Understanding the Cyberwheel Framework: A Mathematical and Computational Foundation for Autonomous Cyber Defense Training

Technical Documentation

July 23, 2025

Abstract

This document provides a comprehensive mathematical and technical explanation of the Cyberwheel framework, a reinforcement learning simulation environment designed for training autonomous cyber defense agents. We present the mathematical formulations underlying network representation, observation spaces, action spaces, reward mechanisms, and the specific implementation of blue agent observations. The framework addresses critical limitations in existing cybersecurity training environments by providing high-fidelity simulations that can scale to networks with millions of nodes while maintaining realistic attack and defense scenarios.

Contents

1 Introduction

1.1 What is Cybersecurity?

Cybersecurity is the practice of protecting digital systems, networks, and data from malicious attacks. In traditional cybersecurity, human analysts monitor networks, detect threats, and respond to incidents. However, modern cyber attacks are becoming increasingly sophisticated and frequent, overwhelming human defenders.

1.2 The Need for Autonomous Defense

Autonomous cyber defense refers to AI systems that can automatically detect, analyze, and respond to cyber threats without human intervention. These systems use machine learning, particularly reinforcement learning (RL), to learn optimal defense strategies through interaction with simulated environments.

1.3 Cyberwheel Overview

Cyberwheel is a simulation framework that creates realistic virtual networks where AI agents can be trained to perform cybersecurity tasks. It models both attacking agents (red agents) and defending agents (blue agents) in a game-like environment where they interact and learn from each other.

2 Mathematical Foundations

2.1 Network Representation

2.1.1 Graph-Theoretic Foundation

A network in Cyberwheel is represented as a directed graph $G = (V, E)$ where:

$$G = (V, E) \tag{1}$$
$$V = H \cup S \cup R \tag{2}$$
$$E \subseteq V \times V \tag{3}$$

Where:

- H = set of hosts (computers/devices)
- S = set of subnets (network segments)
- R = set of routers (network infrastructure)
- E = set of directed edges representing network connections

2.1.2 Host Modeling

Each host $h_i \in H$ is characterized by a state vector:

$$h_i = \langle \mathrm{IP}_i, \mathrm{OS}_i, \mathcal{S}_i, \mathcal{V}_i, \mathrm{compromised}_i, \mathrm{decoy}_i \rangle \tag{4}$$

Where:

- IP_i = IP address assignment
- OS_i = operating system type
- $\mathcal{S}_i$ = set of running services
- $\mathcal{V}_i$ = set of vulnerabilities (CVEs)
- $\mathrm{compromised}_i \in \{0, 1\}$ = compromise status
- $\mathrm{decoy}_i \in \{0, 1\}$ = whether host is a honeypot

2.1.3 Network State

The complete network state at time t is:

$$\mathcal{N}_t = \langle G, \{h_i\}_{i=1}^{|H|}, \{s_j\}_{j=1}^{|S|}, \{r_k\}_{k=1}^{|R|} \rangle \tag{5}$$

2.2 Agent Framework

2.2.1 Red Agent (Attacker)

The red agent represents a cyber attacker following the MITRE ATT&CK framework. Its state includes:

$$\mathrm{RedState}_t = \langle \mathrm{position}_t, \mathrm{knowledge}_t, \mathrm{killchain}_t \rangle \tag{6}$$

Where:

- $\mathrm{position}_t \in H$ = current compromised host
- $\mathrm{knowledge}_t \subseteq H \times S$ = discovered network information
- $\mathrm{killchain}_t \in \{\text{discovery}, \text{reconnaissance}, \text{privilege-escalation}, \text{impact}\}$

2.2.2 Blue Agent (Defender)

The blue agent represents the cyber defender with state:

$$\mathrm{BlueState}_t = \langle \mathcal{D}_t, \mathcal{A}_t, \mathrm{budget}_t \rangle \tag{7}$$

Where:

- $\mathcal{D}_t$ = set of deployed decoys
- $\mathcal{A}_t$ = alert history
- $\mathrm{budget}_t \in \mathbb{R}^+$ = available resources

3 Observation Spaces

3.1 Blue Agent Observation Space

The blue agent's observation space is the most critical component for effective defense. Let's examine the mathematical formulation of the `BlueObservation` class.

3.1.1 Observation Vector Structure

The observation vector $\mathbf{o}_t \in \mathbb{R}^n$ has the following structure:

$$\mathbf{o}_t = \begin{bmatrix} \mathbf{o}_t^{\text{current}} \\ \mathbf{o}_t^{\text{history}} \\ \mathbf{o}_t^{\text{meta}} \end{bmatrix} \tag{8}$$

Where:

- $\mathbf{o}_t^{\text{current}} \in \{0,1\}^{|H|}$ = current timestep alerts
- $\mathbf{o}_t^{\text{history}} \in \{0,1\}^{|H|}$ = cumulative alert history
- $\mathbf{o}_t^{\text{meta}} \in \mathbb{R}^2$ = metadata (padding and decoy count)

3.1.2 Current Alert Vector

For the current timestep alerts:

$$o_t^{\text{current}}[i] = \begin{cases} 1 & \text{if } \exists a \in \mathcal{A}_t : a.\text{src\_host} = h_i \\ 0 & \text{otherwise} \end{cases} \tag{9}$$

Where $\mathcal{A}_t$ is the set of alerts at time t, and h_i is the i-th host in the network.

