Spring 2022

Autonomous Network Defence Using Multi-Agent Reinforcement Learning and Self-Play

Roberto G. Campbell
San Jose State University

AUTONOMOUS NETWORK DEFENCE USING MULTI-AGENT REINFORCEMENT LEARNING AND SELF-PLAY

A Thesis

Presented to

The Faculty of the Department of Computer Engineering

San José State University

In Partial Fulfillment

of the Requirements for the Degree

Master of Science

by

Roberto G. Campbell

May 2022

The Designated Thesis Committee Approves the Thesis Titled

AUTONOMOUS NETWORK DEFENCE USING MULTI-AGENT REINFORCEMENT LEARNING AND SELF-PLAY

by

Roberto G. Campbell

APPROVED FOR THE DEPARTMENT OF COMPUTER ENGINEERING

SAN JOSÉ STATE UNIVERSITY

May 2022

Magdalini Eirinaki, Ph.D.	Department of Computer Engineering
Younghee Park, Ph.D.	Department of Computer Engineering
Brian Coltin, Ph.D.	NASA Ames Research Center

ABSTRACT

AUTONOMOUS NETWORK DEFENCE USING MULTI-AGENT REINFORCEMENT LEARNING AND SELF-PLAY

by Roberto G. Campbell

Early threat detection is an increasing part of the cybersecurity landscape, given the growing scale and scope of cyberattacks in the recent years. Increasing exploitation of software vulnerabilities, especially in the manufacturing sector, demonstrates the ongoing need for autonomous network defence. In this work, we model the problem as a zero-sum Markov game between an attacker and defender reinforcement learning agents. Previous methods test their approach on a single topology or limit the agents to a subset of the network. However, real world networks are rarely fixed and often add or remove hosts based on demand, link failures, outages, or other factors. We do not confine our research to a fixed network in terms of size and topology, but instead are interested in larger networks and varied topologies to determine the scalability and robustness of the approach. We consider additional topologies and a robust training curriculum that incorporates network topologies to build more general, capable agents. We also use PPO which offers a good balance of computational complexity and convergence speed.

ACKNOWLEDGMENTS

I would like to thank my advisor Dr. Magdalini Eirinaki, for the guidance, feedback, and support through this journey. I would also like to thank my co-advisor Dr. Younghee Park, who has been reliable source of critical insights and feedback. And a big thanks to Dr. Brian Coltin and Dr. Michael Furlong who if it weren't for their unwavering belief in me I might not be here typing this.

I would like to thank my family, who without their support at every step I wouldn't have made it this far. Thanks to David, Diane, Maggie, Goretti y'all were the anchor that kept me grounded throughout this process. Y quiero agradecer a mis padres, que han hecho sacrificios inumerables para hacer estos momentos posible.

And finally I'd like to thank my girlfriend, who's remembered to feed the cats while I've been lost in numbers and figures. Thanks for holding the fort down, I couldn't have done this without you.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

LIST OF ABBREVIATIONS

A2C	Advantage Actor-Critic
A3C	Asynchronous Advantage Actor-Critic
API	Application Programming Interface
DPG	Deterministic Policy Gradient
DDPG	Deep Deterministic Policy Gradient
DQN	Deep Q Network
DRL	Deep Reinforcement Learning
IDS	Intrusion Detection System
IPS	Intrusion Prevention System
ISC	Impact Subscore
MARL	Multi-agent Reinforcement Learning
MDP	Markov Decision Process
MBF	Multiple Bloom Filter
MRP	Markov Reward Process
RL	Reinforcement Learning
OT	Operational Technology
PPO	Proximal Policy Optimization
POMDP	Partially Observable Markov Decision Process
SAC	Soft Actor Critic
SDN	Software Defined Network
TCAM	Ternary Content-Addressable Memory

1 INTRODUCTION

1.1 General Trends

Early threat detection is an increasing part of the cybersecurity landscape, given the growing scale and scope of cyberattacks in the recent years. This increased scope means that integrating threat intelligence data from multiple feeds such as honeypots, open source intelligence (OSINT), network monitoring, and DNS analytics is crucial to understanding the threat landscape. It also means that threat analysts must sift through data from a variety of sources in order to detect threat actors and malicious campaigns. By automating some of these processes we free up analyst time to address high priority threats.

According to a recent survey of the threat landscape, vulnerabilities in the cloud have increased by 150% in the last five years [1]. Manufacturing leads as the most attacked industry, with 47% of the attacks due to the exploitation of a vulnerability. Of the attacks on manufacturing, 23% of those were ransomware. Operational technology (OT) used throughout the industry is a frequent target in these attacks and the increasing adoption of Internet of Things (IoT) devices has increased the attack surface available to threat actors. In May 2021, the Colonial Pipeline Company was extorted by a ransomware group [2]. An analysis after the attack discovered that the attackers had used a single password to gain access to a legacy VPN system. The attack demonstrated that the distributed nature of OT and real time performance requirements make it an especially attractive target for attackers. It can also be a matter of national security, as was the case in the Colonial Pipeline attack.

Reinforcement learning (RL) is a rapidly growing sub-field of machine learning that has found applications in gaming, robotics, autonomous vehicles, and finance. Unlike supervised learning which learns from labeled data, RL allows an agent to learn from interacting with an environment and receiving feedback in the form of rewards [3]. The

agent learns a policy which informs how it interacts with the environment, improving its average reward over time. Agents can even cooperate with or compete in an environment similar to the framework of generative adversarial networks in deep learning. Cybersecurity, and in particular network defense, is a good candidate for applying reinforcement learning agents [4], [5].

1.2 Contribution

In this work we propose a RL approach using proximal policy optimization (PPO) to train an agent to choose countermeasures for defending a network. The goal of the defender is to mitigate the intrusion while minimizing the impact on the network by the attacker. We model both defender and attacker as RL agents. The agents compete in a zero-sum game using self-play to improve convergence [6]. In addition, we evaluate the agent in varying environments and with varying simulated threat scenarios in order to determine the effect of observed vulnerabilities in the environment on defender performance. Instead of training on a fixed topology/network size as previous work does [4], [5], [7], [8] we instead evaluate our approach on larger networks and varied topologies to determine the scalability and robustness of the approach. We unify the topologies using a training curriculum to build more general, capable agents. And finally, we choose PPO for training the agents which offers a good balance of computational complexity and convergence speed.

Previous methods test their approach on a single topology or limit the agents to a subset of the network . However, real world networks are rarely fixed and often add or remove hosts based on demand, link failures, outages, or other factors.

In particular, our contributions can be summarized as follows:

1) Propose a modification of the base reward of Gabirondo-Lopez et al. [7] that improves convergence.

2) Design and development of a training curriculum using threat scores and network topology.
3) A training methodology that adds snapshots of agents to opponent pool for self-play using an adaptive win rate threshold.
4) An extensive training and evaluation of the the agents in multiple network topologies.
5) An extensive training and testing of the agents in a variety of threat scenarios

The remainder of this work is organized as follows. A review of the state-of-the-art for reinforcement learning, multi-agent reinforcement learning, and reinforcement learning for software defined networks is given in Section 2. A breakdown of the environment, state, and actions of the Markov game is given in Section 3. This section also includes a discussion of rewards and observations per agent. In Section 4, we give an overview of the training process which includes a discussion of self-play and the curriculum for training. In Section 5, we present the results of our experimental evaluation. In Section 6, we discuss the results and give concluding remarks. In Section 7, we remark on future research directions of the work.

2 LITERATURE REVIEW

2.1 Reinforcement Learning

In reinforcement learning, the interaction of agent and environment is modeled as a Markov Decision Process (MDP). A MDP is a collection of (s_t, a_t) tuples which describe the transition probabilities between states s in the environment given action a at timestep t. The MDP can be augmented with rewards to describe the reward R for state action pair (s_t, a_t). This is known as a Markov Reward Process (MRP) [9].

A partially observable Markov decision process (POMPD) is a Markov decision process with states $s \in S$ that the agent cannot observe. Similarly to MRP, rewards can be added to describe a partially observable Markov reward process (POMRP) [10]. Additionally the POMDP is described by an observation function $s_t \rightarrow O_t$ that maps states s to observations O at time t [6].

Markov decision processes are further extended by increasing the possible agents from 1 to $n >= 2$. These MDPs are known as Markov games [6]. Instead of simply maximizing the sum of expected rewards, we are now interested in the set of policies that characterize a Nash equilibrium between the competing agents. The learned Nash equilibrium can be sub-optimal [11].

The expected sum of rewards in a Markov reward process can described using the Bellman equation:

$$Q(s,a) = r_{s,a} + \gamma \max Q(s', a') \tag{1}$$

where $Q(s,a)$ is the discounted sum of rewards when taking action a at time t in state s. This quantity is also known as the Q value, or the value of action. The reward is discounted by discount factor γ to account for future rewards [10]. In reinforcement learning, we are interested in the distribution of actions, or policy π, that maximizes the expected sum of rewards. The Bellman operator provides a guarantee for finding the

optimal policy through maximizing the value of action at each step. Since Markov processes are memoryless, we only need to consider the current state; the history is unnecessary [6], [11].

In recent years, RL has made advances in increasingly complex environments and across domains. Value-based methods use the expected discounted sum of rewards to choose actions as an approximation of the agent's policy π [3]. In 2016, Mnih et al. stabilized Q learning with deep reinforcement learning by using two separate Q networks, the target network and the online network. The authors added a replay buffer to sample past experience which lead to more sample efficiency, as the network can train on previously seen samples. The online network predicts the Q value and is refreshed every t time steps using the target network [12]. This also breaks the dependency between samples which improves training with deep reinforcement learning (DRL). This approach is referred to as Deep Q Networks, or DQN. Hessel et al. extended the approach by combining improvements for DQN such as n-step DQN, double DQN, and distributional DQN [13]. The method achieved state-of-the-art results on the Atari benchmark.

Policy gradient methods directly estimate policy instead of inferring actions indirectly through Q values such as in value-based methods. The policy π is estimated by the policy network as a distribution over the actions. In the policy gradient method REINFORCE, the Q values and log of the policy define the loss of the policy [14]. However, REINFORCE and similar policy gradient methods are sensitive to high variance [15].

Lillicrap et al. augment the deterministic policy gradient method (DPG) with DRL to learn a distribution over a continuous action space [16]. This method, also known as deep deterministic policy gradients, or DDPG, improves on the performance of benchmark continuous tasks such as Cheetah and Cartpole [17]. Since actions are continuous, they propose sampling from a random noise process instead of the common ε greedy approach for exploration.

Actor-critic methods such as DDPG and advantage actor critic (A2C) use two networks, an actor and critic, to better estimate the policy compared to methods such as REINFORCE. The baseline, or the critic, predicts values that are used to improve the actor's estimate of the policy. This reduces variance but increases bias in the agent's actions [15].

Mnih et al. extend the actor-critic framework by training an actor-critc model asynchronously with parallel environments collecting experience simultaneously on multiple CPU threads. Instead of a replay buffer, the authors propose a different exploration strategy for each agent. This removes the need for a replay buffer and allows the training to converge faster. Additionally, the authors improve the critic's baseline estimate using the advantage $A(s,a) = Q(s,a) - V(s)$ where $A(s,a)$ is the advantage, $Q(s,a)$ is the Q value (value of action), and $V(s)$ is the value of state. The advantage represents the relative quality of the action when taking action a. The authors refer to this approach as Asynchronous Advantage Actor-Critc (A3C) [18].

