SoK: A Comparison of Autonomous Penetration Testing Agents

Raphael Simon
Royal Military Academy
Brussels, Belgium
r.simon@cylab.be

Wim Mees
Royal Military Academy
Brussels, Belgium
w.mees@cylab.be

ABSTRACT

In the still growing field of cyber security, machine learning methods have largely been employed for detection tasks. Only a small portion revolves around offensive capabilities. Through the rise of Deep Reinforcement Learning, agents have also emerged with the goal of actively assessing the security of systems by the means of penetration testing. Thus learning the usage of different tools to emulate humans. In this paper we present an overview, and comparison of different autonomous penetration testing agents found within the literature. Various agents have been proposed, making use of distinct methods, but several factors such as modelling of the environment and scenarios, different algorithms, and the difference in chosen methods themselves, make it difficult to draw conclusions on the current state and performance of those agents. This comparison also lets us identify research challenges that present a major limiting factor, such as handling large action spaces, partial observability, defining the right reward structure, and learning in a real-world scenario.

CCS CONCEPTS

• **General and reference** → **Surveys and overviews**; • **Security and privacy** → *Network security*; • **Computing methodologies** → *Reinforcement learning*.

KEYWORDS

Penetration Testing, Deep Reinforcement Learning, Security Automation, Reinforcement Learning

ACM Reference Format:
Raphael Simon and Wim Mees. 2024. SoK: A Comparison of Autonomous Penetration Testing Agents. In *The 19th International Conference on Availability, Reliability and Security (ARES 2024), July 30–August 02, 2024, Vienna, Austria.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3664476.3664484

1 INTRODUCTION

The world is as connected as it has ever been, and we have long become dependent on it. There have never been more computer systems deployed, that run our critical infrastructures such as health, finance, transportation, energy etc. At the same time, cyber-related threats continue to grow, posing enormous risks, especially for critical infrastructures. Defending those systems is a massive and difficult task considering the general unfairness of the playing field. Defenders have to cover every possible hole while attackers often only need a single entry. The amount of AI-enabled technologies has also been increasing. Cyber security is no exception to this, and an extensive amount of methods have been proposed [31] to help with the detection of attacks.

With the rise of Deep Reinforcement Learning (DRL), especially since agents were able to show human-like performances playing Atari games [18], there have been some attempts at using it within the domain of cyber security as well. Some of the usages include cyber-physical systems and intrusion detection for instance [21]. In this work we, focus on agents which fall into the field of penetration testing (or pentesting). A penetration test is an authorized, simulated attack on a system, network, or application, with the goal of identifying vulnerabilities and weaknesses, enabling organizations to strengthen their security posture.

The promise of such agents is that they can assist human pentesters, helping the expert to focus on more complex tasks, rather than wasting time on simple and repetitive ones. Another possibility could be to deploy the agent in a network to simulate an attack with the goal of generating events for the Security Operations Team (SOC), and help them train specific scenarios and how to handle them. Finally, the logs that are created by the interaction of an agent with the network could be used for the creation of datasets to further refine Machine Learning (ML) detections.

In this paper we present and give a comparative overview of the current autonomous penetration testing agents. We found that the greatest challenges are not just restricted to this field, but rather to deploying real-world agents in general. The motivations for this work are that a good amount of methods have been proposed, while also being distinct from one another. Comparing existing work helps to showcase the approaches that do better than others, while also highlighting shortcomings.

The structure of this paper is as follows: In Section 2 we establish some of the necessary background for (Deep) Reinforcement Learning. Section 3 gives an overview of the current approaches, divided into two parts: the proposed execution environments for the agents, and the agents themselves. Afterwards, in Section 4 we elaborate on the different challenges that are encountered in this field and look at how the current work tries to handle these challenges. Finally, Section 5 presents the future research directions we have identified.

