UNIVERSITY OF CALIFORNIA
SANTA CRUZ

SET PRIORITY: SOLVING AN ARTIFICIAL INTELLIGENCE CYBERSECURITY CHALLENGE WITH A SIMPLE MANUAL AGENT

A thesis submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE AND ENGINEERING

by

Joaquin L. Ortiz

December 2024

Table of Contents

List of Figures

List of Tables

Abstract

Set Priority: Solving an Artificial Intelligence Cybersecurity Challenge with a Simple Manual Agent

by

Joaquin L. Ortiz

In recent years, there has been a focus on the potential of using artificial intelligence agents for network security. In this paper, we consider a scenario where network segmentation protocols require a multi-agent solution, along with an adversary that can evade traditional firewall solutions via email, known as the CAGE 4 challenge. While the challenge is focused on the use of artificial intelligence, we propose a solution that was manually crafted by analyzing the potential alerts provided by the environment with human eyes. The exceptionally good performance of our agent, especially when compared to other submissions to the challenge, leads us to raise questions about the efficacy of such challenges for evaluating the performance of artificial intelligence algorithms.

Acknowledgments

We would like to thank David Torres-Mendoza for designing an agent visualization system for use in debugging my code. Without his visualizer, this paper would not have been possible, as analyzing the behavior of agents using only text files was extremely frustrating if not impossible.

We would also like to thank the creators of the UCTHESIS LaTeXdocument class, found at https://github.com/adamnovak/ucscthesis.

Figure 2.1 used with permission from Group [2023].

Chapter 1

Introduction

Over the past decade, artificial intelligence technologies have gained significant mainstream attention. While some of these technologies have been around for much longer, others, especially generative technologies, are relatively new and are continuously increasing their capabilities. As public access to artificial intelligence tools increases, so does concern for potential malicious use of these tools. One potential malicious use is to drive a cyberattack. An artificial intelligence agent trained on a network simulator may develop a method of attack which performs better than the average human hacker would in the same scenario. The resulting model could then be used to create an automatic network infiltration service. While such a model would likely require some target-specific training, it could still pose quite a threat to traditional security systems.

Due to this possibility, artificial intelligence gyms have been created for the purpose of training defensive agents. Such agents could, in theory, deduce that a network intrusion has occured faster than traditional models and even take actions to confine or expel the intruder without waiting for human input. These agents could also be continuously trained to stay up to date with recently observed threats. Additionally, these agents could exploit the structure of the network to their advantage, as an attacker would likely not have complete knowledge of this structure. For instance, a defensive agent could mislead an attacker to a honeypot network which contains fake data for the attacker to steal.

To further research about this topic, challenges such as CAGE have been created. These challenges present a particular cyberattack scenario, with varying network layouts, observable features, goals, and attackers, and invite participants to create a defensive agent which performs the best according to the challenge's goals and scoring system. The results of these challenges are used to study techniques for training cybersecurity agents and their effectiveness under different scenarios.

In this paper, we will present an atypical solution to the CAGE 4 scenario. Contrary to what might be expected for a solution to an artificial intelligence challenge, the bulk of the solution does not involve artificial intelligence, but instead a manually developed prioritization of actions. Artificial intelligence is used selectively, if at all, in places where uncertainty is present. Despite the simplicity of this solution, it performs equivalently to or better than all known solutions submitted during the CAGE 4 competition.

The main contributions of this paper are summarized as follows:

- This is the one of the first papers in the literature to explore the CAGE 4 scenario, and the unique challenges that the scenario poses when developing an agent.

- We propose only using artificial intelligence where human intelligence fails, or using human intelligence to bootstrap artificial intelligence, instead of relying on reinforcement learning to find a policy without guidance.

- We raise questions about how well a training scenario that can be easily be beaten by a human can serve as an evaluation to determine how effective an AI model is.

- We raise questions about how to effectively balance such scenarios so that it is difficult for a human to arrive at an optimal solution but possible for an AI to do so.

The remainder of the paper is structured as follows. Section 2.2 will explore prior work in this field, followed by the CAGE 4 challenge itself, the semantics of the environment, and the method of evaluation. We will then introduce our agent in Section 3 and describe its observation space, why we decided to create our own observation space instead of using the one provided by the challenge, implementation details, and some potential modifications. In Section 4, we will evaluate the agent before concluding and discussing potential for future research in Section 5.

Chapter 2

Background

2.1 Important Concepts

This paper assumes familiarity with the following concepts, which are used by the CAGE challenge's documentation and in related literature. We are defining these terms as they are used in the context of artificial intelligence for cybersecurity, not necessarily for artificial intelligence in general.

Agent: In an AI environment, an agent is a interface that can receive observational inputs and rewards from the environment, and use these inputs to take actions in order to achieve some goal. For the purposes of this paper, an agent may refer to a program which tries to defend or attack a network or a simulated user of the network.

Action: An high-level abstraction of a general computer security tactic or technique, such as restoring a host using a system image or creating a honeypot in a particular location.

Step: An arbitrary moment of time used to separate the continuous flow of time into discrete units. In general, each agent receives one observation on each step and takes one action on each step.

Agent Teams: Agents are separated into 3 general teams. Blue agents are agents which attempt to defend the network from a cyberattack, while red agents are the agents which perform the attack. Green agents represent the users of the network, and are usually controlled by the simulation. The existence of the green agents serves as an extra challenge to the blue agents, as they provide a source of innocuous activity throughout the network that serves to distract from the red agent's malicious activity.

Host: A single system connected to a network. For the purposes of this paper, a host can either refer to a user terminal through which a green agent will interface with the network or a server which provides services that can be connected to.

Lateral Movement: For a red agent, the act of moving in between hosts and subnets in search of a goal, such as an important server or subnet. The red agent will use lateral movement to learn about the topology of the network so it can reach this goal. In this paper, we specifically use it to refer to the act of moving in between subnets, as this is significantly harder than the act of move in between hosts.

Partial Observability: The inability to have a complete picture of the environment at any given time. In cybersecurity scenarios, all agents have partial observability of the environment. Red agents are generally not aware of the topology of the network when the game begins, and must explore to discover hosts. Also, the red agent cannot see everything that is occurring on a particular host without gaining administrative access to said host. While the blue agent is aware of the topology of the network at the start of the game, it may not be aware of every communication or process that runs in the network, possibly due to a requirement for confidentiality of communications or due to deception on the part of the red agent. And of course, green agents are simply users and do not have complete access to all hosts in the network, nor should they.

Reinforcement Learning (RL): A form of learning which learns to estimate the expected reward for taking a certain action in a given state, as well as which state the agent is expected to end up in as a result. Using these estimations, an agent can either select the action that is expected to result in the best reward-state pair (exploitation) or choose to take an action at random to see what kinds of rewards and/or states occur (exploration). Reinforcement learning works well in stochastic environments, where the transition from state to state is not deterministic, but relies heavily on an accurate reward function.

CybORG: An AI gym for training cybersecurity agents, which complies with the OpenAI gym specification [Standen et al., 2021]. The CAGE challenges are implemented using this gym [Group, 2023].

2.2 Related Work

Several papers have been written about previous iterations of CAGE. These iterations have some significant differences from CAGE 4, but are still useful to track existing methodologies of training artificial intelligence agents for cyber defense. This section is not exhaustive; we are only highlighting works that we found relevant to

Actions			
Blue Agent	Steps	**Red Agent**	Steps
1. **Analyze** files	2	1. **Discover** hosts	1
2. Create **Decoy** service	2	2. **Portscan** host	1 or 3
3. **Kill** detected processes	3	3. **Identify** Decoys	2
4. **Restore** system image	5	4. **Exploit** service	4
5. **Block** subnet	1	5. **Escalate** privilege	2
6. **Unblock** subnet	1	6. **Degrade** services	2

Table 2.1: Actions available in CAGE 4

the issues posed by this challenge and the solution that we created.

Bates et al. [2023] investigated the use of reward shaping to expedite training of agents in the CAGE 2 environment. This is especially important as the CAGE challenges are systems with very sparse, mostly negative rewards. Sparsity of rewards often corresponds with requiring more samples to learn a policy [Crowder et al., 2024]. Their experiments showed that reward shaping can be effective in helping a learning algorithm converge on a policy faster, but that doing reward shaping improperly can cause the algorithm to instead converge on a worse-performing policy Ng et al. [1999].