A3C and REINFORCE are susceptible to policy gradient updates that undo previous learning. This can cycle into further incorrect policy updates as the policy takes actions and makes further gradient updates with the bad network parameters. Proximal policy optimization (PPO) attempts to solve this issue by constraining the gradient updates to a region in the parameter space which should lead to favorable updates in the network's parameters. PPO clips the policy as shown in Equation 2.

$$L = \sum \min(r_t(\theta)A_t, clip(r_t(\theta), 1-\varepsilon, 1+\varepsilon)A_t) \tag{2}$$

Similar to PPO, Haarnoja et al. [19] set out to design a sample efficient model-free RL approach without the need for extensive hyperparameter tuning. The authors propose SAC, or soft-actor critic, which unlike PPO is off-policy. The results reported

state-of-the-art performance on continuous robotics tasks such HalfCheetah-v2 [17], with PPO comparable in Humanoid-v2 but with slower convergence .

2.2 Multi-agent Reinforcement Learning

Multi-agent reinforcement learning (MARL) extends the ideas in single-agent DRL and applies them to Markov games which can be either cooperative or competitive. One of the earliest works in multi-agent reinforcement learning, or MARL, was by Tampuu et al. which trained two independent agents to play Pong. By adjusting the reward, they were able to elicit either cooperative or competitive behavior from the agents [20].

A well known issue of MARL is the non-stationarity of the environment, which violates the Markov property. This refers to the fact that from the each agent's perspective the actions of additional agents form part of the environment. As a result, the moving policy distribution of the agents inhibit convergence [21]. Self-play using opponent pools addresses this issue for competitive environments by freezing the agent's parameters and adding them to a pool of past opponents. Each episode, the opponent pool is sampled from uniformly to ensure the agent does not forget how to play against previously seen strategies [22].

Silver et al. [23] used self-play to train an agent achieve grandmaster level play in the game of Go. The authors used Monte Carlo Tree Search (MCTS) to simulate games. The MCTS tallys up the visit count for each action and updates the Q value for each. We can then consider the policy π a vector of search probabilities proportional to the visit count for each action.

In 2019, OpenAI [24] used PPO to outperform the top players in the world in Dota 2. The group used self-play with a shared base feature extraction Long Short Term Memory (LSTM) for both the policy and value networks. This is inspired by work done by Hausknecht and Stone [25], where they used an LSTM to learn temporal dynamics of Atari games like Pong. This eliminated the need for framestacking for Atari games.

Most recently, Schrittwieser et al. [26] expanded on the search based framework built by AlphaZero in a work titled MuZero. MuZero is a model-based approach that outputs three values at each step: actions, values, and rewards. The model uses a recurrent model internally that takes the observation and possible future action as input. The authors trained the model using self-play in a variety of environments such as chess, Go, and Atari.

2.3 Software Defined Networks

Software Defined Networks (SDN) can be broken into three major architectural components: the controller, the northbound interface, and the southbound interface. The purpose of this design is to separate the data and control plane. The data plane is implemented at the device level, which communicates to the controller through the southbound interface. This allows the controller to add or modify rules that determine how packets are routed dynamically. Each device maintains a table of rules called a flow table which is managed by the controller.

This allows for more dynamic routing, as the devices forward packets to the controller if there a packet does not match any rules in the current flow table. The device forwards the packet to the controller and the controller can update the flow table of the device with the appropriate rule. The northbound interface augments that flexibility by providing an API for use by services and apps to communicate with the controller. By monitoring events in the network from the southbound interface, apps built on the controller can dynamically manage the network and respond to events in the network.

2.4 Reinforcement Learning for Software Defined Networks

RL in cybersecurity can be categorized into three major areas: intrusion detection, game theory, and cyber-physical systems. In this work, we are interested in the game theoretic application to cybersecurity defense. In particular, we plan to abstract an

attacker-defender interaction as a zero-sum game and search for the optimal defense strategy.

2.4.1 Network Management

Since SDNs allow for dynamic updating of network routing rules, reinforcement learning allows for autonomous network management. Network performance measures such as bandwidth, control overhead, latency, and table-hit rate are especially useful when managing the network.

Switches and other physical devices have a limit to the number of rules they can store which is due to the capacity of the Ternary Content-Addressable Memory (TCAM). In an OpenFlow switch, flow entries are stored in the flow table on the TCAM. However, TCAM is limited and can be exceeded by adding an excess amount of entries to the flow table. This can cause flow entries to be forwarded to the controller for processing. An increase of forwarded flow entries increases overhead on the controller due to increased table-miss scenarios for packets that match the entries on the controller. To address the overhead due to network reconfiguration with this constraint, Mu et al. [27] propose a reinforcement learning framework that moves which rules should be stored on device and which should be processed by the controller to minimize table-miss scenarios .

This framework uses the parameters $flowfrequency$ and $flowrecent$ to select entries from a pool and install the selected flow entries onto the OpenFlow device. These parameters are sampled from OpenFlow and represent the flow match frequency and duration of flow, respectively. Based on the agent's prediction of flow frequency and flow recentness, the controller splits the entry pool into two distinct sets: device rules and controller rules. The device rules are installed on the corresponding devices and the controller rules are maintained on the controller. For the results, the authors are interested in reducing the control overhead which is the amount of table-misses in an episode. The performance of the traditional reinforcement learning method Q learning is compared to

DQN. A 40% reduction in the control overhead is set as the target threshold over a training session comprising 101 episodes. DQN reduced the overhead in the 5th episode whereas Q learning reached the goal around the $\sim$60th episode.

The authors also compared the performance of the RL methods discussed with a Multiple Bloom Filter (MBF) [27]. The comparison looks at the table-hit percentage of the three methods. Q learning showed the lowest table-hit rate followed by the MBF. The DQN approach had the highest table-hit rate at 65%, a roughly 10% increase over the MBF .

2.4.2 Threat Detection

In addition to flow management, SDN controllers can leverage reinforcement learning for detecting threats in a cybersecurity context. With the proliferation of Internet of Things (IoT) devices, it has become more important to monitor outgoing traffic. Attackers can compromise the devices and hide illicit activity as spikes in traffic which can lead to packet loss. Dake et al. [28] propose a RL system to detect DDOS and other suspicious activity in an IOT network. The authors focus on three types of attacks: traffic burst, elephant flow, and DDOS in the network. An RL agent is applied to address traffic burst and elephant flow while a second monitors potential DDOS. The two agents work to normalize traffic by working through the controller to install flow rules on the device.

The controller monitors the network and sends information to the agents which the use to modify the environment. Both agents use information such as the link occupancy rate between the flow and the switches, channel link occupancy between packet-in messages, and link occupancy between packet-out messages. Since the first agent is responsible for monitoring traffic burst and elephant flows, it takes two actions in response to the observed network environment. The first assigns the next hop to available switches during packet-out messages. The second increases bandwidth of the affected flow within the demand range of the network. The second agent, however, is responsible

for responding to DDOS in the network and as a result will add a packet drop rule if the flow frequency is greater than a given threshold. To compare the multi-agent approach against the single agent, the author's setup a network using mininet and the SDN controller Ryu with 5 and 8 Openflow switches. The network has 20 and 60 nodes, respectively with traffic sent using TCP and link bandwidth set to 50-100 Mbps.

The authors measured bandwidth and intrusion detection rate in the network for both single agent and multi-agent. For the 60 node scenario, multi-agent RL used 7% less bandwidth by the end of the experiment. Overall, multi-agent consistently used less bandwidth than the single agent approach over the time span measured. The intrusion detection rate is measured as the number of DDOS packets. As the packet transmission rate increases, the multi agent approach scales almost linearly with the packet transmission rate. For the 20 node scenario, multi-agent RL has a 44% improvement in packet detection compared to the baseline. The improvement is similar for the 60 node scenario. In this case, the proposed approach improved the detection by 50%.

2.4.3 Intrusion Prevention

Intrusion prevention systems build on IDS by taking action within the SDN to prevent an attacker from compromising resources. An intrusion prevention system running on an SDN controller can delay or prevent an attacker given feedback from an intrusion detection system.

Han et al. [8] consider a defense agent as well as possible adversarial attacks against RL agents running on an SDN controller. The authors test the effectiveness of the methods of attack and briefly consider potential countermeasures.

The defender can take one of four actions to defend the network: isolate a node and patch, reconnect a node, migrate the critical server, or take no action for that timestep. In response the attacker is modeled as a deterministic policy. The policy successively attempts to compromise nodes in the backbone of the network. The goal of the attacker is

ultimately to compromise the critical server(s), whereas the goal of the defender is to preserve as many critical servers and keep as much of the network available and connected to the critical servers as possible.

The authors propose two types of adversarial attacks for poisoning the training process: flipping reward signs and changing the observed state of the defense agent. The first type of attack samples the training batches of the network and switches the sign of the experience with the greatest value of the gradient of the loss with respect to the reward. Based on experiment results this type of attack succeeds in slowing the training process but eventually the agent's policy does converge to the optimal solution. The second attack attacks the observed state of the agent by changing the state of two nodes in the network for each sampled experience tuple of state, action, reward, and next state. When accounting for a limit on false positive and false negative nodes, the authors found that less than 50% of the nodes were preserved after applying this black box attack.

Gabirondo-Lopez et al. [7] also consider training an intrusion prevention system using reinforcement learning. However, they model the training as a competition between two agents: an attacker and a defender agent. The authors model the environment as a fully connected network of eight nodes. The defender sends messages to the controller in order to interact with the SDN in response to the actions of the attacker. In addition to the main network, there is also a honeynet. This separate network allows the agent to migrate compromised nodes, potentially delaying the attacker.

The training can therefore be thought of as a zero-sum Markov game between the two agents. In the scenario, the attacker has an incomplete view of the network that progressively improves as more nodes are compromised. The attacker can survey a node's links, check for vulnerabilities, and attack a vulnerability once a vulnerability has been detected. The defender can protect against these actions by checking a node's status, isolating a compromised node, migrating the critical node, or migrating a compromised

node to the honeynet. The authors fix the network size at 8 nodes and use MuZero for both agents and train through self-play.

In this thesis we follow a similar approach in modeling the problem as a zero-sum Markov game between the attacker and the defender. Previous methods test their approach on a single topology or limit the agents to a subset of the network [4], [5], [7], [8]. However, real world networks are rarely fixed and often add or remove hosts based on demand, link failures, outages, or other factors. We do not confine our research to a fixed network in terms of size and topology, but instead are interested in larger networks and varied topologies to determine the scalability and robustness of the approach. We consider additional topologies and a training curriculum that incorporates network topologies to build more general, capable agents. We also use PPO which offers a good balance of computational complexity and convergence speed.

3 THE RL TRIAD: ENVIRONMENT, ACTIONS, REWARDS

The agents compete in a simulated network environment inspired by the work done by Gabirondo-Lopez et al. [7]. The goal of the attacker is to compromise the critical node while also compromising as many hosts in the network as possible. The goal of the defender is to protect the critical node. At the same time, the defender should minimize impact on the network by the attacker. The game ends when either agent wins or the episode horizon is reached, in which case the episode is declared a tie. Figure 1 demonstrates the win conditions for each agent. Each agent takes turns performing an action in the environment. The attacker acts first, followed by the defender. Following each step, the win conditions for the environment are checked and the reward is allocated accordingly.

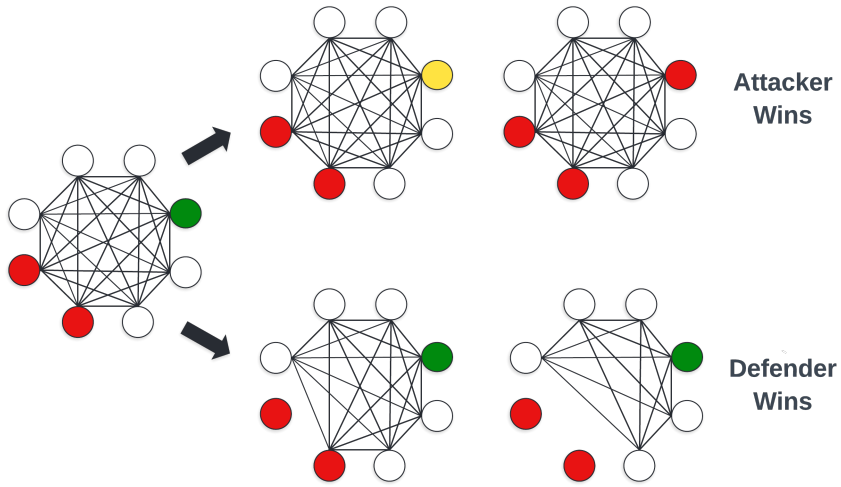

Fig. 1. Win conditions example.

