Defending the unknown: Exploring reinforcement learning agents' deployment in realistic, unseen networks

Alberto Acuto[1,*], Simon Maskell[1] and Jack D.[2]

[1]School of Electrical Engineering, Electronics and Computer Science, The University of Liverpool, Brownlow Hill, Liverpool, L69 3GJ
[2]The Alan Turing Institute

Abstract

The increasing number of network simulators has opened opportunities to explore and apply state-of-the-art algorithms to understand and measure the capabilities of such techniques in numerous sectors. In this regard, the recently released Yawning Titan is one example of a simplistic, but not less detailed, representation of a cyber network scenario where it is possible to train agents guided by reinforcement learning algorithms and measure their effectiveness in trying to stop an infection. In this paper, we explore how different reinforcement learning algorithms lead the training of various agents in different examples and realistic networks. We assess how we can deploy such agents in a set of networks, focusing in particular on the resilience of the agents in exploring networks with complex starting states, increased number of routes connecting the nodes and different levels of challenge, aiming to evaluate the deployment performances in realistic networks never seen before.

Keywords

Reinforcement Learning, Cyber security, Network simulation

1. Introduction

The development of autonomous resilient agents in the context of automated cyber defence (ACD) to counteract the actions of external or malevolent actors is becoming a pivotal research topic from both academy and governmental agencies. In recent years cyber crimes have increased their presence in the day-to-day life of organisations and governmental institutions, and research in automated cyber defence is one of the most developed topics [1]. Novel technologies such as machine learning (ML) and reinforcement learning (RL) are increasingly employed for both defence and attack thanks to their adaptability, cyber resilience and variety of applications. Some defensive examples are ML applications in spam detection [2], malware and intrusion detection [3, 4], offensive applications can relate to the deployment of algorithms to exploit vulnerabilities of infrastructures and limit the visibility and extend duration (or frequency) of

CAMLIS'23: Conference on Applied Machine Learning for Information Security, October 19–20, 2023, Arlington, VA
*Corresponding author.
a.acuto@liverpool.ac.uk (A. Acuto); s.maskell@liverpool.ac.uk (S. Maskell); j.d@turing.ac.uk (J. D.)
https://github.com/A-acuto (A. Acuto); http://www.simonmaskell.com/ (S. Maskell)
0000-0003-0753-5131 (A. Acuto); 0000-0003-1917-2913 (S. Maskell)

threats [5]. However, simpler ML models are prone to react slowly to changes in the system and not in real-time, while RL algorithms tend to be more flexible to the changes and they had been successfully deployed in the detection of spoofing attacks and DDoS attacks [5, 6]. A relevant summary of the current state-of-the-art in environments where this problem is tackled can be found in several review articles [6, 7, 8, 9], and, particularly, in the work by Wang W. et al. [7], in which the authors consider the role of RL as new technology to tackle cyber defence decision making. In order to develop realistic cyber scenarios, a number of autonomous cyber operations gyms (ACO) have been developed [6], some examples are CybORG [10], TTCP CAGE challenges [11, 12, 13] and FARLAND [14]. A recent one consists of a network simulator developed in the context of UK ARCD program[1], Yawning Titan[2] ([15], hereafter, YT), that offers an environment where we can train and explore the capabilities of RL agents to counteract the actions of an enemy red agent. YT is a complex piece of software that models the intrusion of a "red agent" into a network and a defendant "blue agent" needs to counteract the threats posed. In this paper, we have considered several RL algorithms based on discrete actions space, following the YT simulations reward-action mechanism, to train and evaluate the agents' performances in different scenarios from size and complexity in order to test the resilience of the agents. Then, after having identified the best-performing algorithms, we deployed these agents on unseen realistic networks with training done on synthetic cases. This paper is structured as follows: in Subsection 1.1 we present the Yawning Titan software, in Section 2 we highlight briefly the algorithms considered in this work. In Section 3 we present the simulation and experiment design, in detail in Subsection 3.2 we train and evaluate the agents on a set of example networks looking for the best performing algorithm and in Subsection 3.3 we explore the deployment of such agents in realistic network environments. Finally, in Section 4 we summarise the findings. The codes presented in this paper are available on the following github repository https://github.com/A-acuto/RLYawningTitan.

1.1. Yawning Titan

Yawning Titan is a graph-based cyber-security simulation environment that allows the training of intelligent agents to counteract the actions of a red enemy agent that aims to spread into the network. The YT setup specifies the red and blue agents' capabilities (e.g. the usable actions and success rate) and the network's description: connections between the nodes, entry nodes and the presence and location of a "high-value target" (HVT). Each agent has a set of parameters describing the probability of success of the actions and game rules (i.e., how the red agent can spread from a compromised node or if the blue agent can detect failed attacks in the network). In detail, the red agent has a set of actions, one of which is randomly picked, to "attack" a node, move in the network or "do nothing". The blue agent, instead, has a wider set of possible actions which he can perform:

- *Isolate*: removes all edges of the node, cost 10;
- *Restore*: returns the node to its original status (from "compromised" to "safe"), cost 1. The agent can be punished if patches a safe node or if there are too many infected nodes;

[1] Autonomous Resilient Cyber Defence, https://www.gov.uk/government/news/autonomous-resilient-cyber-defence-intelligent-agents.
[2] https://github.com/dstl/YAWNING-TITAN .

