applied sciences

MDPI

Article

Cascaded Reinforcement Learning Agents for Large Action Spaces in Autonomous Penetration Testing

Khuong Tran [1,*,†], Maxwell Standen [2,†], Junae Kim [2,†], David Bowman [2,†], Toby Richer [2,†], Ashlesha Akella [1,†] and Chin-Teng Lin [1,*,†]

1 School of Computer Science, University of Technology Sydney, Ultimo, NSW 2007, Australia
2 Defence Science and Technology Group, Edinburgh, SA 5111, Australia
* Correspondence: khuong.tran@uts.edu.au (K.T.); chin-teng.lin@uts.edu.au (C.-T.L.)
† These authors contributed equally to this work.

Abstract: Organised attacks on a computer system to test existing defences, i.e., penetration testing, have been used extensively to evaluate network security. However, penetration testing is a time-consuming process. Additionally, establishing a strategy that resembles a real cyber-attack typically requires in-depth knowledge of the cybersecurity domain. This paper presents a novel architecture, named deep cascaded reinforcement learning agents, or CRLA, that addresses large discrete action spaces in an autonomous penetration testing simulator, where the number of actions exponentially increases with the complexity of the designed cybersecurity network. Employing an algebraic action decomposition strategy, CRLA is shown to find the optimal attack policy in scenarios with large action spaces faster and more stably than a conventional deep Q-learning agent, which is commonly used as a method for applying artificial intelligence to autonomous penetration testing.

Keywords: cybersecurity; penetration testing; deep reinforcement learning; large action space

check for updates

Citation: Tran, K.; Standen, M.; Kim, J.; Bowman, D.; Richer, T.; Akella, A.; Lin, C.-T. Cascaded Reinforcement Learning Agents for Large Action Spaces in Autonomous Penetration Testing. *Appl. Sci.* **2022**, *12*, 11265. https://doi.org/10.3390/app122111265

Academic Editors: Luis Javier Garcia Villalba, Ana Lucila Sandoval Orozco, Rafael T. de Sousa Jr. and Robson de Oliveira Albuquerque

Received: 30 September 2022
Accepted: 1 November 2022
Published: 7 November 2022

1. Introduction

Developing effective autonomous defences typically requires that models are trained against sophisticated attacks. Penetration testing has been used extensively to evaluate the security of ICT systems. However, this is a time-consuming process. Additionally, establishing a strategy that resembles a real cyber-attack requires in-depth knowledge of the cybersecurity domain. Moreover, the currently available automatic penetration testing tools are not capable of learning new strategies to respond to simple defensive measures. So, instead of testing all possible values when hacking into a system, it is often more desirable to use an autonomous penetration-testing tool that includes artificial intelligence so as to simulate a real attacker with a real attack strategy.

Reinforcement learning is a learning paradigm in which an autonomous decision-making agent explores and exploits the dynamics of its environment and, in so doing, discovers a suitable policy for acting in that environment. This makes reinforcement learning a suitable candidate for tackling the problem of automating the penetration-testing process. Deep reinforcement learning with function approximators as neural networks have achieved multiple state-of-the-art results on a variety of platforms and with a range of applications, as demonstrated by Mnih et al. [1] and Lillicrap et al. [2]. Value-based methods, such as some variants of deep Q-learning (DQN) algorithms [3], have become the standard in many approaches thanks to their simplicity and effectiveness in learning from experienced trajectories [4–6]. However, one of the limitations of applying DQN in practical applications is the complexity of the action space in different problem domains. DQN requires an evaluation for all the actions at the output of the policy network and uses a maximisation operation on the entire action space to select the best action to take in every step. This is problematic for large action spaces as multiple actions may share similar estimated values [7]. Penetration testing is one such practical application where the action space scales exponentially with the number of host computers in different connected sub-networks. Applying conventional deep reinforcement learning to automate penetration testing would be difficult and unstable as the action space can explode to thousands of executable actions even in relatively small scenarios [8].

There are studies that address the intractable problem of having large discrete action spaces. For example, Tavakoli et al. [9] proposed an action branching architecture where actions are grouped into sub-spaces and jointly trained by different network heads. Prior domain knowledge can also be used to embed or cluster similar actions in the action space, which effectively reduces the cardinality of distinct action representations [10]. These investigations inspired us to develop a new action decomposition scheme for devising penetration-testing attack strategies that is efficient, versatile, and works with different network configurations. The proposed method leverages the multi-agent reinforcement learning (MARL) paradigm to decompose the action space into smaller subsets of a space where a typical reinforcement learning agent is capable of learning.

As such, in this paper, we propose a novel architecture to directly address the large discrete action spaces of autonomous penetration testers. Using an adapted binary segment tree data structure, the action space is algebraically decomposed into sub-branches, each of which is an order of magnitude smaller. A combined learning mechanism is then used to cooperatively train the agents in the cascaded structure. The proposed algorithms are tested on CybORG, which is an autonomous cybersecurity simulator platform developed by Standen et al. [11] designed to empirically validate the learning and convergence properties of an architecture. CRLA's performance demonstrates better and more stable learning curves in comparison to a conventional DQN at the same network size.

This paper is organised as follows: Section 2 reviews the literature on reinforcement learning as it applies to penetration testing as well as current investigations of reinforcement learning that can deal with large action spaces. Section 3 provides a background to reinforcement learning and its application to penetration-testing problem formulation. Section 4 explains the details of the proposed algorithm, and Section 5 describes the experiments undertaken for the paper. Finally, Section 6 presents the results and discussion on the current investigation as well as considering future research directions.

2. Related Work

Reinforcement learning has been used as a method of both devising and modelling exploit attacks in cybersecurity settings [8,12,13]. In these studies, penetration testing is formulated as a partially observable Markov decision process (POMDP), where reinforcement learning or deep reinforcement learning is used to model the environment and learn the exploit strategy. Here, the attacker agent can only observe the compromised hosts and/or subnets, not the entirety of the network connections. The agents need to develop a suitable strategy (policy) to gain access to the hidden assets by learning from trial-and-error experiences. These previous studies have shown that both tabular reinforcement learning and deep reinforcement learning are possible methods for solving small-scale networks or for capturing flag challenges, both of which have relatively small action spaces. However, extending deep reinforcement learning to networks with a larger action space remains a challenge, not only in cybersecurity settings but also in the reinforcement learning context [8].

An architecture was proposed by Dulac-Arnold et al. [10], named Wolpertinger, that leverages an actor-critic framework for reinforcement learning. This framework attempts to learn a policy within a large action space problem that has sub-linear complexity. It uses prior domain knowledge to embed the action space and clusters similar actions using the nearest-neighbour method to narrow down the most probable action selection. However, in a sparse reward environment, this method can be unstable as the gradient cannot propagate back to enhance learning of the actor network.

In another study, carried out by Zahavy et al. [14], the use of an elimination signal was proposed to teach the agents to ignore irrelevant actions, which reduces the number of possible actions to choose from. A rule-based system calculates the action elimination signal by evaluating actions and assigning negative points to irrelevant actions. Then, the signal is sent to the agent as part of an auxiliary reward signal to help the agent learn better. This framework is best suited to domains where a rule-based component can easily be built to embed expert knowledge and facilitate the agent's learning by reducing the size of the action set.

In Tavakoli et al. [9] different network branches were incorporated to handle different action dimensions, while a growing action space was trained using curriculum learning to avoid overwhelming the agents [7]. The authors of Chandak et al. [15] focused on learning action representations. Here, the agent was able to infer similar actions using a single representation. In a similar vein, in de Wiele et al. [16], a solution was proposed targeted at learning action distributions. All these approaches sought to find a way to reduce a large number of actions. However, they cannot be used to solve automating penetration testing problems for several reasons. First, the action space in penetration testing is entirely discrete and without continuous parameters. This is different from other areas of testing where the action space is parameterised and comprised of a few major actions that may contain different continuous parameters. Second, each action in a penetration test can have a very different effect, such as attacking hosts in different subnets or executing different methods of exploits. This is again different in nature to a large, discretised action space, where a group of actions can have spatial or temporal correlations [9,10,16].