2 BACKGROUND

2.1 Reinforcement Learning

Reinforcement Learning (RL) is the problem faced by an agent that learns behaviour through trial-and-error interactions with a dynamic environment [13]. The learning happens similarly as it would for animals or us humans, through gaining *reinforcements*, which

Publication rights licensed to ACM. ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s).
ARES 2024, July 30–August 02, 2024, Vienna, Austria
© 2024 Copyright held by the owner/author(s). Publication rights licensed to ACM.
ACM ISBN 979-8-4007-1718-5/24/07
https://doi.org/10.1145/3664476.3664484

here, refers to rewards. RL is based on the *reward hypothesis*: Any goal can be formalized as the outcome of maximizing cumulative reward [33]. The agent's job is to find the optimal policy (π), that maximises the expected reward (r). It does so by executing actions (a) on the environment and using the rewards as a measure to explore which actions are more desirable in which states (s). Since the environment is dynamic, executing actions results in new states, and thus we have an interaction loop.
Actions are chosen according to a *policy*. It defines the agent's way of behaving at a given time [33]. Most often, we use a stochastic policy $\pi(a_t|s_t) = P(a_t|s_t)$. To measure the quality of our policy, we can use value functions. The state-value function is defined as: $v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s]$ where $G_t = \sum_{t=0}^{T} \gamma r_t$ and γ denotes the discount factor. Next to the state-value function, we also have the state-action value function, sometimes also referred to as the Q-function, because it can be thought of as representing the quality of the state-action pair. It is defined as: $q_\pi(a, s) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a]$.
There exists a range of different algorithms to find the optimal policy. We call a method value-based, when the algorithm learns a value function, which can be used to extract the optimal policy from it, by executing the actions which lead to the greatest return. One of the leading and most influential value-based algorithms is Q-Learning, presented by Watkins et al. [39]:

$$Q^{\text{new}}(s_t, a_t) = Q^{\text{old}}(s_t, a_t) + \alpha[r_t + \gamma \max_a Q(s_t, a) - Q^{\text{old}}(s_t, a_t)]$$

The underlying principle involves iteratively updating the Q-values based on the observed rewards and the maximum Q-value of the next state, emphasizing a balance between exploration and exploitation.

2.2 Markov Decision Processes

A Markov decision process (MDP) is a tuple $\langle S, A, P, R, \gamma \rangle$, where $\mathcal{S}$ is the set of states, $\mathcal{A}$, is the set of actions, $P(s'|s, a)$ is the transition probability from state s to s', using action a. $R(s, a, s')$ is the reward obtained after having taken action a in state s and landing in state s', and γ is a discount factor [33]. The MDP is a framework that enables us to clearly formulate the problem of RL and reason about the objective and how to achieve it. Another advantage of formulating the problem as a MDP is that we're able to make use of the Markov Property: $P(S_{t+1} = s'|S_t = s) = P(S_{t+1} = s'|h_{t-1}, S_t = s)$. It means that the agent has all the necessary information in the current state to take the best decision. In an MDP, all states are assumed to have this property.

Several different variations of MDPs exist. One of them, is a partially observable Markov decision process (POMDP). It's a generalization of an MDP which permits uncertainty regarding the state of a Markov process, and allows for state information acquisition [19]. Formally, a POMDP is defined by the tuple $\langle S, A, P, R, \Omega, O, \gamma \rangle$, where S, A, P, R, and γ remain the same as for MDPs. Ω is a set of observations and O is a set of conditional observation probabilities. After agents execute an action, they now obtain an observation $o \in \Omega$, according to the observation probability function $O(o|s, a)$. POMDPs serve to model problems in which state uncertainty is a central issue, and cannot be assumed away. As such, the observations which are obtained from the environment are *non-markovian*. While the environment can still be Markov, the agent does not know it. Still, there are methods with which to construct a Markov agent state, such as taking in a certain amount of history [18] or making use of recurrent neural networks [11].