- *Make node safe*: reduces the vulnerability[3] of a node, cost 0.5. The agent can be punished in the same way as for *Restore* action;
- *Connect*: reinstate all edges of the node, cost 0, the agent is rewarded by 5 points if it reduces the number of isolated nodes;
- *Add deceptive node*: add an extra "fake" node between two nodes to slow the spread of the red agent[4], cost 8. The agent is also punished by 5 points if adds more deceptive nodes than allowed (in our case 3);
- *Do nothing*: self explanatory, cost -0.5. The agent is punished by doing this action if there are a lot of infected nodes.

The blue action space can be modified using a configuration file. The score is obtained from a a combination of the action costs plus the rewards obtained from removing red nodes, the penalties from the actions and final points from ending the game (winning or losing, ± 100 points). The score is parameter the agents need to optimise. The network is the "gym" where the two agents interact and can either be loaded from an existing scenario or prompted by the user. There are multiple ways to describe a network: it is fundamental to map the specific connections across the nodes[5] and with that knowledge, it is possible to generate an adjacency matrix that is read and interpreted by a graph-based Python library (Networkx[6]) which can interact using Pandas. A network is defined by nodes connected with edges, some of whose are defined as "entry nodes" which are the starting points where the red agent will begin its spread. It is possible to add a "high-value target", a valuable asset inside the network, the red agent can target that specific node and, under certain game rules, can be the trigger for the endgame. In the network, the software generates each node and assigns a specific vulnerability, entry nodes tend to have a higher vulnerability score because are infection starting point. At the end of each simulation step, the vulnerabilities are evaluated in terms of the red and blue actions. At the beginning of each simulation, we have a safe network where every node is clean and the game parameters are set (entry and HVT nodes).

2. Algorithms

We have considered algorithms from the library `Stable-baseline3` which represent model-free RL algorithms where an "agent" learns to play by interacting with the environment. A trained agent has the knowledge of the states by performing actions, obtaining a reward (positive or negative) and their effect on the environment. The aim of the agent is to learn the best actions from a policy in order to maximise the total rewards across an episode, which is everything that happens between the first and the last state in the environment (considered like a timestep). We have considered online, model-free RL algorithms[7] with well-documented applications across

[3] The vulnerability score of a node is a metric that is used for evaluating the risk of the node to be attacked. Exposed nodes and nodes neighbouring, connected, to a compromised node have a higher vulnerability, meaning a higher probability of being infected.

[4] Adding a deceptive node does not count as adding a node in the network.

[5] True, for version 1.0.1. In more recent versions, e.g. 2.0.1b, the user can also draw the network.

[6] https://networkx.org/documentation/stable/index.html.

[7] In the case of offline methods we should create a dataset from the simulation and then train the agents on such data, without the live interactions between the agent and the environment.

different simulations [16] and that have a discrete action space. YT environment describes the possible actions on a discrete space, which scales by the number of nodes and usable actions on each node. Other RL algorithm needs box-shaped actions space or images (e.g. in the cases of CNNPOLICIES) which is not straightforward to implement with YT. The policies considered are MLPPOLICIES, where we pass the state vectors of the network in our input model. These policies implement an actor-critic neural network using a multilayer perceptron (with 2 layers of 64 neurons). The algorithms we are considering are Proximal Policy Optimisation (or PPO [17]), Advantage Actor Critic (or A2C [18]) and Deep Q-Network (or DQN [19, 20]).

PPO algorithm works using a policy gradient optimisation based on natural policy gradients. This algorithm is known to perform better in comparison to similar ones, because the training is more stable by avoiding broad policy updates, helping the convergence on an optimal solution and allowing enough time to recover from an action [21]. The algorithm is based on the optimisation of the policy objective function using a gradient descent (or ascent) and uses a "clipped" surrogate of the objective function which prevents too large policy updates.

A2C is an Actor-Critic method based on temporal difference learning[8] that represent the policy function independent of the value function. Our implementation is Advantage Actor Critic and comes from the Asynchronous Advantage Actor-Critic (A3C) without the asynchronous part. This algorithm, as PPO , uses the policy gradient to weigh the actions and reduce the variance by using a large number of samples (created by single agent exploring the action space) hoping that one of these will provide the true estimation.

DQN is a deep reinforcement learning algorithm which uses Q-learning to learn the best action to take in the given state and a deep neural network (or convolutional neural network) is implemented to estimate the value of the Q-function.

2.1. Algorithms hyper-parameters exploration

We consider exploring how the different algorithms react by changing some hyper-parameters such as the discount factor (γ) and the learning rate (lr). γ measures the rewards the agent has achieved in the past, present and future. An agent with $\gamma = 0$, only cares about his first reward (myopic approach), while if $\gamma = 1$, it is interested in all the future rewards. lr is a parameter that measures how often and, how quickly, the Q-values are updated, improving the steps toward the solution, smaller lr can slow the gradient descent while, a larger value, can fail to converge.

In DQN algorithm, we have reduced the buffer size to 10000 (from 10^6), because in the largest network (>50 nodes) there was the risk of requiring more memory than allocated on the HPC[9] cores. By doing so, we both achieved a faster convergence and assured us to not overload the computing nodes.