To the best of our knowledge, this investigation is the first to propose a non-conventional deep reinforcement learning architecture to directly target the large action space problem in penetration-testing settings. The architecture involves an explicit action decomposition scheme based on a structure of cascaded reinforcement learning agents. Here, the agents are grouped hierarchically and each agent is trained to learn within its action subset using the same external reward signal. This is a model-free methodology where no domain knowledge is required to decompose the set of actions. Further, the architecture is proven to scale efficiently to more complex problems. The solution also takes advantage of value-based reinforcement learning to reduce the sample complexity, and, with some modifications, it can be extended to other problem domains where the action space can be flattened into a large discrete set.

3. Background

The problem to be considered in this paper is a discrete-time reinforcement learning task modelled by a Markov decision process or MDP formalised by a tuple $\langle \mathcal{S}, \mathcal{U}, \mathcal{P}, \mathcal{R}, \bigcirc \rangle$. Unlike previous investigations, we do not wish to formulate the penetration testing as a POMDP because we want to clearly demonstrate the effects of the proposed architecture without compounding factors. No previous studies have used any POMDP-specific solutions, and, further, our method can still be applied as a feature to a POMDP solution. At each time step t, the agent receives a state observation $s_t \in \mathcal{S}$ from the environment. In our penetration testing setting, this represents the observable part of the compromised network. To interact with the environment, the agent executes an action $u_t \in \mathcal{U}$, after which it receives another state s_{t+1} according to the state transition probability function $\mathcal{P} : \mathcal{S} \times \mathcal{U} \times \mathcal{S} \to \mathbb{R}$, which returns a probability distribution of the next state s_{t+1} given the current state s_t and action u_t, and a reward signal r_t given by $\mathcal{R} : \mathcal{S} \times \mathcal{U} \to \mathbb{R}$. The agent's aim is to maximise its utility, defined as the total sum of all rewards gained in an episode $R_t = \sum_{\tau=t}^{\infty} \gamma^{\tau-t} r_\tau$, where $\gamma \in [0,1]$ is the discount factor used to determine the importance of long-term rewards.

In MARL, the action space $\mathcal{U}$ is constructed from the action space $\mathcal{A}^i$ of each individual agent. Normally u is a vector comprised of all the individual actions of k agents $u = (a^1, a^2, a^3, ..., a^k)$ [17]. In this paper, u will be used to refer to the final action which will be exercised on the environment and a^i is the action generated by the agent i, which is dubbed as an action component of u.

One common approach in deep reinforcement learning is to use a value-based method, such as deep q-learning or its variants, to learn a (near) optimal policy $\pi : \mathcal{S} \to \mathcal{A}$ which maps the observed states or observations to actions. The state-action value function, as defined below, tells each agent i the expected utility or discounted future return of taking the action a_t^i in the state s_t:

$$\begin{aligned} Q^{\pi,i}(s,a^i) &= \mathbb{E}[R_t|s_t = s, a_t = a^i] \\ &= \mathbb{E}_{s'}[r + \gamma\, \mathbb{E}_{a^{i'} \sim \pi^i(s')}[Q^{\pi,i}(s', a^{i'})]|s_t = s, a_t = a^i] \end{aligned} \tag{1}$$

By recursively solving Equation (1) for all state action pairs, and taking the action with the highest Q-value in the next state, the optimal action value function, which is defined as $Q^*(s,a) = \max_\pi Q_\pi(s,a)$, can be derived. This optimal solution is shown in the Equation (2):

$$Q^{i*}(s,a^i) = \mathbb{E}_{s'}[r + \gamma \max_{a^{i'}} Q^{i*}(s', a^{i'})|s, a^i] \tag{2}$$

In non-trivial problems where there are large numbers of states and actions, $Q(s,a)$ is often approximated by parameterised functions, such as neural networks (e.g., $Q(s,a;\theta)$), in which similar states are estimated with similar utility.

Generally, DQNs or reinforcement learning algorithms are well-known for becoming unstable as the action space becomes larger, for example, of the order of hundreds or thousands of actions [16]. As a result, the performance of these algorithms degrades as the action space increases [10]. With reinforcement learning, the action space will have a different format and representation depending on the type of problem being tackled. Typically, these action spaces can be classified into two main categories: continuous and discrete. In a discrete action problem, the action space is most commonly constructed as a flat action space with each primitive action identified as one integer. Alternatively, the action space might be parameterised into a hierarchically-structured action space with a few main types of actions and sub-actions underneath [18]. It is harder, however, to learn in a parameterised action space. This is because the agent needs to learn two Q-functions or policies—one to choose the main action and another to choose the sub-action or the parameters of each action. On the other hand, converting a hierarchically structured action space into a flat representation could result in a substantially larger sized action space, which also hinders learning with DQN agents. In this paper, we focus on a large and discrete flat action space. Additionally, we propose a multi-agent learning framework where the agents are grouped into a hierarchical structure.

In a flat and discrete setting, the action space is encoded into integer identifiers ranging from LOW_INT to MAX_INT—for instance, from 0 to 999 in an action space of 1000 actions. Each number is translated into a valid primitive action that is executable by the simulator. In our investigation, we do not assume any prior knowledge of the semantics or relationships amongst these individual actions. The main idea of the proposed solution is to decompose the overall action space $\mathcal{U}$ into smaller sets of integers $\mathcal{A}^1, \mathcal{A}^2, \ldots, \mathcal{A}^L$, where $|\mathcal{A}^i| << |\mathcal{U}|$ ($\forall i \in [1, L]$). A primitive action u_t in $\mathcal{U}$ can be formed as a function over the action components ${a_t}^i$ in those smaller sets: $u_t = f(a_t^1, a_t^2, \ldots, a_t^L)$, with ${a_t}^i \in \mathcal{A}^i$. Intuitively, the integer identifiers of the final actions, which can be large in value, are built up algebraically using smaller integer values. The growing numerical representation of the action space is illustrated in Figure 1. In the next section, we explain the intuition behind this architecture in detail, along with how to define the function as well as the smaller sets of action identifiers.

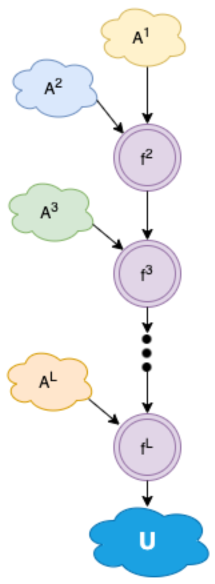


Figure 1. Action space composition. The final action space $\mathcal{U}$ is constructed by chaining multiple action component's subspaces $\mathcal{A}^i$.

4. Methodology

4.1. Algebraic Action Decomposition Scheme

The main contribution of this paper is to propose an action decomposition strategy to reduce large discrete action spaces into manageable subsets for deep reinforcement learning. Our solution also includes a cooperative training algorithm that stabilises the learning process and supports better convergence properties.

As mentioned in the previous section, a large action space U is decomposed into L action component subspaces. However, rather than using different neural network heads to manage these subsets, as proposed by Tavakoli et al. [9], separate DQN agents are used instead. The benefits of this approach are highlighted in the next section.

