Learning automated defense strategies using graph-based cyber attack simulations

Jakob Nyberg
KTH Royal Institute of Technology
Stockholm, Sweden
jaknyb (at) kth.se

Pontus Johnson
KTH Royal Institute of Technology
Stockholm, Sweden
pontusj (at) kth.se

***Abstract*—We implemented and evaluated an automated cyber defense agent. The agent takes security alerts as input and uses reinforcement learning to learn a policy for executing predefined defensive measures. The defender policies were trained in an environment intended to simulate a cyber attack. In the simulation, an attacking agent attempts to capture targets in the environment, while the defender attempts to protect them by enabling defenses. The environment was modeled using attack graphs based on the Meta Attack Language language. We assumed that defensive measures have downtime costs, meaning that the defender agent was penalized for using them. We also assumed that the environment was equipped with an imperfect intrusion detection system that occasionally produces erroneous alerts based on the environment state. To evaluate the setup, we trained the defensive agent with different volumes of intrusion detection system noise. We also trained agents with different attacker strategies and graph sizes. In experiments, the defensive agent using policies trained with reinforcement learning outperformed agents using heuristic policies. Experiments also demonstrated that the policies could generalize across different attacker strategies. However, the performance of the learned policies decreased as the attack graphs increased in size.**

Keywords*—cyber security, machine learning, reinforcement learning, attack graphs, attack modeling, cyber-physical systems, cyber-defense

I. Introduction

Cybersecurity is an important concern in our increasingly digital society. Consequences of cyber crime can be severe, both on an individual and societal level. Apart from managing personal finances and information, digital systems are also used in maintaining infrastructure such as power grids, water supply, and public transportation. The intricacies of these systems provide a large attack surface for malicious actors to exploit. To cover this attack surface, we believe that a combination of automation and human expertise is needed. Using automated systems to handle rote tasks of the decision-making process can allow human operators to focus on higher-level tasks that are yet difficult for a computer to perform.

An autonomic system is a system that can manage itself, and adapt to changes in its environment [1]. A common reference model for autonomic systems is the MAPE-K loop [1]. The MAPE-K loop consists of four steps: *monitor*, *analyze*, *plan*, and *execute*. An autonomic agent monitors its environment through *sensors* provided by the system, and executes actions through *actuators*.

In our implementation of the MAPE-K loop, the sensors are intrusion detection system (IDS) modules for providing alert signals, and the actuators a set of predefined defensive operations, such that can be defined in an software defined network (SDN) controller or host-based IDS system like Wazuh [2].

To analyze the state and plan actions, we use a policy function optimized using reinforcement learning (RL) that uses the IDS signals as input and selects a predefined defensive measure to execute. An issue with RL, especially when proceeded with the prefix "deep", is the need for large amounts of online data collection. Policies are learned by continually observing and interacting with the environment. If running the system is expensive or slow, this can be costly since training an agent can require thousands of episodes of experience. One approach to solve this issue is to train in a simulated environment. The simulator attempts to imitate the inputs, outputs and dynamics of the real system. The learned policy function can then be transferred to a real system with little or no additional training. This is sometimes referred to as *sim-to-real* transfer [3]. This work investigates the first half of the sim-to-real process, the simulation and training. To simulate a cyberattack, we employ approaches from the field of *threat modeling*, which focuses on identifying and analyzing threats to a system. Specifically, we use attack graphs produced using the Meta Attack Language (MAL) [4] to model the cyberattack process.

Although this work focuses on intrusion response, a major challenge in network defense is the *detection* of misuse or compromise. We eschew this issue by assuming that an IDS capable of detecting the ongoing attack is in place in the system we wish to apply our agent in, the output of which is used as input to the decision policy. In practice, the IDS can be a signature-based system, or on based on machine learning. We lighten our assumption by allowing the IDS to be imperfect, failing to register some events and falsely reporting on others. A question of interest to us was how well the defender agent could perform with, or in spite of, the imperfect information from the IDS. We investigated this by training the agent with different volumes of IDS errors, and comparing the performance of the agent against heuristic baselines. We were also interested in the ability of the RL policies to generalize to different attacker strategies. This was investigated by training policies using RL on a number of different attacker strategies, and evaluating the performance of the agent on previously unseen ones. Lastly, we investigated the effect of the size of the attack graph on the performance of the RL policies.

II. Related Work

Cyber intrusion detection and response are broad research fields whose scope extends far beyond that of this work. For the purpose of brevity, we focus this section on other works that employ reinforcement learning for intrusion *response*.

There have been several works that implement different varieties of automated defender agents based on RL [5]–[11]. Huang, Huang, and Zhu [11] list several applications where RL can or may be used to develop a cyber-resilient system. Gabirondo-López, Egaña, Miguel-Alonso, *et al.* [6] trains a variant of AlphaZero using self-play with an attacker and defender agent. Hammar and Stadler [8] presents and tries to solve the defense problem as an optimal stopping problem. Hu, Zhu, and Liu [10] formulates the cyber environment as a partially observable Markov decision process (POMDP) and uses Q-learning to find a solution policy. Several of these works also implement their own cyber attack simulators in order to learn defender policies. Named simulator include CybOrg [12], CyberBattleSim [9] and Yawning Titan [13].