3.1 Environment

The environment is represented as a $k \times k$ graph G, where k is the number of nodes in the network. Similar to an adjacency matrix, G encodes the neighbors of the nodes on the off-diagonal. Additionally, the value on the diagonal $h_i \in G$ represents the node state, where $i \in [0, k-1]$.

Each agent has a corresponding observation of the environment's state as the environment is partially observable from the perspective of both the attacker and the defender. Correspondingly, the environment also maintains a global state which is the authoritative view of the environment. However, the defender has more information of the environment's true state since we assume the defender is interacting with the network through the SDN controller.

We consider the example shown in Figure 2, where the left and right graphs depict the attacker's and defender's observation of the same network respectively. The attacker has compromised two nodes, as shown at the lower left in red. The attacker has also discovered a vulnerability on a third host, the node in yellow at the top left. The colored node in green represents the critical node. The defender's observation on the right shows that one of the compromised nodes has been isolated from the network. The attacker has yet to discover this most recent action by the defender and may try to take an action that will not succeed, based on the true state of the game.

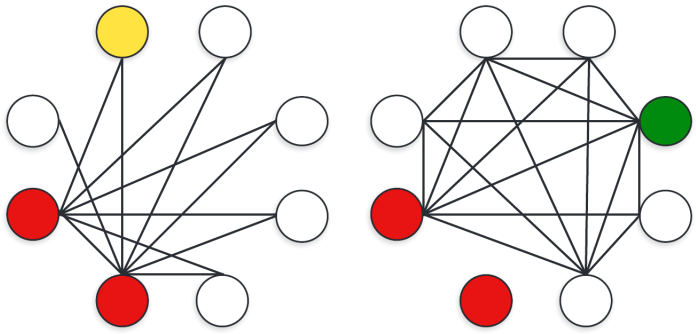

Fig. 2. From left to right: Attacker and defender observation of game state.

3.1.1 Vulnerability Scores

Each node has a vulnerability that can be exploited by the attacker. We represent the vulnerabilities using the NIST Common Vulnerability Scoring System (CVSS) [29], based on work by Gabirondo-Lopez et al. [7]. The CVSS is a framework used by the

National Vulnerability Database (NVD) to categorize and rank vulnerabilities. The scoring system rates vulnerabilities as none, low, medium, and high risk using a vulnerability score with range 1-10 as shown in Table 1. This score, also known as the base score, is what we use in lieu of early warning alerts from an intrusion detection system (IDS). The base score and its subcomponents are used to determine the success of certain actions performed by both the attacker and the defender.

Table 1
CVSS Vulnerability Severity Ratings

Rating	**Base Score Range**
None	0
Low	0.1 - 3.9
Medium	4.0 - 6.9
High	7.0 - 8.9
Critical	9.0 - 10.0

The scoring system includes an additional component called the vulnerability Scope [29]. Scope can belong in one of two states $(Changed, Unchanged)$ where a vulnerability with *Changed* scope affects the given host. Otherwise, if Scope is $Unchanged$ then another host has the affected vulnerability. For this work, we consider the case where all vulnerabilities have Scope *Changed*.

The base score is composed of two subscores [30]: the impact score and the exploitability score [29]. The impact score is defined as:

$$ImpactScore = 1 - (1-C)(1-I)(1-A) \tag{3}$$

where C is the confidentiality, I is the impact, and A is the availability. The components measure the impact to confidentiality, impact, and availability as a result of the exploited vulnerability. The exploitability is defined as:

$$ExploitabilityScore = 8.22 \times AV \times AC \times PR \times UI \tag{4}$$

where AV represents the attack vector, AC the attack complexity, PR the required privilege, and UI the user interaction to carry out the exploit.

With the impact and exploitability, the base score is then defined as:

$$BS = \begin{cases} 0 & ImpactScore = 0, \\ round(\min((ImpactScore + ExploitabilityScore), 10)) & \text{otherwise} \end{cases} \tag{5}$$

As part of the global state there exists a mapping $v_i \rightarrow VulnScores$, where each entry is a tuple of $(ImpactScore, ExploitabilityScore, BaseScore)$. These scores are then used to determine the success of certain actions for both agents. Instead of fixing the scores throughout the environment, the $Impact$ and $Exploitability$ score are sampled using a Gaussian distribution. This leads to forming part of a training curriculum, as we can increase the standard deviation of the parameters to train the agents with progressively more challenging environments [31] ranging from low severity (low exploitability-low impact) up to critical severity (high exploitability-high impact). Using Table 1 as a guide, the parameters of the distribution are modified to test various threat scenarios. Finally, the *BaseScore* is calculated using Equation 5.

For this work, we assume the scores are provided by an intrusion detection system. In a real world application, we can use the vulnerability scores as well as other systems monitoring the network such as anomaly detection on network activity [32].

3.1.2 Topology

One of our objectives is to test our approach across multiple topologies. We model the environment with four topologies as shown in Figure 3: clique, ring, linear, and 2D grid. Each topology can also be scaled by increasing the number of nodes. The allows us to compare across topologies and network sizes to determine the topologies that are most biased in favor of either agent.

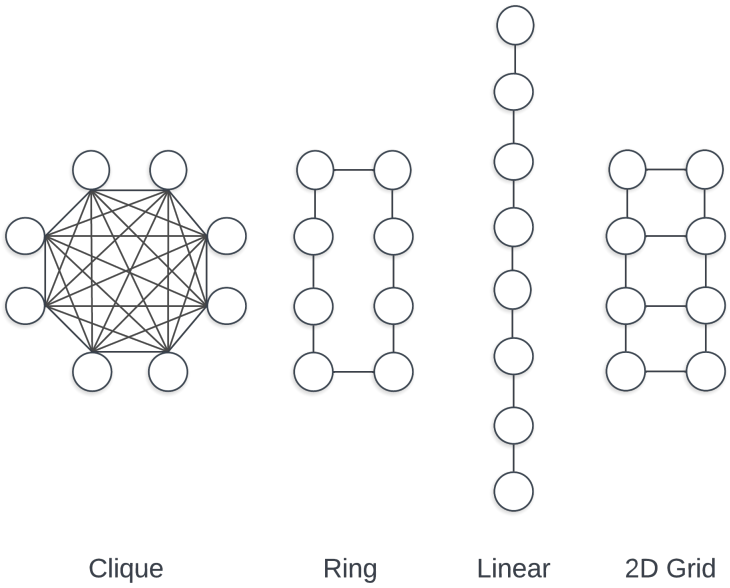

Fig. 3. Network topologies.

The defender's observation is built using the specified topology as a template. The defender's observation is then used to initialize the global observation, with the addition of the compromised node for the attacker. As such, the selected topology does not impact the vulnerability scores in the environment.

3.2 Observations

As previously mentioned, both agents observe a partial observation of the environment state as shown in Figure 4. We maintain a global state which is kept in sync as agents perform actions in the environment. At the start of each new episode, the starting positions for both agents are randomized. For the attacker, one of the nodes is set as compromised and the corresponding observation is updated. For the defender, the starting node is marked as the critical (flag) node. The random start positions prevent the agents from overfitting on the specific hosts (nodes). Figure 5 shows a key for the node states.

At each step, we update the observation for the respective agent given action a_t. First we decompose the action into the target node and target action as shown in Algorithm 1.

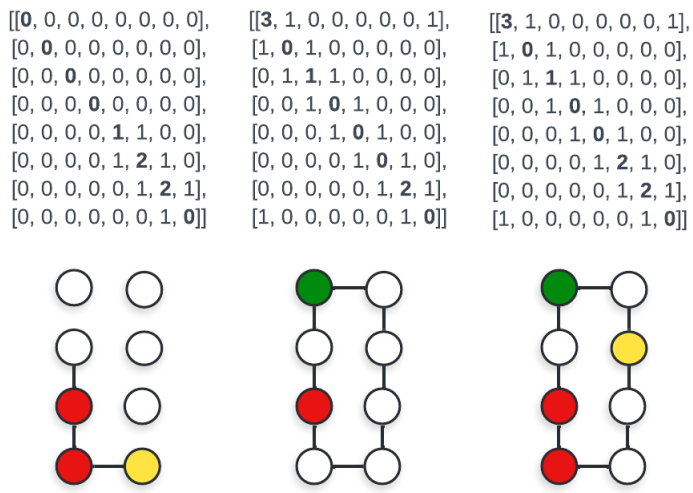

Fig. 4. Ring environment with corresponding observations and state. From left to right: Attacker observation, defender observation, global state

Fig. 5. Color key and state value for node states. From left to right: Normal (0), Vulnerability Scanned (1), Vulnerability Exploited (2), Critical Node (3)

For both agents, we validate the action given the observation and the action subcomponents $a_{t,n}, a_{t,a}$. If the action is valid, the we perform the action (step). Additionally for the attacker, we find a subset of nodes $CN \in V$, where V is the set of nodes in the network. CN is the set of nodes that are one hop away from a compromised node whose vulnerabilities are not exploited (state 0 or 1). If either CN is not empty or the target node is compromised then we take the action on the selected target node.

Algorithm 1 Update Agent Observation

procedure UPDATEOBSERVATION(O_a, A, a_t)

    $a_{t,n}, a_{t,a} = ProcessAction(a_t)$ ▷ Decompose action tuple into target node and target action
    if A = attacker **then**
        $CN \leftarrow FilterCandidateNeighbors(O_a, O_g)$
        if $ValidateAgentStep(a_{t,n}, a_{t,a}, O_a)$ **then**
            $O_a = AttackerStep(O_a, a_t)$
    if A = defender **then**
        if $ValidateAgentStep(a_{t,n}, a_{t,a}, O_a)$ **then**
            $O_a = DefenderStep(O_a, a_t)$

3.3 Actions

3.3.1 Action Space

Similar to [7] we simplify the action space to three actions per agent. This allows us to use the same neural network architecture for both agents while training both policies separately.

The attacker's objective is to compromise the critical node. As such, the agent's actions are directed at exploring the network and exploiting known vulnerabilities. The attacker's step in the environment is shown in Equation 2. An overview of the actions is given in Table Table 2. Figure 6 shows an example of the actions in the environment in the clique topology as well as the corresponding attacker observations.

Table 2
Agent Action Overview

Action	Agent	Description	Cost
Explore Topology	Attacker	Update observation of compromised node	N/A
Scan Vulnerability	Attacker	Scan host for vulnerabilities	N/A
Exploit Vulnerability	Attacker	Exploit vulnerability on host	N/A
Check Status	Defender	Update observation of target node	1
Isolate Node	Defender	Isolate compromised node	6
Migrate Node	Defender	Isolate compromised node	5

Explore Topology: Updates the attacker's observation using the global state. A deterministic action that updates neighbors of the targeted node to the attacker's

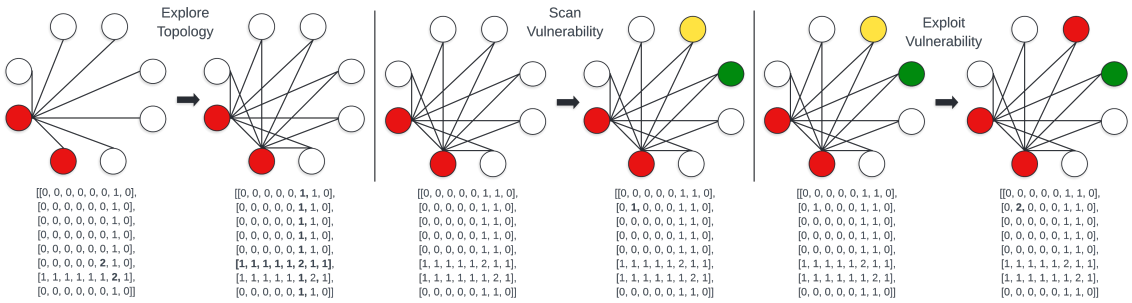

Fig. 6. Example of attacker actions in clique topology.