The idea of decomposing a large action space into smaller action component sets is similar to having a tree structure with multiple layers of action selections. Each node in the tree acts like an action component selection node in that it picks a corresponding component from one of its child nodes. Each node has its own range of identifiers. In order to reach a leaf node with a specific action value, the parent node needs to pick the right children to reach a specified leaf. Figure 2, which is an adaptation from a structured parameterised action space in [19], illustrates the idea of having different action selection layers. Inspired from a segment tree data structure, each non-leaf node in the tree contains a range of identifiers that are reachable from that node. All nodes in the tree share the same branching factor, which is considered to be a hyper-parameter of the model. Heuristics are used to calculate the total number of nodes and the number of levels, as shown in Equation (3), where b is the branching factor. The levels are constructed by filling in the nodes from top to bottom, one level at a time, noting that the final tree should be a complete tree.

$$|T| = \log_b(|\mathcal{U}|) \tag{3}$$

Instead of having a different number of agents in each level to represent ranges of integer values, a linear algebraic function shifts the action component identifier values into appropriate ranges.

$$a^{i+1}_{out} = f(a^i_{out}, a^{i+1}) = a^i_{out} \times \beta^{i+1} + a^{i+1} \tag{4}$$

In Equation (4), the coefficient β^{i+1} represents the number of action component selection nodes or agents at the level $(i+1)$th. The value for β^{i+1} shifts the integer values of the action components to particular output ranges. This results in having only one agent per level, as shown in Figure 3. This action decomposition scheme shows that, instead of having a single reinforcement learning agent explore and learn an entire action space of $|\mathcal{U}|$, which can be extremely large, a few smaller reinforcement learning agents can be used to explore and learn in a much smaller version of the action component subspace. The outputs of these smaller agents are then chained together to produce the final action. The complexity of having multiple agents on different levels is similar to traversing a tree in computer science. The number of levels L is of the order of $L \approx \log_b |\mathcal{U}|$. As a result, only a handful of agents are required to construct an action space even if there are millions of possible actions.

To understand the intuition behind the proposed action composition approach, let us structure those small action component sets into a hierarchy, as shown in Figure 2. Each circle in the illustration represents an agent and the arrows are the possible action components. The three smaller sets of action components, $\mathcal{A}^1$, $\mathcal{A}^2$ and $\mathcal{A}^3$, are chosen and, for simplicity, each has the same action size of 3. The green highlighted path represents the sequence of action components needed to produce the primitive action 21 which is executable by the environment. The branching factor b and the number of levels L are 3 in this example. There are two things worth noting here: First, to reconstruct the original action space $|\mathcal{U}| = 27$, each agent in the hierarchy only needs a small action size of 3. Second, as illustrated in the figure, we need $1 + 3 + 9 = 13$ agents in total to fully specify the 27 actions. However, we do not have to implement all 13 agents. In fact, we only need three agents, one for each level of the action component selection tree.

If the first-level agent picks the action component number 2, and the second-level agent picks the action component number 1, the actual action component identifier created after the second level is $2 * 3 + 1 = 7$, where 3 is the size of the action space of each agent in the second level. Similarly, the final primitive action identifier is $7 * 3 + 0 = 21$, where 0 is the action component taken by the corresponding agent at the third level. To generalise this into a compact formula, let us consider the action component signals a^i and a^{i+1} output by the agents at levels i and $i + 1$, where $0 <= a^i <= |\mathcal{A}^i| - 1$ and $0 <= a^{i+1} <= |\mathcal{A}^{i+1}| - 1$. The action component identifier constructed up to level $i + 1$ is $a^i \times |\mathcal{A}^{i+1}| + a^{i+1}$. The executable primitive action is $u = a^{L-1} \times |\mathcal{A}^L| + a^L$, where L is the number of levels in the action hierarchy.

The design of the small individual action component sets $\mathcal{A}^1, \ldots, \mathcal{A}^L$, as well as selecting the value of L, have to be performed manually as no domain knowledge is incorporated into structuring these action component sets. One rule of thumb is to limit the size of each set to be relatively small, say, 10 to 15 actions. The value of L is then selected accordingly to reconstruct the original action set $\mathcal{U}$. Each agent in the hierarchy can be implemented using any conventional reinforcement learning algorithm, for example, Dueling DQN [4], which is the core reinforcement learning algorithm we have used in this study. This is our Cascaded Reinforcement Learning Agent architecture, or CRLA for short. An operational diagram of the CRLA architecture is depicted in Figure 3.

Agents in the hierarchy can be trained in parallel and independently by optimising their own loss functions (Equation (5)). This involves sampling a batch size of p transitions from the replay buffer, where θ^i are the parameters of the agent i's network and θ^{-i} are the parameters of the target network. The latter parameters are used to stabilise the training, as mentioned in the original DQN paper [1].

$$L^i(\theta^i) = \sum_{j=1}^{p} [(y_j^{i,DDQN} - Q^i(s, a^i; \theta^i))^2] \tag{5}$$

where

$$y^{i,DDQN} = r_t + \gamma \max_{a'^i} Q(s', a'^i; \theta^{-i}) \tag{6}$$

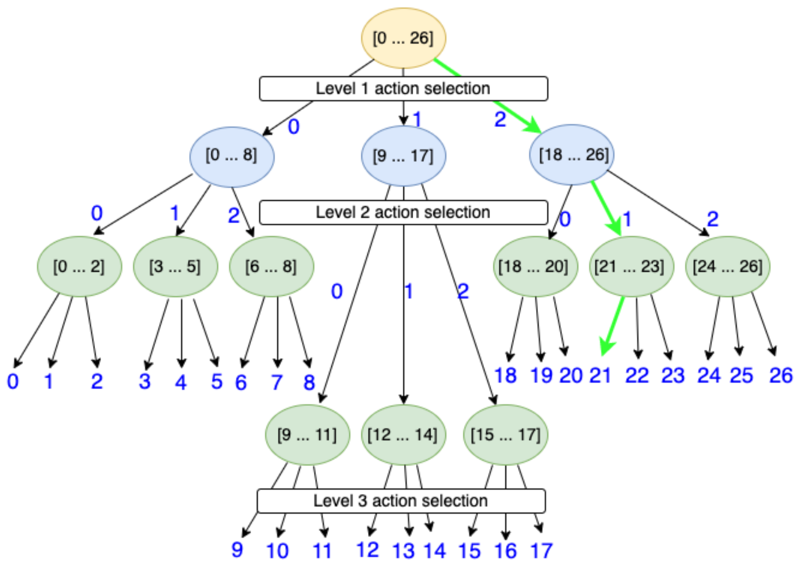


Figure 2. An illustration of a tree-based structure for hierarchical action selection. The primitive action identifiers are located on the leaf nodes. Each internal node contains the action range of its children.

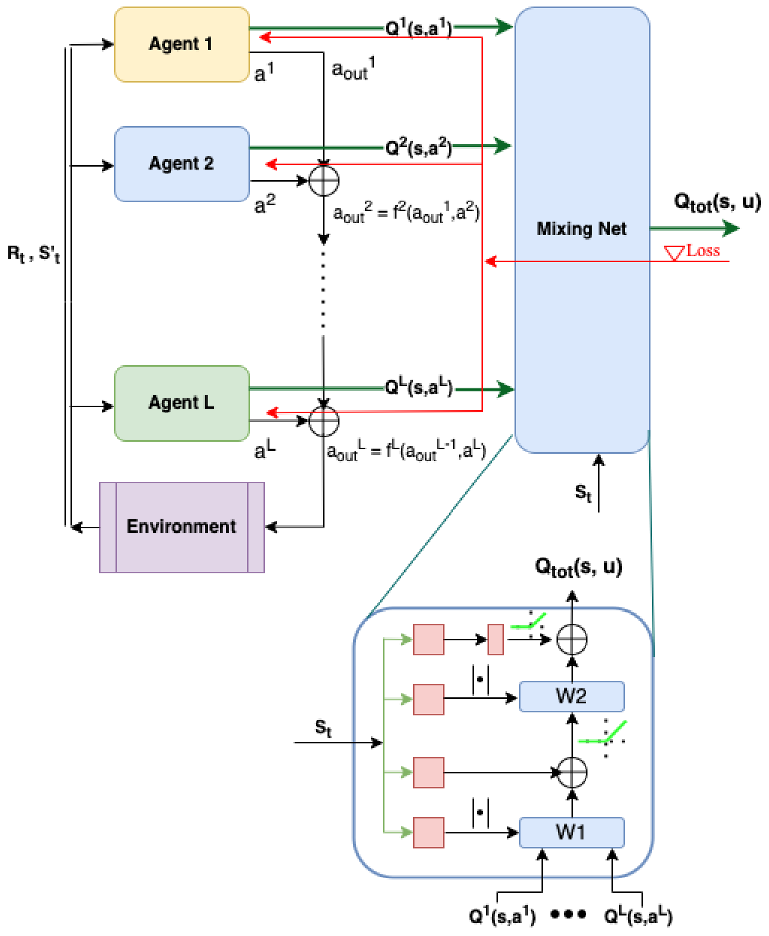


Figure 3. The operational diagram of the proposed CRLA architecture. DRL agents are grouped into a cascaded structure of *L* levels. The state-action values of all agents are fed into a MixingNet implemented as a hyper-network (at bottom of the figure) to optimise the agents altogether. The paths of the gradient are shown as red arrows.