The work of Andrew, Spillard, Collyer, *et al.* [13] bears close resemblance this one. They, too, present a graph-based cyberattack simulator and train a defender agent using the output of the simulator. Unlike this work, however, the authors use causal Bayesian optimization to train a policy function, rather than reinforcement learning. This has the advantage of producing a causal model of the defender's behavior. The causal model can then be used to interpret the defender's decisions, which can be difficult with methods based on neural networks. However, as noted by the authors, using causal optimization methods requires the manual construction of a causal system model. This is a potentially difficult and time-consuming process that may not be feasible for large or complex systems with many variables that influence decision-making. The same attack simulator is also used by Collyer, Andrew, and Hodges [14], but with an agent based on RL instead of a causal models. The authors use a graph embedding method to represent the graph, and use a graph convolutional network to process the graph. This allows the agent to generalize to graphs other than the one it was trained on. They note that for large graphs, the embedding method leads to improved performance in environments not seen by the agent during training.

The work of Wolk, Applebaum, Dennler, *et al.* [15] also share several aspects of this project. They employ a variety of methods based on RL and Proximal Policy Optimization (PPO) to train an automated defender agent on the CAGE Challenge [16], which is built on top of the CybORG simulator [12] cyber environment simulator. They find that an approach using an ensemble of PPO agents performed better than other variants, such as hierarchical PPO. They also evaluate their agents in unseen environments, such as to test how well they may perform in a Real environment, and find that the performance of drops significantly.

Related to the topic of automated defense agents is the concept of automated penetration testing, that instead aims to automate the process of finding and exploiting vulnerabilities in a system [17]–[21]. Automated attacker agent can also serve as opponents for automated defense agents, using adversarial machine learning or game-theoretic solution methods. For this work, we use heuristic policies for the attacker agent, but we believe that the use of RL for the attacker agent is a promising direction for future work.

III. Reinforcement Learning

Reinforcement learning is a field of machine learning focused on the interaction between an *agent* and an *environment* [22, Ch. 1]. The environment is usually modeled as a Markov decision process (MDP), and the agent should learn a policy for performing actions that maximizes a reward signal. Unlike MDP solver methods such as dynamic programming, reinforcement learning does not necessarily require a model of the environment. Instead, the agent interacts with the environment and learns a policy from observations and rewards produced by said environment. There are a number of different algorithms for reinforcement learning, including Q-learning [22, Ch. 6] and policy gradient methods [22, Ch. 13]. Policy gradient methods are focused on directly finding a policy function through gradient descent. The policy function maps a given state s to an action a via the parameters θ, and is denoted as $\pi_\theta(a|s)$. PPO is a policy gradient method that applies a clip function on the policy loss to limit the amount of change in the policy parameters in a single iteration [23]. This is intended to improve the stability of the learning process. The clipped policy loss is defined as

$$L_t^{CLIP}(\theta) = \tag{1}$$

$$\hat{E}_t \left[\min \left(r_t(\theta)\hat{A}_t, \text{clip}(r_t(\theta), 1-\epsilon, 1+\epsilon)\hat{A}_t \right) \right] \tag{2}$$

where $r_t(\theta) = \frac{\pi_\theta(a|s)}{\pi_{\theta_t}(a|s)}$ is the ratio of the new policy to the old policy, ϵ a hyperparameter that controls the clip limits and $\hat{A}_t$ is an advantage function.

IV. System Description

The basis of the system simulation is an *attack graph*. An attack graph is a model to predict and analyze possible events and outcomes of a cyberattack. The attack graphs used are based on those produced using the Meta Attack Language (MAL) [4], which can be used to create digital twins of cyber-physical systems and networks [24]. The graphs defined here share the logic and features of MAL attack graphs. Their construction, however, differs due to not being generated using a list of defined object classes and relations, but rather are constructed manually.

We define an attack graph as a directed graph $G = (V, E)$, where V is the set of nodes, and E is the set of edges between nodes. The nodes are divided into two sets: *attack steps*, A, and *defense steps*, D. Each attack step features a probability distribution specifying the time they require to be *compromised*, the time-to-compromise (TTC). Defense steps, on the other hand, are *enabled*. When a defense step is enabled, it will prevent any attack steps that has the defense step as a parent node from being compromised. A defense step can represent a file not being encrypted, or a firewall rule not existing. Activating the defense, e.g. making the file encrypted, makes the file no longer readable.

When an attack step is compromised, it may give access to other steps in the graph. For example, one attack step may

represent the action of performing a password dictionary attack. If the attack is successful, the attack step is compromised, and the attacker may proceed to a next step in the graph.