3.1.3 Historical Alert Vector

The historical component maintains persistent memory:

$$o_t^{\text{history}}[i] = \max_{\tau=0}^{t} o_\tau^{\text{current}}[i] \tag{10}$$

This creates a "sticky" memory where once a host generates an alert, it remains flagged in the history.

3.1.4 Complete Observation Vector

The complete observation vector construction in the code corresponds to:

3.2 Observation Space Dimensionality

The total observation space dimension is:

$$\dim(\mathcal{O}_{\text{blue}}) = 2|H| + 2 \tag{11}$$

Where:

- $2|H|$ accounts for current and historical alert vectors
- $+2$ accounts for metadata (padding and decoy count)

Algorithm 1 Blue Observation Vector Construction

```
1: Initialize $\mathbf{o}_t \leftarrow \mathbf{0}^n$
2: barrier $\leftarrow |H|/2$
3: for $i = 0$ to barrier $- 1$ do
4:     $o_t[i] \leftarrow 0$ {Reset current alerts}
5: end for
6: for each alert $a \in \mathcal{A}_t$ do
7:     if $a$.src_host $\in$ mapping then
8:         index $\leftarrow$ mapping[$a$.src_host]
9:         $o_t$[index] $\leftarrow 1$ {Current alert}
10:        $o_t$[index + barrier] $\leftarrow 1$ {History}
11:    end if
12: end for
13: $o_t[n-2] \leftarrow -1$ {Padding}
14: $o_t[n-1] \leftarrow |\mathcal{D}_t|$ {Decoy count}
```

4 Action Spaces

4.1 Blue Agent Action Space

The blue agent has a discrete action space $\mathcal{A}_{\text{blue}}$ consisting of:

$$\mathcal{A}_{\text{blue}} = \mathcal{A}_{\text{deploy}} \cup \mathcal{A}_{\text{remove}} \cup \{\text{nothing}\} \tag{12}$$

4.1.1 Deployment Actions

For each subnet $s_j \in S$ and decoy type $d_k \in \mathcal{D}_{\text{types}}$:

$$\mathcal{A}_{\text{deploy}} = \{(\text{deploy}, s_j, d_k) : s_j \in S, d_k \in \mathcal{D}_{\text{types}}\} \tag{13}$$

4.1.2 Removal Actions

Similarly for removal:

$$\mathcal{A}_{\text{remove}} = \{(\text{remove}, s_j, d_k) : s_j \in S, d_k \in \mathcal{D}_{\text{types}}\} \tag{14}$$

4.1.3 Action Space Size

The total action space size is:

$$|\mathcal{A}_{\text{blue}}| = 2 \cdot |S| \cdot |\mathcal{D}_{\text{types}}| + 1 \tag{15}$$

4.2 Red Agent Action Space

The red agent follows a killchain with phase-specific actions:

$$\mathcal{A}_{\text{red}} = \bigcup_{p \in \text{KillChain}} \mathcal{A}_p \tag{16}$$

Where killchain phases include:

- $\mathcal{A}_{\text{discovery}}$ = network scanning actions
- $\mathcal{A}_{\text{reconnaissance}}$ = information gathering
- $\mathcal{A}_{\text{privilege-escalation}}$ = elevation attempts
- $\mathcal{A}_{\text{impact}}$ = damage/disruption actions

5 Reward Functions

5.1 General Reward Structure

The reward function $R : \mathcal{S} \times \mathcal{A}_{\text{red}} \times \mathcal{A}_{\text{blue}} \to \mathbb{R}$ considers both immediate and recurring rewards:

$$R_t = R_t^{\text{immediate}} + R_t^{\text{recurring}} \tag{17}$$

5.2 Blue Agent Reward Function

For the blue agent, the reward considers deception effectiveness:

$$R_t^{\text{blue}} = R_{\text{red}}^{\text{deception}} + R_{\text{blue}}^{\text{action}} + R_t^{\text{recurring}} \tag{18}$$

5.2.1 Deception Reward

When red agent attacks a decoy:

$$R_{\text{red}}^{\text{deception}} = \begin{cases} 10 \cdot |R_{\text{red}}^{\text{base}}| & \text{if target is decoy and attack succeeds} \\ -|R_{\text{red}}^{\text{base}}| & \text{if target is real host and attack succeeds} \\ 0 & \text{otherwise} \end{cases} \tag{19}$$

5.2.2 Action Cost

Blue actions have associated costs:

$$R_{\text{blue}}^{\text{action}} = \begin{cases} R_{\text{deploy}} & \text{if action is deploy and succeeds} \\ R_{\text{remove}} & \text{if action is remove and succeeds} \\ R_{\text{failure}} & \text{if action fails} \\ 0 & \text{if action is nothing} \end{cases} \tag{20}$$