4.2. Cooperative Multi-Agent Training

The multi-agent training approach discussed in the previous section resembles the popular independent Q-learning (IQL) method [20] in which independent agents are trained simultaneously in the same environment. Despite its simplicity, IQL performs surprisingly well in practice and is usually a good benchmark against more advanced MARL algorithms [21]. The major drawback of IQL is that it does not consider non-stationarity in the policies of individual agents, nor do the agents take each other's policies or actions into account when updating their own policy. As a result, changes in observation are not the direct result of each agent's behaviour, and, hence, there is no convergence guarantee even with infinite exploration.

To provide a sound theoretical guarantee of convergence with CRLA, we adapted a well-known multi-agent training network called QMIX [17]. QMIX trains an individual agent's network properly while taking the changing policies of the other agents into consideration. QMIX is shown in the right half of Figure 3 under the name Mixing Net. Generally, QMIX enables the training of a joint action value function by aggregating all the individual functions and optimising a combined action value function $Q_{tot}(s, u)$ with $u = f(a^1, a^2, \ldots, a^L)$.

We had two reasons for choosing QMIX. First, we needed a coordination mechanism that would allow individual agents to account for changes to its own update as a result of another agent's policy. QMIX provides an end-to-end learning scheme that takes in all the values of individual action components (e.g., $Q^1(s, a^1), \ldots, Q^L(s, a^L)$), plus the state information, in order to learn an optimal $Q^*_{tot}(s, u)$. It does this by minimising the following loss function:

$$L(\theta) = \sum_{j=1}^{p} [(y_j^{tot} - Q_{tot}(s_t, u; \theta)^2] \tag{7}$$

and $y^{tot} = r_t + \gamma \max_{u'} Q(s', u'; \theta^-)$

Different functions and network architectures can then be used to construct Q_{tot} [22]. We tried to construct Q_{tot} through a summation function and a feed-forward neural network but, eventually, we selected a two-layered hyper-network to output the weight of the Mixing Net as this yielded a better result experimentally (Equation (8)):

$$Q_{tot}(s_t, u; \theta) = g(Q^1(s, a^1), \ldots, Q^L(s, a^L); \theta_{mixer}) \tag{8}$$

This hyper-network was proposed by Ha et al. [23] and used in Rashid et al. [17]. It has the advantages of using a small number of parameters $W1$ and $W2$ to output the weights for a larger network, in this case the Mixing Net. As a result, it helps to avoid a linear time complexity when scaling up to more agents or inputs. In previous studies on QMIX, the Mixing Net takes in a set of state-action values from homogenous agents as well as the environment's true states. This is different from our setting since each agent in the hierarchy can have a variable number of action components and we do not have access to the environment states. With respect to implementation, we had to concatenate h time steps of observations as the inputs to the Mixing Net. We also leveraged the vectorised computing feature in Pytorch [24] to efficiently adapt to the heterogenous input values. During back-propagation, the learning signal is distributed to each agent, which means they can consider other agents' policies when updating their own parameters to achieve a global optimum. This behaviour would not be possible without the use of this coordinating network.

The second reason for using QMIX is that its parameters ($W1$ and $W2$ in Figure 3) are learned using a two-layer hyper-network with a non-negative activation function. This extra function is used to constrain the parameters of the Mixing Net to be positive, allowing QMIX to represent any factorisable and non-linear monotonic function $\frac{\partial Q_{tot}}{\partial Q^i} >= 0$ for all agents i. This results in an equality between taking the argmax for Q_{tot} and taking the argmax for each individual Q^i (Equation (9)):

$$\underset{\mathbf{u}}{\operatorname{argmax}}\, Q_{tot}(\tau, \mathbf{u}) = \begin{pmatrix} \operatorname{argmax}_{a^1} Q_1(\tau, a^1) \\ \vdots \\ \operatorname{argmax}_{a^n} Q_n(\tau, a^n) \end{pmatrix} \tag{9}$$

In summary, if all agents learn their own optimal policy cooperatively, the entire architecture can derive the optimal global policy.

4.3. CRLA Implementation

Even though CRLA is inspired by a MARL framework, there are distinct features that help CRLA learn efficiently during training. First, all agents share the same state vector and reward signal. The only difference between the agents is the action it selects. Consequently, the multi-agent training and action selection can be configured to run in parallel instead of serially. All agents can also share a replay buffer containing the tuple (s, u, r, s', d), where the action u is a vector comprising all the individual action components a^i.

With an appropriate configuration for the multi-agent DQN training, and the use of a vectorisable programming framework, such as Pytorch [24], all agents can be trained and used for inference in parallel. Moreover, the training is optimised to be faster than for a serial multi-agent approach. In addition, since the tuple of information is mostly the same for all agents (except for the outputted action component of each agent, as shown in Figure 4), this is also true for the use of the replay buffer, where a single replay buffer is used for all agents.

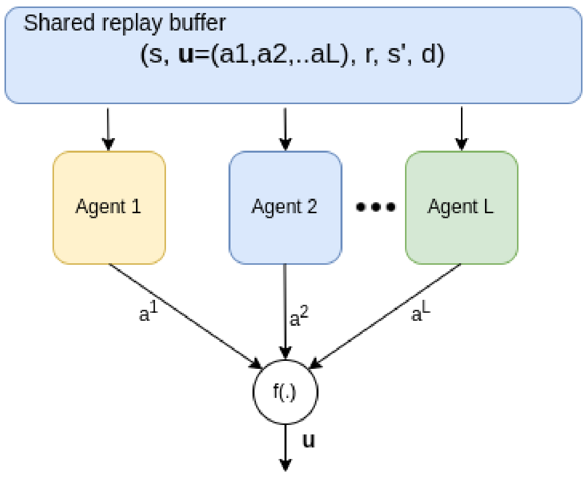


Figure 4. Parallel training and execution. All agents share the replay buffer from which experiences can be sampled in batches and used for training in parallel.

4.4. Algorithms Pseudocode

The pseudocode for the main CRLA algorithm is given in Algorithm 1 and the sub-routine for initialising the agents given the environment action space $\mathcal{U}$ and a branching factor b is provided in Algorithm 2.