Algorithm 2 Attacker Agent Step

1: **procedure** ATTACKERSTEP($O_a, a_{t,n}, a_{t,a}$)
2: $exploitabilityRatio = O_g[exploitabilityScore][a_{t,n}]/10$
3: **if** $a_{t,a} = 0$ **then** ▷ Explore Topology
4: $setNeighbors(O_a, a_{t,n}, O_g)$
5: **if** $a_{t,a} >= 0$ **then** ▷ Scan or Exploit Vulnerability
6: $actionSuccess = random()$
7: **if** $actionSuccess < exploitabilityRatio$ **then**
8: $O_a[a_{t,n}][a_{t,n}] \leftarrow a_{t,a}$
9: $O_g[a_{t,n}][a_{t,n}] \leftarrow a_{t,a}$

observation. Since the attacker can only update the observation of exploited nodes, this action will fail if attempted on other node states. If the node has been isolated by the defender, this action has no effect.

Scan Vulnerability: Scans the target node for vulnerabilities. The success of this action depends on the *ExploitabilityScore* of the node in question as demonstrated in Algorithm 2 line 2. Target node must be normal (state 0).

Exploit Vulnerability: Exploits the scanned vulnerability. Target node must be already scanned (state 1) for action to succeed. If the critical node is exploited by the attacker, the attacker wins and the game ends.

The defender's objective is to protect the critical node while minimizing impact to the network and its services. The defender's action allow it to update its observation and

Algorithm 3 Defender Agent Step

procedure DEFENDERSTEP($O_a, a_{t,n}, a_{t,a}$)

$baseScore \leftarrow O_g[baseScore][a_{t,n}]$
$targetNodeState \leftarrow O_g[networkGraph][a_{t,n}]$
$actionSuccessThreshold \leftarrow \frac{baseScore}{10 \times \log_2(k)}$
$actionSuccess = random()$
if $a_{t,a} = 0$ **then**
    if $targetNodeState = 0$ **then** ▷ Check status
        if $actionSuccess < actionSuccessThreshold$ **then**
            $O_a[a_{t,n}][a_{t,n}] \leftarrow 2$
            $addCountermeasureCost(a_{t,a})$
if $a_{t,a} = 1$ **then** ▷ Isolate compromised node
    $setNeighbors(O_a, a_{t,n}, 0)$ ▷ Set edges for node to zero
    $addCountermeasureCost(a_{t,a})$
if $a_{t,a} = 2$ **then** ▷ Migrate critical node
    if $targetNodeState = 2$ **then**
        if $actionSuccess < actionSuccessThreshold$ **then**
            $O_a[a_{t,n}][a_{t,n}] \leftarrow 2$
            $addCountermeasureCost(a_{t,a})$
    if $targetNodeState \neq 2$ **then**
        $criticalNode \leftarrow indexOf(O_a, 3)$
        $O_a[criticalNode][criticalNode] \leftarrow 0$
        $O_a[a_{t,n}][a_{t,n}] \leftarrow 3$

isolate known compromised nodes. The defender may also migrate the critical node to another location in the network. Each action has a corresponding cost as shown in Table 2, which is factored into the defender's reward [7], [32]. An overview of the defender's actions is shown in Figure 7.

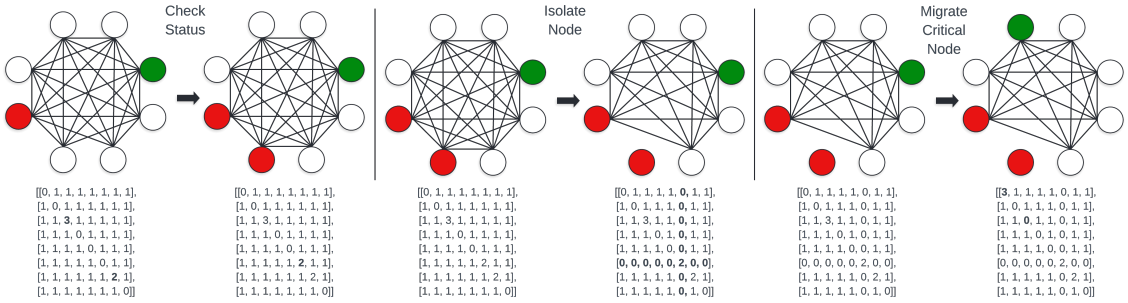

Fig. 7. Example of defender actions in clique topology.

Check Node Status: Updates the defender's observation as shown in Algorithm 3 line 6. If a node is compromised, the likelihood of discovering the exploited vulnerability is

$$P[O_d(h_i) = 2 | O_g(h_i) = 2] = \frac{BaseScore}{10log_2(k)} \tag{6}$$

where k is the number of nodes in the network, O_d is the defender's observation, O_g is the global state, and h_i is the index of the host (node).

This formulation is a slight modification from the probability proposed in Gabirondo-Lopez et al. [7]. We change the term in the denominator from $10k$ to $10log(k)$.This keeps the probability a function of the number of nodes while scaling better to higher values of k. For example, for 16 hosts, the defender has a 6.25% chance of identifying the vulnerability in the scenario with the highest *BaseScore* using the probability proposed in [7]. With our proposed modified status check, it becomes 25%. In a low risk scenario ($BaseScore = 1$), that drops to 0.625% and 2.5%, respectively.

Isolate Node: Once the defender has discovered an exploited vulnerability in the network, it can act to mitigate the attack. The defender can do so by isolating the node, cutting off the node from the rest of the network. This is a deterministic action that removes the neighbors from the targeted host and updates the global observation O_g.

Migrate Node: Finally, the defender can migrate the critical node to another node in the network. First, the defender checks if its observation matches the observation in the global state. This is the same check as in the *Check Node Status* action. If the target host is compromised according to the global state and the check succeeds, the defender updates its observation. If the target host is not compromised according to the global state, then the node is migrated. Note that since the *Check Node Status* action is stochastic, the check may fail to update the observation.

3.3.2 Action Masking

Early on in training, both agents may select invalid actions as part of the exploration process. For example, the attacker may attempt to exploit a previously exploited node. Similarly, the defender may attempt to isolate a node that has not been compromised according to its observation.

Due to the zero-sum nature of the game, rewards are sparse, i.e. almost all rewards for reward r_t in the set of tuples (s_t, a_t, r_t) over an episode are zero except for the reward when the game ends. In environments with dense rewards, the agent will learn to avoid invalid actions over time, as the value estimates converge. The sparse rewards contribute to instability in value estimates. This is known as the credit assignment problem [33]. To address this, we apply a mask to the agent's actions as shown in Figure 8. The action mask is a tuple of shape $(k, 3)$, where the values in the masks $m \in [0, 1]$. If the action mask value is zero then the probability of the invalid action is set to zero. Otherwise the value is one and the agent can perform the action. The action mask is differentiable which allows the agent to learn which actions are invalid [34].

The action mask does not completely prevent the agent from taking invalid actions. Since the agent chooses both the target node and the target action independently, we only consider invalid actions in each subcomponent of the action space. Therefore the agent may choose a node-action tuple that is invalid but each sub-action can be valid. For example, we can see in Figure 8 that the defender can perform an action on every node except the node on the top right, which corresponds to the critical node. The defender can also perform every action possible since it can Migrate or Check Status on a node with status *Normal* or *Isolate Node* a compromised node. For the attacker, we can that nodes that are not connected to a compromised node are masked. Additionally, the attacker can perform all actions on the observed nodes.

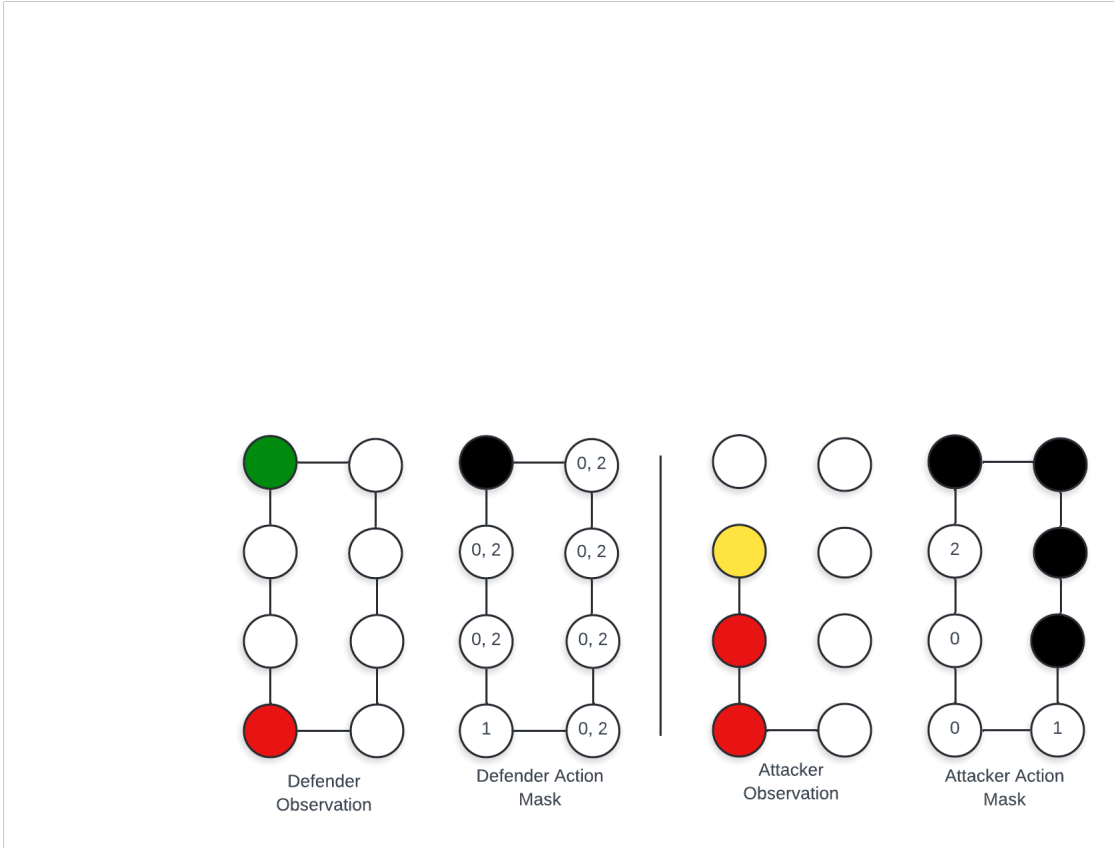

Fig. 8. Agent action masks.

3.4 Rewards

3.4.1 Reward Shaping

We consider two rewards: the reward defined by Gabirondo-lopez et al. [7] and our proposed, modified version of this baseline, defined in Equation 7:

$$R_w = \begin{cases} 10 \times TI & \text{winner = attacker,} \\ max(r_{min}, SR - 10 \times TI - TC) & \text{winner = defender,} \\ 0 & \text{otherwise} \end{cases} \tag{7}$$

where SR represents the steps remaining in the episode, TI is the sum of the impact scores of the compromised nodes in the network, and TC is the total cost of the countermeasures implemented by the defender. r_{min} is a parameter with values $r_{min} \in [0,1]$, where $r_{min} = 0$ is the value for the baseline and $r_{min} = 1$ is the value for the modified reward.