In accordance to MAL, attack steps can be of one of two types: AND and OR. AND-steps require all parent steps to be compromised in order to be compromised themselves, whereas OR-steps require only one parent step to be compromised. AND-steps can be used to represent preconditions such as an account requiring a password to be compromised. OR-steps, on the other hand, can be used to represent alternative paths, such as a user account being compromised by either a password dictionary attack or a phishing campaign. Figures 1b and 1a show visualizations of handcrafted attack graphs used for experimentation. AND-steps are indicated by dashed incoming edges.

We define the *attack surface* as a set of attack steps, $A_{as} \subset A$, that fulfill the following conditions:

1) There is an edge from a compromised attack step to the step.
2) One parent step is compromised, if the step is an OR-step, or all parent steps are compromised, if the step is an AND-step.
3) A defense step that is a parent node to the attack step is not enabled.

A. Episode Structure

In order to train a defender agent, we model an attack-defend capture-the-flag games as an MDP. The goal of the attacker is to capture as many targets, *flags*, as possible. The defender attempts to stop the attacker, and is penalized for each flag compromised.

The game is played with an attack graph acting as the basis of actions that can be taken by the attacker and defender. A subset of attack steps in the graph are assigned as flags, $F \in A$. Episodes are initiated with a single attack step in the attack graph being compromised, the entry point for the attacker.

The game is played in discrete time-steps, with the attacker and defender both acting within the same time-step. Every time-step, the attacker can select any attack step from the attack surface to work on. For every time-step the attacker works on an attack step, the TTC value is reduced by 1. When the TTC reaches 0, the attack step is compromised. As the defender's set of actions, it can select any defense step from the set of defense steps that are not enabled. When a defense step is enabled, all child steps are removed from the attack surface and can no longer be compromised by the attacker. Steps that have been compromised by the attacker are no longer considered compromised if defended. The defender can also choose to do nothing for a time-step. An episode ends when the attack surface is empty, meaning that there are no attack steps that can be reached by the attacker.

B. State Description

The state of the environment is expressed by two vectors, $\vec{A} = (a_0, \ldots, a_{|A|}) \in \{0,1\}^{|A|}$ and $\vec{D} = (d_0, \ldots, d_{|D|}) \in \{0,1\}^{|D|}$. $\vec{A}$ describes the state of all attack steps, and $\vec{D}$ the state of all defense steps. For $\vec{A}$, 1 represents that the step is compromised, and 0 that it is not. For $\vec{D}$, 1 represents that the defense is enabled, and 0 that it is disabled.

We assume that the network that is being defended has an IDS capable of tracking the state of every attack step in the attack graph. We also assume that this process may be faulty, and that the IDS may sometimes fail to report an attack step as compromised, or report a step as compromised when it is actually not. Thus, we define another vector, $\vec{O} = (o_0 \ldots, o_{|A|}) \in \{0,1\}^{|A|}$, to describe the observation of the attack steps produced by the IDS. As the state changes over time, the subscript t is used to denote the state at time-step t, e.g. $\vec{A}_t$ is the state of all attack steps at time-step t.

We define the accuracy of the IDS by a false positive rate (FPR) and false negative rate (FNR). Every time-step, depending on the FPR and FNR, the state of the system can be incorrectly reported by the IDS. The FPR and FNR can be expressed as the conditional probabilities

$$\text{FPR} = P(o_{ti} = 1 | a_{ti} = 0) \qquad i \in \{0, \ldots, |A|\} \tag{3}$$

$$\text{FNR} = P(o_{ti} = 0 | a_{ti} = 1) \qquad i \in \{0, \ldots, |A|\} \tag{4}$$

C. Attacker

The attacker agent uses a policy function to select actions defined by the attack graph, deciding which attack step to work on at a given time-step. The attacker agent can only select attack steps that are in the attack surface. Four different search policies were used in experiments for the attacker agent.

a) Random: A search policy that selects a random attack step from the attack surface to work on each time-step.

b) Breadth-first: A search policy that compromises all steps on the current depth of the attack surface before moving on to the next depth. The policy is randomized by shuffling the order that attack steps are compromised when multiple are available at the same depth.

c) Depth-first: A search policy that compromises each branch of the attack surface until it reaches an attack step with no children, and then backtracks to the start of another branch. The policy is randomized by shuffling the order in which branches are traversed when multiple are available.

d) Pathfinder: A policy that incorporates full information of the attack graph. It calculates the shortest path to each flag, and then targets flags in order of increasing TTC cost for the path. If blocked by a defense step en route, the policy will recalculate the shortest path to the targeted flag. If no path is available, it will target the next flag in the list.