Algorithm 1 CRLA—Cascaded Reinforcement Learning Agents

```
1:  Initialise a shared replay buffer $\mathcal{D}$
2:  Initialise a list of agent $\mathbf{A}$ = initialise_agents(env)
3:  Initialise QMIXER($\theta$) and QMIXER_target($\theta$)
4:  for i_episode = 0 ... MAX_EPISODES do
5:      Initialise an empty set of compromised hosts $\mathbf{H} = \emptyset$
6:      Reset the environment $S_0$ = env.reset()
7:      $s = S_0$
8:      for step = 0 ... MAX_STEPS do
9:          for agent in agent_list do
10:             $a^i = \epsilon$_greedy($s, \epsilon^i$)
11:         end for
12:         Construct final primitive action: $\mathbf{u} = f(a^0, a^1, a^2, ..., a^k)$
13:         Collect $(s, u, r, s', d)$ by executing action $\mathbf{u}$ on the environment
14:         Perform reward shaping and update $\mathbf{H}$ if needed:
15:                 $r$ = reward_function(state, $\mathbf{H}$, $r$)
16:         Store $(s, \mathbf{u}, r, s', d)$ into $\mathcal{D}$
17:         Sample a batch of $(s, \mathbf{u}, r, s', d)$ from $\mathcal{D}$ for learning
18:         Compute q-value for each agent $\mathbf{q}^i(s, a^i)$
19:         Compute target q-value for each agent $\mathbf{q}^{target-i}(s', a'^i)$
20:         $Q(s, \mathbf{u})$ = concat($\mathbf{q}^i(s, a^i)$) from all agents
21:         $Q^{target}(s, \mathbf{u})$ = concat($\mathbf{q}^{target-i}(s', ai^i)$) from all agents
22:         Compute $\mathbf{Q}_{tot}(s, \mathbf{u})$ = QMIXER($Q(s, \mathbf{u})$, $s$)
23:         Compute $\mathbf{Q}_{tot}^{target}(s, \mathbf{u})$ = QMIXER($Q^{target}(s, \mathbf{u})$, $s$)
24:         Compute TD target $y^{tot} = r + \gamma \mathbf{Q}_{tot}^{target}(s, \mathbf{u})$
25:         Perform SGD to minimise $L(\theta) = MSE(\mathbf{Q}_{tot}(s, \mathbf{u}), y^{tot})$
26:     end for
27: end for
```

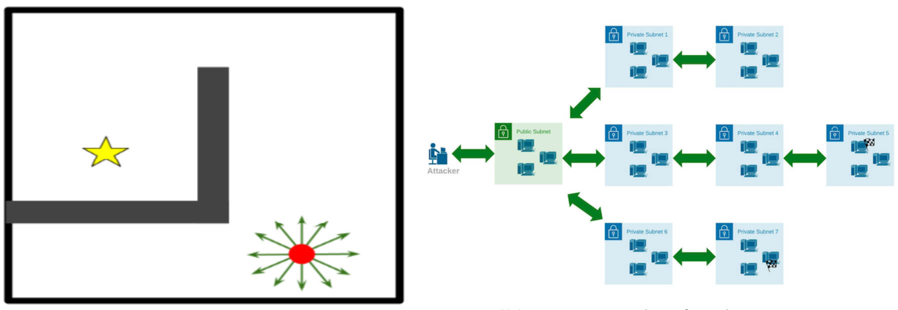
Algorithm 2 Initialise the agents

```
1: def initialise_agents(env: Environment, b: branching factor)
2: begin
3: $|U|$ = env.action_space
4: $|T| = \log_b(|U|)$
5: Build a complete tree with branching factor b
6: Initialise a Dueling DQN agent for each action selection level
7: Return agent_list
8: end
```

5. Experiment Setup

5.1. Toy Maze Scenario

As a proof of concept to verify the performance of the proposed algorithm, we set up an experiment with a toy-maze scenario. An agent, marked as the red dot in Figure 5a, tries to control some actuators to move itself toward the target—the yellow star. The agent can essentially move continuously in any direction. However, the actuators' output is discretised into a variable number of action outputs to make this a discrete action environment. The size of the action set is $|\mathcal{U}| = 2^n$, where n is the number of actuators. These are represented by green arrows pointing in the direction of movement displacement. The agent receives a small negative reward for every step in the maze and, if it can get to the target within 150 steps, it receives a reward of 100. The environment was adapted from Chandak et al. [15] to create different testing scenarios with a variable number of actions. The fact that the agent does not get any positive reward unless it reaches the target within the pre-specified number of steps makes this a challenging scenario with sparse rewards.



(**a**) Maze scenario

(**b**) An example of 24 hosts scenario

Figure 5. Two simulated scenarios. (**a**) A toy-maze-based scenario. (**b**) A CybORG scenario of 24 hosts.

5.2. The CybORG Simulator

To validate the performance of CRLA in a more practical scenario, we used an autonomous penetration testing simulator as our testbed. The authors of Standen et al. [11] developed CybORG to provide an experimental environment for conducting AI research in a cybersecurity context. The testbed is designed to enable an autonomous agent to conduct a penetration test against a network. Apart from being open source and having the appropriate OpenAI gym interface for applying different deep reinforcement learning algorithms, this environment represents a real application where we are faced with different aspects of complexity. The sparsity of reward signals, a large discrete action space, and a discrete state representation mean we cannot rely on any computer vision techniques to facilitate learning. Details of CybORG's complexity and how we executed the experiments follow.

The simulation provides an interface for the agent to interact with the network. A vector based on the observed state of the network is also provided to the agent. Through this interface, the agent performs actions that reveal information about the simulated network and they successfully penetrate the network by capturing flags on the hosts within it.

The simulation can implement various scenarios. These scenarios detail the topology of the target network, the locations of the flags, and specific information about the hosts, such as the operating system and services the host is running. The target network features a series of subnets that contain hosts. The hosts in a subnet may only act upon hosts in the same subnet or adjacent subnets. An illustrative representation of a possible scenario configuration with 24 hosts and one attacker agent is presented in Figure 5b. In this example scenario, there are two hidden flags: one on a host in subnet 5 and one in subnet 7. There are three hosts in each of the subnets. The scenarios can differ in terms of the number of hidden flags and the locations of the hosts that contain the flags, along with the number of hosts. This determines the complexity of the state and action spaces. In the 24-host example, 550 actions are available to an attacking agent. The simulation includes three action types: host-to-host actions, host-to-subnet actions, and on-host actions. Each action may be performed on a host, and each host-to-host action or host-to-subnet action may target another host or subnet, respectively.

In each scenario, p hosts are grouped into q subnets. If a simulation supports m types of host-to-host actions, n types of host-to-subnet actions, and o types of on-host actions, the action space for the host-to-host actions is $m \times p \times (p-1)$, the action space for the host-to-subnet actions is $n \times p \times q$, and the action space for the on-host actions is $o \times p$. This leads to a large potential action space, even in a small network.

The success of the actions depends on both the state of the simulation and the probability of success. The state of the simulation consists of various types of information, such as the operating system on each host and the services that are listening on each host. The agent sees an unknown value for each value it does not know. As the agent takes actions in the simulation, its awareness of the true state of the simulation increases. An action fails if the agent does not have remote access to a target host or subnet. However, an action also has a probability of failing even if the agent has access to the target. This probability reflects the reality that attempts to exploit vulnerabilities are seldom, if ever, 100% reliable.

Our simulation was deliberately configured to pose a highly sparse environment. Agents began the game with no knowledge of the flags' locations, so finding the hosts with the flags or locating the hidden assets was completely random. A large positive reward was given for capturing each hidden flag, while a small positive reward was given for every host that was successfully attacked. The flags were hidden from the state space; hence, the agent could not determine the location of the flags by observing the state vector. This simulation differs significantly from the typical reinforcement learning environments in which either the objective or the target is visible to the agent.

5.3. Neural Network Architecture

A diagram of the neural network's architecture is shown in Appendix A with a comparison between a single-agent DDQN and the proposed CLRA. Given a scenario with four agents, the size of each layer in CLRA is one-fourth of the corresponding layer in the single-agent architecture. The sizes were selected after a fine-tuning process wherein the networks were able to converge stably using the minimum number of neurons. The action space of each DDQN agent in CRLA was varied from 8 to 12. As a result, there were at most four DDQN agents created for scenarios of up to 5000 actions. Due to the running complexity of the CybORG simulator, the algorithm's performance was tested with scenarios of up to 100 hosts and a maximum action space of 4646 actions. All the experiments were conducted on a single RTX6000 GPU.

6. Results

6.1. Maze

Figure 6 presents a performance comparison of policy training between a single-agent DDQN and the proposed CRLA architecture. The tested scenario had an action size of 4096 with $n = 12$ actuators. The results were reproduced over multiple random seeds with the shaded regions showing the variances between runs. The left panel shows the cumulative scores of the agent as the training proceeded, while the right panel shows the number of steps required to reach the goal. Ideally, the cumulative scores would increase towards a maximum reward at the end of the training and the number of steps would reduce to a stable value.