The cost per countermeasure is specified in Table 2. The costs are derived from the work by Chung et al. [32], which specified an intrusiveness and cost per countermeasure. The total countermeasure cost specified in Table 2 is the sum of the equivalent action's intrusiveness and cost. For the *Check Status* defender action, we use the countermeasure cost used in [7] since there is no equivalent action in the countermeasures used by Chung et al. [32].

We modify the reward R_w to reward the defender to isolate the attacker at the early stages of training [35]. Even if the defender's reward term before clipping $SR - 10 \times TI - TC$ is negative, some positive reward is useful feedback for the defender in the early stages of training. High TC or high TI which might otherwise result in a tie for the defender in the original reward would still be a win and a reward of $R_w = 1$, as long as the defender has successfully isolated the attacker. This is also useful feedback for the attacker as well, as it receives a reward of -1 instead of 0.

4 TRAINING PROCESS

4.1 Self-play

We use self-play to train the agents, similar to the work proposed in Bansal et al [22]. Instead of playing against the most recent opponent, we maintain, for each agent, an opponent pool of previously seen opponents. The attacker and defender then take turns playing games against an opponent in their respective opponent pools. We make a further distinction here between agents and policy, one that's useful in this context. The environment has two agents: the attacker and defender. The environment only has two agents but there can be many policies, each of which maps to a specific agent. We define a policy mapping function $\pi_* \rightarrow A_t$. This allows us to map the opponents in the opponent pool to agents in the environment.

Before training, opponent pools are initialized with one random policy for each respective opponent pool. There are two main trainable policies: the attacker and the defender. The two main trainable policies only play against the snapshot policies in the opponent pools, as shown in the setup in Figure 9.

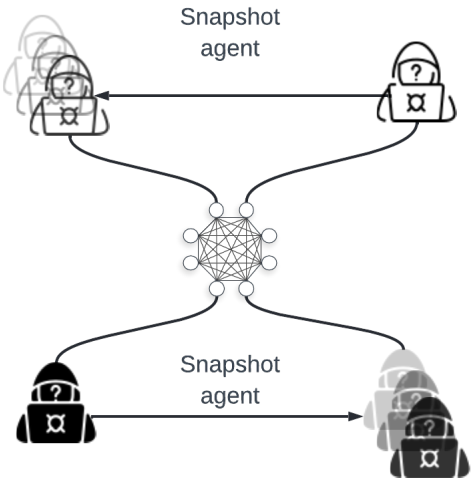

Fig. 9. Self play overview.

4.1.1 Opponent Selection

We consider two methods of adding new policies to the pools. The first is adding policies uniformly every d timesteps [22]. The second is using a league-based play setup, where policies are only added if they meet a threshold win rate [21].

Contrary to previous work, instead of setting a fixed threshold win rate throughout the entirety of training, we set an initial win threshold WT_a that adjusts to the performance of the agent(s) a. At each evaluation interval, the agent's win rate WR_a is assessed. If $WR_a >= WT_a$, then the agent's policy is added to the respective opponent pool. We then increase WT_a by a fixed amount $WinThreshOffset = 0.025$.

4.1.2 Opponent Sampling

Instead of a uniform sample of opponents [22], we consider a split opponent pool with a bias towards more recent agents similar to the approach in Oh et al [21]. First, we define three quantities: the probability of sampling one of the top $d\%$ agents p_{recent} , and the probability of sampling one of the $(1-d)\%$ agents $p_{past} = 1 - p_{recent}$. Over training, we decay the value of p_{recent} to gradually increase the chance to train against previously policies. This is done to address the issue of catastrophic forgetting.

4.2 Curriculum Learning

4.2.1 Vulnerability Scores

As part of the curriculum for training, we gradually increase the standard deviation for sampling both the *Impact* and *Exploitability* score. This allows the environment to sample from vulnerabilities with increasingly varying *Exploitability* and *Impact* scores. Our agents can then adapt to uncertainty in the vulnerability scores in the environment, compared to overfitting to fixed vulnerability scores in the training environment. The standard deviation starts at 0.01 and increases to a max value of 1 as shown in the Figure 10.

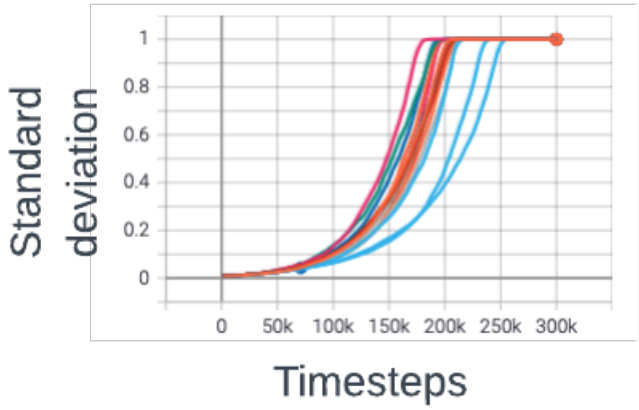

Fig. 10. Increase of standard deviation for exploitability score over training for sample experiment. Standard deviation for impact score follows a similar increase from $0.01 \rightarrow 1$. Colors represent different sample runs.

4.2.2 Network Topology

Beyond the four fixed topologies, we propose a dynamic network construction strategy as shown in Algorithm 4 that builds progressively more dense networks during training. The inspiration comes from the observation that the linear network is the least dense, single connected component. If we continue to add edges, eventually we converge to the fully connected network (clique). Also, the linear network is the least challenging environment from the perspective of the defender. This means we can start training with a linear network and progressively add topologies to the space of environments. Adding denser and denser topologies allows the defender to learn in progressively more challenging environments. And since we still sample the previous topologies, we address the issue of completely forgetting what was learned on sparser networks (catastrophic forgetting).

The strategy works as follows. At the start of each episode, an empty observation O_d is initialized with all zeros. For each node in the observation, we add the neighbors for a linear topology which acts as a baseline. Then for each successive neighbor, the choice of adding the edge is based on the result of sampling a binomial distribution with one trial and the probability $p = min(sampleThreshold, Random())$. The $sampleThreshold$ parameter is then adjusted during training from $0 \rightarrow 1$ to raise the probability threshold for adding edges in the network. And since $Random()$ samples from a random uniform distribution for values in the range $(0, 1)$, we sample from the entire space up to $sampleThreshold$.

This strategy, along with the previously mentioned threat randomization, creates a robust curriculum for training the agents.

Algorithm 4 Build dynamic topology

```
1:  procedure BUILDDYNAMICNETWORK(k, critPos, sampleThreshold)
2:      O_d ← 0
3:      for i ∈ O_d do
4:          if i + 1 < k then
5:              O_d[i][i + 1] ← 1
6:          if i − 1 >= 0 then
7:              O_d[i][i − 1] ← 1
8:          for j ∈ (i + 2, k) do
9:              edgeAddSuccess = Binomial(1, min(sampleThreshold, Random())   ▷ Add edge based on
    sample from binomial distribution
10:             if edgeAddSuccess = 1 then
11:                 O_d[j] ← 1
12:         O_d[critPos][critPos] ← 3                                  ▷ Set the critical node state
```

4.3 Training Setup

We use the RL framework Rllib with Ray which simplifies the training loop [36]. In addition, we use the PettingZoo framework which allows us to use an OpenAI Gym-like API to interact with the multi-agent environment [37] [38]. Since our environment has a small memory footprint, we can create multiple environments and collect samples in

parallel. We use tensorflow as the deep learning backend and use 4 CPUs for training: 2 dedicated for model predictions and gradient updates and 2 for running the environment. Each CPU dedicated for running the environment has 4 environments running concurrently.

5 EXPERIMENTAL EVALUATION

As part of our evaluation, we compare the win rate and tie rate achieved by the agent using the baseline reward demonstrated in Equation 7 against our proposed reward for the clique and ring topologies. We set the network size to 8 and train for 300k steps.

The vulnerability scores play a large role in the success of the both the defender and the attacker's actions and as such we are interested in quantifying the affect of the *Impact* and *Exploitability* on the defender's performance. We investigate the effect of the vulnerability scores on agent performance by modeling a variety of threat scenarios, using the Severity Ratings in Table 1 as a guide.

To evaluate the scalability of the approach, we train the dynamic training curriculum for varying network sizes. We use the network of size 8 as a baseline and train with sizes 16, 32, and 64.

As mentioned previously, real life networks are dynamic and we want to train the network to account for that. Nodes are added to networks, removed from the network, and links can change between nodes. We train agents on the discussed topologies and compare against the proposed curriculum. The objective is to demonstrate the effectiveness of the curriculum in training robust defensive agents.

Unless otherwise noted, we train all agents for a total of 300k timesteps. Agents are added to the opponent pool every 20k steps and agents are sampled from the pool with $d = 0.2$ and $p_{recent} = 0.8$. The default network size is set to 8 nodes. The mean $Impact = 4.31$ and the mean $Exploitability = 2.59$, again unless otherwise noted. We use the proposed one-clipped reward, except when comparing against the baseline zero-clipped reward.

5.1 Rewards

We begin by comparing the modified reward against the baseline, zero-clipped reward proposed by Gabirondo-Lopez et al. [7]. The performance of the two reward functions is compared across the clique and the ring topologies.

Figure 11 shows the average of the win rate (in red) as well as the confidence interval (in grey). For the proposed one-clipped reward, we can see that the win rate starts high and ends slightly higher for the ring than for the clique. For the reward in Figure 12, there is no appreciable significant difference for the clique between the two rewards. However, we can see that the one-clipped reward significantly improved the reward for the ring topology.

Fig. 11. Average defender win rate for one- and zero-clipped baseline reward with confidence interval. The max and min values are highlighted for each cell.

We observe a similar result in Table 3, where the reward is the average of the max value. For the one-clipped reward, we see a higher reward and lower tie rate for games

Fig. 12. Average defender reward for one- and zero-clipped baseline reward with confidence interval. The max and min values are highlighted for each cell.

played by the defender. The attacker also has lower average reward for both the clique and the ring with respect to the modified reward. Overall, we observe a lower tie rate and greater defender win rate of the proposed one-clipped reward over the baseline zero-clipped reward.

Table 3
Reward Results

Topology	**clique**		**ring**	
Reward	**zero-clipped**	**one-clipped**	**zero-clipped**	**one-clipped**
Max Attacker Reward	14.99	11.66	-1.7	-10.45
Attacker Win Rate	22.2%	22.0%	14.6%	12.8%
Attacker Tie Rate	37.6%	30.8%	45.3%	33.0%
Max Defender Reward	56.85	56.37	64.42	66.57
Defender Win Rate	46.5%	53.2%	46.1%	59.3%
Defender Tie Rate	35.9%	29.8%	42.6%	31.0%

5.2 Threat Scenarios

For the threat scenarios, we are interested in evaluating the effect of the vulnerability scores on the agent's win rate and expected reward. We consider the $ImpactScore \in [1, 2.5, 5]$ and $ExploitabilityScore \in [1, 2.5, 5]$. This creates 4 risk scenarios as showwn in Figure 13: Three in the low severity range, three in the medium severity range, two in the high severity range, and the last when the $BaseScore = 10$ and severity is critical. The *BaseScore* is calculated using Equation 5.

