that order within 12 timesteps. The reason the expected score is about 4.0, is because the random Blue agent is choosing a restore action and taking a 1 point penalty 4 times each game on average.

It is interesting to note that the Competitive Red Agent outperforms the Dedicated Red agent by over 3 points against a Random opponent. Even though both are conservative scores, this result is another consequence of the dedicated Red agent being trained for just one specific opponent.

The 0.03 expected score that the trained Blue agents receive against a random Red opponent could be due to the Red agents rarely selecting a correct combination of actions to compromise a User host. But more likely, this indicates that the Blue agents still have a small probability of incorrectly selecting a restore action once every 30 to 40 games. This is a consequence of every policy being neural network approximations: no action probability is every truly zero. Even the competitive Red agent only scans the User subnet on 99.98% of the time on the first turn of the game. As previously discussed, more training time and larger batches could have likely improved this accuracy even further, and the final chapter of this thesis will discuss certain quality control techniques that could improve this accuracy in future research.

5.7. Evaluation Summary

Through this experiment, it is confirmed that fictitious play can be used in the CybORG environment to approximate the optimal policies for game-playing agents. The trained agents are able to identify the set of viable actions at each key decision point in the game tree, by using policies that were developed through fictitious play. This, more than any other outcome, demonstrates the potential for competitive RL in cyber incident simulation environments.

The exploitability of agents was shown to gradually decrease across training for both the Red and Blue agents, but these improvements were not always consistent every generation. With more generations of training, lower learning rates, or larger batch sizes, the new generations of agents could likely have shown more reliable improvement with each generation. However, this would have drastically slowed training, and the policies trained here did still correctly converge to approximate the optimal minmax policies.

Finally, a measurement of the expected score between trained agents showed that no neural network approximation trained in this way is 100% accurate. Even Red's first move of the game, which should be the same every time, has an error of 0.02%. The sample games included in Appendix A almost certainly leave room for small probability optimizations at the key decision points, that would marginally improve the minmax scores that were approximated here. The final chapter of this thesis will recommend some quality control techniques that could improve this accuracy even further in future research.

6. Conclusion

This chapter will conclude the thesis by emphasizing what this research contributes to the ACO domain. As well, it will highlight some open research questions that were encountered during this process and could be the aim of future work in the domain.

6.1. Thesis Contribution

This thesis is exploratory research to examine whether proven competitive RL algorithms could be equally effective in simulated ACO environments. The evaluation included in Chapter 5 provides a proof of concept that competitive RL, specifically fictitious play, could be a tool for finding game theory optimal policies in the eligible environments.

However, this experiment is limited by the realism of available ACO simulation environments. The environment examined in this thesis, like all open-source ACO environments at this time, is not realistic enough to contribute any meaningful tools to cybersecurity. An AI trained using RL in these environments could not assist with any real-world task, because many essential details for enacting these policies in the real world have been abstracted away in ACO simulators.

This is precisely why ongoing research for more realistic ACO simulations is so important. Existing literature, some of which was discussed in Chapter 3, has shown encouraging results to suggest that RL could be viable for training an AI to assist in the duties of a cybersecurity analyst defending a specific network. Like any RL application, the potential for these tools can only be truly explored once a sufficiently realistic ACO environment is available for simulation, such that any learned policy can be used to take action on a real-world network.

This thesis has no implication for existing cybersecurity tools or practices, but it does contribute to the continued development of realistic ACO environments, with the hope that these environment will eventually be accurate models that are used for developing cybersecurity tools. It contributes to the continued development of these environments in two important ways.

First, it provides a methodology that can still serve as a foundation for competitive RL in more realistic environments. The fictitious play design used in this thesis was based on existing competitive applications in other domains. Similarly, many of the details discussed in the methodology in Chapter 4 will stay the same regardless of environment complexity. In particular, the cycle of learning-from and contributing-to separate Blue and Red agent pools during opponent sampling

is irrelevant to the environment's realism, and this thesis has shown it is effective. As well, using exploitability to show convergence of ACO agents towards optimal policies is a new idea introduced in this thesis. This is a convenient metric to compare the quality of different policies in competitive ACO environments.

Second, this thesis provides a means for finding the optimal policies for other environments and scenarios that are available today. The optimal behaviors of competitive agents reveal insights about the realism of that environment and suggests priorities for continued development. Do the optimized agents match the expected behavior of a human analyst? Or, do they exploit some unrealistic feature of the environment? If optimized agents rely heavily on an unrealistic abstraction this suggests a priority for future development, because adding additional realism elsewhere in the environment will not change how learning agents abuse the most egregious flaws.

The competitive policies trained in Chapter 5 do abuse a number of abstractions in the CybORG environment to arrive at their optimized policies. Perhaps the most egregious, is that the CybORG environment models a cyber incident as taking place in a fixed number of timesteps. The competitive agents have learned behaviors that take advantage of the abstract nature of this scenario.