5.2.3 Recurring Rewards

Recurring rewards model ongoing costs/benefits:

$$R_t^{\text{recurring}} = \sum_{i=1}^{|\mathcal{D}_t|} R_{\text{maintain}} + \sum_{j=1}^{|\mathcal{C}_t|} R_{\text{compromise}} \tag{21}$$

Where:

- $\mathcal{D}_t$ = active decoys at time t
- $\mathcal{C}_t$ = compromised hosts at time t
- $R_{\text{maintain}} < 0$ = maintenance cost per decoy
- $R_{\text{compromise}} < 0$ = ongoing damage per compromised host

6 Learning Dynamics

6.1 Reinforcement Learning Formulation

The cyber defense problem is formulated as a Markov Decision Process (MDP):

$$\text{MDP} = \langle \mathcal{S}, \mathcal{A}, \mathcal{P}, \mathcal{R}, \gamma \rangle \tag{22}$$

Where:

- $\mathcal{S}$ = state space (network configurations)
- $\mathcal{A}$ = action space (defense actions)
- $\mathcal{P} : \mathcal{S} \times \mathcal{A} \times \mathcal{S} \to [0, 1]$ = transition probabilities
- $\mathcal{R} : \mathcal{S} \times \mathcal{A} \to \mathbb{R}$ = reward function
- $\gamma \in [0, 1]$ = discount factor

6.2 Policy Optimization

The blue agent learns a policy $\pi_{\text{blue}} : \mathcal{S} \to \mathcal{A}$ that maximizes expected cumulative reward:

$$\pi^* =_\pi \mathbb{E}\left[\sum_{t=0}^{\infty} \gamma^t R_t | \pi\right] \tag{23}$$

6.3 Proximal Policy Optimization (PPO)

Cyberwheel uses PPO with the objective:

$$L^{\text{PPO}}(\theta) = \mathbb{E}_t\left[\min\left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1 - \epsilon, 1 + \epsilon)\hat{A}_t\right)\right] \tag{24}$$

Where:

- $r_t(\theta) = \frac{\pi_\theta(a_t|s_t)}{\pi_{\theta_{\text{old}}}(a_t|s_t)}$ = probability ratio
- $\hat{A}_t$ = estimated advantage at time t
- ϵ = clipping parameter (typically 0.2)

7 Alert and Detection Mechanisms

7.1 Alert Generation

When a red agent performs an action, it potentially generates an alert a_t:

$$a_t = \langle \text{src\_host}, \text{dst\_hosts}, \text{techniques}, \text{services}, \text{timestamp} \rangle \tag{25}$$

7.2 Detection Probability

Each detector has a probability $p_{\text{detect}}(a, d)$ of detecting action a:

$$p_{\text{detect}}(a, d) = \prod_{i=1}^{|a.\text{techniques}|} p_i^{(d)} \tag{26}$$

Where $p_i^{(d)}$ is the detection probability for technique i by detector d.

7.3 False Positive Generation

Detectors may also generate false positives with rate λ_{fp}:

$$P(\text{false positive at time } t) = 1 - e^{-\lambda_{\text{fp}} \Delta t} \tag{27}$$

8 Core Architecture Implementation

8.1 Main Entry Point

The system provides four main operational modes through `__main__.py`:

Listing 1: Main CLI Interface Implementation

```
# cyberwheel/__main__.py
import sys
from cyberwheel.utils import (train_cyberwheel,
   evaluate_cyberwheel,
                             run_cyberwheel,
                                run_visualization_server)

if __name__ == "__main__":
    if len(sys.argv) > 2:
        mode = sys.argv.pop(1)  # train, evaluate, visualizer,
           run
        config = sys.argv.pop(1)

        if mode == 'train':
            train_cyberwheel(args)
        elif mode == 'evaluate':
            evaluate_cyberwheel(args)
        elif mode == 'visualizer':
            run_visualization_server(config)
        elif mode == 'run':
            run_cyberwheel(args)
```

8.2 Network Representation Implementation

The network is implemented using NetworkX graphs in `network_base.py`:

Listing 2: Network Base Implementation

```
# cyberwheel/network/network_base.py
import networkx as nx
from cyberwheel.network.host import Host
from cyberwheel.network.subnet import Subnet
from cyberwheel.network.router import Router

class Network:
    def __init__(self, name: str = "Network", graph: nx.Graph =
        None):
        self.graph = graph if graph else nx.DiGraph(name=name)
        self.name = name

        # Categorize network components
        self.hosts = {name: host for name, host in self
                     if isinstance(host, Host)}
        self.subnets = {name: subnet for name, subnet in self
                       if isinstance(subnet, Subnet)}
        self.decoys = {hn: host for hn, host in self.hosts
                      if host.decoy}

        # Separate user and server hosts
        self.user_hosts = HybridSetList({hn for hn, host in self.
           hosts
                                         if "workstation" in host.
                                            host_type.name.lower()
                                            })
        self.server_hosts = HybridSetList({hn for hn, host in
           self.hosts
                                           if "server" in host.
                                              host_type.name.lower
                                              ()})
```