The winning submission to CAGE 1, created by Foley et al. [2022], used a specific subclass of reward shaping known as intrinsic curiosity, which encourages agents to explore states that are considered "novel". Implementing curiosity into one of their agents led to a drastic improvement in said agent's score. They observed that such curiosity "overcomes the problem of developing strategies in the presence of randomness in the adversary". They used a similar strategy to develop an agent for CAGE 2 [Foley et al., 2023], but were not able to achieve the same level of success without also modifying the observation provided by the challenge to contain additional information.

The same team used a different strategy when approaching CAGE 3, which is a multi-agent environment with highly random elements, such as the local and behavior of the red agents and the behavior of the green agents. In Hicks et al. [2023], they found that a naive application of PPO completely failed to learn a policy, and instead found that it was best to design a well-performing expert agent, an agent based on an expertly designed algorithm as opposed to learning a policy, which could be used to bootstrap learning agents in the same environment.

2.3 CAGE Challenge

The CAGE 4 Challenge [Group, 2023] takes place in a network whose layout is pictured in Figure 2.1. Due to the high security of the network, certain subnets cannot allow connections from external blue agents. As a result, these subnets require a separate dedicated blue agent, represented in the figure using the shield icon. Likewise, if the red agent moves laterally into one of these subnets, it must operate a separate agent in that subnet, unable to communicate with its parent. However, one of the agents is still in charge of three of the subnets; we refer to this agent as the HQ agent. All blue agents share the same score, which is provided by green agents, simulated users that will deduct points if their work is interrupted.

Each subnet contains between 1 and 10 user terminals as well as between 1 and 6 servers, collectively known as hosts. Each host runs a set of random services which a red agent can exploit to gain access. One of the servers is responsible for maintaining the subnet's firewall. If the red agent gains administrative access to this special server, it can allow itself to temporarily communicate through the firewall and achieve lateral movement.

Over the course of the simulation, the subnets will go through three phases. In the first phase, known as the planning phase, all links in Figure 2.1 are active. After one-third of the simulation has elapsed, the subnets will move on to **mission phase A**. In mission phase A, green agents within the subnets labeled Restricted Zone A will not be able to communicate with hosts in Restricted Zone B and vice versa, and green agents within the subnet labeled Operational Zone A will not be able to communicate with other subnets at all. After another one-third of the simulation has elapsed, **mission phase B** will begin, which is similar except that Operational Zone B is affected instead of Operational Zone A. These restrictions are voluntary on the part of the green agents; the red agent may ignore them, requiring preventative action from the blue agent.

Unlike red agents, blue agents are able to have limited communication with each other. On each step, a blue agent can send 8 bits of data to adjacent blue agents. The operational zones are an exception: blue agents in these zones are not able to transmit messages due to the high security protocols, but may still receive messages from blue agents in the corresponding restricted zone. The content and meaning of these bits is entirely up to the agents

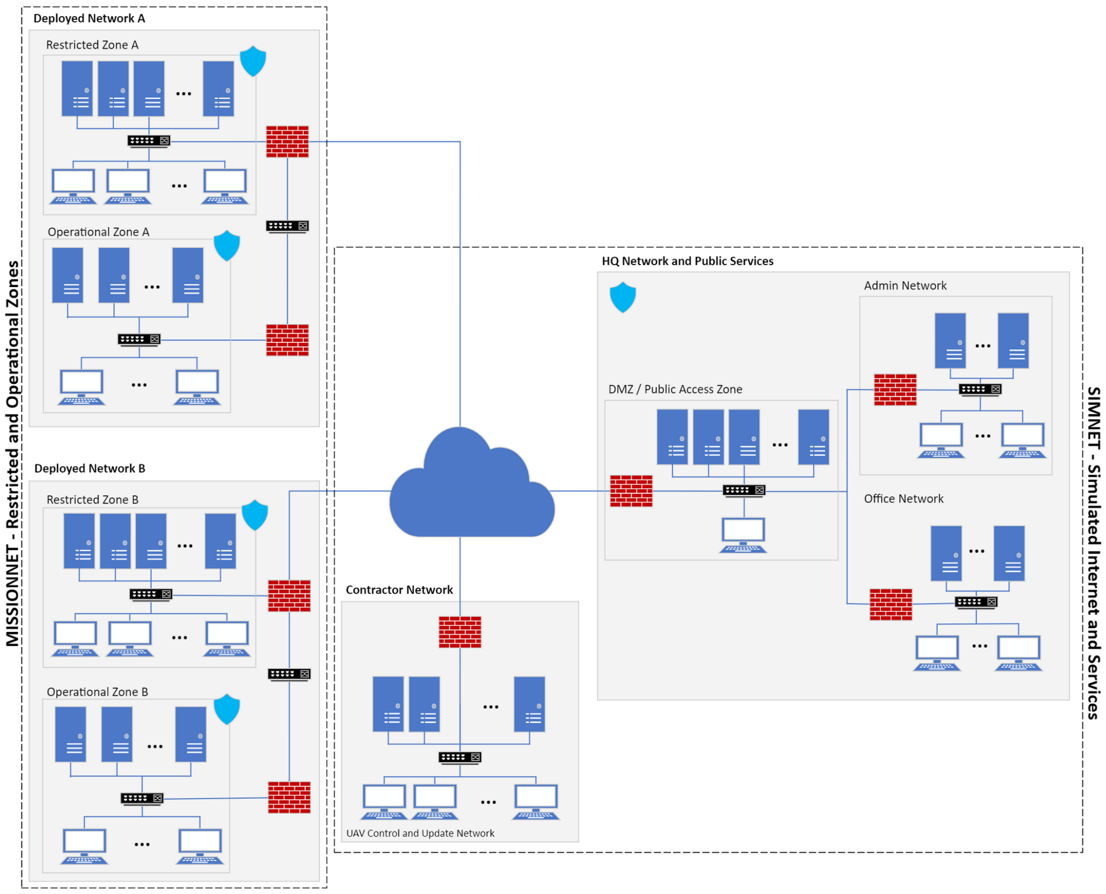

Figure 2.1: CAGE 4 Subnet Diagram

themselves.

The red agent always enters the network through a compromised user terminal in the contractor subnet. The contractor does not have a blue agent installed in their subnet, and the servers in the contractor's subnet are vital to the green agents' work, so it is not possible to simply quarantine the subnet or the affected terminal. Instead, the blue agents must respond to the red agent's attempts to move laterally out of the contractor subnet before the red agent can damage important systems.

2.4 Agent Actions

The blue and red agents each have six high-level actions, as shown in Table 2.1. Each action corresponds to a basic kind of attack or defense used in real world scenarios. Each agent can only run one action at a time, and some actions may take multiple steps to finish. The effects of an action will only apply at the end of this waiting period. The waiting period includes the step that the action was run; a one-step action will yield results on the next step. Actions cannot be canceled once started. Below, we describe semantics of actions that we found notable.

Each host can only support a limited number of **Decoy**s before they begin interfering with the host's regular services. The number of decoys that can be used on a given host depends on the services that a host runs at the start of the game, as decoys can only listen on certain ports.

Kill cannot arbitrarily kill processes; it will only kill processes on the specified host's "Kill list". Processes are added to a "Kill list" if they trigger a security alert, as detailed in 2.6. Processes cannot be killed if they are running with elevated permissions. This action takes 3 steps and always succeeds, even if it failed to kill a process.

Blocks will prevent all communication between two subnets, except for emails. Blocks will not be removed automatically; they will last until **Unblock**ed.

Portscans can either be done aggressively, taking 1 step, or stealthily, taking 3 steps. We will cover the difference between the two in Section 2.6

The red agent can run **Degrade** in one of two modes: *OT* mode or *Service* mode. In OT mode, the red agent will attempt to kill a particular internal service known as the OT service. If the specified host is not running an instead of the OT service, the action will fail. In Service mode, the reliability of all non-OT services on the specified host will be reduced. Unreliable services have a 20% chance to reject an otherwise valid connection, and this chance increases by 20% every time Degrade is run in Service mode.

2.5 Green Agents

Green agents are simulation controlled agents that represent the users of the network. One green agent is created for each user host in the network. The actions of green agents drive the scoring system used by the challenge, and also provide a source of innocuous network and process activity that may cause a false alarm. Green agents use an entirely separate set of actions, having only two: **LocalWork** and **RemoteWork**. Every step, the green agent will execute one of the two actions at random. Both actions take only one step to complete.

LocalWork: The green agent attempts to connect to a service running locally, on the same host as the green agent itself. If there are multiple services, one is selected at random. As this is a connection, an unreliable service may reject the connection, causing the action to fail.