For example, a Red agent in the CybORG environment might choose to "exploit" a host that it already controls, and this is a viable strategy because the Blue opponent might simultaneously choose to "restore" the machine on the same timestep. Because the Blue action resolves first, Red maintains a foothold on the machine. Agents also rely on the termination time for the scenario. The Blue agent might freely allow Red to escalate privileges in the final timestep if this does not change the game's outcome.

Any Markov Game requires some sort of guaranteed termination condition for RL to take place. Otherwise, the estimated value of any state could be infinite: a learning agent could maximize its return by choosing to never terminate a scenario. Therefore, the termination conditions during training are as important as the reward function for shaping an agent's learned behavior.

A sufficiently motivated attacker could maintain a foothold for weeks or longer before attempting a single tactic. Or, that attacker could attempt to complete their cyber killchain in the span of hours. In modelling more realistic environments, developers will be forced to commit to a timescale that is less nebulous than "timesteps". This decision will require a more precise goal for the learning agent, that should be rooted in a well-defined real-world application for a trained AI.

The CybORG environment does not attempt to definite the timescale for the

simulated cyber incident, but in a more realistic environment, the specific application and scope of the AI will need to be more defined. The results of this thesis demonstrate why this would be a worthwhile priority for future development: the optimal policies of agents in this environment rely heavily on the discrete time model in a way that does not reflect real-world behavior.

6.2. Future Work

There are two broad categories of obvious potential research to build on this thesis: Exploring competitive RL in more complex and realistic ACO environments, and exploring more elegant competitive RL designs to improve on this algorithm's performance.

The methodology included in Chapter 4 is designed to support fictitious play in any eligible ACO environment. Any attempt to use fictitious play in an alternative ACO simulator, or even in a future release of CybORG, would be a useful contribution to the domain. The scenario in this thesis was intentionally simple as a proof of concept, even attempting this exact methodology in the same environment but with a more realistic network topology would help explore the potential and limitations for this algorithm. Comparing alternative scenarios (such as different network topologies and observable features) or alternative game designs (such as different reward functions and termination conditions) could provide a more nuanced critique for this application of fictitious play.

This thesis included a design for fictitious play in a relatively simple ACO scenario. However, as ACO environments become more realistic and complex, this design will almost certainly need to be adapted to rely on the less abstract features of those environments. Some of these adaptations can be investigated immediately. For example, the scenario used in this thesis included a simple observation vector. The CybORG wrappers make it easy for a learning AI to deduce whether a host was scanned, or targeted with an exploit. As well, when Blue analyzes a host, it is given the true state of the Red agent on that host. In this way, the game is played at an abstract tactical level. At the same time, CybORG does simulate each agents lower-level activity, such as what exploit was used on an available service with a known vulnerability. Future research will need to consider what a viable observation vector will look like in a realistic simulation, but the CybORG environment has enough details to explore more detailed, low-level observation vector immediately.

More elegant competitive RL designs are equally important for continued research in this domain. As discussed in Section 6.1, this research is primarily

meant to contribute to the continued development of more realistic ACO environments. Many existing competitive RL applications for other domains include quality-control techniques that ensure their computation time is spent efficiently. These quality-control techniques will become especially beneficial, and perhaps essential, in more complex ACO environments.

Perhaps the most obvious tool for a competitive AI (that was excluded from this thesis) is a forward search. Many other competitive applications use policies that are trained using fictitious play while employing some kind of forward search, such as a Monte-Carlo Search Tree (MCTS), to improve their accuracy. A forward search is, intuitively, how an agent spends time "thinking" about its next action instead of immediately using the output of the actor network.

To conduct a forward search, the trained AI must maintain a local copy of the simulation environment. Then, before committing to an action, it simulates out many possible game trees from the current state to a predetermined depth. This way, the agent is relying on value estimations at the depth of the search tree, instead of estimating value at the current state. A forward search could augment the algorithm from this thesis to improve the performance of a trained competitive agent in an ACO environment. Suitable research could involve training optimized agents using fictitious play and then comparing their performance with and without a search tool, in the most realistic available ACO simulation environment.

Another potential technique to improve the quality of competitive agents, or at least to reduce the time required to approximate the optimal policy, is more intelligent opponent sampling. In this research, uniform opponent sampling was used so that each new generation was trained against the average of all past opponents. This ensured gradual convergence as each new generation was encouraged to avoid the vulnerabilities of the previous generation, because of the newest opponent in the opponent pool.

However, as noted in Chapter 5, newer generations had a reduced impact on their existing pool over time. This is because, when using uniform sampling, early generations have a significant impact on the existing pool, but later generations only make a small change to the existing pool of policies. The scenario examined in this thesis was simple enough that uniform sampling could still be used to arrive at an optimal policy, but it is feasible that a more complex scenario (including another scenario in the CybORG environment) could not be solved by this method of uniform sampling. This might occur if new generations were having a negligible impact on their policy pools before a Nash Equilibrium had been reached. Uniform sampling should still lead to the optimal policies after

an infinite number of generations, but it could be computationally infeasible in a more complex ACO scenario.