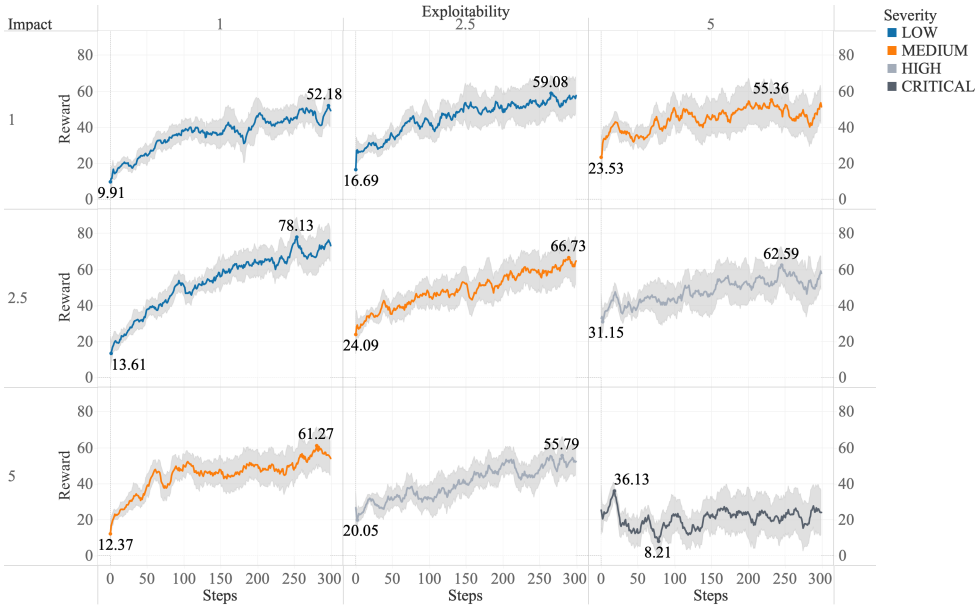

Fig. 13. Average defender reward for mean impact $ImpactScore \in [1, 2.5, 5]$ and $ExploitabilityScore \in [1, 2.5, 5]$. The max and min values are highlighted for each cell.

Figure 13 shows the reward trend downwards as the *Exploitability* increases. This aligns with our understanding of the attacker's actions, which are more likely to succeed and therefore discover and exploit vulnerabilities when exploitability is higher, as shown in Equation 2. As the *Impact* increases, we can see the defender reward increase when *Impact* increases from $1 \rightarrow 1.5$. However, that increase is followed by a drop in reward when *Impact* increases from $2.5 \rightarrow 5$. This is because the average total impact caused by

the attacker is much higher than the slight increase from $1 \rightarrow 2.5$ and factors into the penalty term of the defender's reward.

Even though the reward decreases, we can see in Table 4 that the defender win rate increases for increasing values of *Impact*, with the exception when the $Severity = CRITICAL$. Figure 14 shows the win rate categorized by severity. In summary, we observe more wins for the defender as impact increases and more wins for the attacker as exploitabililty increases.

Table 4
Threat Scenarios Results

Exploitability	**1**			**2.5**			**5**		
Impact	**1**	**2.5**	**5**	**1**	**2.5**	**5**	**1**	**2.5**	**5**
Max Attacker Reward	-7.9	-25.7	-11.4	-15.6	-12.2	-3.8	-6.8	-2.7	30.4
Attacker Win Rate	8.3%	5.2%	6.9%	14.1%	12.6%	10.5%	22.8%	18.0%	18.2%
Attacker Tie Rate	50.2%	37.2%	29.9%	45.3%	36.3%	34.2%	38.5%	31.2%	38.6%
Max Defender Reward	62.1	81.2	72.5	78.5	77.3	65.6	81.5	84.2	66.8
Defender Win Rate	45.5%	62.3%	65.4%	46.2%	55.9%	59.8%	46.1%	55.0%	47.4%
Defender Tie Rate	48.1%	33.8%	29.5%	42.7%	34.5%	32.4%	36.9%	30.8%	39.3%

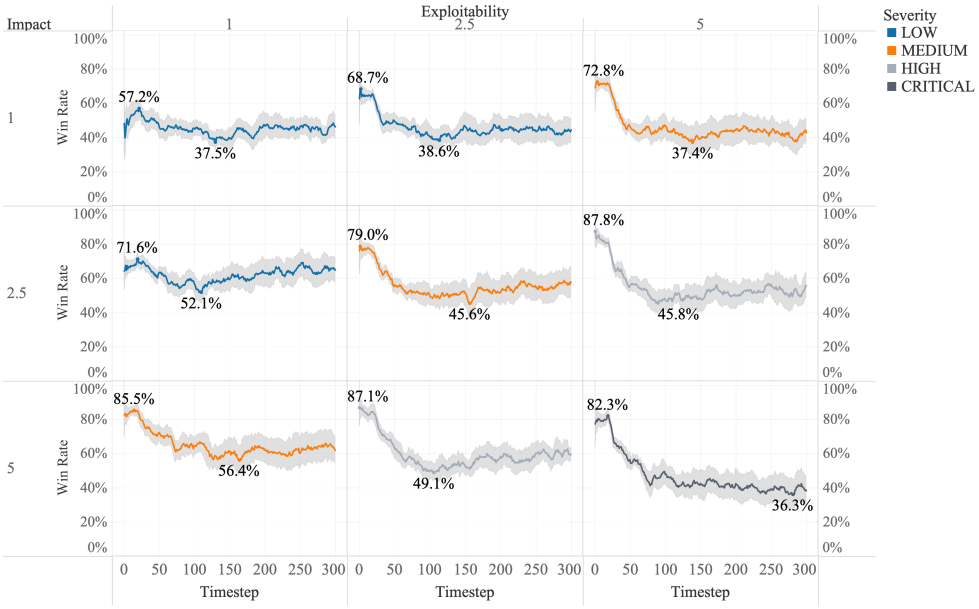

Fig. 14. Average defender win rate for $ImpactScore \in [1, 2.5, 5]$ and $ExploitabilityScore \in [1, 2.5, 5]$ with confidence interval. The max and min values are highlighted for each cell.

5.3 Network Size

For the size (number of nodes) of the network, we consider four values: 8, 16, 32, and 64. We use 8 as the baseline and use the same horizon (200 steps) for all episodes. We increase the total steps to 400k and unlike the rewards, we use a single topology (the dynamic topology curriculum) to train the agents. Table 5 shows a sharp drop in defender reward when increasing $k \rightarrow 16$, with a smaller drop when k increases to 32. Similarly in Figure 15 we can see the win rate follows a similar pattern, with the sharpest drop when $8 \rightarrow 16$ than when $16 \rightarrow 32$. However, the defender's loss is not completely the attacker's gain, as the majority of games end in a tie when $k > 16$.

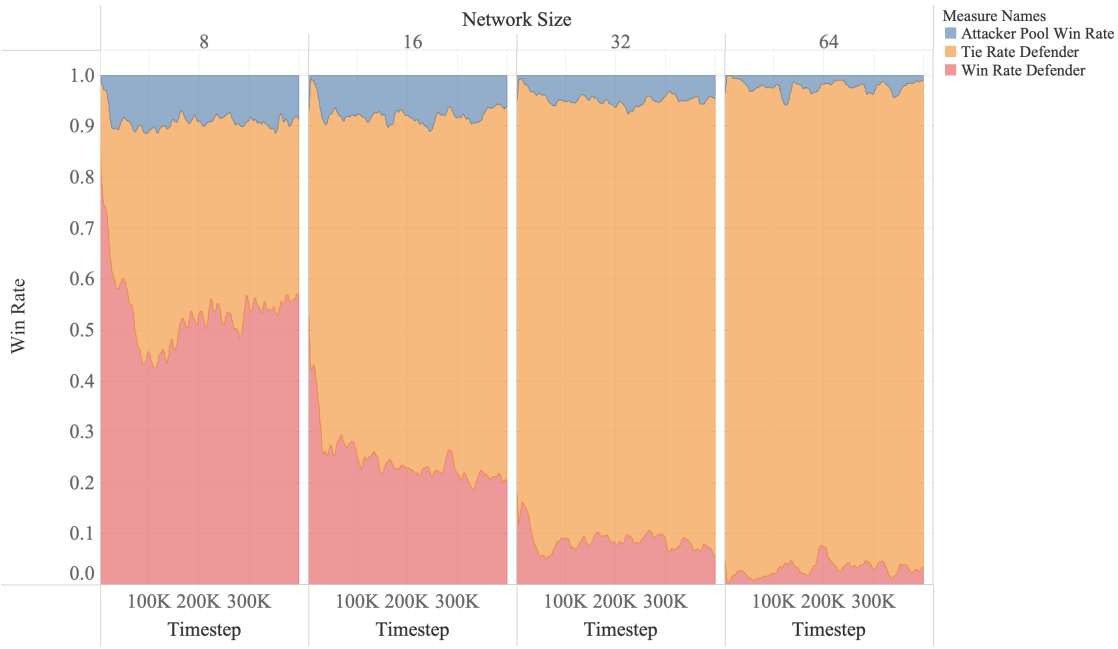

Fig. 15. Average defender win rate over training for Network Size with network size $k \in [8, 16, 32, 64]$. Max and min values highlighted for each cell.

By fixing the horizon across the experiments, we fix the total possible positive reward based on Equation 7. With increased network sizes the penalty terms *TI* and *TC* increase. This is due to the increased impact the attacker can achieve in the network and the total costs of the countermeasures required to successfully respond.

Table 5
Network Size Results

Network Size	**8**	**16**	**32**	**64**
Max Attacker Reward	-0.36	9.23	11.25	6.26
Attacker Win Rate	11.2%	8.5%	5.6%	3.1%
Attacker Tie Rate	40.3%	69.4%	87.3%	94.4%
Max Defender Reward	76.46	22.25	5.78	4.40
Defender Win Rate	52.4%	24.5%	8.6%	2.9%
Defender Tie Rate	38.6%	67.8%	86.9%	94.6%

5.4 Training Curriculum

We test the dynamic topology curriculum against two topologies: clique and linear. All topologies also use the vulnerability (threat) randomization. Therefore, we are isolating the dynamic topology from the threat score component of the curriculum. We compare with clique and linear as the two are at either ends of the space explored by the dynamic topology. Linear is seen early on in training and clique encountered as training progresses.

Figure 16 shows that the dynamic topology performs as well or better than the benchmark topologies. In particular, it performs as well or better than the linear topology, which as mentioned is the topology most biased in favor of the defender. This is supported by the max defender reward in Table 6, where the reward and the win rate for the dynamic topology is greater than for clique and linear. Also, less games end in a tie when the agent is trained using the dynamic topology. This demonstrates that the curriculum leads to both higher reward and higher win rate compared to training with a fixed topology.

Table 6
Training Curriculum Results

Topology	**clique**	**linear**	**dynamic**
Max Attacker Reward	17.64	-4.75	-7.56
Attacker Win Rate	19.72%	12.34%	10.92%
Attacker Tie Rate	31.3%	29.8%	29.0%
Max Defender Reward	66.89	73.26	75.41
Defender Win Rate	54.08%	62.22%	64.47%
Defender Tie Rate	30.4%	28.3%	26.8%

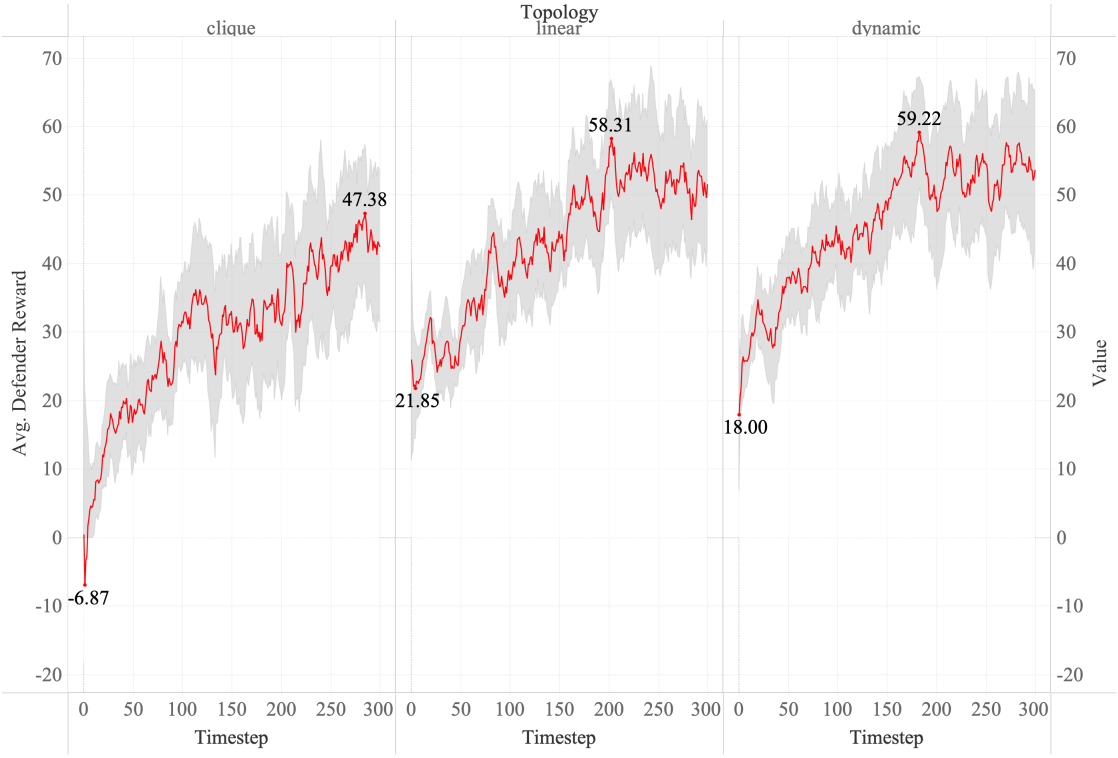

Fig. 16. Average defender reward with confidence interval comparing clique and linear topologies with dynamic topology curriculum. Max and min values highlighted for each cell