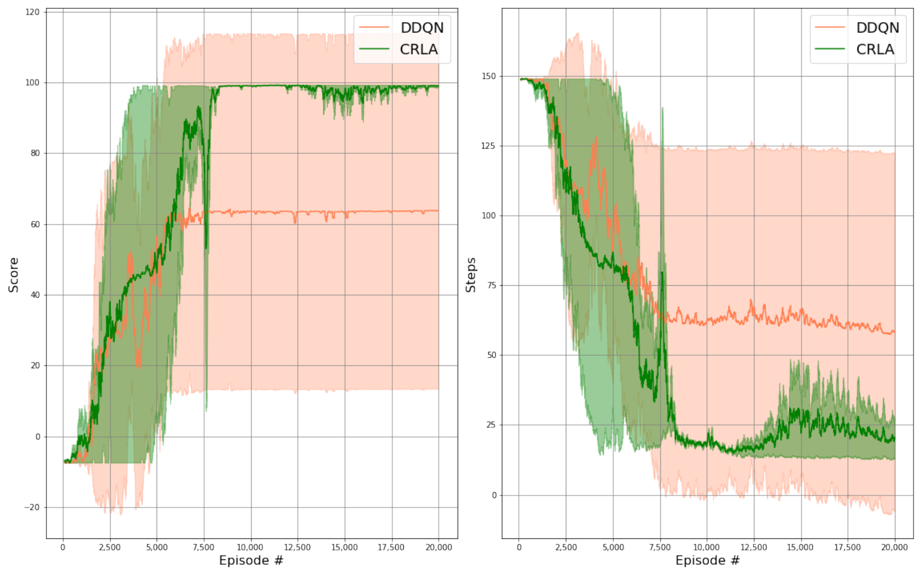


Figure 6. Results of CRLA and single-agent DDQN on a maze scenario with 4096 actions. Left panel: the cumulative scores throughout the training. Right panel: the required number of steps to capture the flag throughout the training.

6.2. CybORG

We tested the proposed algorithm in multiple CybORG scenarios with different numbers of hosts to see how well the CRLA learned given action spaces of increasing size. Each performance in Figure 7 was repeated five times with different random seeds to ensure reproducibility. We examined two indicators to verify performance: the maximum score attainable by the agents (shown in the left panel) and the number of steps to capture the flag (shown in the right panel). For each scenario, the maximum score the agent could receive was approximately 20 points (minus some small negative rewards for invalid actions taken to reach the assets).

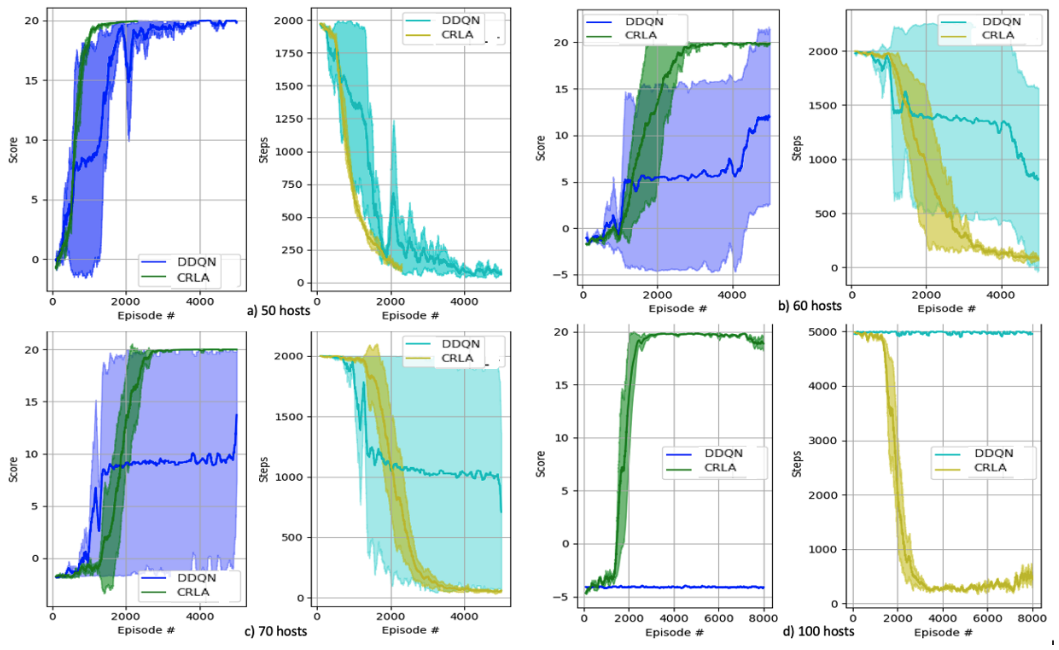


Figure 7. Results of CRLA and single DDQN on different CybORG scenarios. Sub-figures from left to right, top to bottom: (**a**) 50-hosts scenario, (**b**) 60-hosts scenario, (**c**) 70-hosts scenario, (**d**) 100-hosts scenario.

We tested the CRLA algorithm in a variety of scenarios with different configurations of hosts and action spaces, as shown in Table 1. In terms of scenario complexity, we varied the number of hosts from 6 to 100 and increased the action space from 49 to 4646 actions. However, we only added two more agents to train. In all the scenarios tested, CRLA's convergence results were either similar to or superior to the DDQN agent, with the dependent factors being the complexity of the scenario and the size of the action space.

Figure 7 presents the algorithms' performance on four scenarios where the complexity was significant enough to showcase the superiority of the algorithm while not being too computationally expensive for repeated training. In scenarios where both DDQN and CRLA were able to learn the optimal policy, CRLA showed faster and more stable convergence than DDQN. DDQN's performance on scenarios with 60 and 70 hosts was unstable as it only successfully learned the optimal policy in one out of four runs. In contrast, CRLA held up its performance on the scenario with 100 hosts where DDQN failed to achieve any progress during training.

For the 100 hosts scenario, DDQN was not able to explore the action space at all, nor could it learn the policy, while CRLA took approximately 4000 episodes to grasp the scenario and begin to converge on the optimal policy. Further, the policy learned by CRLA in each of the tested scenarios was optimal in terms of having a minimum number of actions taken.

Table 1. Configurations of tested scenarios.

Hosts	State Space	Action Space	Number of Agents
6	39	49	2
9	55	120	2
12	71	182	2
18	217	342	2
24	285	550	2
50	573	1326	3
60	685	1830	4
70	797	2414	4
100	1133	4646	4

This simulation serves as the first demonstration of applying deep reinforcement learning to an automatic penetration-testing scenario with such a large action space [25].

6.3. Cooperative Learning with QMIX

We verified CRLA's learning, with and without the use of QMIX, by testing the two variants on the 60-host scenario. It can be seen from Figure 8 that CRLA with QMIX demonstrated slightly better stability with a smaller variance between runs in comparison to the independent CRLA without QMIX. The cooperative training using QMIX gives a better theoretical guarantee of convergence. However, implementing this empirically requires extra hyper-parameter tuning for the hyper-network.