More elegant sampling techniques have been used across a variety of competitive applications to improve training speed and performance. The most effective opponent sampling techniques are used today to optimize games, but these same techniques will likely be equally important for realistic ACO simulations. Quality based sampling is perhaps the technique that was cited most frequently during the literature review in Chapter 3 for highly complex game environments. Historic sampling is another option to ensure that new generations are trained against the most performant opponents, and ensures that even late generations of training have a significant impact on training future opponents. Any of these alternatives would introduce a new variation of fictitious play for ACO, and would be valuable research topics that provide another novel contribution to the domain.

6.3. Closing Remarks

This thesis provides a proof of concept that fictitious play can be used to solve a simple cybersecurity scenario in the popular ACO simulation environment, CybORG. However, even CybORG is not nearly realistic enough for the trained policies to be of any real-world use. The availability of sufficiently realistic training environments is the greatest challenge that prevents RL from creating ACO tools today. The algorithm demonstrated in this thesis can be used to discover optimal policies in eligible ACO environments. These optimal policies can suggest priorities for future development, by revealing which abstract features an optimized agent relies on the most.

Furthermore, this fictitious play algorithm can act as a foundation for solving complex scenarios in more realistic environments. When these environments become available, fictitious play should be tested with other quality-control techniques, such as a forward search and intelligent opponent sampling.

Competitive RL research is most often used to develop an AI that is highly performant in some sort of classical game. Games make excellent demonstration tools for these algorithms, but the value of this technology is in its potential to solve real-world optimization problems. Cybersecurity and ACO are domains where ongoing research in competitive RL could contribute to the development of real-world tools in the future.

Bibliography

[1] C. Zhong, J. Yen, and P. Liu, "Can cyber operations be made autonomous? an answer from the situational awareness viewpoint," *Adaptive Autonomous Secure Cyber Systems*, 2020.

[2] K. Kim, F. A. Alfouzan, and H. Kim, "Cyber-attack scoring model based on the offensive cybersecurity framework," *Applied Sciences 11 no. 16*, 2021.

[3] Mitre attck framework. (accessed: 22.03.2023). [Online]. Available: https://attack.mitre.org/

[4] K. Zhang, Z. Yang, and T. Basar, "Multi-agent reinforcement learning: A selective overview of theories and algorithms," *Handbook of Reinforcement Learning and Control, 321-384*, 2021.

[5] D. Silver *et al.*, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm," *arXiv:1712.01815*, 2017.

[6] M. Moravcik, M. Schmid, B. Lisy, D. Morrill, N. Bard, T. Davis, K. Waugh, M. Johnson, and M. Bowling, "Deepstack: Expert-level artificial intelligence in heads-up no-limit poker," *Science 356.6337*, 2017.

[7] A. Nguyen-Tuong *et al.*, "Xandra: An autonomous cyber battle system for the cyber grand challenge," *IEEE Security Privacy 16.2*, 2018.

[8] L. Li, R. Fayad, and A. Taylor, "Cygil: A cyber gym for training autonomous agents over emulated network systems," *IJCAI-21 1st International Workshop on Adaptive Cyber Defense*, 2021.

[9] Caldera. (accessed: 01.04.2023). [Online]. Available: https://github.com/mitre/caldera

[10] Microsoft. Cyberbattlesim. (accessed: 08.03.2023). [Online]. Available: https://github.com/microsoft/CyberBattleSim

[11] A. Molina-Markham, C. Miniter, B. Powell, and A. Ridley, "Network environment design for autonomous cyberdefense," *arXiv preprint*, 2021.

[12] M. Standen, M. Lucas, D. Bowman, T. J. Richer, J. Kim, and D. Marriot, "Cyborg: A gym for the development of autonomous cyber agents," *IJCAI-21 1st International Workshop on Adaptive Cyber Defense*, 2021.

[13] S. Vyas, J. Hannay, A. Bolton, and P. Burnap, "Automated cyber defence: A review," *arXiv preprint arXiv:2303.04926*, 2023.

[14] R. Sutton and A. Barto, *Reinforcement Learning, An Introduction*, 2nd ed. MIT Press, 2020.

[15] K. Leyton-Brown and Y. Shoham, *Essentials of Game Theory, a Concise Multidisciplinary Introduction*, 2008.

[16] C. Watkins and P. Dayan, "Q-learning," *Machine Learning 8.3*, 1992.

[17] J. Schulman and Others, "Proximal policy optimization algorithms," *arXiv preprint arXiv:1707.06347*, 2017.

[18] D. Balduzzi and Others, "Open-ended learning in symmetric zero-sum games," *International Conference on Machine Learning*, 2019.

[19] P. Sun and Others, "A framework for competitive self-play based distributed multi-agent reinforcement learning," *arXiv preprint arXiv:2011.12895*, 2020.

[20] F. Timbers and Others, "Approximate exploitability: Learning a best response in large games," *arXiv preprint arXiv:2004.09677*, 2020.

[21] M. Zinkevich *et al.*, "Regret minimization in games with incomplete information," *Advances in Neural Information Processing Systems 20*, 2007.

[22] G. Brown, "Iterative solution of games by fictitious play," *Act. Anal. Prod Allocation 13.1: 374*, 1951.