D. Defender

The defender agent takes alert signals from the IDS as input, and makes a decision on which defense to enable. It can also choose to do nothing for a time-step. Decisions are made using a policy function, with $\vec{A}$ concatenated with $\vec{D}$ as input. The action space consists of all defense steps in D that are not enabled, plus a do-nothing action. In experiments, we evaluated three choices of defender policy function:

a) Random: A policy that selects a random available defense step each time-step.

b) Tripwire: A conditional policy, that emulates the behavior of "if-this-then-that" rules present in IDS frameworks such as Wazuh [2]. An example rule may be to automatically block traffic from a certain host if a port scan is detected. Within the attack graph environment, this is translated to activating a defense step when one of its child step is reported as compromised.

c) Reinforcement Learning: A policy trained using RL. The policy function was parametrized by a fully-connected neural network and optimized using the PPO algorithm. The neural network had an input layer of size $|A| + |D|$, a set number of hidden layers, and an output layer of size $|D| + 1$. The number of hidden layers and their sizes were treated as hyperparameters. $\tanh$ was used as the activation function for the hidden layers.

E. Reward

The defender agent has two goals in the game: to minimize the number of compromised flags, and to minimize the operational cost of the defense. The operational cost is an abstraction that could, for example, represent the cost of downtime resulting from disabling a service or network for security purposes. A defender with the singular goal of defending the flags could easily maximize its reward by shutting off access to the entire system immediately and be done, as the attacker will have nothing to do at that point. However, this would prevent normal users from accessing the system, which we consider undesirable. Defensive measures are thus assumed to have a set cost c_d. With $\vec{F}_t = \{f_0, \dots f_{|F|}\} \in \{0,1\}^{|F|}$ denoting flag states, the reward for the defender agent at a given time-step t is defined as

$$r_d(t) = -\sum_{i=0}^{|D|} d_i c_d - \sum_{i=0}^{|F|} f_i c_f \quad d_i, f_i \in [0,1],\ c \in \mathbb{R} \tag{5}$$

where c_d is the cost of activating a defense step $d \in D$, c_f is the cost of taking a flag $f \in F$. Note that $f_i = 1$ only if the flag was taken at time-step t, meaning that the penalty is only incurred once for each flag taken. The defense penalty, on the other hand, is incurred on every time-step for every defense step enabled, i.e. $d_i = 1$. There are no positive terms in the reward function, making the maximum possible reward 0. The minimum reward would be incurred if the defender enables all defenses from the first time-step, and the attacker still takes all flags during the episode. As the defender agent can only disable one defense per time-step, the minimum cumulative reward for an episode of length l is

$$-c_d \left(\sum_{i=1}^{|D|-1} i + |D|(l - (|D| - 1)) \right) - c_f |F| \tag{6}$$

V. Experiments

Three experiments were performed to evaluate the simulator and defender agent: A comparison between the RL policy and heuristic policies at different levels of IDS accuracy, a comparison between RL policies trained with different attackers and an analysis of how the policy training is affected as the graph size increases.

RL policy training and evaluation was performed on a Google Cloud virtual machine equipped with an Nvidia V100 GPU, 12 CPU cores, and 30 GB of RAM. The implementation used the Python library Ray RLLib [25] for implementations of the reinforcement learning algorithms. All policies trained with RL were run for 500 PPO policy iterations. All policies were evaluated by running 500 episodes. Each policy took roughly 20 minutes to train and evaluate.

Experiments were performed three times with different seeds for the random number generator. The neural networks used two hidden layers with 128 nodes.

All TTC values except those explicitly set to 0 were initialized at the start of each episode by sampling from the exponential distribution $f(x;\beta) = \frac{1}{\beta} e^{-\frac{1}{\beta}x}$, where β is the mean TTC value assigned to each attack step.

The costs were set as $c_f = 1.5 \cdot \sum_{a \in A} \text{TTC}(a)$ and $c_d = 1$ for all flags and defense steps respectively, where $\text{TTC}(a)$ denotes the TTC value for attack step a. The choice of c_f was made such that the cost of losing a flag would be greater than disabling a defense step on the first time-step.

A. Sensor Fault Resistance

A desired property of a good defender agent is the ability to operate in spite of imperfect information. Therefore, an experiment was performed to study the effects of false alerts and missed alerts on the defender. The rate of errors in the observations is defined using two values, the FPR and the FNR. Five values for the FPR and FNR were selected: $0\,\%$, $12.5\,\%$, $25\,\%$, $72.5\,\%$ and $100\,\%$. A partial grid search was performed over combinations the error rates, for a total of 15 points. Points where $FNR > 1 - TNR$ were excluded since they are equivalent to those where $FNR < 1 - TNR$, but with reverse definitions of the true and false labels.

Three defender agents, one using policies trained with PPO, one using the tripwire policy and one using the random policy were compared. Figure 1a depicts the attack graph used as the environment, with depth-first search as the attacker policy. For the RL agents, one policy was trained and evaluated for each combination of FPR and FNR values.

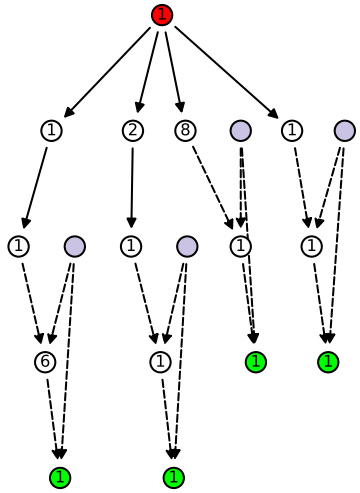
B. Attacker Comparison

A second experiment was performed to study the generalization capabilities of the defender agent trained using RL. As such, only the PPO policy was used for this experiment. The agent was trained with one attacker policy, and then evaluated against all attacker policies listed in Section IV-C. A mixture of the attacker policies was used as an additional variant, where the attacker policy was chosen at random at the beginning of each episode. Figure 1b depicts the attack graph used. The FPR and FNR were set to $10\,\%$.