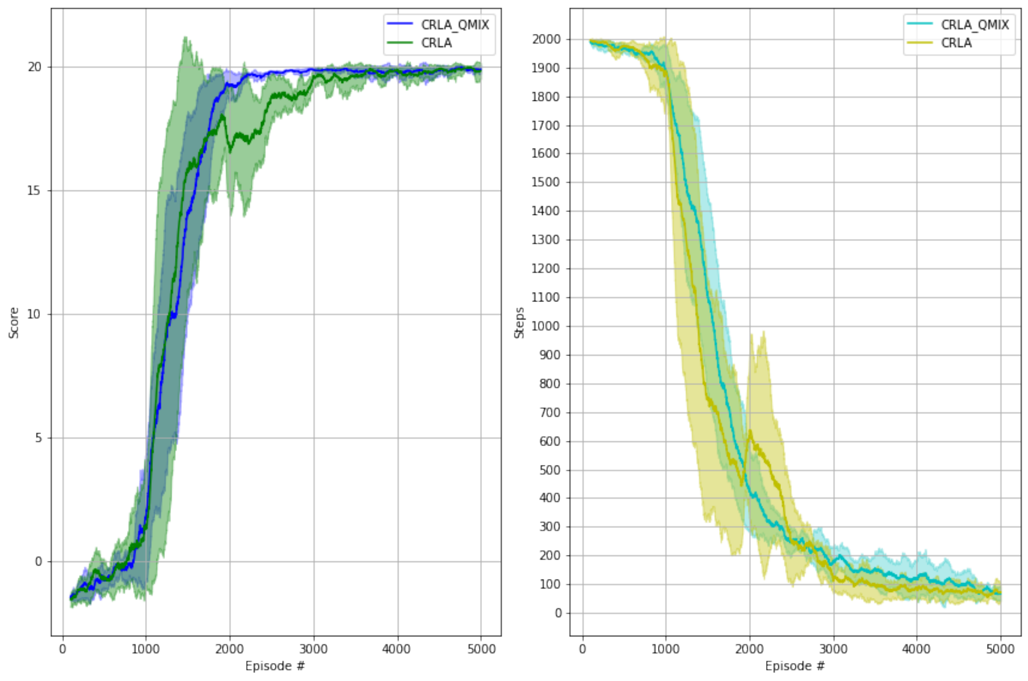


Figure 8. Performances of CRLA with and without QMIX on the 60-hosts scenario.

6.4. Discussion

To examine how well each agent in CRLA learned, we looked into the state and action representations that the trained agents learned by visualising the state representations using the t-SNE method [26]. Figure 9 shows the results. As can be seen, the state representations for the 50-host scenario were quite similar across the three agents. Surprisingly, the learned state representations were mapped into nine separate clusters, which matched the nine private subnets in the 50-host configuration. This configuration itself had one public subnet and nine private subnets, each with five hosts, even though this knowledge was not provided or observable by the agents. The coloured markers represent the identifier of the action component each agent would take in certain states. The action space of 1326 was mapped into smaller subsets of 10 to 15 action components per agent.

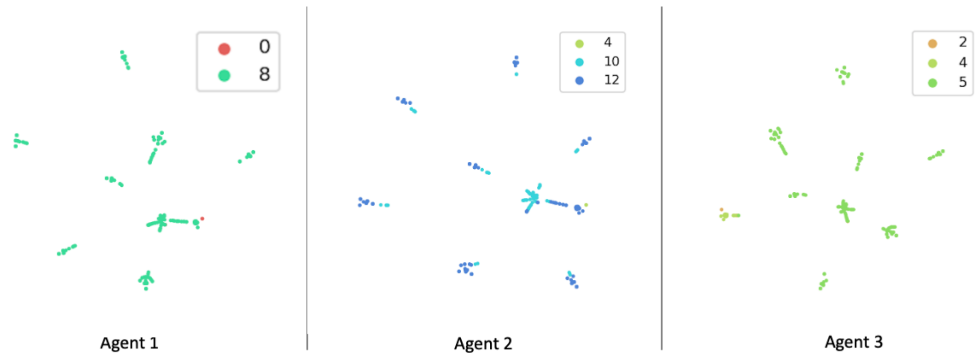


Figure 9. State representation from the 3 agents of the 50-hosts scenario. The clusters are coloured based on the action components.

After training, each agent learned that only two to three action components in its own subsets were needed to optimally capture the hidden assets. This visualisation opens up new research directions where further examination of the state and action representations might yield better progress in applying deep reinforcement learning to even more complex penetration-testing scenarios.

7. Conclusions

This paper introduced a new cascaded agent reinforcement learning architecture called CRLA to tackle penetration-testing scenarios with large discrete action spaces. The proposed algorithms require minimum prior knowledge of the problem domain while providing competitively better performance than conventional DQN agents. We validated the algorithms on simulated scenarios from CybORG. In all tested scenarios, CRLA showed superior performance to a single DDQN agent.

The proposed algorithm is also scalable to larger action spaces with a sub-linear increase in network computational complexity. This is because the individual networks are trained and used for inference separately and their outcomes are used sequentially to compute the final action. Despite CRLA's promising results, various challenges remain to be resolved. Future work will extend CRLA to incorporate the learning of sub-goals. This should make CRLA perform better in environments where the rewards are much sparser. Additionally, efficient exploration in large action spaces is still an outstanding problem that deserves further research.

Author Contributions: Conceptualization and methodology, K.T. and A.A.; software, K.T. and M.S.; formal analysis, K.T., J.K., M.S., D.B., T.R., A.A. and C.-T.L.; supervision, J.K., T.R. and C.-T.L.; writing—review and editing, K.T., M.S., J.K., D.B., T.R., A.A. and C.-T.L. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Australian Defence Science Technology Group (DSTG) under Agreement No: MyIP 10699.

Conflicts of Interest: The authors declare no conflict of interest.

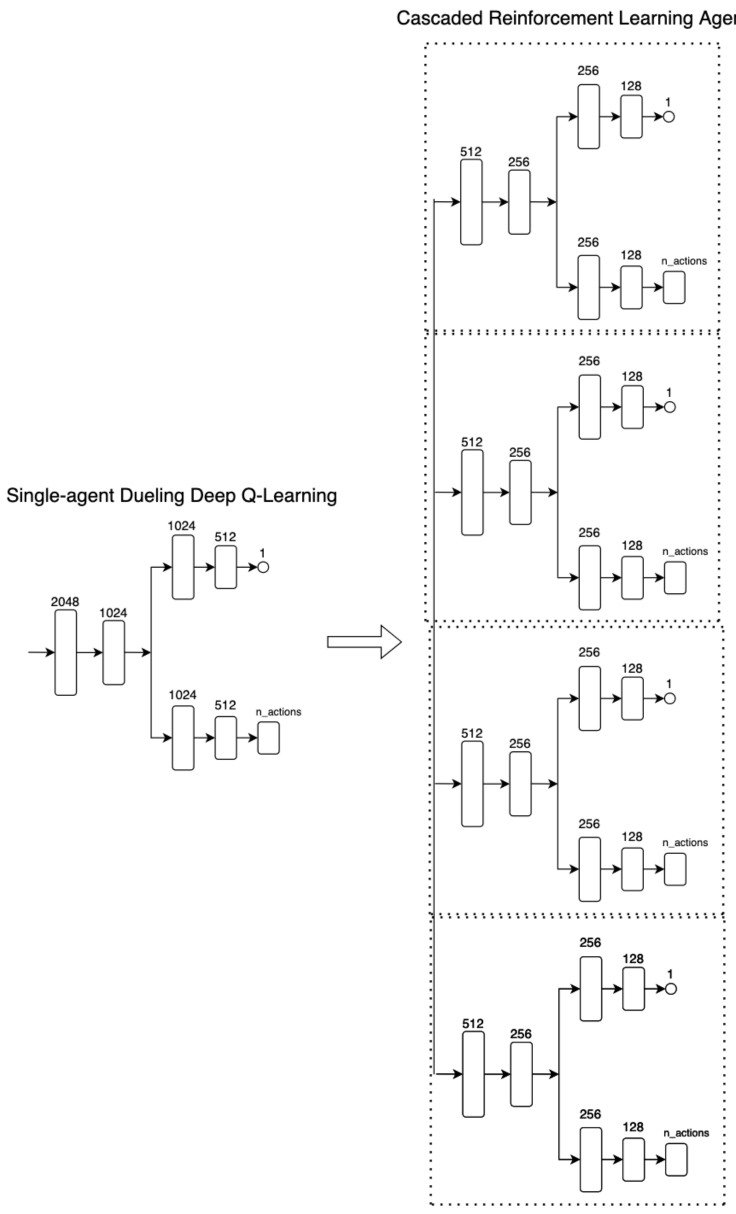
Abbreviations

The following abbreviations are used in this manuscript:

CRLA	Cascaded reinforcement learning agents
MDP	Markov decision process
POMDP	Partially observable MDP
RL	Reinforcement learning
CybORG	Cyber Operations Research Gym
DQN	Deep Q-network
GPU	Graphical processing unit

Appendix A. Network Comparison

Figure A1 shows a comparison between the network architectures of a single-agent Dueling DQN and the CRLA network.