[23] J. Heinrich, M. Lanctot, and D. Silver, "Fictitious self-play in extensive-form games," *International conference on machine learning*, 2015.

[24] J. Heinrich and D. Silver, "Deep reinforcement learning from self-play in imperfect-information games," *arXiv preprint arXiv:1603.01121*, 2016.

[25] M. Jaderberg *et al.*, "Human-level performance in 3d multiplayer games with population-based reinforcement learning," *Science 364.6443*, 2019.

[26] C. Berner *et al.*, "Dota 2 with large scale deep reinforcement learning," *arXiv preprint arXiv:1912.06680*, 2019.

[27] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Transactions on Neural Networks and Learning Systems*, 2021.

[28] R. Elderman, L. Pater, A. Thie, M. Drugan, and M. Wiering, "Deep reinforcement learning for cyber security," *ICAART (2)*, 2017.

[29] J. Bland *et al.*, "Machine learning cyberattack and defense strategies," *Computers & security 92*, 2020.

[30] X. He, H. Dai, and P. Ning, "Faster learning and adaptation in security games by exploiting information asymmetry," *IEEE Transactions on Signal Processing 64.13*, 2016.

[31] Y. Han *et al.*, "Reinforcement learning for autonomous defence in software-defined networking," *Decision and Game Theory for Security: 9th International Conference*, 2018.

[32] Mininet Project. Mininet. (accessed: 09.03.2023). [Online]. Available: http://mininet.org/

[33] R. Maeda and M. Mimura, "Automating post-exploitation with deep reinforcement learning," *Computers Security 100*, 2021.

[34] P. Speicher *et al.*, "Towards automated network mitigation analysis," *Proceedings of the 34th ACM/SIGAPP symposium on applied computing*, 2019.

[35] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba. (2016) Openai gym. https://github.com/openai/gym. (accessed: 23.02.2023).

[36] "Cyber autonomy gym for experimentation challenge 1," https://github.com/cage-challenge/cage-challenge-1, 2021, created by Maxwell Standen, David Bowman, Son Hoang, Toby Richer, Martin Lucas, Richard Van Tassel.

[37] J. Schwartz and H. Kurniawatti, "Nasim: Network attack simulator," https://networkattacksimulator.readthedocs.io/, 2019.

[38] O. Vinyals *et al.*, "Grandmaster level in starcraft ii using multi-agent reinforcement learning," *Nature 575.7782*, 2019.

[39] Y. Seo. Laser tag pytorch nfsp. (accessed: 13.03.2023). [Online]. Available: https://github.com/younggyoseo/pytorch-nfsp

[40] E. Steinberger, "Pokerrl," https://github.com/TinkeringCode/PokerRL, 2019, (accessed: 30.03.2023).

[41] A. Goldwaser and M. Thielsher, "Deep reinforcement learning for general game playing," *Proceedings of the AAAI conference on artificial intelligence. Vol. 34. No. 02*, 2020.

[42] J. Clark and D. Amodei, "Faulty reward functions in the wild," *Internet: https://blog.openai.com/faulty-reward-functions*, 2016.

[43] D. Roijers *et al.*, "A survey of multi-objective sequential decision-making," *Journal of Artificial Intelligence Research 48*, 2013.

[44] E. Liang *et al.*, "Rllib: Abstractions for distributed reinforcement learning," *International Conference on Machine Learning*, 2018.

[45] M. Foley *et al.*, "Autonomous network defence using reinforcement learning," *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, 2022.

[46] M. Andrychowicz *et al.*, "What matters in on-policy reinforcement learning? a large-scale empirical study," *arXiv preprint arXiv:2006.05990*, 2020.

A. Sample Games

This appendix contains example games between trained agent policies. Each of these games is referenced as part of the evaluation in Chapter 5. The three games included here are labeled based on their corresponding policy from the evaluation. The "competitive" agents are the Red and Blue policies generated through fictitious play. The "dedicated" agents were trained using single-agent RL to optimize against the competitive policies, as part of the experiment validation.

These games show the raw text file output for each game. When these samples are created, they include:

1. The Blue agent action probabilities.
2. The Blue action that was selected.
3. The Red agent action probabilities.
4. The Red action that was selected
5. A table to show the new Red observation
6. The reward for that timestep,
7. The new score.

The Red agent action space contains 19 discrete actions, and for most timesteps the majority of actions had a probability less than 0.02%. Rather than show the full list of action probabilities for the Red agent, Red action probabilities are only listed if they had at least a 1% chance of being selected.

As discussed in Chapter 5, it was observed that restore actions were the only Blue actions that influenced the game progression. As a result, Blue often maintained a 0% chance of selecting any restore action, and its action probabilities were distributed among the remaining actions. If the Blue agent policy supported a restore action, then a certain amount of probability would be allocated to that action, and the remaining probability would still be distributed among the non-restore actions.

For the sake of brevity, instead of printing an additional 16 lines for every timestep, the Blue agent policy is only shown if there was at least a 1% chance of selecting a restore action. Otherwise, the Blue agent action-probabilities and the selected action are hidden, as neither impacted the progression of that game.