There are studies (i.e., [22]) that demonstrate and compare various RL agents in different contexts by changing and tuning the various hyper-parameters. For the means of this paper, we have not fine-tuned our agents to perform in the networks because we wanted to have control

[8]Temporal difference learning methods are a class of model-free reinforcement learning algorithms which learn by bootstrapping the current estimate of the value function.

[9]High performance computing. The training of the agents were performed on HPC CPU cores at University of Liverpool computing facility.

over the way the performances may differ. We have chosen the best models to deploy in realistic networks according to the testing on sample networks.

3. Simulation setup

We have trained agents on a set of networks comprised of a small case of 18 nodes (25 edges), a medium one of 50 nodes (>250 edges) and the largest case of 100 nodes (>500 edges) with an increasing number of entry points (3, 5, 10). A trained agent on an 18 nodes network cannot be deployed onto a larger or smaller network because the dimension of the action space is bound to the possible states in the specific case.

The simulation setup has a red agent that can spread only via connected nodes with 45% infection success rate and 15% chance of spreading from connected infected nodes, the endgame rule is that the red agent wins if it takes over 80% of the network, 500 timesteps are the target for a blue victory. The HVT is chosen randomly and furthest away from the entry points. We train and analyse the performances of the various algorithms on a set of example networks, using the same network for both training and evaluation, then, we test the best algorithms on a series of realistic network configurations after training on similar networks with the same amount of nodes.

3.1. Training the agents

We perform the training of the various agents in the networks without any hyper-parameter tuning. Then, we train the agents by modifying just one parameter at a time, γ and lr on the same networks. We set up 5×10^5 timesteps and we consider the convergence when we have not measured any improvements of the average rewards for up to four consecutive evaluations (a single evaluation is the average of rewards over 50 timesteps).

Table 1
We present the algorithms and their hyper-parameters modified in the training phase, the first uses the standard hyper-parameter values. We divide the training on the three network sizes (left, central and right column) showing the training time (in seconds) and the final score obtained in each case.

Algorithm		18 Nodes		50 Nodes		100 Nodes	
		Training time [s]	Final score	Training time [s]	Final score	Training time [s]	Final score
PPO	$\gamma = 0.99, \mathrm{lr} = 0.0003$	1722	-130	3899	-108	7264	-120
	$\gamma = 0.75$	1702	-122	4375	-107	5972	-102
	$\mathrm{lr} = 0.001$	1714	-117	4337	-120	6655	-100
A2C	$\gamma = 0.99, \mathrm{lr} = 0.0007$	2221	-99	3149	-331	13660	-558
	$\gamma = 0.75$	2235	-109	3467	-353	4540	-2220
	$\mathrm{lr} = 0.001$	2221	-110	3463	-352	13700	-547
DQN	$\gamma = 0.99, \mathrm{lr} = 0.0005$	1655	-114	6800	-235	12389	-409
	$\gamma = 0.75$	1617	-121	5714	-283	14755	-400
	$\mathrm{lr} = 0.001$	1641	-119	5120	-308	13852	-476

In table 1, we compare the training of the agents showing the training time (in seconds) and the final scores. All agents converge to an optimal solution before the end of the training, we find, also, that adding more nodes (larger action space) the time requirements are higher. The final score can give us an indication of the expected results during the trials, however, we should not be surprised by any different results. If we have to compare the training times: PPO seems to be the quickest to converge in all three network sizes. The A2C algorithm reaches the stability quite quickly, even if it is not the better performing at the end of the training session, the PPO agents tend to have steady and stable growth in performances during the training while DQN agents have a constant behaviour: in all training, they have very little gain during the initial steps, then they rapidly improve their performances taking over the A2C performances as well.

3.2. Deploy the trained agents

In this section, after having presented the training, we compare the performances of the agents on the same seeded networks and we obtain the mean and standard deviation of the scores achieved (using `Stable-baseline3` evaluator function EVALUATE_POLICY). We compare the performances with the scores obtained by a random agent on the same networks, this agent randomly chooses a node and one action from the available. By testing the agents' performances on the same seeded network we aim to evaluate the agents on a constrained set of examples of starting points, red agent actions, reducing the variability of the games and aiming to understand better how they behave.
In figure 1, we compare the agents' performances against the same networks. In diamonds we present the standard hyper-parameters setup, crosses for the algorithms with $\gamma = 0.75$ and in triangles the case with $\mathrm{lr} = 0.001$. We show the PPO results using blue dots, orange for the A2C case and green for DQN algorithm. We can see that in the case of 18 nodes, all algorithms have similar and comparable performances (around -130 as mean reward). In the 50 nodes case, we measure a more significant difference in performances, in particular in the DQN case, almost 5 times lower than PPO results. Instead, in the 100 nodes case we measure a considerable lower score in the PPO case (in which the mean reward passes from a few hundred to -2000) while, on the opposite side, DQN shows higher scores. As stated earlier, we changed the buffer size of DQN , by doing so we measure a significant improvement in the performances highlighting a strong positive influence of this hyper-parameter. Both A2C and DQN algorithms, with a reduced discount factor, perform better than PPO in the largest network case. By increasing the network size, the agents have more opportunities to take action, and more chances of having negative rewards because the red agent is able to spread more. Therefore, the simulations are longer, and many of them resulted in the blue agent victory since the agent was able to slow the spread by making more expensive actions such as adding deceptive nodes. In cases of large networks, it is important to see also how spread are the final scores, measured by the standard deviation, and understand how an algorithm can overall perform.