If a red agent is active in a subnet adjacent to the one the green agent's host is a member of, and the LocalWork action succeeds, there is a 1% chance that the green agent will accidentally open a phishing email while performing work. This will lead to the red agent gaining access to the green agent's host. Phishing emails can be received and opened even if the sender and recipient are separated by a firewall that has been Blocked. This is believed to be a bug, and makes it more difficult to stop the lateral movement of red agents.

RemoteWork: A server host in the same subnet or an adjacent subnet is selected at random. If that server host provides an OT service, the green agent attempts to connect to that service, otherwise it will select a service from that host randomly. The action will fail if the connection is blocked due to firewall rules created by Block, if the service being connected to has been killed, or if the service being connected to rejects the connection due to being affected by Degrade.

2.6 Partial Observation

In practice, security analysts cannot fully observe the state of a network, and attacker actions can be hidden or missed by security monitors. The CAGE challenge models this problem as a partially observable environment, where the blue agent can only observe partial information about the state of the network. In this section we describe the actions that may trigger these security monitors, resulting in an alert, their chances of doing so, and the resulting observations.

When an alert occurs, information about the alert and the associated host is made visible to the blue agent that has jurisdiction over the subnet in which the alert occurred. Each alert is classified as a file, connection, or process, which determines which fields may be present in the alert. If a connection alert crosses subnet bounds, the alert only appears to the blue agent with jurisdiction over the subnet that received the connection, not the blue agent with jurisdiction over the subnet that sent it.

```
{
    'Files': [{
        'Density': 0.9, 'File Name'
    : 'cmd.sh', 'Path': '/tmp/'
    }],
    'Connections': [{
        'local_address': '
    10.0.90.221', 'local_port': 25,
    'remote_address': '10.0.90.250',
     'remote_port': 53258
    }],
    'Processes': [{'PID': 2651}],
    'Interface': [
        {'ip_address': '10.0.90.221
    '}
    ],
    'System info': {
        'Architecture': 'x64',
        'Hostname': '
    admin_network_user_host_3',
        'OSDistribution': 'ubuntu',
        'OSType': 'linux',
        'OSVersion': '20.04'
    }
}
```

Listing 2.1: Example of alerts from an Ubuntu host.

An aggressive **Portscan** has a 75% chance triggering a connection alert for each port, while a stealthy Portscan has a 25% chance of triggering a connection alert for each port. However, if the host has a Decoy, the connection to the Decoy's port will *always* trigger a connection alert.

```
[
    {'local_address': '10.0.90.2',
    'local_port': 25, '
    remote_address': '10.0.90.196',
    'remote_port': 52597},
    {'local_address': '10.0.90.2',
    'local_port': 22, '
    remote_address': '10.0.90.196',
    'remote_port': 51368},
    {'local_address': '10.0.90.2',
    'local_port': 3390, '
    remote_address': '10.0.90.196',
    'remote_port': 50271}
]
```

Listing 2.2: Example of alerts caused by a single Portscan.

Exploit: Has a 95% chance of triggering a connection and process alert on success and a 100% chance of triggering a connection alert on failure. If a process alert is triggered, the detected process will be added to the Kill list. Exploit also creates a payload file called "cmd.sh" that will trigger a file alert when Analyze is used.

```
'Connections': [
{'local_address': '10.0.90.221', '
    local_port': 25, 'remote_address
    ': '10.0.90.250', 'remote_port':
     53258}
],
'Processes': [{'PID': 2651}],
```

Listing 2.3: Example of a connection and process alert caused by Exploit. The lack of fields in the process alert indicates that information is being hidden.

Escalate: Although Escalate cannot actively cause alerts, it does create a payload file called "escalate.sh" that will trigger a file alert when scanned by Analyze.

```
{
    'Density': 0.9,
    'File Name': 'escalate.sh',
    'Path': '/tmp/'
}
```

Listing 2.4: Example of a file created by Escalate. Density may vary but File Name is always the same.

Green Agents: Green agents have a 1% chance to raise a false positive connection alert every time they take a successful action.

```
{'local_address': '10.0.74.39', '
    remote_address': '10.0.86.254',
    'remote_port': 49711}
```

Listing 2.5: Example of an false positive alert caused by RemoteWork

```
{'local_address': '10.0.90.221', '
    local_port': 53258}
```

Listing 2.6: Example of a false positive alert caused by LocalWork

Unlike successful runs of Exploit, an infiltration triggered by a phishing email encountered during LocalWork does *not* create a file, nor can it ever cause an alert. This makes phishing emails an especially dangerous way for red agents to move laterally, as they are completely untraceable. However, the red agent must Escalate before it can deal damage to the host, which gives Analyze a chance to trigger an alert for the payload file created by Escalate.

2.7 Scoring

The goal of the blue agent is to maximize service availability in the presence of an attacker that is attempting to kill and/or reduce the reliability of services. This goal is reflected in the scoring system provided by the challenge, where points are deducted should the attacker successfully kill or destabilize a service in a way that disrupts a green agent's attempt to do some kind of Work. These penalties vary during the three phases of the game to represent how certain system are more valuable if they are currently performing critical operations.

Penalties are assessed whenever a green agent fails to do Work, either because of Degraded or killed services, or, in the case of RemoteWork, a Block that prevents the connection from going through. Due to how critical it is to operation of the network, an immediate penalty is assessed should the red agent successfully kill the OT service on any host. Additionally, a single-point penalty is applied for using Restore on a host to represent the downtime incurred, regardless of if the host is a server or a user.

During mission phases, the operational and restricted zones for the currently running mission have increased penalties for red agent activity. This is to represent the importance of the work that the hosts in that subnet are currently performing. The phase does not affect the Restore penalty.

Despite there being no blue agent to defend the contractor network, it has a penalty during the planning phase. This is likely to prevent blue agents from simply Blocking the contractor network to prevent initial infection of the network.

Subnet	Phase	Penalty
Contractor	Planning	-5
Contractor	A or B	0
Operational A	A	-10
Restricted A	A	-2
Operational B	B	-10
Restricted B	B	-2
(other)		-1

Table 2.2: Rewards, separated by subnet and phase

2.8 Red Agent Behavior

For evaluation purposes, a state machine red agent is provided by the CAGE 4 challenge. The state machine of this agent is described by Figure 2.2, with a separate instance of the state machine for each known host. Although we did not analyze this state machine for the purposes of developing our solution, we are including it here as it is part of the evaluation used for the challenge and it may provide some insights as to why certain solutions perform they way they do. At the start of a step where is it able to take an action, the red agent will select a host at random and select a state transition as follows.

Hosts begin in the Known state. The red agent can run Portscan on a Known host to get information about which ports are open, putting that host in the Scanned

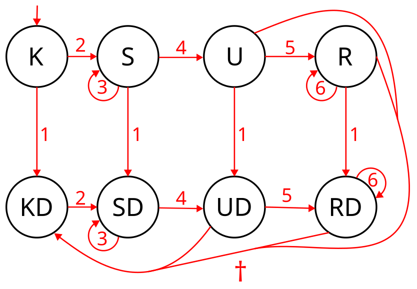

Figure 2.2: Red agent state machine. Numbers correspond to actions in Table 2.1. † occurs if the red agent loses access to a host.

state. From here, the red agent can run rounds of Identify to determine which ports are likely decoys, and eventually run Exploit on a port which is believed to not be a decoy. On success, the host will be put in the Unrooted state. After this, the red agent will Escalate to move to the Rooted state, and then repeatedly Degrade that host as long as it has access.

States ending in D indicate that the subnet which contains that host has already been Discovered. The red agent has a high chance of executing Discover during any state which does not end in D, and will transition all hosts discovered as a result of that action to the equivalent D state.

2.9 Observation Wrapper

Although the blue agent observations described in Section 2.6 are suitable for parsing by Python code, they are not suitable for inputting to an RL agent algorithm. Most algorithms require that the input observation be a multi-discrete array, a one-dimensional array of integers, each with a defined range of possible values. Because of this, the creators of CAGE 4 designed an observation wrapper which transforms the raw, unwrapped observation previously described into a multi-discrete array. The layout of the resulting array can be seen in Table 2.3. Below, we describe each of the fields in the array.