2.3 Deep Reinforcement Learning

All components of RL are functions. The policy is a mapping from states to actions, or probabilities of the action set. The value functions are a mapping from states to real numbers. The model is a mapping from state to states and the state update is a mapping from states and observations to states. This opens the door to use Neural Networks (NN) and Deep Learning (DL) techniques to learn or approximate these functions, which has been proven to be incredibly powerful. On the downside, we also have to take care, as we violate certain assumptions from supervised learning, such as the data being independent and identically distributed, and the problems being stationary.

2.3.1 *Value-based methods.*

The most commonly used value-based DRL method is a Deep Q-Network (DQN), introduced by Mnih et al. [18]. It is based on Q-Learning and parameterises an approximate value-function $Q(s, a; \theta_i)$. In which θ_i are the weights at the current iteration. Practically speaking, the Q-network takes the current observations as input, and outputs the Q-value estimates. The action to be taken is chosen like we would with Q-Learning, according to which maximises the Q-function. After a certain amount of iterations, the weights of the NN are updated according to a loss function, to move them closer to a target. For this update, a certain amount of past experiences are randomly sampled, to remove correlations, from a replay buffer.
Several different improvements to vanilla DQN have been proposed since then, such as Double-Deep Q-Network, which utilises double learning to reduce the over-estimation problem that standard Q-Learning methods have [10]. Improvements have also been brought to the way past experiences are sampled. While sampling uniformly from past experiences makes sense mathematically speaking, these samples don't necessarily contain the experiences that are best to replay to the agent i.e., that result in the best behaviour optimization. To help address this problem, Schaul et al. [27] introduced Prioritized Experience Replay.

2.3.2 *Policy Gradient methods.*

Instead of learning a value function to extract a policy, policy gradient methods directly learn a *parameterised policy*. The policy is now parameterised by a vector $\boldsymbol{\theta} \in \mathbb{R}^{d'}$ and we write the policy as $\pi(a|s, \boldsymbol{\theta})$ [33]. Actions are then chosen according to probabilities, instead of values. This results in a stochastic policy. To learn the policy parameter, stochastic gradient ascent is performed according to some performance measure. The goal is to maximize the expected discounted return of the policy.
Methods that both learn a policy and a value function, are called *actor-critic* methods. The 'actor' refers to the policy, while the 'critic' is a value function, that gives feedback on the value of the action that was picked [33].

Advanced Actor-Critic. (A2C) is a DRL algorithm that combines the strengths of actor-critic methods with advanced training techniques. It utilizes two neural networks: an actor network to select actions and a critic network to evaluate state-action pairs. A2C

optimizes both networks simultaneously, enabling efficient learning and faster convergence. It employs parallel environments for experience collection, enhancing sample efficiency. By updating the actor and critic networks independently, A2C mitigates issues such as high variance in policy gradients, leading to stable and effective training. A2C is widely used in various applications due to its simplicity, scalability, and robust performance.

Proximal Policy Optimization. (PPO) is a cutting-edge DRL algorithm designed for stability and sample efficiency [28]. PPO operates within an actor-critic framework, utilizing a policy network to select actions and a value function network to estimate state values. Noteworthy for its "clip and trust" strategy, PPO constrains policy updates to a specified range, preventing large policy changes and enhancing stability during training. By iteratively collecting experiences and updating the policy with multiple epochs, PPO strikes a balance between exploration and exploitation. Its robustness, simplicity, and capacity to handle continuous and discrete action spaces make PPO a prominent choice in contemporary reinforcement learning research and applications.

3 STATE OF THE ART

The state of the art in this field consists of two big components. 1) The execution environments, in the form of different simulators, which are used to train agents. 2) The penetration testing agents themselves. The execution environments are presented first, since some of the agents discussed in the following section, make use of them. They are also an important part of training agents in general. Lastly, we also consider agents that are not directly trained for penetration testing as a whole, but do perform one of the subdisciplines (like vulnerability scanning) or which are trained for pentesting, but do not make use of DRL.