We want to extend this analysis while changing the starting conditions, in detail: add isolated nodes, compromised nodes, a mixture of isolated and compromised nodes, changing the number of edges and the red agent's skills.
In figure 2 and 3, we summarise the mean reward scores obtained by the various agents in the networks applying the proposed changes on the network. Isolated and compromised nodes

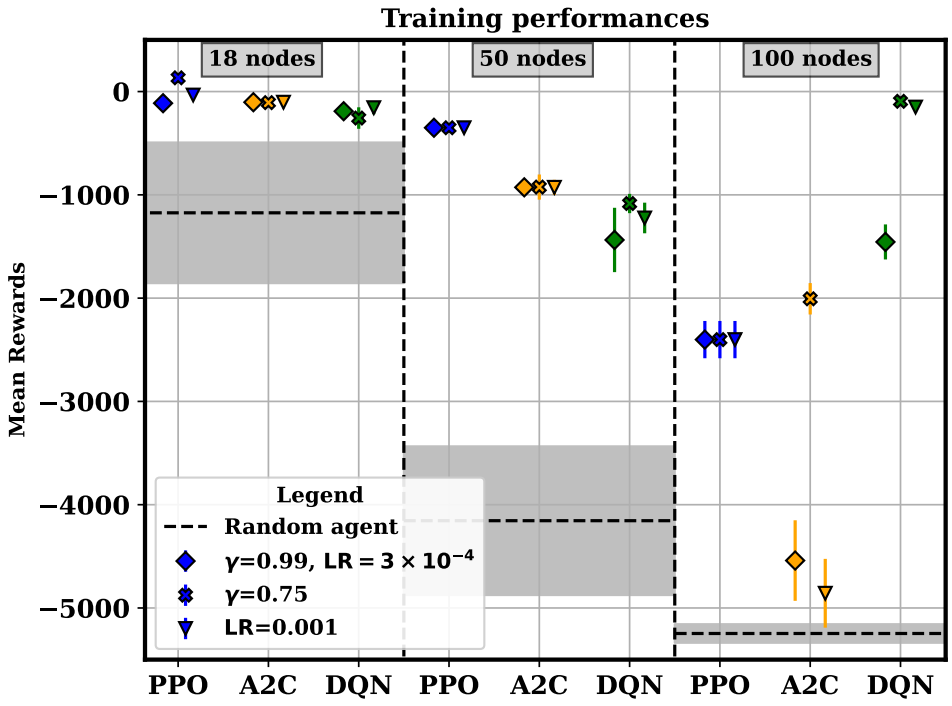

Figure 1: Agents' performance comparison in the different networks (left panel 18 nodes, centre 50 nodes and 100 nodes right panel). We show the algorithms with standard hyper-parameters using diamonds, in crosses the case modifying γ and in triangles when modifying the lr. We use different colors to easily identify the algorithms: blue for PPO , orange for A2C and green for DQN . We show as well the 1σ deviations of the scores with colored lines. The grey band is the random agent scores.

were randomly chosen, therefore it may have happened that some nodes were both isolated and compromised at the same time. Changing the number of edges in the network creates, or removes, routes for the red agent to spread, but also allows the blue agent to defend better the network by adding more deceptive nodes or isolating cross-road nodes, reducing the effectiveness of the spread. Changing the red agent skills increases, or reduces, the simulation challenge level because a red agent that has a higher success rate spreads more quickly and it is more difficult to react to, on the other hand, a less effective red agent would leave more time for the agent to fix the nodes.

We can see that, in most cases, tweaking the network (nodes isolated or compromised) does not result in a measurable change in the final scores. For instance, the PPO algorithm obtains a mean of 130 points in all three variations when we lower γ. Considering the DQN algorithm, we find a similar picture in which even if the scores differ from the standard case, the variations trials have comparable results between them. Considering the explorations with fewer or more edges, we measure a noteworthy difference in the mean rewards between the PPO and the other algorithms. PPO agent, even with the variations, shows a lower score removing edges, while this is not the case for the other two agents, however the random agent performances are significantly lower. On the other hand, adding edges increases the variations for the random agent but the