8.3 Host Modeling Implementation

Each host is modeled with comprehensive attributes:

Listing 3: Host Implementation

```
# cyberwheel/network/host.py
class Host(NetworkObject):
    def __init__(self, name: str, host_type: HostType, subnet:
        Subnet):
        super().__init__(name)
        self.host_type = host_type
        self.subnet = subnet
        self.services = []  # Running services
        self.vulnerabilities = set()  # CVE identifiers
        self.compromised = False
        self.decoy = False  # Whether this is a honeypot
        self.os = host_type.os

    def add_vulnerability(self, cve: str):
        """Add CVE vulnerability to host"""
        self.vulnerabilities.add(cve)

    def is_vulnerable_to(self, technique):
        """Check if host is vulnerable to specific attack
           technique"""
        return bool(self.vulnerabilities.intersection(technique.
           cve_list))
```

8.4 Blue Agent Implementation

The blue agent implements defensive strategies:

Listing 4: Blue Agent Base Implementation

```
# cyberwheel/blue_agents/rl_blue_agent.py
class RLBlueAgent(BlueAgent):
    def __init__(self, network: Network, args: YAMLConfig):
        self.network = network
        self.observation = BlueObservation(
            shape=2 * len(network.hosts),
            mapping={host.name: i for i, host in enumerate(
               network.hosts)},
            detector_config=args.detector_config
        )
        self.action_space = self.create_action_space()

    def create_action_space(self, max_size: int):
        """Create discrete action space for decoy deployment/
           removal"""
        return spaces.Discrete(max_size)

    def act(self, action: int) -> BlueAgentResult:
        """Execute blue agent action based on RL policy decision
           """
        action_type, subnet, decoy_type = self.decode_action(
           action)

        if action_type == "deploy":
            return self.deploy_decoy(subnet, decoy_type)
        elif action_type == "remove":
            return self.remove_decoy(subnet, decoy_type)
        else:
            return BlueAgentResult("nothing", "", True, 0)
```

8.5 Blue Actions Implementation

Specific defensive capabilities are implemented as action classes:

Listing 5: Deploy Decoy Action Implementation

```python
# cyberwheel/blue_actions/actions/DeployDecoyHost.py
class DeployDecoyHost(BlueAction):
    def __init__(self, network: Network, subnet_name: str,
                 decoy_config: dict):
        self.network = network
        self.subnet_name = subnet_name
        self.decoy_config = decoy_config

    def execute(self) -> BlueAgentResult:
        """Deploy a honeypot on specified subnet"""
        subnet = self.network.subnets.get(self.subnet_name)
        if not subnet:
            return BlueAgentResult("deploy_decoy", "", False, 0)

        # Create decoy host
        decoy_host = Host(
            name=f"decoy_{len(self.network.decoys)}",
            host_type=self.decoy_config["type"],
            subnet=subnet
        )
        decoy_host.decoy = True

        # Add to network
        self.network.add_host(decoy_host)
        self.network.decoys[decoy_host.name] = decoy_host

        return BlueAgentResult("deploy_decoy", decoy_host.name,
           True, 1)
```

8.6 Red Agent Implementation

The red agent follows MITRE ATT&CK killchain phases:

Listing 6: ART Agent Implementation

```python
# cyberwheel/red_agents/art_agent.py
class ARTAgent(RedAgent):
    def __init__(self, network: Network, args: YAMLConfig):
        self.network = network
        self.killchain_phases = [
            "discovery", "reconnaissance",
            "privilege-escalation", "impact"
        ]
        self.current_phase = "discovery"
        self.position = self.select_initial_host()
        self.knowledge = set()  # Discovered hosts/subnets

    def act(self) -> RedAgentResult:
        """Execute current killchain phase"""
        if self.current_phase == "discovery":
            return self.discovery_action()
        elif self.current_phase == "reconnaissance":
            return self.reconnaissance_action()
        elif self.current_phase == "privilege-escalation":
            return self.privilege_escalation_action()
        elif self.current_phase == "impact":
            return self.impact_action()

    def discovery_action(self):
        """Perform network discovery (ping sweep, port scan)"""
        technique = self.select_discovery_technique()
        target_subnet = self.position.subnet

        # Execute ping sweep
        discovered_hosts = self.ping_sweep(target_subnet)
        self.knowledge.update(discovered_hosts)

        return RedAgentResult(
            action=technique,
            target_host=self.position,
            success=True,
            action_results=discovered_hosts
        )
```

8.7 Red Actions and MITRE ATT&CK Implementation

Attack techniques are implemented based on Atomic Red Team:

Listing 7: MITRE ATTCK Technique Implementation

```
# cyberwheel/red_actions/art_techniques.py
class ScheduledTask(Technique):
    mitre_id = "T1053.005"
    name = "Scheduled Task"
    technique_id = "attack-pattern--005a06c6-14bf-4118-afa0-
       ebcd8aebb0c9"
    kill_chain_phases = ['execution', 'persistence', 'privilege-
       escalation']
    supported_os = ['windows']
    cve_list = {'CVE-2019-1064', 'CVE-2018-8440', ...}

    atomic_tests = {
        'fec27f65-db86-4c2d-b66c-61945aee87c2': AtomicTest({
            'name': 'Scheduled Task Startup Script',
            'command': 'schtasks /create /tn "malicious_task" /sc
                onlogon /tr "cmd.exe /c calc.exe"',
                'cleanup_command': 'schtasks /delete /tn "
                   malicious_task" /f',
                'supported_platforms': ['windows']
            })
        }

    def execute(self, target_host: Host) -> bool:
        """Execute technique if host is vulnerable"""
        if not self.is_applicable(target_host):
            return False

        # Check CVE vulnerabilities
        if self.cve_list.intersection(target_host.vulnerabilities
           ):
            target_host.compromised = True
            return True
        return False
```

8.8 Detection System Implementation

Alert generation and detection mechanisms:

Listing 8: Alert and Detection Implementation

```
# cyberwheel/detectors/alert.py
class Alert:
    def __init__(self, src_host: Host = None, techniques: List[
       Technique] = [],
                 dst_hosts: List[Host] = [], services: List[
                    Service] = []):
        self.src_host = src_host
        self.techniques = techniques
        self.dst_hosts = dst_hosts
        self.services = services
        self.timestamp = time.time()

# cyberwheel/detectors/detector_base.py
class Detector:
    def __init__(self, detection_probabilities: dict):
        self.detection_probs = detection_probabilities

    def obs(self, perfect_alerts: List[Alert]) -> List[Alert]:
        """Apply detection logic to perfect alerts"""
        detected_alerts = []

        for alert in perfect_alerts:
            for technique in alert.techniques:
                prob = self.detection_probs.get(technique.
                   mitre_id, 0.0)
                if random.random() < prob:
                    detected_alerts.append(alert)
                    break
        # Add false positives
        if random.random() < self.false_positive_rate:
            detected_alerts.append(self.generate_false_positive()
               )

        return detected_alerts
```

8.9 Blue Observation Implementation

The Python implementation in `blue_observation.py` corresponds to the mathematical formulation as follows:

Listing 9: Blue Observation Vector Construction

```
# cyberwheel/observation/blue_observation.py
class BlueObservation(Observation):
    def __init__(self, shape: int, mapping: Dict[Host, int],
                 detector_config: str):
        self.offset = 2
        self.shape = shape + self.offset
        self.mapping = mapping
        self.obs_vec = np.zeros(self.shape)
        self.len_obs = shape
        self.detector = DetectorHandler(detector_config)

    def create_obs_vector(self, alerts, num_decoys):
        # Clear current alerts (first half of vector)
        barrier = self.len_obs // 2
        for i in range(barrier):
            self.obs_vec[i] = 0

        # Process alerts
        for alert in alerts:
            alerted_host = alert.src_host
            if alerted_host and alerted_host.name in self.mapping
               :
                index = self.mapping[alerted_host.name]
                self.obs_vec[index] = 1          # Current alert
                self.obs_vec[index + barrier] = 1 # History

        # Add metadata
        self.obs_vec[-2] = -1          # Padding
        self.obs_vec[-1] = num_decoys # Decoy count

        return self.obs_vec
```

8.10 Reward System Implementation

The reward calculation implements deception-based incentives:

Listing 10: Reward Function Implementation

```
# cyberwheel/reward/rl_reward.py
class RLReward(Reward):
    def calculate_reward(self, red_action: str, blue_action: str,
                        red_success: bool, blue_success: bool,
                        target_host: Host, blue_id: str = -1,
                        blue_recurring: int = 0) -> float:

        target_host_name = target_host.name
        decoy = target_host.decoy

        # Deception reward logic
        if red_success and not decoy and target_host_name in
           valid_targets:
            # Red succeeded on real host - negative reward
            r = self.red_rewards[red_action][0] * -1
            r_recurring = self.red_rewards[red_action][1] * -1
        elif red_success and decoy and target_host_name in
           valid_targets:
            # Red succeeded on decoy - large positive reward
            r = self.red_rewards[red_action][0] * 10
            r_recurring = self.red_rewards[red_action][1] * 10
        else:
            r = 0
            r_recurring = 0

        # Blue action costs
        if blue_success:
            b = self.blue_rewards[blue_action][0]
        else:
            b = 0

        # Handle recurring rewards
        if r_recurring != 0:
            self.add_recurring_red_action('0', red_action, decoy)

        if blue_recurring == -1:
            self.remove_recurring_blue_action(blue_id)
        elif blue_recurring == 1:
            self.add_recurring_blue_action(blue_id, blue_action)

        return r + b + self.sum_recurring()
```

8.11 Training and Evaluation Implementation

PPO-based training with parallel environments:

Listing 11: Training Implementation

```
# cyberwheel/utils/trainer.py
class Trainer:
    def configure_training(self):
        # Create parallel environments
        env_funcs = [make_env(self.env, self.args, self.networks,
            i, False)
                  for i in range(self.args.num_envs)]

        self.envs = (async_call(env_funcs) if self.args.async_env
                  else gym.vector.SyncVectorEnv(env_funcs))

        # Initialize PPO agent
        self.agent = RLAgent(
            self.envs.single_observation_space,
            self.envs.single_action_space
        )

        self.optimizer = optim.Adam(self.agent.parameters(),
                                    lr=self.args.learning_rate)

    def train(self, update: int):
        """Execute one training update"""
        # Collect experience
        obs = torch.zeros((self.args.num_steps, self.args.
           num_envs) +
                          self.envs.single_observation_space.shape
                             )
        actions = torch.zeros((self.args.num_steps, self.args.
           num_envs) +
                              self.envs.single_action_space.shape)
        rewards = torch.zeros((self.args.num_steps, self.args.
           num_envs))

        # Experience collection loop
        for step in range(0, self.args.num_steps):
            with torch.no_grad():
                action, logprob, _, value = self.agent.
                   get_action_and_value(obs[step])

            next_obs, reward, done, info = self.envs.step(action.
               cpu().numpy())

            obs[step] = torch.tensor(next_obs)
            actions[step] = action
            rewards[step] = torch.tensor(reward)

        # PPO update
        self.ppo_update(obs, actions, rewards)
```

8.12 Configuration System Implementation

The YAML-driven configuration system enables modularity:

Listing 12: Environment Configuration Example

```yaml
# cyberwheel/data/configs/environment/train_blue.yaml
experiment_name: "cyber_deception_training"
seed: 1
total_timesteps: 100000000
num_envs: 30
num_steps: 50

# Environment Parameters
environment: CyberwheelRL
network_config: "200-host-network.yaml"
host_config: "host_defs_services.yaml"
decoy_config: "decoy_hosts.yaml"
red_agent: "art_agent.yaml"
blue_agent: "rl_blue_agent.yaml"
reward_function: "RLReward"
detector_config: "detector_handler.yaml"

# RL Parameters
learning_rate: 2.5e-4
gamma: 0.99
clip_coef: 0.2
ent_coef: 0.01
```

Listing 13: Network Configuration Example

```yaml
# cyberwheel/data/configs/network/small-network.yaml
name: "SmallTestNetwork"
subnets:
  - name: "dmz"
    cidr: "192.168.1.0/24"
    hosts:
      - name: "web-server"
        type: "WebServer"
        ip: "192.168.1.10"
      - name: "mail-server"
        type: "MailServer"
        ip: "192.168.1.11"
  - name: "internal"
    cidr: "10.0.0.0/24"
    hosts:
      - name: "workstation-1"
        type: "WindowsWorkstation"
        ip: "10.0.0.100"
      - name: "workstation-2"
        type: "LinuxWorkstation"
        ip: "10.0.0.101"
routers:
  - name: "main-router"
    interfaces:
      - subnet: "dmz"
        ip: "192.168.1.1"
      - subnet: "internal"
        ip: "10.0.0.1"
```

8.13 Visualization Implementation

Real-time network visualization using Dash and GraphViz:

Listing 14: Visualization Server Implementation

```
# cyberwheel/utils/run_visualization_server.py
import dash
from dash import dcc, html, dash_table
import plotly.graph_objects as go
import networkx as nx

def create_network_visualization(network_state, episode_data):
    """Generate network topology visualization"""
    G = network_state.graph
    pos = nx.spring_layout(G)

    # Create nodes
    node_trace = go.Scatter(
        x=[pos[node][0] for node in G.nodes()],
        y=[pos[node][1] for node in G.nodes()],
        mode='markers+text',
        text=[node for node in G.nodes()],
        textposition="middle center",
        marker=dict(
            size=20,
            color=['red' if network_state.hosts[node].compromised
                   else 'blue' if network_state.hosts[node].decoy
                   else 'green' for node in G.nodes()]
        )
    )

    # Create edges
    edge_trace = go.Scatter(
        x=[pos[edge[0]][0] for edge in G.edges()] +
          [pos[edge[1]][0] for edge in G.edges()],
        y=[pos[edge[0]][1] for edge in G.edges()] +
          [pos[edge[1]][1] for edge in G.edges()],
        mode='lines',
        line=dict(width=2, color='gray')
    )

    return go.Figure(data=[edge_trace, node_trace])

app = dash.Dash(__name__)

@app.callback(
    dash.dependencies.Output('network-graph', 'figure'),
    [dash.dependencies.Input('step-slider', 'value')]
)
def update_graph(selected_step):
    """Update visualization based on selected timestep"""
    episode_data = load_episode_data(selected_step)
    return create_network_visualization(episode_data.
       network_state, episode_data)
```

8.14 Emulation Bridge Implementation

Integration with Firewheel for real-world testing:

Listing 15: Firewheel Emulation Integration

```
# cyberwheel/emulation/firewheel_bridge.py
class FirewheelBridge:
    def __init__(self, scenario_config: str):
        self.scenario_config = scenario_config
        self.vm_mapping = {}  # Simulation hosts -> VM instances

    def convert_scenario(self, network: Network) -> str:
        """Convert Cyberwheel network to Firewheel plugin"""
        firewheel_config = {
            "vms": [],
            "networks": [],
            "services": []
        }

        # Convert hosts to VMs
        for host_name, host in network.hosts.items():
            vm_config = {
                "name": host_name,
                "os": host.os,
                "memory": "2GB",
                "vcpus": 2,
                "interfaces": [{
                    "network": host.subnet.name,
                    "ip": str(host.ip)
                }],
                "services": [service.name for service in host.
                   services]
            }
            firewheel_config["vms"].append(vm_config)

        # Convert subnets to networks
        for subnet_name, subnet in network.subnets.items():
            network_config = {
                "name": subnet_name,
                "cidr": str(subnet.cidr)
            }
            firewheel_config["networks"].append(network_config)

        return yaml.dump(firewheel_config)
    def execute_action(self, action: str, target_host: str) ->
        bool:
        """Execute simulation action in emulated environment"""
        vm_instance = self.vm_mapping.get(target_host)
        if not vm_instance:
            return False

        # Translate action to shell commands
        commands = self.translate_action_to_commands(action)

        # Execute via SSH
        return self.execute_remote_commands(vm_instance, commands
            )
```

8.15 Scalability Features

Support for large-scale networks:

Listing 16: Large Network Generation

```
# cyberwheel/network/network_generation/large_network_generator.
   py
class LargeNetworkGenerator:
    def generate_enterprise_network(self, num_hosts: int =
        1000000,
                                    num_subnets: int = 2000) ->
                                        Network:
        """Generate large-scale enterprise network"""
        network = Network(name=f"Enterprise_{num_hosts}_hosts")

        # Generate subnet hierarchy
        subnets_per_tier = [10, 100, num_subnets]  # 3-tier
            architecture

        for tier, subnet_count in enumerate(subnets_per_tier):
            for i in range(subnet_count):
                subnet = Subnet(
                    name=f"subnet_tier{tier}_{i}",
                    cidr=f"10.{tier}.{i}.0/24"
                )
                network.add_subnet(subnet)

        # Distribute hosts across subnets
        hosts_per_subnet = num_hosts // num_subnets
        host_types = ["WindowsWorkstation", "LinuxServer", "
            WebServer"]

        for subnet in network.subnets.values():
            for j in range(hosts_per_subnet):
                host_type = random.choice(host_types)
                host = Host(
                    name=f"{subnet.name}_host_{j}",
                    host_type=HostType(host_type),
                    subnet=subnet
                )

                # Add realistic vulnerabilities
                self.add_realistic_vulnerabilities(host)
                network.add_host(host)

        return network

    def add_realistic_vulnerabilities(self, host: Host):
        """Add CVE vulnerabilities based on host type and OS"""
        vulnerability_db = {
            "WindowsWorkstation": ["CVE-2021-34527", "CVE
               -2021-1675"],
            "LinuxServer": ["CVE-2021-4034", "CVE-2022-0847"],
            "WebServer": ["CVE-2021-44228", "CVE-2021-45046"]
        }

        base_vulns = vulnerability_db.get(host.host_type.name,
           [])
        for vuln in base_vulns:
            if random.random() < 0.3:  # 30% chance per
               vulnerability
                host.add_vulnerability(vuln)
```

9 Experimental Validation

9.1 Comprehensive Performance Metrics

The framework supports extensive evaluation metrics:

$$\text{Deception Rate} = \frac{\text{Attacks on Decoys}}{\text{Total Attacks}} \tag{28}$$

$$\text{Protection Rate} = \frac{\text{Real Hosts Protected}}{\text{Total Real Hosts}} \tag{29}$$

$$\text{Resource Efficiency} = \frac{\text{Successful Deceptions}}{|\mathcal{D}_t|} \tag{30}$$

$$\text{Detection Latency} = \bar{t}_{\text{detection}} - \bar{t}_{\text{attack}} \tag{31}$$

$$\text{False Positive Rate} = \frac{\text{False Alerts}}{\text{Total Alerts}} \tag{32}$$

9.2 Decoy Placement Strategy

Experiments show that agents learn to place decoys strategically:

$$\pi^*_{\text{deploy}}(s_j) \propto P(\text{red agent visits } s_j) \cdot \text{value}(s_j) \tag{33}$$

Where value(s_j) represents the strategic importance of subnet s_j.