A.1. Competitive Blue versus Competitive Red

```
---- Turn 1 ----

Red Policy:
Red selects ('DiscoverSystems', 'User') with probability 99.98%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.213.112/28 | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.213.112/28 | 10.0.213.119 |      User0      |  False  | Privileged |
| 10.0.213.112/28 | 10.0.213.120 | UNKNOWN_HOST: 0 |  False  |    None    |
| 10.0.213.112/28 | 10.0.213.121 | UNKNOWN_HOST: 1 |  False  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +-0.0
Score: 0.0


---- Turn 2 ----

Red Policy:
('DiscoverServices', 'User1'): 88.57%
('DiscoverServices', 'User2'): 9.98%
('DiscoverServices', 'User3'): 1.39%
Red selects ('DiscoverServices', 'User1') with probability 88.57%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.213.112/28 | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.213.112/28 | 10.0.213.119 |      User0      |  False  | Privileged |
| 10.0.213.112/28 | 10.0.213.120 | UNKNOWN_HOST: 0 |   True  |    None    |
| 10.0.213.112/28 | 10.0.213.121 | UNKNOWN_HOST: 1 |  False  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +-0.0
Score: 0.0


---- Turn 3 ----

Red Policy:
Red selects ('ExploitServices', 'User1') with probability 99.65%

+-----------------+--------------+-----------------+---------+------------+
```

```
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.213.112/28 | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.213.112/28 | 10.0.213.119 |      User0      |  False  | Privileged |
| 10.0.213.112/28 | 10.0.213.120 |      User1      |   True  | Privileged |
| 10.0.213.112/28 | 10.0.213.121 | UNKNOWN_HOST: 1 |  False  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.1


---- Turn 4 ----

Blue policy:
('Restore', 'User1'): 13.36%
Blue selects ('Analyse', 'User3') with probability 5.26%

Red Policy:
('DiscoverServices', 'User2'): 82.49%
('DiscoverServices', 'User3'): 9.65%
('PrivilegeEscalate', 'User1'): 7.61%
Red selects ('DiscoverServices', 'User2') with probability 82.49%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.213.112/28 | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.213.112/28 | 10.0.213.119 |      User0      |  False  | Privileged |
| 10.0.213.112/28 | 10.0.213.120 |      User1      |   True  | Privileged |
| 10.0.213.112/28 | 10.0.213.121 | UNKNOWN_HOST: 1 |   True  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.2


---- Turn 5 ----

Red Policy:
Red selects ('ExploitServices', 'User2') with probability 99.88%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.213.112/28 | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.213.112/28 | 10.0.213.119 |      User0      |  False  | Privileged |
```

```
| 10.0.213.112/28 | 10.0.213.120 |      User1      |   True  | Privileged |
| 10.0.213.112/28 | 10.0.213.121 |      User2      |   True  | Privileged |
+-----------------+--------------+-----------------+---------+------------+

Reward: +0.2
Score: 0.4


---- Turn 6 ----

Blue policy:
('Restore', 'User1'): 10.92%
('Restore', 'User2'): 22.88%
Blue selects ('Analyse', 'Op_Server0') with probability 7.17%

Red Policy:
('PrivilegeEscalate', 'User1'): 90.81%
('PrivilegeEscalate', 'User2'): 8.89%
Red selects ('PrivilegeEscalate', 'User1') with probability 90.81%

+------------------+--------------+-----------------+---------+------------+
|       Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+------------------+--------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 5 | 10.0.182.20  | UNKNOWN_HOST: 6 |  False  |    None    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+------------------+--------------+-----------------+---------+------------+

Reward: +0.2
Score: 0.6


---- Turn 7 ----

Blue policy:
('Restore', 'User2'): 1.74%
Blue selects ('Remove', 'User2') with probability 27.36%

Red Policy:
Red selects ('DiscoverServices', 'Op_Server0') with probability 99.90%

+------------------+--------------+-----------------+---------+------------+