- `Phase` is a single number that reports the current phase, with the planning phase being 0, mission phase A being 1, and mission phase B being 2. The range of this value is $[0, 2]$.
- `Subnet ID` is a single number that tells the agent which subnet it has jurisdiction over. The range of this value is $[0, 7]$, as the contractor network counts as a subnet despite there not being a blue agent there.
- `Blocked subnets` is a series of 8 bits, one for each of the subnets in the simulation, that encode whether a Block exists between the subnet indicated by `Subnet ID` and the subnet indicated by the bit.
- `Should Subnet be Blocked` is a se-

Name	Length	Values
Phase	1	Planning, Mission A, Mission B
Subnet ID	1	See 2.1
Blocked Subnets	8	No, Yes
Should Subnet be Blocked	8	No, Yes
Process Alert on Previous Step	16	No, Yes
Connection Alert on Previous Step	16	No, Yes
Repeat last 5 rows for additional subnets, if this is the HQ agent		
Communication Vector	$4 * 8$	0, 1
Padding	0 or 98	0

Table 2.3: Fields of the wrapped observation array provided by the challenge

ries of 8 bits, one for each of the subnets in the simulation, that encode whether a Block should be placed between the subnet indicated by `Subnet ID` and the subnet indicated by the bit in order to enforce the current phase's communication policy, as detailed in Section 2.3.

- `Process Alert on Previous Step` is a series of 16 bits, one for each of the possible hosts in the subnet, that encode whether a process alert occurred on that host on the previous step. If there are less than 16 hosts in a subnet, bits which do not correspond to any host will always be 0.

- `Connection Alert on Previous Step` is the same as `Process Alert on Previous Step`, but for connection alerts instead of process alerts. Connection alerts will only appear on the host that received the connection that caused the alert, not the one that sent it.

- If this is the HQ agent, the above 5 fields (`Subnet ID` to `Connection Alert`) will be repeated for each of the three subnets the HQ agent has jurisdiction over.

- The `Communication Vector` encodes messages received from other agents as four series of 8 bits. If the current blue agent cannot receive a message from a given blue agent due to the communication policies, the message will be represented as all zeroes in the observation space.

- `Padding` may be optionally used to support algorithms which do not allow for variable observation sizes. If used, it will be appended to the arrays of non-HQ blue agents so that they match the size of the array of the HQ blue agent.

Chapter 3

Set Priority Agent

Following our analysis of the various observations in the CAGE 4 challenge, we determined that there were patterns which could be used to reliably differentiate between false and true positive alerts. However, these patterns were being hidden by the observation wrapper provided by the challenge. As a result, we decided to create our own observation wrapper which would use these patterns to sort hosts into different sets. While we intended to use the sets as input to a reinforcement learning algorithm, we realized that we could instead write a simple algorithm to prioritize actions using these sets.

Ultimately, a good solution to this challenge is one that is able to balance maintaining availability of network services with the dangers of acting on false positive alerts, while also attempting to account for false negatives by proactively searching for red agents. Using these principles, we created a design for a simple agent to use as a starting point using the following basic ideas:

- If we know that Restore must be run on a host, meaning that either the red agent on the host has Escalated or we were not alerted to the initial infection of the host, we should run Restore on it as soon as possible to prevent the red agent from dealing damage to the host.
- If we know about malicious processes on a host, we should Kill them as soon as possible before they have the chance to Escalate. As Kill takes less steps than Restore, and the window for Killing is limited, Killing should take priority over Restoring.
- As damage a server has the potential to affect more users than damage to a single user, we should prioritize running Kill or Restore on servers before doing so on users. However, Kill on

a user should still take priority over Restore on a server.

- When there are not active threats on our subnets, we should create as many Decoys as possible on each host, as Decoys greatly increase the chance of Portscans causing alerts and somewhat increase the chance of Exploits failing. This requires us to know how many more Decoys can be created on a host at any given time, as different hosts may not be able to support the same number of Decoys.
- To increase the chance that a red agent causes alerts no matter where it tries to go, we should aim to create Decoys evenly. In other words, we first create one Decoy on all hosts, then create a second Decoy on all hosts, and so on, instead of creating three Decoys on one host before moving on to the next.
- When we have nothing else to do, we should Analyze all hosts in turn so that we can detect infections that we may have missed, either due to a phishing email or, rarely, a false negative alert from Exploit.
- Despite this, false negative Exploits are rare, so it may be beneficial to only Analyze user hosts, as only users can open phishing emails.
- Following a successful Restore or Kill, there is no need to Analyze a host for some time, as it is unlikely that it will be reinfected immediately.

Note that these design objectives leave three major ambiguities: what to do if we know about a malicious process but suspect that it has already Escalated, what information to put into messages between agents, and how to determine if we should run Restore on a host. We intended to identify possible answers to the first two ambiguities by developing and testing various mutations of our agent that implemented various possible answers, and intended to define the answer to the last one as part of our observation space.

3.1 Alert Patterns

In Section 2.6, we listed all possible causes of alerts. Upon analyzing these further, we found that the presence or non-presence of keys will always be the same for any given cause of alert. For instance, if Portscan or Exploit causes a connection alert, the resulting alert will always have a `local_port` and a `remote_port`. However, if RemoteWork causes a connection alert, the resulting alert will only have a `remote_port` and not a `local_port`. Similarly, a connection alert caused by LocalWork will only have a `local_port` and not a `remote_port`. This pattern, based on presence or absence of fields, is very powerful, as it allows for us to **reliably distinguish** between false positive and true positive connection alerts. As there is no way for false positive file or process alerts to occur, this single pattern could allow an agent to learn to distinguish and ignore all false positive alerts, making said agent's job much simpler.

3.2 Observation Wrapper Issues

Unfortunately, we found that the observation space described in Section 2.9 often **omits or oversimplifies critical information**. This includes information that allows for detecting the patterns mentioned above, but also extends to actions as simple as Analyzing the files on a host or identifying the source of an attack. Both of these are critical to implementing the ideas behind our algorithm.

The issues with the observation space that we found most pressing are as follows:

- The `Alert on Previous Step` fields only report alerts that occurred on the previous step. **Once another step passes, the alert is forgotten.** As the agent may not be able to immediately act on an alert, the agent requires some form of internal memory or observation stacking so as to not forget about alerts. Due to the length of the Restore action in particular, and the possibility that any number of alerts could happen during any turn, the task of remembering all of the alerts quickly becomes difficult to manage if the responsibility is left to the agent.

- The wrapped observation **does not include any information about Decoys**, which makes it difficult to determine how many decoys are currently deployed on a host and how many additional decoys can be deployed. As the number of decoys that can be deployed on each host varies between episodes, this information would be useful to avoid wasting steps.

- The wrapped observation **does not include any fields for file alerts**, which makes it impossible to retrieve results from Analyze. The action is effectively unusable with this wrapper.

- Although there are patterns that allow differentiating between false and true positive alerts, the wrapper **simplifies the alerts down to single bits** that signify the presence or absence of a alert for each host. This makes it impossible for the agent to exploit these patterns.

- As the wrapper **only shows information about where connection alerts were received**, it is impossible to determine which host sent a connection that caused an alert.

3.3 New Observation Space

Due to the lack of key information in the original wrapped observation, we deemed it necessary to create our own observation wrapper that could preserve or parse the aforementioned patterns for our agent. The goal of the new observation wrapper is to distinguish hosts that require Restoring, Killing, more Decoys, or Analyzing, and to pass that information to the agent, which will choose the action. Due to the simplicity of the distinguishing patterns, we chose to hardcode the parsing of the patterns into the observation wrapper, and use the results of those patterns to create two sets of hosts. While creating the wrapper, we observed that we could create further sets to represent additional features about the hosts in the environment, and use information about host membership in those sets as a multi-binary input to an agent.

The core two sets of our new observation space are **the set of hosts that Kill should be used on** and **the set of hosts that Restore should be used on**. These are abbreviated as the Kill set and the Restore set.

The Kill set is simple to define: it is the set of all hosts which have a non-empty Kill list. Killing when we don't have anything to Kill is a no-op, so it should be avoided. As only true positive alerts from successful Exploits lead to processes being added to a host's Kill list, the set becomes the set of hosts on which we have been alerted to a successful Exploit.

As Escalate cannot result in an alert, the Restore set is not as simple to define. However, if we receive a true positive connection alert, we know that the red agent must have a presence on the host that sent the connection. If the host that sent the connection is not already in the Kill set, it is not possible to use Kill to remove the red agent from that host. Instead, the host should be added to the Restore set. In the event that the host is already in the Kill set, it is added to the Escalate set anyway, with dual set membership indicating that it is not clear if the red agent has run Escalate on this host or not. In a real world scenario, it is likely that a intruder would attempt to gain privilege escalation on a machine before using it to attack other machines, especially if the blue agent is not aware of the presence of the intruder.