9.3 Experimental Results Implementation

Code for experimental analysis and results:

Listing 17: Experimental Analysis Implementation

```
# cyberwheel/utils/experiment_analyzer.py
class ExperimentAnalyzer:
    def __init__(self):
        self.metrics = {
            'deception_rate': [],
            'protection_rate': [],
            'resource_efficiency': [],
            'episode_rewards': [],
            'attack_success_rate': []
        }

    def analyze_episode(self, episode_data: dict):
        """Analyze single episode performance"""
        total_attacks = episode_data['red_actions']
        decoy_attacks = sum(1 for action in total_attacks
                           if action['target_decoy'])

        deception_rate = decoy_attacks / len(total_attacks) if
           total_attacks else 0
        self.metrics['deception_rate'].append(deception_rate)

        # Calculate protection rate
        real_hosts_attacked = sum(1 for action in total_attacks
                                 if not action['target_decoy']
                                    and action['success'])
        total_real_hosts = episode_data['network_size'] -
           episode_data['num_decoys']
        protection_rate = 1 - (real_hosts_attacked /
           total_real_hosts)
        self.metrics['protection_rate'].append(protection_rate)

        # Resource efficiency
        successful_deceptions = sum(1 for action in total_attacks
                                   if action['target_decoy'] and
                                      action['success'])
        num_decoys = episode_data['num_decoys']
        efficiency = successful_deceptions / num_decoys if
           num_decoys > 0 else 0
        self.metrics['resource_efficiency'].append(efficiency)

    def generate_report(self) -> dict:
        """Generate comprehensive experimental report"""
        return {
            'mean_deception_rate': np.mean(self.metrics['
               deception_rate']),
            'std_deception_rate': np.std(self.metrics['
               deception_rate']),
            'mean_protection_rate': np.mean(self.metrics['
                protection_rate']),
            'convergence_episode': self.find_convergence_point(),
            'learning_curve': self.metrics['episode_rewards']
        }
```

10 Conclusion

The Cyberwheel framework provides a mathematically rigorous foundation for training autonomous cyber defense agents. The observation space design enables agents to learn from both immediate and historical alert patterns, while the reward structure incentivizes effective deception strategies. The use of realistic network topologies and MITRE ATT&CK-based attack patterns ensures that trained agents can transfer effectively to real-world scenarios.

The mathematical formulations presented here demonstrate how complex cybersecurity concepts can be encoded into machine learning frameworks, enabling the development of AI systems capable of defending against sophisticated cyber threats.

10.1 Summary of Key Components

The framework's comprehensive implementation includes:

1. **Network Simulation**: NetworkX-based graph representation with realistic host modeling
2. **Agent Framework**: Separate red (offensive) and blue (defensive) agents with distinct capabilities
3. **MITRE ATT&CK Integration**: 295+ documented attack techniques with CVE/CWE mappings
4. **Observation Spaces**: Dual-structure observations combining current alerts and historical memory
5. **Reward Systems**: Sophisticated reward functions emphasizing deception effectiveness
6. **Detection Mechanisms**: Realistic alert generation with configurable detection probabilities
7. **Scalability Features**: Support for networks with millions of hosts
8. **Visualization Tools**: Real-time network state visualization and episode replay
9. **Configuration System**: YAML-driven modularity for easy experimentation
10. **Emulation Bridge**: Integration with Firewheel for real-world testing

10.2 Research Impact and Applications

The Cyberwheel framework enables several critical research directions:

- **Cyber Deception**: Strategic honeypot placement and management
- **Adaptive Defense**: Learning optimal responses to evolving attack patterns
- **Multi-Agent Coordination**: Collaborative defense strategies
- **Transfer Learning**: Simulation-to-reality knowledge transfer
- **Adversarial Learning**: Co-evolution of attack and defense strategies

11 Future Directions

11.1 Multi-Agent Extensions

Future work could extend to multi-agent scenarios:

$$\mathcal{A}_{\text{blue}} = \bigcup_{i=1}^{n} \mathcal{A}_{\text{blue}}^{(i)} \tag{34}$$

Where multiple blue agents coordinate defense strategies.

11.2 Continuous Action Spaces

Extension to continuous action spaces for fine-grained resource allocation:

$$\mathcal{A}_{\text{continuous}} = [0, 1]^{|\text{resources}|} \tag{35}$$

11.3 Adversarial Training

Co-evolution of red and blue agents through adversarial training:

$$\min_{\pi_{\text{blue}}} \max_{\pi_{\text{red}}} \mathbb{E}[R(\pi_{\text{blue}}, \pi_{\text{red}})] \tag{36}$$

References

[1] Oesch, S., Chaulagain, A., Austria, P., Weber, B., Sadovnik, A., Watson, C., Dixson, M., Roberson, B. Towards a High Fidelity Training Environment for Autonomous Cyber Defense Agents. *Workshop on Cyber Security Experimentation and Test (CSET 2024)*, 2024.

[2] Schulman, J., Wolski, F., Dhariwal, P., Radford, A., Klimov, O. Proximal Policy Optimization Algorithms. *arXiv preprint arXiv:1707.06347*, 2017.

[3] MITRE Corporation. MITRE ATT&CK Framework. `https://attack.mitre.org/`, 2024.

[4] Red Canary. Atomic Red Team. `https://atomicredteam.io/`, 2024.