|       Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+------------------+--------------+-----------------+---------+------------+
```

```
| UNKNOWN_SUBNET: 5 | 10.0.182.20  | UNKNOWN_HOST: 6 |   True  |    None    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+-------------------+--------------+-----------------+---------+------------+

Reward: +0.2
Score: 0.8


---- Turn 8 ----

Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 99.92%

+-------------------+--------------+-----------------+---------+------------+
|       Subnet      |  IP Address  |     Hostname    | Scanned |   Access   |
+-------------------+--------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 5 | 10.0.182.20  |    Op_Server0   |   True  |    User    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+-------------------+--------------+-----------------+---------+------------+

Reward: +0.2
Score: 1.0


---- Turn 9 ----

Blue policy:
Blue selects ('Restore', 'Op_Server0') with probability 90.59%

Red Policy:
('ExploitServices', 'Op_Server0'): 19.38%
('PrivilegeEscalate', 'Op_Server0'): 80.10%
Red selects ('PrivilegeEscalate', 'Op_Server0') with probability 80.10%

+-------------------+--------------+-----------------+---------+------------+
|       Subnet      |  IP Address  |     Hostname    | Scanned |   Access   |
+-------------------+--------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 5 | 10.0.182.20  |    Op_Server0   |   True  |    None    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
```

```
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+-------------------+--------------+-----------------+---------+------------+

Reward: +1.2
Score: 2.2


---- Turn 10 ----

Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 99.94%

+-------------------+--------------+-----------------+---------+------------+
|       Subnet      |  IP Address  |     Hostname    | Scanned |   Access   |
+-------------------+--------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 5 | 10.0.182.20  |    Op_Server0   |   True  |    User    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+-------------------+--------------+-----------------+---------+------------+

Reward: +0.2
Score: 2.4


---- Turn 11 ----

Blue policy:
Blue selects ('Restore', 'Op_Server0') with probability 84.72%

Red Policy:
('ExploitServices', 'Op_Server0'): 98.84%
('PrivilegeEscalate', 'Op_Server0'): 1.03%
Red selects ('ExploitServices', 'Op_Server0') with probability 98.84%

+-------------------+--------------+-----------------+---------+------------+
|       Subnet      |  IP Address  |     Hostname    | Scanned |   Access   |
+-------------------+--------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 5 | 10.0.182.20  |    Op_Server0   |   True  |    User    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
```

```
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+-------------------+--------------+-----------------+---------+------------+

Reward: +1.2
Score: 3.6


---- Turn 12 ----

Blue policy:
Blue selects ('Restore', 'Op_Server0') with probability 15.35%

Red Policy:
Red selects ('PrivilegeEscalate', 'Op_Server0') with probability 97.16%

+-------------------+--------------+-----------------+---------+------------+
|       Subnet      |  IP Address  |     Hostname    | Scanned |   Access   |
+-------------------+--------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 5 | 10.0.182.20  |    Op_Server0   |   True  |    None    |
| UNKNOWN_SUBNET: 3 | 10.0.182.24  | UNKNOWN_HOST: 4 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.113 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.213.112/28  | 10.0.213.119 |      User0      |  False  | Privileged |
|  10.0.213.112/28  | 10.0.213.120 |      User1      |   True  | Privileged |
|  10.0.213.112/28  | 10.0.213.121 |      User2      |   True  | Privileged |
+-------------------+--------------+-----------------+---------+------------+

Reward: +1.2
Score: 4.8
```