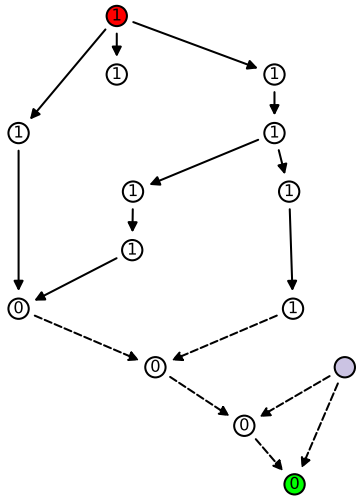
C. Graph Size

Real-life attack graphs can be huge due to the complexity of real-life systems [4]. Thus, an experiment was performed to study the effects of graph size on policy learning.

Four graphs of sizes 20, 40, 60 and 80 steps were generated using a semi-random attachment procedure. Each graph has



(a) Attack graph used for FPR/FNR sweep experiment. Each of the four flags can be defended using a defense step.



(b) Attack graph used for attacker policy comparison experiment.

Fig. 1: Manually constructed attack graphs used in experiments. Defense steps are colored purple, flags green and attacker entry points red. AND-steps have dashed incoming edged. Average TTC values are drawn as numbers in nodes.

$\frac{|A|}{20}$ flags. Each flag step had a defense step attached to it, ensuring that each flag was defensible. A policy was trained for each graph size using PPO and evaluated together with the tripwire policy. The same hyperparameter values were used for training on all graph sizes. The FPR and FNR were set to $10\,\%$.

VI. Results

Sections VI-A, VI-B and VI-C present the results of the experiments described in Section V. Results for all experiments are averaged over three runs with different seeds for the random number generator (RNG). The RNG affects the TTC values sampled at that episode initialization, attacker behavior, and neural network parameter initialization.

A. Sensor Fault Resistance

Figure 2 presents performance metrics for the three defender agents as surface plots. There are two figures for each policy, one showing the average percentage of flags captured by the attacking agent, and one showing the average reward for the 500 episodes of evaluation. Episodes were, in average, 13 time-steps long for agents using PPO and tripwire policies, and 15 time-steps for the agent using a random policy. The agent using the random policy has a consistent performance across the different combinations, as it does not use observations to begin with. The amount of flags captured was higher than for the other policies, and the reward was thus lower.

For all but two combinations of error rates, the learned policies produce a higher average reward than the tripwire policy. For the two combinations where the FPR ≥ 0.5 and the FNR $= 0$, the difference in reward was no longer statistically significant ($p < 0.05$).

The performance of both learned polices and the tripwire policy decrease as the FPR increases. The PPO policies performs better when the FNR increases compared to the tripwire policy, the performance of which rapidly decreases with an increased FNR.

Averaged across all combinations, the PPO policies have an average cumulative reward of -50, compared with -73 for the tripwire policy and -72 for the random policy.

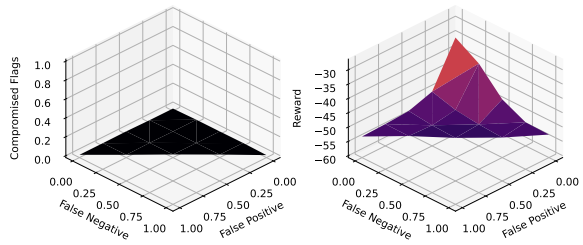
B. Attacker Comparison

Figure 3 shows bar plots of the rewards and percentage of flags compromised. Episodes were, in average, 9 time-steps long. The longest recorded episode was 19 time-steps, and the shortest 6 time-steps.

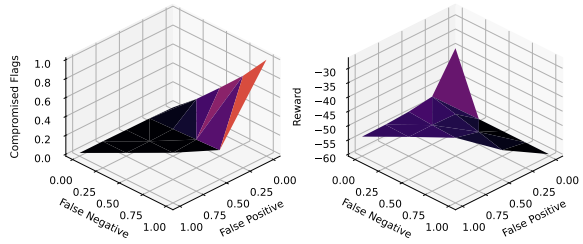
There is a significant difference between the rewards produced by the agents when faced with different attacker policies than those they were trained on. The policy trained against the depth-first attacker produces the lowest reward compared to the others when faced with the depth-first attacker. However, it performs much better on average when faced with other strategies. Inversely, the policy trained against the breadth-first attacker has the best overall performance compared to the other policies when faced with only one attacker, but is significantly worse when faced with others. When averaged over all attackers, the difference in reward between policies not trained against the breadth-first attacker were not statistically significant ($p < 0.05$).