Simply relying on connection and process alerts to act comes with the caveat that not all red agent actions can result in connection and process alerts. Due to the existence of phishing emails, it is possible for a red agent to move laterally and deal damage without the blue agent recieving a single alert. As a result, it is best to encourage the agent to actively scan hosts in its network using Analyze. Although phishing emails do not create files, the red agent must Escalate before it can deal damage, and Analyze will always raise a file alert on a host where Escalate has been used. To aid in this, the wrapper provides a deque of **hosts to run Analyze on** and a singleton set containing the host at the front of the queue, abbreviated as the Analyze queue and Analyze set respectively. As phishing emails can only infect user terminals, server hosts do not need to be in the deque. Whenever a blue agent runs Analyze, Restore, or Remove on a host, the host is moved to the back of the deque so that hosts which have not recently had some form of removal or analysis drift to the front of the deque.

The results of Analyze (file alerts) can also be used to place hosts into the Restore set. Being alerted to a file named `escalate.sh` should add a host to the Restore set. However, being alerted to a file named `cmd.sh` should also add a host to the Restore set, but only if it is not already in the Kill set. If a host has a `cmd.sh` file but is not in the Kill set, the PID of the process which created that file and is acting as the red agent's shell is unknown, and thus Kill will not work. As such, Restore must be used instead to evict the red agent from the host.

As Decoys cause Portscans to reliably cause alerts and can entice the red agent to use Exploit on the wrong port, the wrapper contains three sets and one mapping relating to decoys. The three sets are the **set of hosts that can support n additional Decoys**, where $n \in [1, 3]$, abbreviated as D_n. The mapping is a mapping of hosts to the number of additional Decoys that the host could support at the beginning of the game. The mapping is maintained so that a host can be added to the correct set after it is Restored, as doing so will wipe out all Decoys. When a Decoy is created, the host with the new Decoy will be moved from set D_{n-1} to D_n, unless $n = 0$, in which case the host will simply be removed from D_1.

Now that the observation wrapper has these sets, the observation passed to the agent can be encoded as a binary matrix. Each item in the matrix indicates set membership for a certain host, with each host having 6 bits. A 1 bit indicates that the host is in that set, and a 0 bit indicates that the host is not in that set.

Set bits for each Host
Kill set
Restore set
D_3
D_2
D_1
Analyze set

Table 3.1: Layout of the set-based observation.

Once the host sets were defined, we attempted to design sets related to Blocking subnets. Our original plan was to make sets of subnets that were Blocked and subnets that should be Blocked, but we realized that this would effectively be identical to the original observation wrapper after encoding set membership as bits. Instead, we used knowledge of the problem to determine the each agent only needs to worry about blocking one subnet, if any, during one phase of the simulation, as seen in Table 3.2. The observation itself then only requires information about whether that one subnet is/should be Blocked instead of holding information for all 8 subnets. This also allowed us to decrease the size of the action space by removing Block and Unblock actions that do not pertain to that subnet.

3.4 Set Priority (Matrix) Agent

Given the sets provided by the observation wrapper, a simple agent can be created using the following algorithm, which manually prioritizes actions using the observation's sets:

1. If the phase changed since the last action, and this agent has a subnet which should be Blocked or Unblocked during a certain phase, Block/Unblock that subnet.

2. If the Kill set is not empty, pop a host from the set and use Kill on it. After the action is complete, move this host to the back of the Analyze queue and remove it from the Kill and Restore sets[1]. As a red agent on a server can do more damage than a red agent on a

[1]This must be done after the action completes

Agent Home Subnet	Subnet to Block	Phase
Operational Zone A	Restricted Zone A	Mission A
Restricted Zone A	Restricted Zone B	Mission A
Operational Zone B	Restricted Zone B	Mission B
Restricted Zone B	Restricted Zone A	Mission B
HQ Subnets	None	

Table 3.2: List of Blocks supported by our observation wrapper. The limited number of Blocks greatly decreases the size of the wrapped observation.

user, prioritize running Kill on server hosts before doing so user hosts.

3. If the Restore set is not empty, pop a host from the set and use Restore on that host. After the action is complete, reset the Decoy set for that host to what it was at the start of the simulation, move this host to the back of the Analyze queue, and remove it from the Kill and Restore sets. Similarly to Kill, prioritize server hosts.

4. If any of the sets D_3, D_2, or D_1 are not empty, pop a host from the set with the highest number and deploy a decoy on that host.

5. If no other items apply, take the host on top of the Analyze queue, move it to the back of the queue, and run Analyze on it.

This priority of operations can be implemented through an if-else chain or by multiplying a hardcoded matrix of priority values with the binary observation space. For simplicity, we provide a matrix for only the host portion of the observation, not the subnet portion. The matrix is diagonally tiled so that each host's observations are associated with actions on that host. Prioritization of servers can be accomplished by ordering the observation and action spaces so that server hosts come before user hosts, and implementing the matrix such that the first index which leads to the global maxima will be used as the action.

```
def prioritize_action(last_action:
    Action):
     if last_action is IN_PROGRESS:
         # Have to wait for it to
    finish
         return Sleep()
     elif last_action is SUCCESS and
      last_action is Restore or
    Remove:
         host = last_action.host
         if host in analyze_queue:
```

as the host could be re-added to either set by alerts occurring as the action executes. This would result in the action being needlessly executed a second time, wasting steps.

```
        move host to back of
analyze_queue
    remove host from kill_set
    remove host from
restore_set

if phase == phase_to_block and
not is_blocked:
    Block subnet specified in
Block table
    set is_blocked to true
elif is_blocked and phase !=
phase_to_block:
    Unblock subnet specified in
 Block table
    set is_blocked to false
elif kill_set is not empty:
    pop host from kill_set,
preferring servers over users
    run Kill on host
elif restore_set is not empty:
    pop host from restore_set,
preferring servers over users
    run Restore on host
    reset number of decoys on
host
elif D_n is not empty for n in
(3, 2, 1):
    pop host from D_n
    create Decoy on host
    if n > 1:
        add host to D_(n-1)
else:
    host = analyze_queue.front
    move host to back of
analyze_queue
    run Analyze on host
```

Listing 3.1: Pseudocode implementation of Set Priority Agent

Using a modified version of the environment that ran two agents in lockstep with each other, we verified that the agent behavior encoded by Table 3.3 is identical to the behavior encoded by our Python implementation.

While we did intend to add more sophisticated handling for the case where a host is in both the Kill set and the Restore set at the same time, as well as sending and using the contents of messages, we found that the agent outlined here performed quite well on its own. We did attempt some modifications to the agent that used messages to communicated about cross-subnet attacks, but this did not appear to have any affect on the agent's performance. We also attempted to take more variables into account when determining whether to use Kill or Restore, but doing so naively resulted in significantly poorer performance, so we deemed that more analysis was necessary.

However, we found one potential modification that could work for this specific scenario. As phishing emails, which are the primary cause of lateral movement, are not affected by Blocks, we entertained the idea that Blocks may not be effective in this

$$\begin{array}{ccccc} \text{Kill} & \text{Restore} & \text{Decoy} & \text{Analyze} & \\ \left[\begin{matrix} 7 \\ 0 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix}\right. & \begin{matrix} 0 \\ 6 \\ 0 \\ 0 \\ 0 \\ 0 \end{matrix} & \begin{matrix} 0 \\ 0 \\ 4 \\ 3 \\ 2 \\ 0 \end{matrix} & \left.\begin{matrix} 0 \\ 0 \\ 0 \\ 0 \\ 0 \\ 1 \end{matrix}\right] & \begin{matrix} \text{Kill set} \\ \text{Restore set} \\ D_3 \\ D_2 \\ D_1 \\ \text{Analyze set} \end{matrix} \end{array}$$

Table 3.3: Matrix encoding of Set Priority agent.

scenario by creating a version of the agent which did not perform Blocks or Unblocks. The agent's performance was completely unaffected by this change, suggesting that Blocks are not an effective method of network defense in this challenge.

Through development of this agent, we utilized a visualization tool to assist in debugging the agent. For instance, the visualization system allowed us to realize that we had made a typographical error when creating the subnet table, which drastically reduced scores until it was fixed. We believe that this visualization system allowed us to effectively audit the behavior of our agent and determine if it was missing signs of red agent infection or attempting to perform an invalid action due to an error in our coding. We believe that having a system to allow visualization or similar oversight over an agent's actions is key to agent development.

Chapter 4

Results

With the hardcoded priority matrix, the agent achieves an average score in the range of -105 to -115 over 100 episodes, with a standard deviation of around 35. This score would result in the agent placing within the top two if it were submitted to the CAGE competition, **despite not using inter-agent communication at all**. A comparison to the top 5 submitted agents is available in Table 4.1. In the remainder of this section, we will analyze our agent's performance and what in means in terms of reinforcement learning and the challenge, as well as some ways that our solution could be improved through selective addition of artificial intelligence.