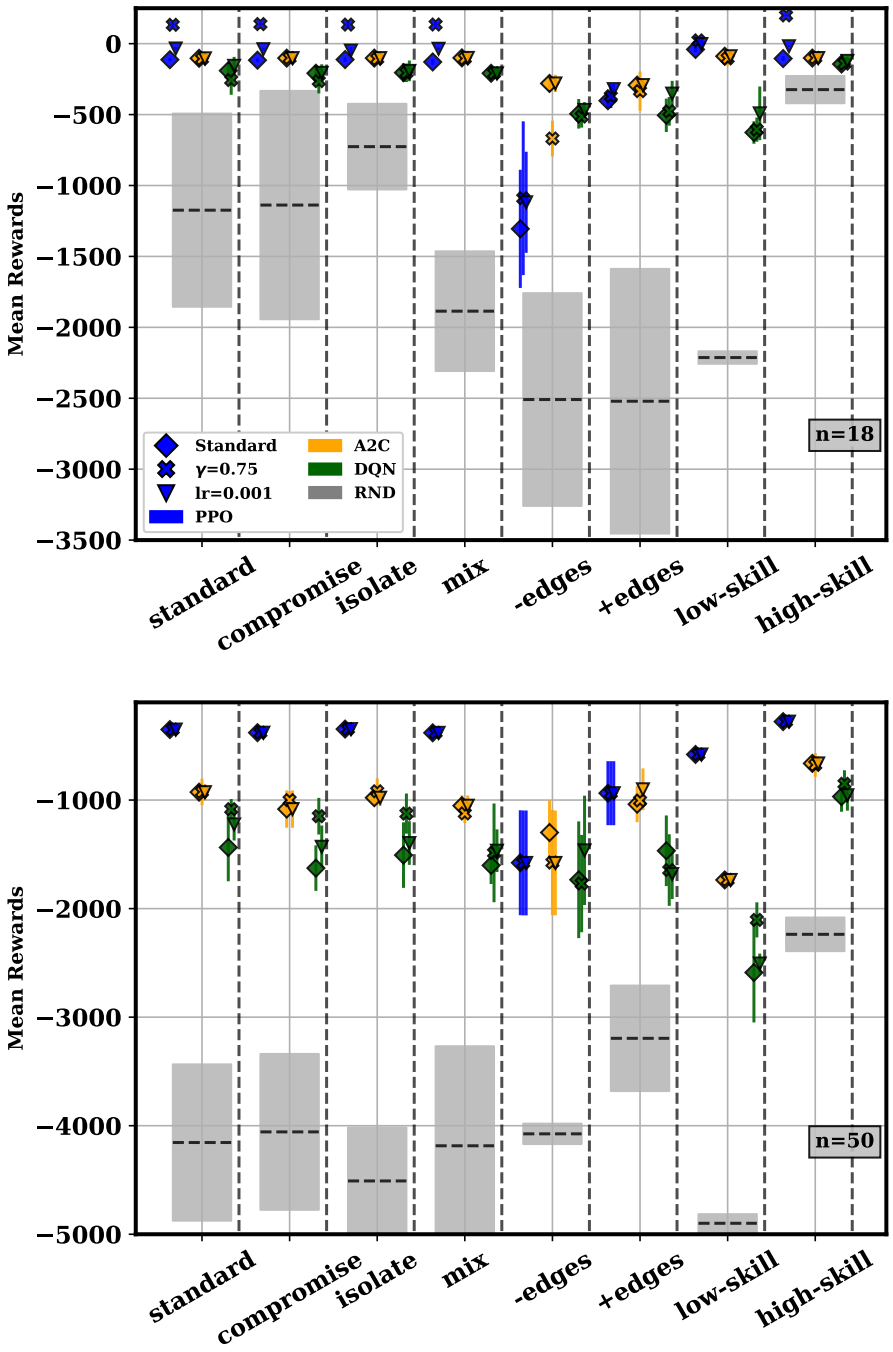

Figure 2: Mean rewards for the agents on the three networks (18 nodes top, 50 nodes bottom and 100 nodes in figure 3), in blue symbols we show the PPO results, A2C in orange and DQN in green. The three symbols show the different changes on the algorithm: the diamond is the standard version of the algorithm, the cross is using $\gamma = 0.75$ and the triangle is with $\mathrm{lr} = 0.001$. The random agent (RND) is shown in grey with the mean value as dashed line and the grey area shows 1σ deviation. The 1σ deviation on the agents scores is shown using y-errorbars. On the x-axis we show the various extension tested as adding compromised nodes and adding or removing edges.

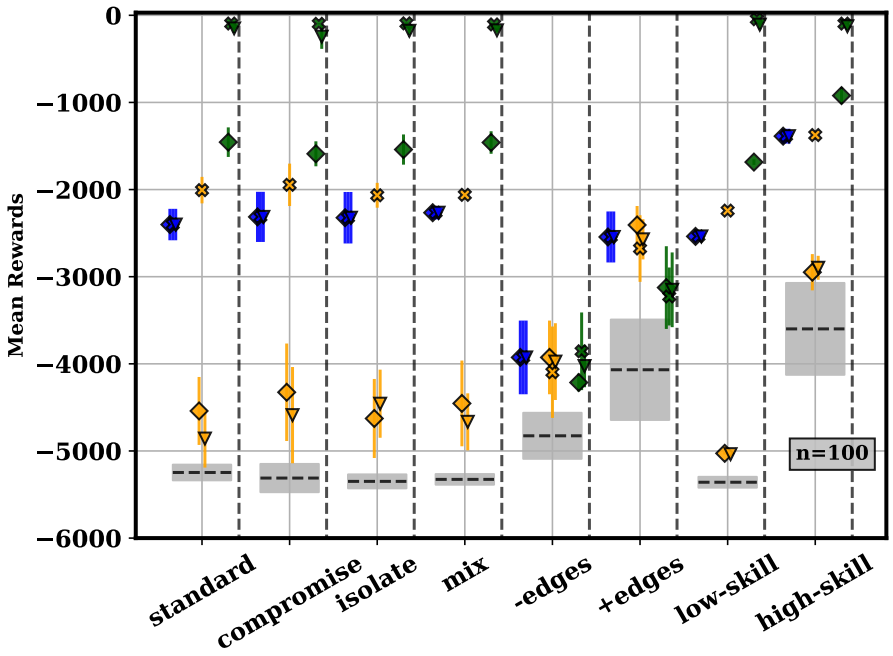

Figure 3: Same figure as 2, but for the 100 nodes case.