C. Graph Size

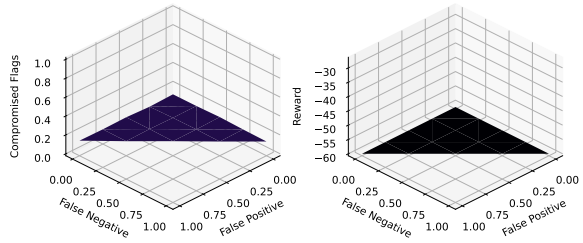
The performance of the policies learned by PPO decrease as the graph size increases. For all graph sizes greater than 20, the RL policies produced worse results than the tripwire policy.



(a) Surface plots showing performance metrics for defender agent using RL policies. The percentage of flags taken by the attacker remains low for all values of the FPR and FNR.



(b) Surface plots showing performance metrics for defender using the tripwire policy. As the FNR increases, performance decreases significantly.



(c) Surface plots showing performance metrics for defender taking random actions. The reward is consistently low compared to other methods, and the amount of flags lost to the attacker is higher.

Fig. 2: Surface plots showing metrics for different defender policies. Results are averages over three runs with different RNG seeds.

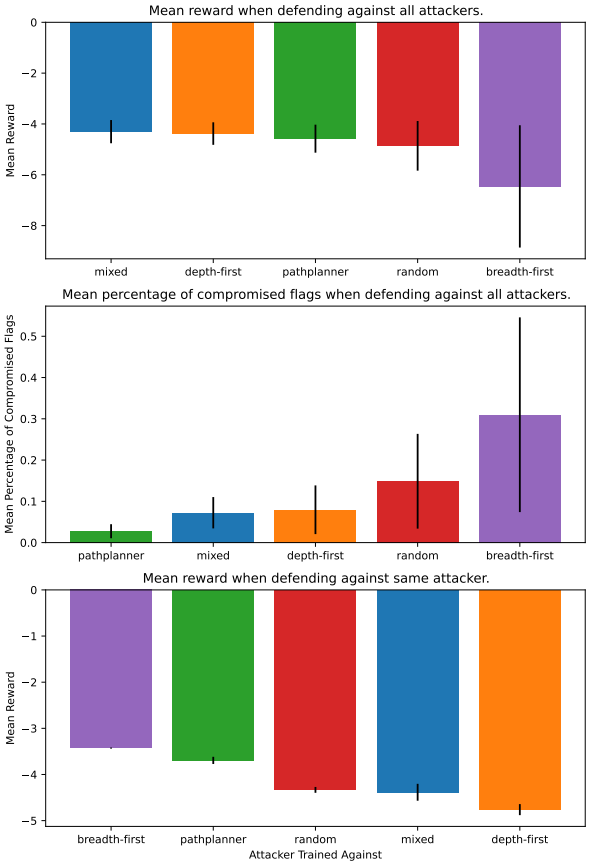


Fig. 3: Three plots showing PPO policy performance against different attacker policies during evaluation. The top and middle plots show the average reward and flags captured when faced with attacker policies not encountered during training. The bottom plot shows the reward when evaluated against the attacker policy used during training. The metrics are averaged over three runs with different RNG seeds.

The tripwire policy also decreases in performance as the graph size increases, but the decrease is less than for the RL policies.

VII. Discussion

The experimental results presented in Section VI-A demonstrate that the defender agent using a policy with RL can produce a higher reward than one using the tripwire policy. Through qualitative analysis, we can see that the reason for this was that the RL defender learns to enable defenses earlier than the tripwire policy, thus ending episodes more quickly. The RL policy was also more resilient to missed alerts than the tripwire policy. This was likely because the RL policy was able to use signals from more than one attack step to compensate for missed alerts, whereas for the tripwire policy, if the IDS fails to report the state for the particular step the policy was monitoring, the policy will not enable that defense, potentially leading to a compromised flag.

A. Other Baselines

We compared the RL policy against two heuristic policies, neither of which were optimal. Comparing against the optimal policy would be beneficial to judge the effectiveness of the learned policies. In order to find the optimal policy for a particular configuration, one could apply POMDP solver methods. These may become difficult to apply, however, for larger graphs as the number of possible states is 2^n, two to the power of attack steps. The state space thus grows exponentially. This also makes it difficult to compare the neural network policy to a tabular RL policy, like Q-learning that store a value for each state-action pair.

B. Scaling

A large drawback of the RL method was the time it took to train the neural network policy function. The training time only grew larger as graphs sizes increased and, inversely, the performance of the policies decreased. The decrease in performance with increased graph sizes comes from a number of factors. One of these was optimization robustness. As the graph size increases, the size of the neural network grows linearly. Despite the size of neural network and reward signals changing, the same set of hyperparameters was used for all graph sizes. A procedure to find appropriate hyperparameters for each graph size would likely be necessary to make training more robust. One could also take an orthogonal approach and apply same neural network to all graph sizes. This would require a different approach to how the graph state was represented, as the current approach is not invariant to changes in the graph structure or size. Graph convolutional networks are a possible solution to this problem, as demonstrated in previous work [14].