4.1 Analysis

For the sake of analysis, we modified the environment so that we could decompose the score and determine when, where, and why the agent received penalties. We discovered that the majority of our penalties were caused by the red agent doing damage to contractor hosts during the planning phase. While the contractor network does have a penalty associated with it during the planning phase, we found that all penalties related to the contractor network were caused by failures of LocalWork within the contractor network itself. As there is no way for any blue agent to defend the contractor network, we re-evaluated what agents we could using a modified environment that did not count penalties for LocalWork in the contractor network, which can be seen in Table 4.2.

For our agent, nearly all of our remaining penalties are as a result of running Restore. Removing the penalty for running Restore increases our score further to -8.9 ± 8.1, and Cybermonic's score to -54.4 ± 46.0.

But what penalties actually led to these scores? To test this, we modified the

CAGE 4 Submission Scores

Team or Agent Name	Score
Set Priority Agent	-110.4 ± 33.8
UC	-113.8
LANCER	-118.5
PUNCH	-142.7
Cybermonic	-193.7 ± 65.2
BlueSTAR	-302.0
Sleep Agent	-6621.5 ± 1483.4

Table 4.1: Set Priority Agent compared to the top 5 submissions to the CAGE 4 competition, plus a "Sleep Agent" that takes no actions during the game. As few teams have made their agent's code publicly available, standard deviations for most of the agents cannot be given.

Scores without Contractor Penalties

Agent	Score
Set Priority Agent	-41.4 ± 8.4
Cybermonic	-135.1 ± 61.0
Sleep	-6646.1 ± 1325.2

Table 4.2: Scores with contractor subnet penalties removed, as it is impossible to defend the contractor subnet.

environment to dump information about penalties and certain actions to a file and ran it for 100 episodes, aiming to determine the following:

- Which subnets, ignoring the contractor network, result in penalties for the blue agent.
- When these penalties occur.
- The circumstances which lead to a host being placed in both the Kill set and the Restore set, and if the amount of time since the blue agent was first alerted to a malicious process on a host can help determine if the process has Escalated.
- How common use of Kill is on hosts where the red agent has already Escalated.
- How common it is for red agents to Escalate while Kill is still executing.

In Figure 4.1, we graph the number of penalties received in each subnet over the course of the 100 episodes. The disproportionately large number of penalties received for OT service kills in the operational zone subnets is due to these subnets being guaranteed to have an OT service on each host, as opposed to only appearing at random. To demonstrate this, we graph the number of attempted OT service kills for

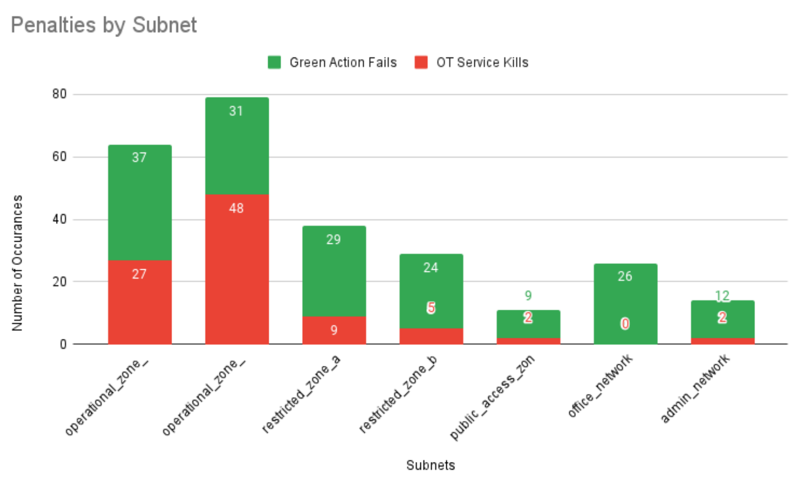

Figure 4.1: Penalties separated by subnet. This counts the number of penalties, not how many points they were worth. Green bars indicate failed green agent actions, while red bars indicate successful OT service kills.

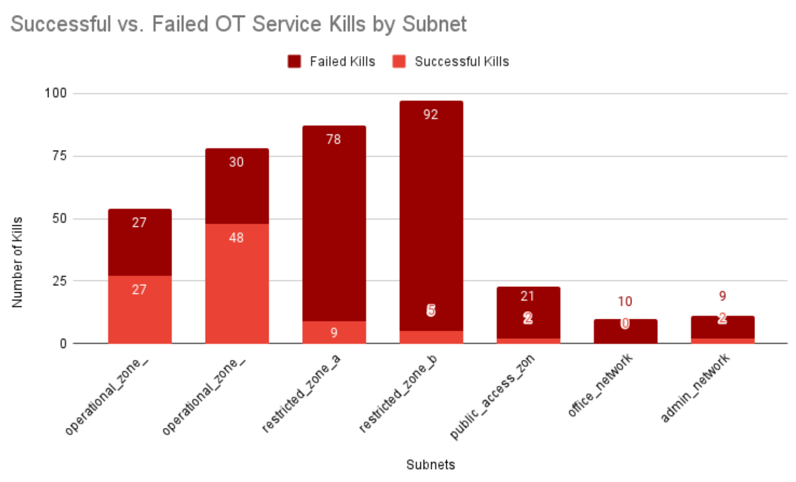

Figure 4.2: Attempted OT service kills separated by subnet.

each subnet in Figure 4.2, which shows that the red agent is attempting to kill the OT service more in restricted zones than in the operational zones. This is likely due to red agent's being more likely to enter the restricted zone via phishing emails, but being quickly detected and evicted before the OT service can actually be killed.

A trend in both figures is the red agent's near complete failure to disrupt activity in the public access and administrative subnets. We believe that the reason for this is that these two subnets, along with the office subnet, are shared by a single red agent, much like how they are shared by a single blue agent as seen in Figure 2.1. As a red agent chooses which host to act on completely randomly, having to choose between three subnets worth of hosts makes it more likely that the HQ red agent will attempt to spread itself out to other hosts instead of focusing on securing its position on hosts it has already infected. As a result, the HQ blue agent is much more likely to detect the red agent before it can deal damage to an infected host. However, this does not explain why the office subnet was disproportionately affected by green agent failures; we are unsure as to why it occurred in our data, and attempts to collect more data were inconclusive in determining if this was a pattern.

We were initially concerned about the amount of successful OT service kills in the operational zones, as those have the potential for high penalties depending on which phase they occur. As such, we graphed the penalties for only the two operational subnets, aggregated by timestamp, as seen in Figure 4.3. The vast majority of the penalties occur during the planning phase, during which the blue agents have yet to finish creating Decoys. The lack of Decoys both reduces the risk of a red agent causing an alert during a Portscan and nullifies the risk of an Exploit failing. This allows the red agent to move between hosts in a subnet much faster, increasing the risk for damage. Additionally, there is a chance that a red agent will delay Discovering other hosts and instead focus on dealing damage to the one host it has access to. As the blue agent will be too busy creating Decoys to run Analyze, this will go undetected until the red agent attempts to Portscan or Exploit another host. Regardless, once the Decoys are set up, the red agent's false negative rate sharply decreases and regular scans can detect otherwise silent Escalations, making it difficult to hide.

Out of curiosity, we also decided to graph the number of Kills vs the num-

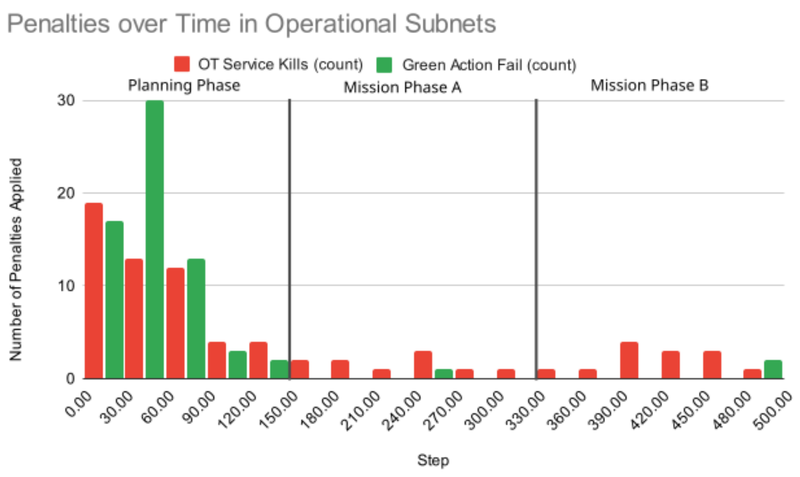

Figure 4.3: Penalties in operational subnets separated by the time at which they occurred. Grey bars have been added to approximately separate the phases.