5.5 Self Play

5.5.1 Opponent Selection

For the win threshold opponent selection, we set 4 initial threshold values $WT \in [0, 0.25, 0.5, 0.75]$. Similar to the network training curriculum experiment, we test on the clique and linear topologies with the vulnerability randomization curriculum enabled. For the opponent sampling, we set the initial $d = 0.2$ and $p_{recent} = 0.8$. We train 15 training runs for each set of parameters and each training run is trained for 100k steps. In Figure 17, we see the reward is similiar for both topologies. This is likely due to the fact that the attacker win rate hovers around 20%. When the initial win rate threshold is too high then no new policies are added to the attacker pool. This can lead to high reward

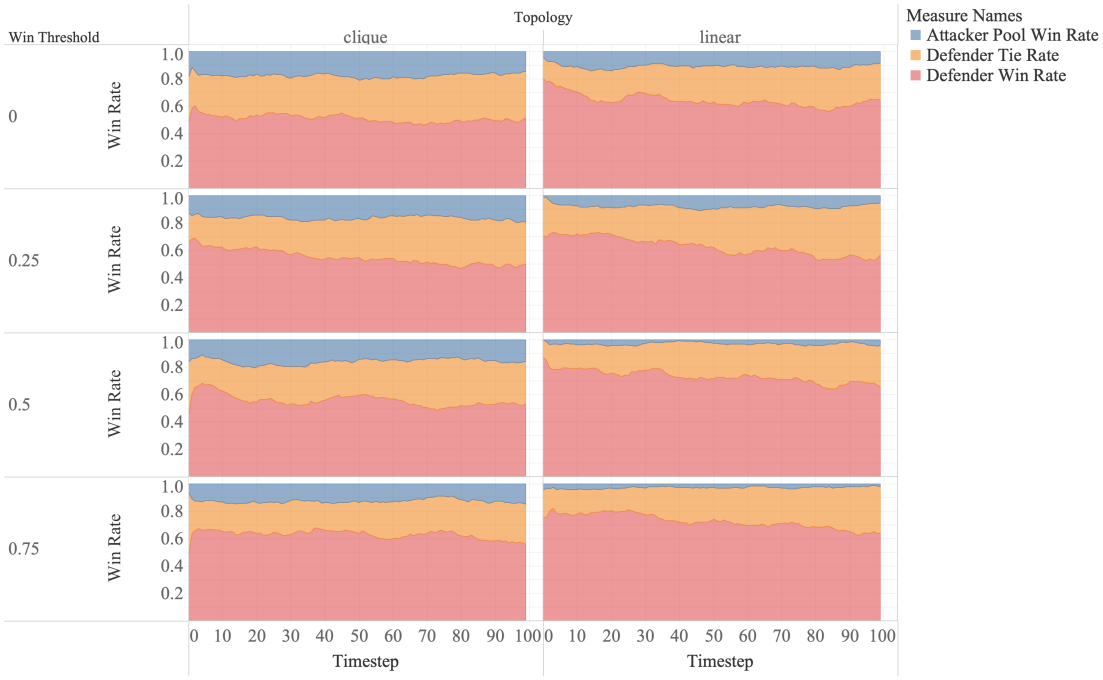

Fig. 17. Share of win rate over time for defender against attacker pool policies. Results shown for clique and linear topologies across 4 win threshold values.

and win rate for the defender, due to the agent overfitting on the single, initial policy in the pool. We can see the policies added over time in Figure 18. Less policies are added overall for the attacker in general. When the initial WT for the attacker is greater than 0.5, no new policies are added.

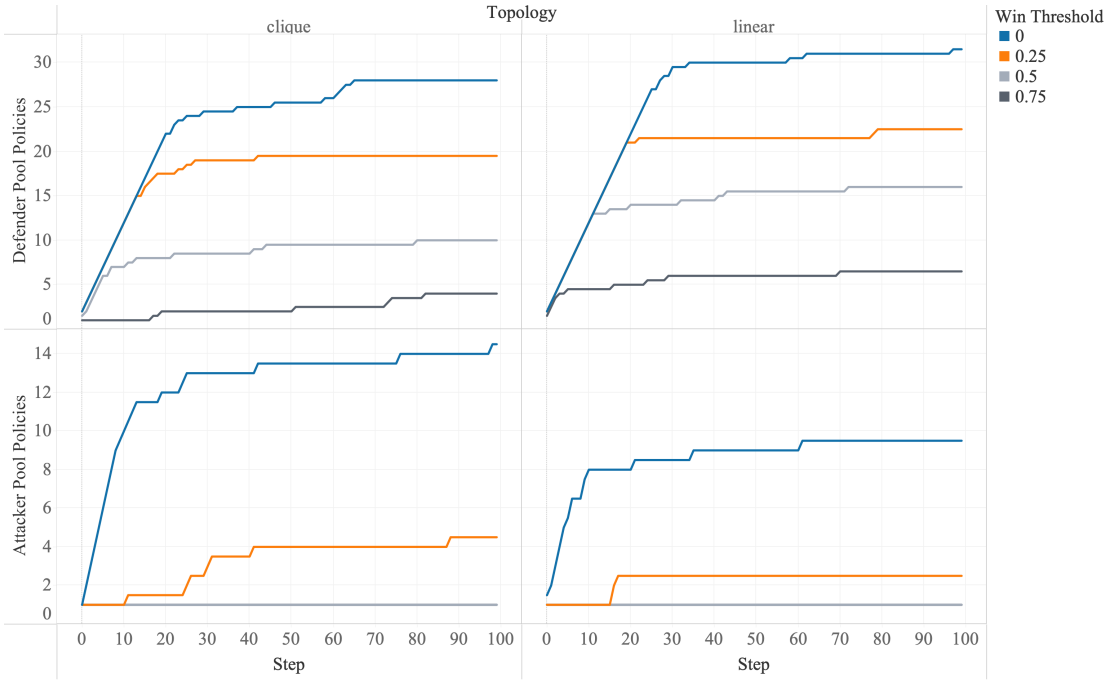

Fig. 18. Agents added to respective agent pool over 100k steps for two topologies: clique and linear. Results are further broken down by win threshold.

6 CONCLUSION

In this work, we have presented an intrusion prevention agent that learns how to select defensive actions by playing a zero-sum Markov game against an attacker in a network environment. Our method built on an existing approach to autonomous network defense and modified the proposed reward to improve convergence. We focus on evaluating the scalability of the approach by considering larger networks and varied topologies. This reflects real world network conditions as links may fail, outages can occur, and the number of hosts may change. To reflect these conditions, we unified the topologies as part of a robust training curriculum.

Our results show that our proposed, modified reward improves both reward and win rate for the defender. The characteristic high win rate and low reward in the beginning demonstrates that the defender is winning games at a high penalty which otherwise would be marked as a tie using the baseline reward. This is most likely due to the defender taking many high cost actions in order to isolate the attacker. As training progresses, the defender wins episodes by choosing actions more conservatively.

An important aim of this work was to quantify the effect of the vulnerability scores on the defender's performance. We show that the defender can generalize well across different threat scenarios. An environment with higher *Impact* vulnerabilities, on average, is more favorable to the defender. An environment wit higher *Exploitability* vulnerabilities, on average, is more favorable to the attacker.

We also note that network size has a large impact on defender reward, with each node increasing the action space of the agents and the potential impact of the attacker. In addition, the increased node size increases the actions needed to win the game for the defended which in turn increases the total cost (TC) term in the reward. We refer the reader to the supplemental material in Appendix A for more information.

For the self-play implementation, we chose to select opponents at a fixed interval as a baseline. We compared that approach with a selection strategy using the opponent win rate. The greater the win rate threshold, the less policies added to the pool of attackers. To address this, we set the initial win rate threshold and increased whenever the agent successfully met the threshold. This allows the win rate to vary between the two agents and we see that the lower the initial win rate threshold the lower the defender win rate as more opponents are added to the pool of attackers.

And finally, we remark on the training curriculum. The dynamic topology curriculum is a simple and powerful approach to gradually expose the defender agent to virtually all the topologies in a network of a given size. With the addition of the vulnerability scores, the agent learns in a challenging threat environment that works together to build a robust curriculum.

7 FUTURE WORK

As part of the network size evaluation, we noted that the episode horizon led to most games ending in a tie once the network size > 16. To remove the horizon as a bottleneck, we can evaluate the setup with a higher horizon and additionally add more network sizes.

Recurrent networks can address the partial observability of the environment, which could provide some benefit to the defender agent [25]. This may be helpful if the game was modified to a more realistic simultaneous game, instead of the current sequential, turn based approach.

We can expand the scope of the work by adding countermeasures to the action space of the defender, similar to the work in [32]. A possible future direction in the near term would be to align the agent observation more closely with information received from an IDS. In that way, we can integrate information from additional sources that may be useful for the RL agent for selecting defensive actions. The defender has a binary view of nodes in the network, either *Node Compromised* or *Normal*: more information about suspicious behavior and network activity would allow the agent to consider a wider set of countermeasures. This could involve using the *Impact* and *Exploitability* as part of the defender's observation but that assumes that all vulnerabilities on the network are previously seen and documented. Additionally, we are interested in implementing the defensive countermeasures in an SDN framework such as Ryu.

Literature Cited

[1] I. Security, "Xforce threat intelligence index 2022." https://www.ibm.com/security/data-breach/threat-intelligence (Accessed April 3, 2022).

[2] U. D. of Energy, "Colonial pipeline cyber incident." https://www.energy.gov/ceser/colonial-pipeline-cyber-incident (Accessed April 1, 2022).

[3] K. Arulkumaran, M. P. Deisenroth, M. Brundage, and A. A. Bharath, "Deep reinforcement learning: A brief survey," *IEEE Signal Processing Magazine*, vol. 34, no. 6, pp. 26–38, 2017.

[4] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Transactions on Neural Networks and Learning Systems*, pp. 1–17, 2021.

[5] K. Hammar and R. Stadler, "Finding effective security strategies through reinforcement learning and self-play," in *2020 16th International Conference on Network and Service Management (CNSM)*, pp. 1–9, 2020.

[6] A. DiGiovanni and E. C. Zell, "Survey of self-play in reinforcement learning." 2021.

[7] J. Gabirondo-López, J. Egaña, J. Miguel-Alonso, and R. Orduna Urrutia, "Towards autonomous defense of sdn networks using muzero based intelligent agents," *IEEE Access*, vol. 9, pp. 107184–107199, 2021.