three algorithms behave similarly. A similar conclusion can be found in the instances of red agent skills: a less effective red agent (low skill) makes the game longer resulting in scores built on top of more expensive actions and fewer rewards from fixing the network. A more effective red agent is more aggressive and spreads quickly and the blue agent gains more points in fixing the nodes, which results in shorter game lengths because it is more difficult to stop and prevent the final escalation. Indeed, this is also verified by the small spread of the results from the random agent. In the case of a highly skilled red agent the performances of the random agent and the trained ones are really similar, even if still distinguishable.

In the 50 node scenario, bottom panel of figure 2, we see a significant difference between the various algorithms, PPO agent is always better performing compared to the others, being DQN agent the lowest among the three. Modifying the network result in similar spread of the scores even varying the parameters as γ and lr.

In 100 nodes case, figure 3, we see that A2C with lr modified has similar scores with PPO agents results in almost all experiments, while DQN obtains significantly different results, in particular by modifying γ. Interestingly, we do not see much difference when we modify the number of edges with all models scoring similar results. A2C agents are the worst performing, with results significantly lower than the other agents. Summarising the findings and analysis done in these cases we can say:

- for given algorithm (and hyper-parameter choice) adding isolated or compromised nodes, does not change significantly the performances in comparison to a clean starting network;
- the scores in larger networks are lower because the game length is generally higher, this

happens because the red agent is not able to overtake the network quickly, therefore the blue agent has more time to do expensive actions, while on smaller networks it is more rewarding fixing the nodes even if that results in loss;
- changing the network topology can trigger significant changes in the agents response. Adding more edges results in higher scores. However we measure similar performances between the agents and variations of hyper-parameters;
- the red agent skills has a significant impact on the performances but more importantly has the same reaction on the blue agent, we understand by noting small differences between all scores (small standard deviations).

Given this panoramic view of these results we can say that both PPO and A2C are performing well in small networks, in particular by changing the discount factor (γ), while adjusting the buffer size and γ in the DQN case conveys in performing well, and in a stable manner, in larger network cases.

3.3. Agent deployment on realistic networks

In the previous sections, we have explored the performances of RL algorithms trained and tested onto the same networks, without any resemblance to reality, trying to understand the best algorithm and check their resilience in the changes. In this subsection, we focus on a sample of cases using realistic networks configurations for testing the agents' behaviour. We consider three cases with 22, 55 and 60 nodes. We have extrapolated these networks from a portion of a larger existing network of computers considering only nodes connected, i.e. nodes that are connected between them but they do not share a connection with the core of the network are not considered. We deploy in the first instance an agent trained using an A2C algorithm, PPO with the standard setup for the 55 node network and DQN with $\gamma = 0.75$ in the latter. We train the agents on example networks with the same amount of nodes but with a different configuration (number of edges), and we evaluate the changes in performances in the realistic ones, which are effectively novel networks to the agents. In table 2, we summarise the network statistics for the synthetic networks used in training, the realistic environments in deployment and the algorithms selected. The average clustering measures how many connections are between the nodes: nodes connected with more edges have a value closer to 1. The triangles are a set of three nodes where each node has a relationship to the other two. These quantities can describe the complexity of the network, in particular it is clear that the realistic networks are much simpler (lower average clustering and number of triangles) in comparison to the synthetic ones.

In figure 4, we compare the performances of the trained agents in five different scenarios on the network used for training and the realistic one, we show the random agent performances for comparison. As before, we test the agents while modifying the network by adding isolated nodes, compromised, a mixture of those and against a red agent with lower and higher skills level. In this figure the y-axis is in log-scale for easier comparison. The goal of this analysis is to understand how an agent trained on a different network performs in a realistic network without re-training.

In the case of A2C , we can see that the scores in the training network are almost identical and, almost, two times higher in comparison to the ones obtained in the realistic scenario exploration. It is interesting to note that the scatter from the five realisations is small in the training networks,

Table 2
We present the synthetic networks used for training and the realistic networks used in deployment. The network statistics are the number of nodes, edges, high-value targets (HVT) and entry nodes. The algorithms used are: A2C and PPO with standard hyper-parameters and DQN with $\gamma = 0.75$ and buffer size = 10000. Finally, we present the average clustering value and the number of triangles present in the network: the average clustering measures how much the nodes are closer and connected with edges (nodes more connected have this measure closer to one), and the triangles defined as three nodes where each one has a relationship to the other two.

Mode	#Nodes	#Edges	#HVT	Entry nodes	Algorithm	Average clustering	#Triangles
Train	22	113	1	21	A2C	0.45	540
Deploy	22	21	1	21	A2C	0	0
Train	55	730	5	10	PPO	0.49	9500
Deploy	55	54	5	10	PPO	0	0
Train	60	901	4	12	DQN	0.5	13400
Deploy	60	62	4	12	DQN	0.045	6

while in the realistic ones there is a larger variance of results. This result maybe be connected to the peculiar shape of the realistic network and its relatively low number of routes the red agent can choose. The random agent performances are significantly lower in comparison to other agents.