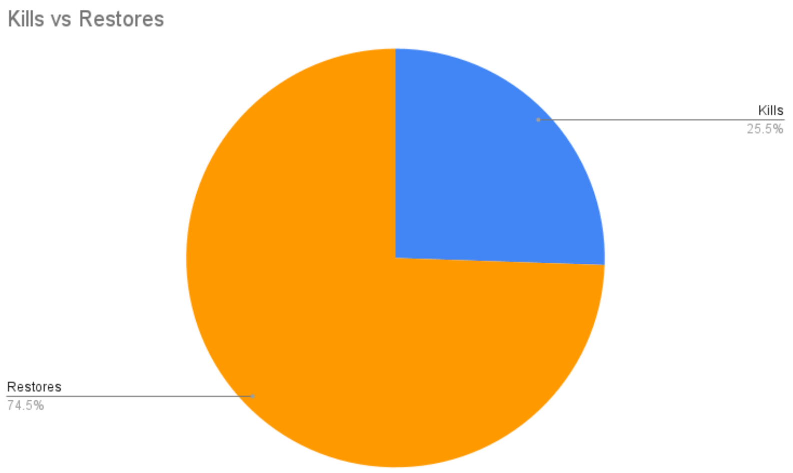

Figure 4.4: Comparison of number of Kills to number of Restores.

ber of Restores as Figure 4.4. This shows that a large proportion of reactions were Restores rather than Kills, meaning that the blue agent did not know which PID to kill. This is likely because infections from phishing emails are more common than infections as a result of lateral movement, as the blue agent is adept at stopping attempts at lateral movement quickly but cannot stop phishing emails from infecting hosts. We intended to graph this by subnet, but there was no meaningful difference to this ratio between subnets.

We intended to discuss the case where a host is added to both the Kill set and the Restore set, and how we could potentially use other factors, such as how long it had been since the infection had occurred and how large the subnet was, to estimate whether or not the host was actually rooted. However, it turned out that this had only occurred 7 times in our dataset, and in every one of those instances the red had not Escalated that host by the time an action was taken in response. Additionally, a response was almost always taken within 1 step of the agent becoming aware of the issue, rendering the number of steps since the initial process alert unsuitable as an input to differentiate between the two cases, as shown in Figure 4.5.

The case where Kill was used on a host where the red agent had already Escalated occurred twice in our dataset. In both cases, the blue agent had only become aware of the red agent's presence on the turn it reacted. Additionally, the case where the red agent Escalated after the blue agent had already begun running Kill, causing the Kill to silently fail, occurred only 19 times. As a result, we are unable to provide meaningful commentary.

4.2 Discussion

From the results above, it is clear that this simple manually created agent performs surprisingly well against RL agents submitted to the competition. As most teams have not made their code available, we cannot create a thorough analysis of strategies used by other agents. However, we believe the reason that agents encountered difficulty achieving a high score is due to aspects of the challenge itself that may confuse RL algorithms. Additionally, we believe that the fact that this score was achievable with a rather simple algorithm says more about the simulation than it does the algorithm.

The environment itself poses challenges that make it difficult for RL agents to converge on a working policy. Namely,

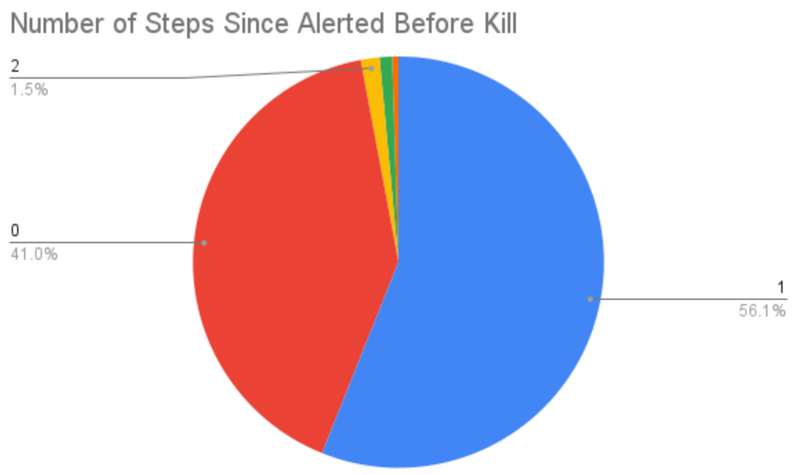

Figure 4.5: Number of steps that pass in between an agent first receiving a process alert and the agent using Kill in response to that alert. The equivalent pie chart for Restores is extremely similar.

there is a very poor relationship between actions and rewards, and there are no positive rewards at all, both of which hinder agent learning greatly. A situation like this would greatly benefit from per-host and per-agent rewards, as this would allow an actor-critic algorithm to determine which hosts should have been acted on at what times and critique the actor accordingly. Although there are per-turn rewards, the cause of the reward is not necessarily related to what the actor was currently or recently doing, especially if another agent was the cause of the reward. When the actor begins training, it is not clear that actions, observations, and penalties from a certain host have little to no relation with the actions, observations, and penalties from other hosts, and this lack of specific feedback only impedes the actor from learning this fact.

One possible solution to this issue is using a graph-based observation. By separating the hosts into graph nodes, the algorithm can more easily establish a relationship between actions, observations, and hosts, even without per-host rewards. At least one other team, Cybermonic, appears to have implemented this solution, with their agent managing to achieve 5th place with 50,000 iterations of training. However, we would rather not speak for them, so we will not analyze their agent thoroughly. Regardless, their method for avoiding these issues seems to be relatively successful.

Somewhat more concerning is that a simple if-else chain such as this was able to perform so well in such an environment. We originally intended to use this agent to assist in initial training of a later, RL-powered agent, but were surprised by its performance even when using a buggy version that would block the wrong subnets, and continue to find its performance relative to other submissions to the challenge equally surprising. We believe that the reason for the high score of our agent comes from the challenge being too simple, and not an accurate representation of a real world scenario. Specifically, we believe that it is too easy to distinguish false positive alerts from true positive alerts, as discussed in Section 2.6, and that the red agent used in the challenge is not sophisticated enough to emulate a real attacker.

Although we did not analyze the red agent's behavior when creating our agent, our agent performs excellently against the red agent because our agent assumes that the red agent does not try to deceive us. While the red agent has no actions explicitly pertaining to deception, it has the ability to remove all of it's sessions on a host at any time while leaving damage caused by Degrade. However, the red agent in the challenge never uses this ability. Additionally, the red agent selects a host completely randomly when deciding what to do next, with no regard for what it was doing on the previous step. This means that the red agent does not have much of a strategy for taking over the network or performing lateral movement, and it can be countered by simply reacting to its presence as opposed to trying to spot a pattern in its actions, as there is none.

Chapter 5

Future Work

5.1 Reward Shaping

One potential way to bootstrap training of an RL algorithm, especially in a scenario where rewards or sparse or are not a good indicator of whether individual actions are a step in the right direction or not, is to modify the rewards to perform reward shaping. In reward shaping, the rewards from the environment are augmented or replaced with rewards that are tailored to entice the agent into performing actions that bring it close to the real rewards. Reward shaping is effective in helping agent converge more quickly [Bates et al., 2023].

An alternative is to initialize the model of the agent itself to one that is known to work decently, and then let the agent train from there. This helps the agent get over the initial stage of randomly performing actions in the hopes of getting some reward and instead focus on improving an already functioning model. However, depending on the hyperparameters, scenario, and initial model, it is possible for the agent to converge on a non-optimal local maxima due to being unwilling to stray too far from the model. This can occur if the global optimal policy is too different from the initial model for the agent to converge towards. Additionally, as models get more complex, it becomes more difficult to select "good" initial values as doing so requires expertise in both the domain and the model in question [Hu et al., 2020].

We planned to use our agent to perform either reward shaping, by rewarding an agent for following the same "rules" as our agent did, or by model initialization, but we decided not to do so. This decision was spurred by the reward system for the challenge being extremely negative, causing us to believe that to counter the negative effect of the challenges rewards, we would have to make our shaping rewards so large that we would end up with an extremely overfitted model. Regardless, we believe that a simple manual agent such as ours has potential to be used in reward shaping, and we would like to see this idea further explored in future works.