Single-agent Dueling Deep Q-Learning
2048
1024
1024
512
1
1024
512
n_actions
Cascaded Reinforcement Learning Agents
512
256
256
128
1
256
128
n_actions

Figure A1. Neural network architectures comparison between the single-agent duelling deep Q-learning and the proposed cascaded reinforcement learning agents approach.

References

1. Mnih, V.; Kavukcuoglu, K.; Silver, D.; Rusu, A.A.; Veness, J.; Bellemare, M.G.; Graves, A.; Riedmiller, M.; Fidjeland, A.K.; Ostrovski, G.; et al. Human-level control through deep reinforcement learning. *Nature* **2015**, *518*, 529. [CrossRef] [PubMed]
2. Lillicrap, T.P.; Hunt, J.J.; Pritzel, A.; Heess, N.; Erez, T.; Tassa, Y.; Silver, D.; Wierstra, D. Continuous control with deep reinforcement learning. *arXiv* **2015**, arXiv:1509.02971.
3. Watkins, C.J.; Dayan, P. Q-learning. *Mach. Learn.* **1992**, *8*, 279–292. [CrossRef]
4. Wang, Z.; Schaul, T.; Hessel, M.; Van Hasselt, H.; Lanctot, M.; De Freitas, N. Dueling network architectures for deep reinforcement learning. *arXiv* **2015**, arXiv:1511.06581.
5. Hessel, M.; Modayil, J.; van Hasselt, H.; Schaul, T.; Ostrovski, G.; Dabney, W.; Horgan, D.; Piot, B.; Azar, M.G.; Silver, D. Rainbow: Combining Improvements in Deep Reinforcement Learning. In Proceedings of the Thirty-second AAAI conference on artificial intelligence 2018, New Orleans, LA, USA, 2–7 February 2018.
6. Fan, J.; Wang, Z.; Xie, Y.; Yang, Z. A Theoretical Analysis of Deep Q-Learning. In *Proceedings of Machine Learning Research, Proceedings of the 2nd Conference on Learning for Dynamics and Control, Berkeley, CA, USA, 11–12 June 2020*; Bayen, A.M., Jadbabaie, A., Pappas, G., Parrilo, P.A., Recht, B., Tomlin, C., Zeilinger, M., Eds.; ML Research Press: Maastricht, The Netherlands, 2020; Volume 120, pp. 486–489.
7. Farquhar, G.; Gustafson, L.; Lin, Z.; Whiteson, S.; Usunier, N.; Synnaeve, G. Growing Action Spaces. In Proceedings of the International Conference on Machine Learning 2020, Virtual, 24–26 November 2020.
8. Schwartz, J.; Kurniawati, H. Autonomous Penetration Testing using Reinforcement Learning. *arXiv* **2019**, arXiv:cs.CR/1905.05965.
9. Tavakoli, A.; Pardo, F.; Kormushev, P. Action branching architectures for deep reinforcement learning. In Proceedings of the Thirty-Second AAAI Conference on Artificial Intelligence, New Orleans, LA, USA, 2–7 February 2018.
10. Dulac-Arnold, G.; Evans, R.; van Hasselt, H.; Sunehag, P.; Lillicrap, T.; Hunt, J.; Mann, T.; Weber, T.; Degris, T.; Coppin, B. Deep reinforcement learning in large discrete action spaces. *arXiv* **2015**, arXiv:1512.07679.
11. Standen, M.; Lucas, M.; Bowman, D.; Richer, T.J.; Kim, J.; Marriott, D. CybORG: A Gym for the Development of Autonomous Cyber Agents. *arXiv* **2021**, arXiv:2108.09118.
12. Hu, Z.; Beuran, R.; Tan, Y. Automated Penetration Testing Using Deep Reinforcement Learning. In Proceedings of the 2020 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW), Genoa, Italy, 7–11 September 2020; pp. 2–10.
13. Zennaro, F.M.; Erdodi, L. Modeling penetration testing with reinforcement learning using capture-the-flag challenges and tabular Q-learning. *arXiv* **2020**, arXiv:2005.12632.
14. Zahavy, T.; Haroush, M.; Merlis, N.; Mankowitz, D.J.; Mannor, S. Learn What Not to Learn: Action Elimination with Deep Reinforcement Learning. In Proceedings of the Advances in Neural Information Processing Systems, Montreal, QC, Canada, 3–8 December 2018.
15. Chandak, Y.; Theocharous, G.; Kostas, J.; Jordan, S.M.; Thomas, P.S. Learning Action Representations for Reinforcement Learning. In Proceedings of the International Conference on Machine Learning 2019, Long Beach, CA, USA, 10–15 June 2019.
16. de Wiele, T.V.; Warde-Farley, D.; Mnih, A.; Mnih, V. Q-Learning in enormous action spaces via amortized approximate maximization. *arXiv* **2020**, arXiv:cs.LG/2001.08116.
17. Rashid, T.; Samvelyan, M.; de Witt, C.S.; Farquhar, G.; Foerster, J.N.; Whiteson, S. QMIX: Monotonic Value Function Factorisation for Deep Multi-Agent Reinforcement Learning. In Proceedings of the International Conference on Machine Learning 2018, Stockholm, Sweden, 10–15 July 2018.
18. Masson, W.; Konidaris, G.D. Reinforcement Learning with Parameterized Actions. In Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence 2016, Phoenix, AZ, USA, 12–17 February 2016.
19. Fan, Z.; Su, R.; Zhang, W.; Yu, Y. Hybrid Actor-Critic Reinforcement Learning in Parameterized Action Space. *arXiv* **2019**, arXiv:1903.01344.
20. Tan, M. Multi-Agent Reinforcement Learning: Independent versus Cooperative Agents. In Proceedings of the Tenth International Conference on Machine Learning (ICML 1993), Amherst, MA, USA, 27–29 July 1993; pp. 330–337.
21. Leibo, J.Z.; Zambaldi, V.F.; Lanctot, M.; Marecki, J.; Graepel, T. Multi-agent Reinforcement Learning in Sequential Social Dilemmas. *arXiv* **2017**, arXiv:1702.03037.
22. Sunehag, P.; Lever, G.; Gruslys, A.; Czarnecki, W.M.; Zambaldi, V.F.; Jaderberg, M.; Lanctot, M.; Sonnerat, N.; Leibo, J.Z.; Tuyls, K.; et al. Value-Decomposition Networks For Cooperative Multi-Agent Learning. *arXiv* **2017**, arXiv:1706.05296.
23. Ha, D.; Dai, A.M.; Le, Q.V. HyperNetworks. *arXiv* **2016**, arXiv:1609.09106.
24. Paszke, A.; Gross, S.; Chintala, S.; Chanan, G.; Yang, E.; DeVito, Z.; Lin, Z.; Desmaison, A.; Antiga, L.; Lerer, A. Automatic differentiation in PyTorch. In Proceedings of the NIPS 2017 Autodiff Workshop, Long Beach, CA, USA, 9 December 2017.
25. Ghanem, M.C.; Chen, T.M. Reinforcement Learning for Efficient Network Penetration Testing. *Information* **2020**, *11*, 6. [CrossRef]
26. Van der Maaten, L.; Hinton, G. Visualizing data using t-SNE. *J. Mach. Learn. Res.* **2008**, *9*, 2579–2605.