Another complicating factor was the rules of the attack-defense game. When the graph size increases, so does the episode length. The episode ended only when the attacker no longer can perform any actions. Even if the defender enables all defenses, the attacker can still traverse the remaining nodes, thus extending the episode length. Adjusting the episode end conditions could help alleviate this issue, such as imposing a time limit for the attacker. Another issue that comes from long episodes is that the PPO policies are stochastic. This means that the longer the episode, and the more actions are available, the more likely it is that the defender will eventually take a low-probability action. As there is no way to undo actions, the defender can eventually enable every defense even if the attacker does nothing. This could be addressed by adding a probability threshold for enabling a defense, or by always selecting the most likely one.

C. Generalization

Generalization is a desirable property for the defender agent, as it would allow the agent to be used in different environments without having to retrain the policy. This was also one of the main motivations for using deep RL, as we would like to be able to learn policies in environments that are easily simulated, in order to be able to test them in more realistic environments down the line. Figure 3 demonstrates that the policy can generalize across different attacker policies. However, so can the tripwire policy, as it is invariant to the behavior of the attacker. A more interesting task is to generalize to different graph environments. In the current implementation, a learned policy function is only applicable to a single graph. The graph topology has to be static, and no attack steps can be added or removed. Additionally, the agent was only presented with the node states, and not the topology of the graph. Both of these problems may be addressed by using a model that can incorporate the graph topology, like a graph neural network. Another task to test generalization is to evaluate trained policies against different volumes of IDS noise, to determine which level is best to train the policy on.

D. The Sim-to-Real Gap

The defender agent is trained in a simulated environment, with the intention of being used in a real environment. Within the field of robotics, it has been shown that the transfer of learned policy functions from simulation to real world can deteriorate performance [3]. This is a consequence of differences between the real world and the simulation model, leading to worse policy performance. For this work, the target environment is a computer network. One issue related to the field of cybersecurity in particular is the construction of realistic attack scenarios. Even if the network itself is modeled accurately, the attacks and adversaries the defender trains against may be unrealistic.

VIII. Conclusion

An automated cyber defense agent using policies trained with RL was implemented and evaluated. The agent was trained and tested in a simulated environment based on MAL attack graphs. The RL policy was compared against heuristic policies when faced with different volumes of IDS noise. We observed that the agents implemented using RL could learn policies that were better than the heuristic policies, and that they were more resilient to missed alerts. From our second experiment, we also observed that the neural network-based policy had the ability to generalize to different adversarial attackers, and that some adversaries were better training opponents than others. Unfortunately, the performance of the learned policies decreased as the graphs increased in size. Future work will focus on the ability for the defender agent to generalize across graphs, and crossing the gap between simulation and the real world, transferring a defender agent trained in simulation to an actual computer network.

References

[1] J. Kephart and D. Chess, "The vision of autonomic computing," *Computer*, vol. 36, no. 1, pp. 41–50, Jan. 2003, Conference Name: Computer, ISSN: 0018-9162. DOI: 10.1109/mc.2003.1160055.

[2] Wazuh. "Wazuh the open source security platform." (2023), [Online]. Available: https://wazuh.com/ (visited on 01/04/2023).

[3] W. Zhao, J. P. Queralta, and T. Westerlund, "Sim-to-real transfer in deep reinforcement learning for robotics: A survey," in *2020 IEEE Symposium Series on Computational Intelligence (SSCI)*, IEEE, Dec. 2020, pp. 737–744. DOI: 10.1109/ssci47803.2020.9308468.

[4] P. Johnson, R. Lagerström, and M. Ekstedt, "A meta language for threat modeling and attack simulations," in *Proceedings of the 13th International Conference on Availability, Reliability and Security*, ACM, Aug. 2018, pp. 1–8. DOI: 10.1145/3230833.3232799.

[5] 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. DOI: 10.1109/TNNLS.2021.3121870.

[6] 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. 107 184–107 199, 2021. DOI: 10.1109/ACCESS.2021.3100706.

[7] Y. Han, D. Hubczenko, P. Montague, *et al.*, "Adversarial reinforcement learning under partial observability in autonomous computer network defence," in *2020 International Joint Conference on Neural Networks (IJCNN)*, L. Bushnell, R. Poovendran, and T. Başar, Eds., ser. Lecture Notes in Computer Science, Cham: IEEE, Jul. 2018, pp. 145–165, ISBN: 978-3-030-01554-1. DOI: 10.1109/ijcnn48605.2020.9206634.

[8] K. Hammar and R. Stadler, "Learning intrusion prevention policies through optimal stopping," in *2021 17th International Conference on Network and Service Management (CNSM)*, IEEE, Oct. 2021, pp. 509–517. DOI: 10.23919/cnsm52442.2021.9615542.