In the network with 55 nodes we use the PPO algorithm: we measure values close to -4500 (close to the random agent performances) in the training network for most of the different scenarios, while in the real network the scores are around -1500, even in the case of varying the red agent's skill. We can conclude that the difference in the topology of the network has played a significant role in this analysis, overcoming as well, the impact of varying the red agents' skill, which has shown a larger effect in the previous analysis.

We have deployed the DQN algorithm in the final case considered. We find a similar behaviour we have seen in the 22 nodes case, with scores in the training network really close (around -300) one to the other and higher in comparison to the realistic scenario (close to -1000). In the realistic network we measure a larger variability in the final scores, again, we measure that the largest impact on the scores is due to the red agents' skills. The random agent scores are significantly lower in all tests.

In light of these results, we can state that the YT framework allows the training of RL agents and their deployment is transferable from synthetic to realistic networks with minimal loss of performance. In particular, we can validate that the agents' scores are lower in comparison to the ones obtained from agents trained and evaluated on the same networks, however, these are still much more improved from random scores. This analysis highlights that changing the networks' topology has not invalidated the performances of such agents.

4. Conclusion and future work

In this paper, we have performed training and evaluation of RL agents in a set of networks using Yawning Titan ACO capabilities, comparing how their performances change by modifying the

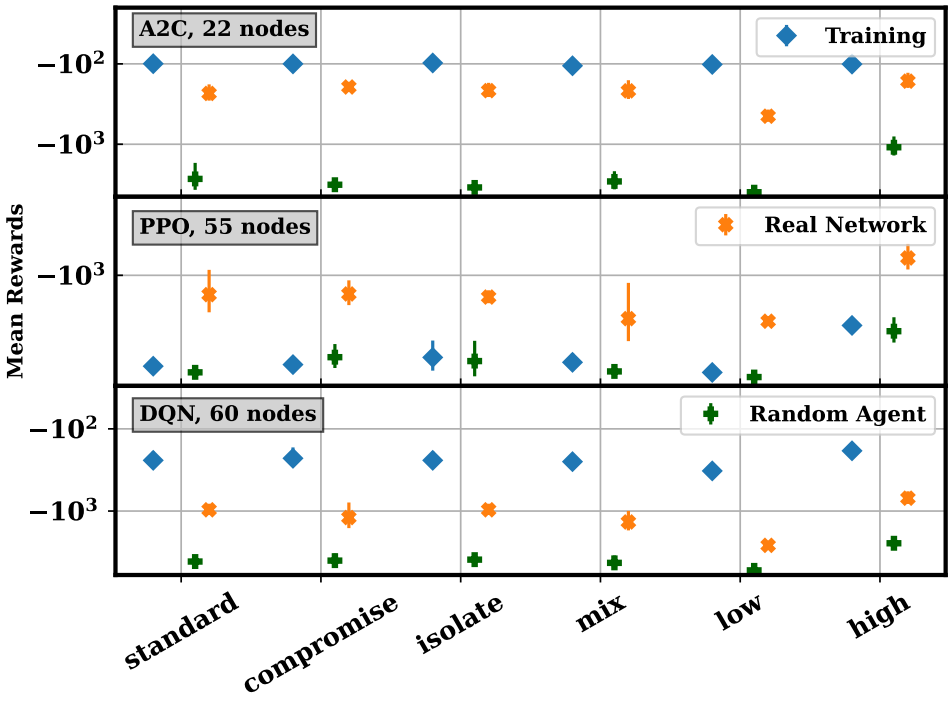

Figure 4: Agents' performance comparison between training networks and realistic networks. In the top panel we present the case with 22 nodes using A2C algorithm, in the central panel the 55 nodes network using PPO and finally, in the bottom panel we present the DQN agent applied to a 60 node network. The green crosses are the random agent scores, blue diamonds are the training scores and the realistic cases are in orange crosses. Please note that the y-axis, differently from the previous figures, is in log-scale for easier comparison.

status of the network and methods hyper-parameters looking for the best algorithm to deploy in realistic networks. The main findings are that by increasing the number of nodes, the mean reward per simulation is lower, highlighting a positive correlation with the action space dimension and varying the red agent's skills has the larger impact on the results.

We did not measure significant differences in the scores while modifying the status of the nodes (being compromised or isolated), on the other hand adding (or removing) edges in the network and augmenting (or reducing) the red agent skills showed interesting differences in the performances. We find that the RL algorithms considered can react well to network changes by measuring the level of performances across the various tests. By exploring the hyper-parameters tuning, the discount factor (γ) seems to have the most positive impact in the training and evaluation processes, in comparison to the limited results obtained by changing the learning rate (lr).

This work has shown and proved the possibility of using Yawning Titan in training agents that could be considered in realistic cyber-defence environments with minimal computational requirements. The tests we have carried out have shown that the changes in the agents' performances were arising from the different network topologies and not from changes in the network status itself. We have shown, as well, that it is possible to deploy an agent trained on a different topology

with minimal loss of performance, and in some cases (e.g., 50 nodes networks) we have measured an improvement in the mean scores. These results show the possibility to train intelligent agents in synthetic networks and deploy such agents in realistic networks without re-training. However, little exploration has been done in modifying or exploiting the current rewards of actions inside the simulations, exploring different winning setups (e.g., allowing the end game if the high-value target is taken) and more complex scenarios (e.g., more red agents, complex decision making) and other Markov decision process algorithms.