A.2. Competitive Blue versus Dedicated Red

```
---- Turn 1 ----

Red Policy:
Red selects ('DiscoverSystems', 'User') with probability 99.99%

+----------------+-------------+-----------------+---------+------------+
|     Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+----------------+-------------+-----------------+---------+------------+
| 10.0.31.160/28 | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.31.160/28 | 10.0.31.165 |      User0      |  False  | Privileged |
| 10.0.31.160/28 | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
| 10.0.31.160/28 | 10.0.31.170 | UNKNOWN_HOST: 0 |  False  |    None    |
+----------------+-------------+-----------------+---------+------------+
```

```
Reward: +-0.0
Score: 0.0


---- Turn 2 ----

Red Policy:
Red selects ('DiscoverServices', 'User1') with probability 100.00%

+----------------+-------------+-----------------+---------+------------+
|     Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+----------------+-------------+-----------------+---------+------------+
| 10.0.31.160/28 | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.31.160/28 | 10.0.31.165 |      User0      |  False  | Privileged |
| 10.0.31.160/28 | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
| 10.0.31.160/28 | 10.0.31.170 | UNKNOWN_HOST: 0 |   True  |    None    |
+----------------+-------------+-----------------+---------+------------+

Reward: +-0.0
Score: 0.0


---- Turn 3 ----

Red Policy:
Red selects ('ExploitServices', 'User1') with probability 100.00%

+----------------+-------------+-----------------+---------+------------+
|     Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+----------------+-------------+-----------------+---------+------------+
| 10.0.31.160/28 | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.31.160/28 | 10.0.31.165 |      User0      |  False  | Privileged |
| 10.0.31.160/28 | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
| 10.0.31.160/28 | 10.0.31.170 |      User1      |   True  |    User    |
+----------------+-------------+-----------------+---------+------------+

Reward: +-0.0
Score: 0.0


---- Turn 4 ----

Blue policy:
('Restore', 'User1'): 13.36%
Blue selects ('Remove', 'Op_Host0') with probability 7.64%

Red Policy:
```

```
Red selects ('PrivilegeEscalate', 'User1') with probability 99.88%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
| UNKNOWN_SUBNET: 5 | 10.0.165.27 | UNKNOWN_HOST: 6 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.1


---- Turn 5 ----

Red Policy:
Red selects ('DiscoverServices', 'Op_Server0') with probability 99.98%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
| UNKNOWN_SUBNET: 5 | 10.0.165.27 | UNKNOWN_HOST: 6 |   True  |    None    |
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.2


---- Turn 6 ----

Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 99.98%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
| UNKNOWN_SUBNET: 5 | 10.0.165.27 |    Op_Server0   |   True  |    User    |
```

```
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.3


---- Turn 7 ----

Blue policy:
('Restore', 'Op_Server0'): 53.90%
Blue selects ('Remove', 'User3') with probability 11.75%

Red Policy:
('ExploitServices', 'Op_Server0'): 92.20%
('PrivilegeEscalate', 'Op_Server0'): 7.78%
Red selects ('ExploitServices', 'Op_Server0') with probability 92.20%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
| UNKNOWN_SUBNET: 5 | 10.0.165.27 |    Op_Server0   |   True  |    User    |
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.4


---- Turn 8 ----

Blue policy:
('Restore', 'Op_Server0'): 46.79%
Blue selects ('Restore', 'Op_Server0') with probability 46.79%

Red Policy:
Red selects ('PrivilegeEscalate', 'Op_Server0') with probability 99.75%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
```

```
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
| UNKNOWN_SUBNET: 5 | 10.0.165.27 |    Op_Server0   |   True  |    None    |
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +1.1
Score: 1.5


---- Turn 9 ----

Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 99.99%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
| UNKNOWN_SUBNET: 5 | 10.0.165.27 |    Op_Server0   |   True  |    User    |
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +0.1
Score: 1.6


---- Turn 10 ----

Blue policy:
('Restore', 'Op_Server0'): 51.09%
Blue selects ('Analyse', 'User1') with probability 11.22%

Red Policy:
Red selects ('PrivilegeEscalate', 'Op_Server0') with probability 99.89%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
|   10.0.165.16/28  | 10.0.165.27 |    Op_Server0   |   True  | Privileged |
```

```
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +1.1
Score: 2.7


---- Turn 11 ----

Blue policy:
Blue selects ('Restore', 'Op_Server0') with probability 98.25%

Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 100.00%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
|   10.0.165.16/28  | 10.0.165.27 |    Op_Server0   |   True  |    User    |
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +1.1
Score: 3.8


---- Turn 12 ----

Blue policy:
('Restore', 'Op_Server0'): 6.49%
Blue selects ('Remove', 'Op_Server0') with probability 15.43%

Red Policy:
Red selects ('PrivilegeEscalate', 'Op_Server0') with probability 99.98%

+-------------------+-------------+-----------------+---------+------------+
|       Subnet      |  IP Address |     Hostname    | Scanned |   Access   |
+-------------------+-------------+-----------------+---------+------------+
| UNKNOWN_SUBNET: 3 | 10.0.165.18 | UNKNOWN_HOST: 4 |  False  |    None    |
|   10.0.165.16/28  | 10.0.165.27 |    Op_Server0   |   True  | Privileged |
```

```
|   10.0.31.160/28  | 10.0.31.161 | UNKNOWN_HOST: 2 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.165 |      User0      |  False  | Privileged |
|   10.0.31.160/28  | 10.0.31.167 | UNKNOWN_HOST: 1 |  False  |    None    |
|   10.0.31.160/28  | 10.0.31.170 |      User1      |   True  | Privileged |
+-------------------+-------------+-----------------+---------+------------+

Reward: +1.1
Score: 4.9
```