5.2 LLMs

Although the challenge is based around the use of model-based reinforcement learning techniques, Large Language Models (LLMs) have also been used for reinforcement learning Monea et al. [2024]. LLMs were even the subject of a previous paper by Rigaki et al. [2024], but this paper used a pre-trained LLM, and did not train an LLM specifically for the purpose of cybersecurity. Regardless, the LLM agent performed remarkably well with a simple prompt. We would like to see how an LLM specifically trained for this challenge might be able to perform against an LLM not specifically trained for this challenge. We would especially like to see how such LLMs would be able to explain and justify their actions on each step, which would make monitoring the actions an agent takes much easier. Similarly to our proposal for reward shaping, we would like to use the set priority agent shown in this paper as a base for LLM training. We intend to release a paper covering our efforts to do so in the future.

5.3 Selective Use of A.I.

Although our agent works for the purposes of this challenge, there are two particular areas of uncertainty that might benefit from the use of machine learning for the purposes of binary classification.

1. How do we determine if a communication, file, or process alert is a false positive or not?
2. If we know that a host has malicious processes, after what point should we assume the process has Escalated? In the challenge, the red agent may not Escalate on the host before using it to attack other hosts, and the red agent selects a host to act on completely randomly, so it is difficult to determine if the red agent may have acted on a particular host or not.

Unfortunately, this challenge is not particularly conducive to either of these. As previously mentioned in Section 2.6, the connection alerts in this challenge have easy-to-spot patterns that can reliably distinguish between red agent activity and green agent activity, while file and process alerts can never be false positives. Meanwhile, the randomness inherent to the red agent of this challenge makes it unlikely that Escalation will occur before the blue agent has a chance to respond, making it difficult to train a classifier on the resulting data. The blue agent already responds to alerts as soon as possible, as seen in Figure 4.5, and in most cases is able to do so on the same step that it is first alerted to the presence of the red agent.

Outside of CAGE 4, performing this kind of detection is rather difficult, as cybersecurity analysts are often performing "treadmill work", trying to catch up to attackers as new exploits are discovered and used maliciously [Gyimah et al., 2024]. Such rapid work, combined with the difficulty of analyzing potentially malicious code and data using machine learning [Erdemir et al., 2024] and the number of samples required to train an RL agent makes it difficult to use such agents for a small but very open-ended subtask such as "Figure out if this is a false positive." As for the second potential subtask, it has limited applicability in the real world, as infections frequently take a long time to be discovered [Gyimah et al., 2024]; much longer than the rapid "steps" provided by the simulation.

5.4 Difficulty of Challenges

Most importantly, we believe that the simulation itself should be modified to be more realistic. Mainly, we would prefer that the scenario not have a method for reliable distinction of false and true positive alerts and that red agents be more strategic, as well as there being a wider variety of red agent strategies to prevent overfitting against a single strategy.

Overall, we believe that the challenge is too simplistic to serve as an accurate representation of cybersecurity scenarios, This is somewhat concerning, as the point of the challenge is to evaluate how well various methods of training work in a real world scenario, and this simplicity somewhat detracts from those results. Although it is expected for a security agent to work a high level, a red agent that does not act with any strategy and the ability to reliably distinguish between false and true positives mean that this challenge can be solved with a manually designed algorithm. Additionally, while this challenge emphasizes the use of agent communication, our solution is very successful despite not using it at all, unlike the expert solution used by Hicks et al. [2023] in CAGE 3. While some level simplicity may be necessary to train AI agents in a reasonable time, we believe that the challenge should be more difficult for humans to create a state-of-the-art solution to.

Chapter 6

Conclusion

We have shown that a manual solution is capable of meeting the CAGE 4 challenge despite not exploiting all the tools at its disposal, such as inter-agent communication and, depending on the version of the agent used, even subnet Blocks. We have analyzed the performance of this agent and determined that while there are locations of uncertainty in the agent which might benefit from the additional of artificial intelligence to recognize patterns that we did not see, the environment is ultimately too simplistic yet random to pick out any meaningful data. While it could be argued that our solution misses the point of the challenge, that being to analyze the effectiveness of different methods of training in a cybersecurity environment, we would like to also question if an environment that can be solved through extremely simple pattern matching is an effective way to measure the performance of a training method.

In the future, we would like to see later iterations of the challenge introduce patterns that are more difficult for humans to pick out simply by looking at the observations. As there are few other papers in the literature that cover this challenge at time of writing, we are uncertain of the exact methods used by other teams, but we would like to see why such methods returned the results they did and especially why they failed to match the performance of a simple agent like this. In addition, we would like to focus on more unconventional solutions to these challenges that do not rely on traditional RL models and neural networks.

Appendix A

Additional Files

This thesis submission includes the source code files for the agent which we created. This agent may be run by following the instructions described in [Group, 2023]. We also include the raw results dumped by our modified environment.

- `combined_results.csv`
- `set_priority_wrapped`
  - `dummy_agent.py`
  - `SetBasedWrapper.py`
  - `submission.py`
- `set_priority`
  - `dummy_agent.py`
  - `submission.py`

Bibliography

Elizabeth Bates, Vasilios Mavroudis, and Chris Hicks. Reward shaping for happier autonomous cyber security agents. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, AISec '23, page 221–232, New York, NY, USA, 2023. Association for Computing Machinery. ISBN 9798400702600. doi:10.1145/3605764.3623916. URL https://doi.org/10.1145/3605764.3623916.

Douglas C. Crowder, Darrien M. McKenzie, Matthew L. Trappett, and Frances S. Chance. Hindsight experience replay accelerates proximal policy optimization, 2024. URL https://arxiv.org/abs/2410.22524.

Ecenaz Erdemir, Kyuhong Park, Michael J. Morais, Vianne R. Gao, Marion Marschalek, and Yi Fan. Score: Syntactic code representations for static script malware detection, 2024. URL https://arxiv.org/abs/2411.08182.

Myles Foley, Chris Hicks, Kate Highnam, and Vasilios Mavroudis. Autonomous network defence using reinforcement learning. In *Proceedings of the 2022 ACM on Asia Conference on Computer and Communications Security*, ASIA CCS '22, page 1252–1254, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450391405. doi:10.1145/3488932.3527286. URL https://doi.org/10.1145/3488932.3527286.

Myles Foley, Mia Wang, Zoe M, Chris Hicks, and Vasilios Mavroudis. Inroads into autonomous network defence using explained reinforcement learning, 2023. URL https://arxiv.org/abs/2306.09318.

TTCP CAGE Working Group. Ttcp cage challenge 4. https://github.com/cage-challenge/cage-challenge-4, 2023.

Nana Kankam Gyimah, Judith Mwakalonge, Gurcan Comert, Saidi Siuhi, Robert Akinie, Methusela Sulle, Denis Ruganuza, Benibo Izison, and Arthur Mukwaya. An automl-based approach for network intrusion detection, 2024. URL https://arxiv.org/abs/2411.15920.

Chris Hicks, Vasilios Mavroudis, Myles Foley, Thomas Davies, Kate Highnam, and Tim Watson. Canaries and whistles: Resilient drone communication networks with (or without) deep reinforcement learning. In *Proceedings of the 16th ACM Workshop on Artificial Intelligence and Security*, CCS '23, page 91–101. ACM, November 2023. doi:10.1145/3605764.3623986. URL http://dx.doi.org/10.1145/3605764.3623986.

Yujing Hu, Weixun Wang, Hangtian Jia, Yixiang Wang, Yingfeng Chen, Jianye Hao, Feng Wu, and Changjie Fan. Learning to utilize shaping rewards: A new approach of reward shaping, 2020. URL https://arxiv.org/abs/2011.02669.

Giovanni Monea, Antoine Bosselut, Kianté Brantley, and Yoav Artzi. Llms are in-context reinforcement learners, 2024. URL https://arxiv.org/abs/2410.05362.

Andrew Y. Ng, Daishi Harada, and Stuart J. Russell. Policy invariance under reward transformations: Theory and application to reward shaping. In *Proceedings of the Sixteenth International Conference on Machine Learning*, ICML '99, page 278–287, San Francisco, CA, USA, 1999. Morgan Kaufmann Publishers Inc. ISBN 1558606122.

Maria Rigaki, Ondřej Lukáš, Carlos Catania, and Sebastian Garcia. Out of the cage: How stochastic parrots win in cyber security environments. In *Proceedings of the 16th International Conference on Agents and Artificial Intelligence*, page 774–781. SCITEPRESS - Science and Technology Publications, 2024. doi:10.5220/0012391800003636. URL http://dx.doi.org/10.5220/0012391800003636.

Maxwell Standen, Martin Lucas, David Bowman, Toby J. Richer, Junae Kim, and Damian Marriott. Cyborg: A gym for the development of autonomous cyber agents, 2021. URL https://arxiv.org/abs/2108.09118.