Extension of the current work can be exploring algorithms with proper hyper-parameter tuning, exploitation of the current reward scheme, offline learning methods and inclusion of multi-agent algorithms and time evolving networks.

Acknowledgments

The authors thank the reviewers for their useful comments that improved the quality of the paper and thank the useful discussions with Neil Dhir. This project was financially supported by a contract with the Alan Turing Institute.

References

[1] V. Krishna Viraja, P. Purandare, A Qualitative Research on the Impact and Challenges of Cybercrimes, in: Journal of Physics Conference Series, volume 1964 of *Journal of Physics Conference Series*, 2021, p. 042004. doi:10.1088/1742-6596/1964/4/042004.

[2] M. Crawford, T. M. Khoshgoftaar, J. D. Prusa, A. N. Richter, H. A. Najada, Survey of review spam detection using machine learning techniques, Journal of Big Data 2 (2015).

[3] L. Xiao, X. Wan, X. Lu, Y. Zhang, D. Wu, Iot security techniques based on machine learning: How do iot devices use ai to enhance security?, IEEE Signal Processing Magazine 35 (2018) 41–49. doi:10.1109/MSP.2018.2825478.

[4] A. L. Buczak, E. Guven, A survey of data mining and machine learning methods for cyber security intrusion detection, IEEE Communications Surveys & Tutorials 18 (2016) 1153–1176. doi:10.1109/COMST.2015.2494502.

[5] Y. Huang, L. Huang, Q. Zhu, Reinforcement learning for feedback-enabled cyber resilience, 2021. arXiv:2107.00783.

[6] S. Vyas, J. Hannay, A. Bolton, P. P. Burnap, Automated Cyber Defence: A Review, arXiv e-prints (2023) arXiv:2303.04926. doi:10.48550/arXiv.2303.04926. arXiv:2303.04926.

[7] W. Wang, D. Sun, F. Jiang, X. Chen, C. Zhu, Research and challenges of reinforcement learning in cyber defense decision-making for intranet security, Algorithms 15 (2022). URL: https://www.mdpi.com/1999-4893/15/4/134. doi:10.3390/a15040134.

[8] A. Burke, Robust artificial intelligence for active cyber defence, Alan Turing Insitute. (2017).

[9] R. Buettner, D. Sauter, J. Klopfer, J. Breitenbach, H. Baumgartl, A review of recent advances in machine learning approaches for cyber defense, in: 2021 IEEE International Conference

on Big Data (Big Data), 2021, pp. 3969–3974. doi:10.1109/BigData52589.2021.9671918.
[10] M. Standen, M. Lucas, D. Bowman, T. J. Richer, J. Kim, D. Marriott, Cyborg: A gym for the development of autonomous cyber agents, arXiv preprint arXiv:2108.09118 (2021).
[11] CAGE Challenge 1, arXiv, 2021.
[12] TTCP CAGE Challenge 2, 2022.
[13] T. C. W. Group, Ttcp cage challenge 3, https://github.com/cage-challenge/cage-challenge-3, 2022.
[14] A. Molina-Markham, C. Miniter, B. Powell, A. Ridley, Network environment design for autonomous cyberdefense, 2021. arXiv:2103.07583.
[15] A. Andrew, S. Spillard, J. Collyer, N. Dhir, Developing optimal causal cyber-defence agents via cyber security simulation, in: Workshop on Machine Learning for Cybersecurity (ML4Cyber), 2022.
[16] A. Raffin, A. Hill, A. Gleave, A. Kanervisto, M. Ernestus, N. Dormann, Stable-baselines3: Reliable reinforcement learning implementations, Journal of Machine Learning Research 22 (2021) 1–8. URL: http://jmlr.org/papers/v22/20-1364.html.
[17] J. Schulman, F. Wolski, P. Dhariwal, A. Radford, O. Klimov, Proximal policy optimization algorithms, arXiv preprint arXiv:1707.06347 (2017).
[18] V. Mnih, A. P. Badia, M. Mirza, A. Graves, T. Lillicrap, T. Harley, D. Silver, K. Kavukcuoglu, Asynchronous methods for deep reinforcement learning, in: International conference on machine learning, PMLR, 2016, pp. 1928–1937.
[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, M. Riedmiller, Playing atari with deep reinforcement learning, arXiv preprint arXiv:1312.5602 (2013).
[20] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, et al., Human-level control through deep reinforcement learning, nature 518 (2015) 529–533.
[21] B. Liu, Q. Cai, Z. Yang, Z. Wang, Neural proximal/trust region policy optimization attains globally optimal policy, arXiv preprint arXiv:1906.10306 (2019).
[22] M. Andrychowicz, A. Raichuk, P. Stańczyk, M. Orsini, S. Girgin, R. Marinier, L. Hussenot, M. Geist, O. Pietquin, M. Michalski, S. Gelly, O. Bachem, What Matters In On-Policy Reinforcement Learning? A Large-Scale Empirical Study, arXiv e-prints (2020) arXiv:2006.05990. doi:10.48550/arXiv.2006.05990. arXiv:2006.05990.