[8] Y. Han, B. I. P. Rubinstein, T. Abraham, T. Alpcan, O. De Vel, S. Erfani, D. Hubczenko, C. Leckie, and P. Montague, "Reinforcement learning for autonomous defence in software-defined networking," in *Decision and Game Theory for Security* (L. Bushnell, R. Poovendran, and T. Başar, eds.), (Cham), pp. 145–165, Springer International Publishing, 2018.

[9] R. Howard, *Dynamic Programming and Markov Processes*. Cambridge, MA: Technology Press of Massachusetts Institute of Technology, 1960.

[10] M. Lapan, *Deep Reinforcement Learning Hands-on*. Birmingham, UK: Packt Publishing Limited, 2018.

[11] M. L. Littman, "Markov games as a framework for multi-agent reinforcement learning," in *Machine Learning Proceedings 1994* (W. W. Cohen and H. Hirsh, eds.), pp. 157–163, San Francisco, CA: Morgan Kaufmann, 1994.

[12] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. A. Riedmiller, "Playing atari with deep reinforcement learning." 2013.

[13] M. Hessel, J. Modayil, H. van Hasselt, T. Schaul, G. Ostrovski, W. Dabney, D. Horgan, B. Piot, M. Azar, and D. Silver, "Rainbow: Combining improvements in deep reinforcement learning," *Proceedings of the AAAI Conference on Artificial Intelligence*, vol. 32, Apr. 2018.

[14] R. J. Williams, "Simple statistical gradient-following algorithms for connectionist reinforcement learning," *Mach. Learn.*, vol. 8, p. 229–256, may 1992.

[15] P. Hernandez-Leal, B. Kartal, and M. E. Taylor, "Is multiagent deep reinforcement learning the answer or the question? A brief survey." 2018.

[16] T. P. Lillicrap, J. J. Hunt, A. Pritzel, N. Heess, T. Erez, Y. Tassa, D. Silver, and D. Wierstra, "Continuous control with deep reinforcement learning," in *4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, May 2-4, 2016, Conference Track Proceedings* (Y. Bengio and Y. LeCun, eds.), 2016.

[17] E. Todorov, T. Erez, and Y. Tassa, "Mujoco: A physics engine for model-based control," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, pp. 5026–5033, 2012.

[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, and K. Kavukcuoglu, "Asynchronous methods for deep reinforcement learning," in *Proceedings of The 33rd International Conference on Machine Learning* (M. F. Balcan and K. Q. Weinberger, eds.), vol. 48 of *Proceedings of Machine Learning Research*, (New York, New York, USA), pp. 1928–1937, PMLR, 20–22 Jun 2016.

[19] T. Haarnoja, A. Zhou, K. Hartikainen, G. Tucker, S. Ha, J. Tan, V. Kumar, H. Zhu, A. Gupta, P. Abbeel, and S. Levine, "Soft actor-critic algorithms and applications." 2018.

[20] A. Tampuu, T. Matiisen, D. Kodelja, I. Kuzovkin, K. Korjus, J. Aru, J. Aru, and R. Vicente, "Multiagent cooperation and competition with deep reinforcement learning," *PLOS ONE*, vol. 12, pp. 1–15, 04 2017.

[21] I. Oh, S. Rho, S. Moon, S. Son, H. Lee, and J. Chung, "Creating pro-level ai for a real-time fighting game using deep reinforcement learning," *IEEE Transactions on Games*, vol. 14, no. 2, pp. 212–220, 2022.

[22] T. Bansal, J. Pachocki, S. Sidor, I. Sutskever, and I. Mordatch, "Emergent complexity via multi-agent competition," in *International Conference on Learning Representations*, 2018.

[23] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. P. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm." 2017.

[24] C. Berner, G. Brockman, B. Chan, V. Cheung, P. Debiak, C. Dennison, D. Farhi, Q. Fischer, S. Hashme, C. Hesse, R. Józefowicz, S. Gray, C. Olsson, J. Pachocki, M. Petrov, H. P. de Oliveira Pinto, J. Raiman, T. Salimans, J. Schlatter, J. Schneider, S. Sidor, I. Sutskever, J. Tang, F. Wolski, and S. Zhang, "Dota 2 with large scale deep reinforcement learning." 2019.

[25] M. Hausknecht and P. Stone, "Deep recurrent q-learning for partially observable mdps," in *2015 AAAI Fall Symposium Series*, 2015.

[26] J. Schrittwieser, I. Antonoglou, T. Hubert, K. Simonyan, L. Sifre, S. Schmitt, A. Guez, E. Lockhart, D. Hassabis, T. Graepel, *et al.*, "Mastering atari, go, chess and shogi by planning with a learned model," *Nature*, vol. 588, no. 7839, pp. 604–609, 2020.

[27] T.-Y. Mu, A. Al-Fuqaha, K. Shuaib, F. M. Sallabi, and J. Qadir, "Sdn flow entry management using reinforcement learning," *ACM Trans. Auton. Adapt. Syst.*, vol. 13, nov 2018.

[28] D. K. Dake, J. D. Gadze, G. S. Klogo, and H. Nunoo-Mensah, "Multi-agent reinforcement learning framework in sdn-iot for transient load detection and prevention," *Technologies*, vol. 9, no. 3, 2021.

[29] P. Mell, K. Kent, and S. Romanosky, "Common vulnerability scoring system," 2006-12-29 2006.

[30] N. I. of Standards and Technology, "Common vulnerability scoring system." https://nvd.nist.gov/vuln-metrics/cvss/v3-calculator (Accessed Nov 1, 2021).

[31] Y. Bengio, J. Louradour, R. Collobert, and J. Weston, "Curriculum learning," in *ICML '09*, 2009.

[32] C.-J. Chung, P. Khatkar, T. Xing, J. Lee, and D. Huang, "Nice: Network intrusion detection and countermeasure selection in virtual network systems," *IEEE Transactions on Dependable and Secure Computing*, vol. 10, pp. 198–211, 2013.

[33] M. Minsky, "Steps toward artificial intelligence," *Proceedings of the IRE*, vol. 49, no. 1, p. 8–30, 1961.

[34] S. Huang and S. Ontañón, "A closer look at invalid action masking in policy gradient algorithms." 2020.

[35] A. Ng, D. Harada, and S. J. Russell, "Policy invariance under reward transformations: Theory and application to reward shaping," in *ICML*, 1999.

[36] E. Liang, R. Liaw, R. Nishihara, P. Moritz, R. Fox, J. Gonzalez, K. Goldberg, and I. Stoica, "Ray rllib: A composable and scalable reinforcement learning library." 2017.

[37] J. K. Terry, B. Black, A. Hari, L. Santos, C. Dieffendahl, N. L. Williams, Y. Lokesh, C. Horsch, and P. Ravi, "Pettingzoo: Gym for multi-agent reinforcement learning." 2020.

[38] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "Openai gym." 2016.

Appendix A

SUPPLEMENTAL MATERIAL

A.1 Network Size

For the network size results, we are interested in the relationship between the episode length and the resulting performance (defender win rate, defender reward). We can see in Figure 19 that the average episode length increases as the network size increases, with both $k = 32$ and $k = 64$ almost at the horizon length of 200. This shows that the horizon is acting as a bottleneck and we can likely reduce the number of ties if the agents are given more steps in the episode.

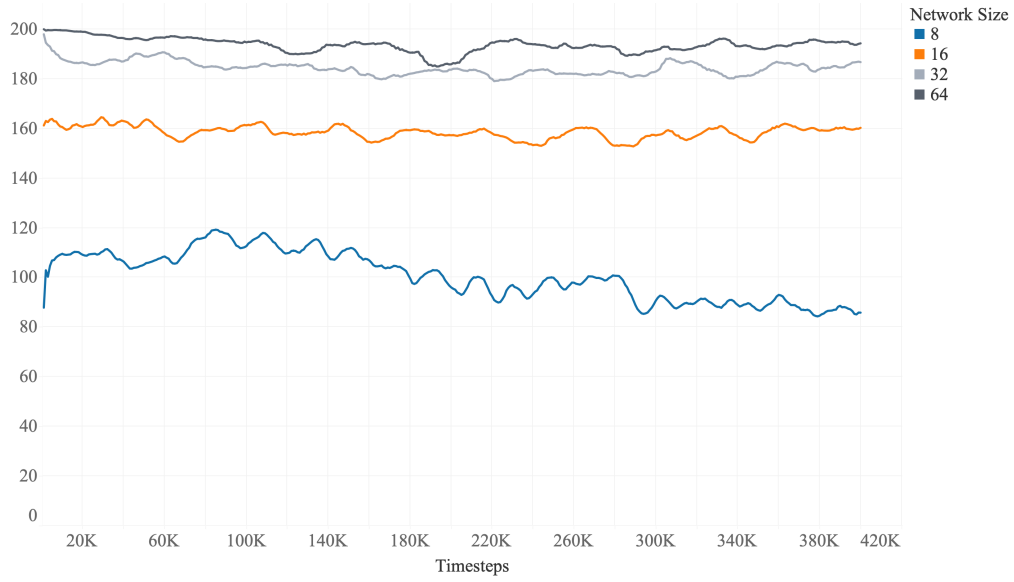

Fig. 19. Average episode length for defender episodes. Results are averaged over 15 runs.

In Figure 20, we can see the total cost term *TC* of the reward per network size. Values are recorded when the defender wins an episode. The total cost converges for all network size values, with some instability for size 64 which could be due to the low win rate.

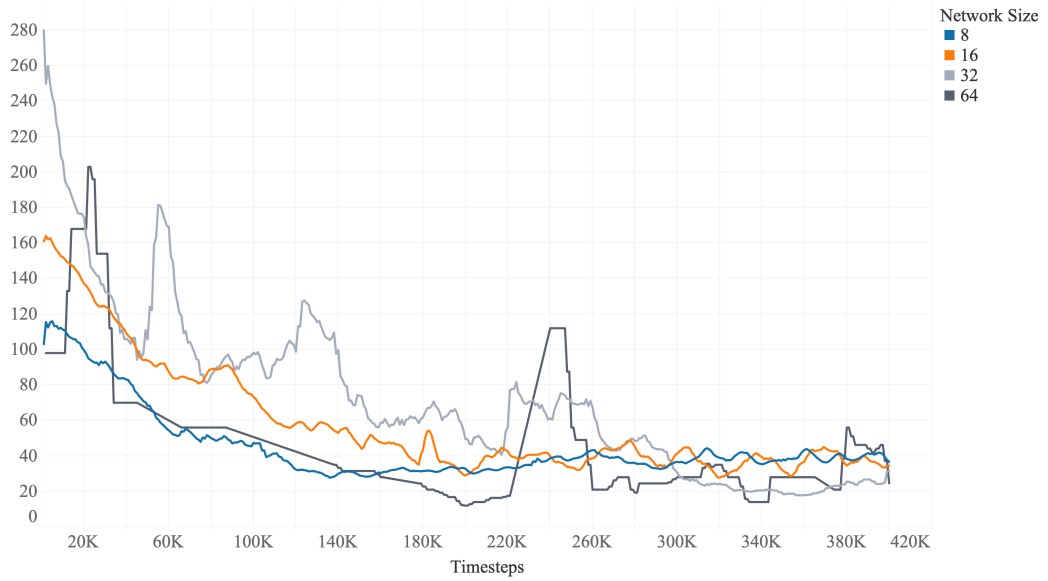

Fig. 20. Average total cost for defender countermeasures. Results are averaged over 15 runs.

A.2 Opponent Sampling

When sampling opponents from an opponent, the pool is split into two pools and sampling probability assigned to each agent based on the values of p_{recent} and d. In addition, we decay p_{recent} over the training. Shown in Figure 21 is an example for a pool with 10 opponents.

Fig. 21. Sample opponent pool when $p_{recent} = 0.8$ and $d = 0.2$. Values are probabilities for selecting respective agent.