A.3. Dedicated Blue versus Competitive Red

```
---- Turn 1 ----

Red Policy:
Red selects ('DiscoverSystems', 'User') with probability 99.98%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 | UNKNOWN_HOST: 1 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 | UNKNOWN_HOST: 0 |  False  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +-0.0
Score: 0.0


---- Turn 2 ----

Red Policy:
('DiscoverServices', 'User1'): 88.57%
('DiscoverServices', 'User2'): 9.98%
('DiscoverServices', 'User3'): 1.39%
Red selects ('DiscoverServices', 'User2') with probability 9.98%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 | UNKNOWN_HOST: 1 |   True  |    None    |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 | UNKNOWN_HOST: 0 |  False  |    None    |
+-----------------+--------------+-----------------+---------+------------+
```

```
Reward: +-0.0
Score: 0.0


---- Turn 3 ----

Red Policy:
('DiscoverServices', 'User1'): 2.73%
('ExploitServices', 'User2'): 96.89%
Red selects ('ExploitServices', 'User2') with probability 96.89%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 | UNKNOWN_HOST: 0 |  False  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.1


---- Turn 4 ----

Red Policy:
('DiscoverServices', 'User1'): 54.60%
('DiscoverServices', 'User3'): 2.30%
('PrivilegeEscalate', 'User2'): 42.62%
Red selects ('DiscoverServices', 'User1') with probability 54.60%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 | UNKNOWN_HOST: 0 |   True  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +0.1
Score: 0.2


---- Turn 5 ----
```

```
Red Policy:
Red selects ('ExploitServices', 'User1') with probability 99.82%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 |      User1      |   True  | Privileged |
+-----------------+--------------+-----------------+---------+------------+

Reward: +0.2
Score: 0.4


---- Turn 6 ----

Blue policy:
('Restore', 'User1'): 99.93%
Blue selects ('Restore', 'User1') with probability 99.93%

Red Policy:
('PrivilegeEscalate', 'User1'): 90.81%
('PrivilegeEscalate', 'User2'): 8.89%
Red selects ('PrivilegeEscalate', 'User1') with probability 90.81%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
+-----------------+--------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 |      User1      |   True  |    None    |
+-----------------+--------------+-----------------+---------+------------+

Reward: +1.1
Score: 1.5


---- Turn 7 ----

Red Policy:
Red selects ('ExploitServices', 'User1') with probability 99.70%

+-----------------+--------------+-----------------+---------+------------+
|      Subnet     |  IP Address  |     Hostname    | Scanned |   Access   |
```

```
+-----------------+-------------+-----------------+---------+------------+
| 10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
| 10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
| 10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
| 10.0.173.128/28 | 10.0.173.138 |      User1      |   True  | Privileged |
+-----------------+-------------+-----------------+---------+------------+

Reward: +0.2
Score: 1.7


---- Turn 8 ----

Red Policy:
('PrivilegeEscalate', 'User1'): 3.87%
('PrivilegeEscalate', 'User2'): 96.03%
Red selects ('PrivilegeEscalate', 'User2') with probability 96.03%

+------------------+-------------+-----------------+---------+------------+
|       Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+------------------+-------------+-----------------+---------+------------+
|  10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
|  10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
|  10.0.173.128/28 | 10.0.173.138 |      User1      |   True  | Privileged |
| UNKNOWN_SUBNET: 5 |  10.0.239.6  | UNKNOWN_HOST: 6 |  False  |    None    |
| UNKNOWN_SUBNET: 3 |  10.0.239.9  | UNKNOWN_HOST: 4 |  False  |    None    |
+------------------+-------------+-----------------+---------+------------+

Reward: +0.2
Score: 1.9


---- Turn 9 ----

Red Policy:
Red selects ('DiscoverServices', 'Op_Server0') with probability 99.98%

+------------------+-------------+-----------------+---------+------------+
|       Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+------------------+-------------+-----------------+---------+------------+
|  10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
|  10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
|  10.0.173.128/28 | 10.0.173.138 |      User1      |   True  | Privileged |
| UNKNOWN_SUBNET: 5 |  10.0.239.6  | UNKNOWN_HOST: 6 |   True  |    None    |
| UNKNOWN_SUBNET: 3 |  10.0.239.9  | UNKNOWN_HOST: 4 |  False  |    None    |
```

```
+------------------+-------------+-----------------+---------+------------+
```

```
Reward: +0.2
Score: 2.1
```

```
---- Turn 10 ---
```

```
Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 99.94%
```

```
+------------------+-------------+-----------------+---------+------------+
|       Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+------------------+-------------+-----------------+---------+------------+
|  10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
|  10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
|  10.0.173.128/28 | 10.0.173.138 |      User1      |   True  | Privileged |
| UNKNOWN_SUBNET: 5 |  10.0.239.6  |    Op_Server0   |   True  |    User    |
| UNKNOWN_SUBNET: 3 |  10.0.239.9  | UNKNOWN_HOST: 4 |  False  |    None    |
+------------------+-------------+-----------------+---------+------------+
```

```
Reward: +0.2
Score: 2.3
```

```
---- Turn 11 ----
```

```
Red Policy:
Red selects ('ExploitServices', 'Op_Server0') with probability 98.84%
```

```
+------------------+-------------+-----------------+---------+------------+
|       Subnet     |  IP Address |     Hostname    | Scanned |   Access   |
+------------------+-------------+-----------------+---------+------------+
|  10.0.173.128/28 | 10.0.173.129 |      User2      |   True  | Privileged |
|  10.0.173.128/28 | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.173.128/28 | 10.0.173.136 |      User0      |  False  | Privileged |
|  10.0.173.128/28 | 10.0.173.138 |      User1      |   True  | Privileged |
| UNKNOWN_SUBNET: 5 |  10.0.239.6  |    Op_Server0   |   True  |    User    |
| UNKNOWN_SUBNET: 3 |  10.0.239.9  | UNKNOWN_HOST: 4 |  False  |    None    |
+------------------+-------------+-----------------+---------+------------+
```

```
Reward: +0.2
Score: 2.5
```

```
---- Turn 12 ----
```

```
Red Policy:
Red selects ('PrivilegeEscalate', 'Op_Server0') with probability 97.16%

+-------------------+--------------+-----------------+---------+------------+
|       Subnet      |  IP Address  |     Hostname    | Scanned |   Access   |
+-------------------+--------------+-----------------+---------+------------+
|  10.0.173.128/28  | 10.0.173.129 |      User2      |   True  | Privileged |
|  10.0.173.128/28  | 10.0.173.132 | UNKNOWN_HOST: 2 |  False  |    None    |
|  10.0.173.128/28  | 10.0.173.136 |      User0      |  False  | Privileged |
|  10.0.173.128/28  | 10.0.173.138 |      User1      |   True  | Privileged |
|   10.0.239.0/28   |  10.0.239.6  |    Op_Server0   |   True  | Privileged |
| UNKNOWN_SUBNET: 3 |  10.0.239.9  | UNKNOWN_HOST: 4 |  False  |    None    |
+-------------------+--------------+-----------------+---------+------------+

Reward: +1.2
Score: 3.7
```