[9] C. Seifert, J. Bono, J. Parikh, and W. Blum. "Gamifying machine learning for stronger security and AI models." (2020), [Online]. Available: https://www.microsoft.com/security/blog/2021/04/08/gamifying-machine-learning-for-stronger-security-and-ai-models/ (visited on 07/27/2021).

[10] Z. Hu, M. Zhu, and P. Liu, "Adaptive cyber defense against multi-stage attacks using learning-based POMDP," *ACM Transactions on Privacy and Security*, vol. 24, no. 1, pp. 1–25, Nov. 2020, ISSN: 2471-2566, 2471-2574. DOI: 10.1145/3418897.

[11] Y. Huang, L. Huang, and Q. Zhu, "Reinforcement learning for feedback-enabled cyber resilience," *Annual Reviews in Control*, vol. 53, pp. 273–295, Jan. 1, 2022, ISSN: 1367-5788. DOI: 10.1016/j.arcontrol.2022.01.001. [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1367578822000013.

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

[13] A. Andrew, S. Spillard, J. Collyer, and N. Dhir, "Developing optimal causal cyber-defence agents via cyber security simulation," in *International Conference on Machine Learning 2022, Workshop on Machine Learning for Cybersecurity*, *Workshop on Machine Learning for Cybersecurity*, vol. abs/2207.12355, 2022. DOI: 10.48550/arXiv.2207.12355. arXiv: 2207.12355.

[14] J. Collyer, A. Andrew, and D. Hodges, "ACD-G: Enhancing autonomous cyber defense agent generalization through graph embedded network representation," en, in *International Conference on Machine Learning 2022, Workshop on Machine Learning for Cybersecurity*, Jul. 2022. [Online]. Available: https://dspace.lib.cranfield.ac.uk/handle/1826/18288.

[15] M. Wolk, A. Applebaum, C. Dennler, *et al.*, "Beyond cage: Investigating generalization of learned autonomous network defense policies," in *NeurIPS 2022, Reinforcement Learning for Real Life (RL4RealLife) Workshop*, Nov. 28, 2022. DOI: 10.48550/ARXIV.2211.15557. arXiv: 2211.15557 [cs.LG].

[16] *Cyber autonomy gym for experimentation challenge 2*, https://github.com/cage-challenge/cage-challenge-2, Created by Maxwell Standen, David Bowman, Son Hoang, Toby Richer, Martin Lucas, Richard Van Tassel, Phillip Vu, Mitchell Kiely, 2022.

[17] H. Holm, "Lore a red team emulation tool," *IEEE Transactions on Dependable and Secure Computing*, pp. 1–1, 2022, ISSN: 1545-5971, 1941-0018, 2160-9209. DOI: 10.1109/tdsc.2022.3160792.

[18] R. Gangupantulu, T. Cody, P. Park, *et al.*, "Using cyber terrain in reinforcement learning for penetration testing," in *2022 IEEE International Conference on Omni-layer Intelligent Systems (COINS)*, IEEE, Aug. 2022, pp. 1–8. DOI: 10.1109/coins54846.2022.9855011.

[19] M. C. Ghanem and T. M. Chen, "Reinforcement learning for efficient network penetration testing," *Information*, vol. 11, no. 1, 2020, ISSN: 2078-2489. DOI: 10.3390/info11010006. [Online]. Available: https://www.mdpi.com/2078-2489/11/1/6.

[20] T. Cody, "A layered reference model for penetration testing with reinforcement learning and attack graphs," in *2022 IEEE 29th Annual Software Technology Conference (STC)*, 2022, pp. 41–50. DOI: 10.1109/STC55697.2022.00015.

[21] Z. Hu, R. Beuran, and Y. Tan, "Automated penetration testing using deep reinforcement learning," in *2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)*, 2020, pp. 2–10. DOI: 10.1109/EuroSPW51379.2020.00010.

[22] R. S. Sutton and A. G. Barto, *Reinforcement Learning*. Springer US, 2018, ISBN: 9781461366089, 9781461536185. DOI: 10.1007/978-1-4615-3618-5.

[23] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, and O. Klimov, "Proximal policy optimization algorithms," *CoRR*, vol. abs/1707.06347, 2017. arXiv: 1707.06347.

[24] M. Masi, G. P. Sellitto, H. Aranha, and T. Pavleska, "Securing critical infrastructures with a cybersecurity digital twin," *Software and Systems Modeling*, Jan. 2, 2023, ISSN: 1619-1366, 1619-1374. DOI: 10.1007/s10270-022-01075-0.

[25] E. Liang, R. Liaw, R. Nishihara, *et al.*, "RLlib: Abstractions for distributed reinforcement learning," in *Proceedings of the 35th International Conference on Machine Learning, ICML 2018, Stockholmsmässan, Stockholm, Sweden, July 10-15, 2018*, J. G. Dy and A. Krause, Eds., ser. Proceedings of Machine Learning Research, vol. 80, PMLR, 2018, pp. 3059–3068. [Online]. Available: http://proceedings.mlr.press/v80/liang18b.html.