3.1 Execution Environments

To train (Deep) Reinforcement Learning agents, we need the right environment for them to interact with, gather experiences, and learn from those. We've already introduced the mathematical framework in Section 2. Here, we're going to present the different simulated environments that have been proposed to train agents. Naturally, these environments represent networks with vulnerable machines in which the agent is confronted to a growing number of available choices, and needing to perform exploration operations and applying the right exploit to a service vulnerable to it. While these can be considered "toy-problems", they deliver an important framework to evaluate different algorithms.

NASim. The Network Attack Simulator (NASim) was one of the fist simulators presented. Developed by Schwartz [29] during his BSc Thesis. Afterwards, it became a stand-alone project, still being developed by the original author [30] and the open source community. The simulator features hosts that can be configured with different services and deployed in subnets with firewalls between them. The agents can perform three different types of actions: Exploitation, Privilege Escalation, Scan. An illustration is provided by Figure 1. The gymnasium[1] gym interface is also provided, which makes for easy development, and interaction with the environment.

[1]https://gymnasium.farama.org/

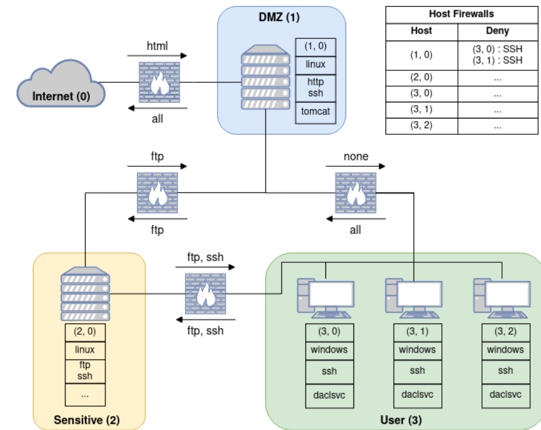

Figure 1: Example NASim Network [30]

Scenarios can be written in `yaml`. Pre-defined and benchmark scenarios are also available.

CybORG. Standen et al. present a work-in-progress gym for Autonomous Cyber Operations (ACO) [32]. Their design includes a simulated as well as an emulated environment, with a shared interface, to be able to easily switch from the simulated environment to an emulated one. The difference is that the actions are executed then on VMs or containers, rather than a graph with transition probabilities. Scenarios may be defined for the Cyber Operations Research Gym (CybORG), which describe the 'game' that agents are aiming to solve or compete in. Hosts can also be configured with a variety of different information for the agent to obtain during the game. The choice of possible actions made available to the agent are left to the user in the form of the scenario definition. Interaction is done through a gym interface. The results of the actions are filtered and merged to present a single observation back to the agent [32].In the paper they talk about their design and the addition of emulation as well [32]. But effectively, the emulation part has never been implemented and the project has been discontinued, as pointed out by Janisch et al. [12].

CyberBattleSim. The Microsoft Defender Research Team developed their own simulator. The CyberBattleSim (CBS) environment consists of a network of computer nodes and is parameterised by a fixed network topology and a set of predefined vulnerabilities. The simulated attacker's goal is to take ownership of some portion of the network by exploiting these planted vulnerabilities [35]. The simulator also features a stochastic defender agent that tries to detect the presence of the attacker and tries to contain the attack. The environment is also designed to be partially observable. Compared to other simulators, it focuses more on the post-breach lateral movement state of a cyber attack, rather than the entire Penetration Test of a network. It features quite a rich set of properties which can be defined by a node, representing: Services, Vulnerabilities, Firewall attributes, Privilege level, etc. and it can be quite exhaustive to define an entire network. Addressing this problem, Esteban et al. [8] developed a web-interface to more easily configure a scenario.

NASimEmu. Janisch et al. have extended the existing NASim project with an emulation functionality through a shared interface [12]. Thus, the action space remains the same (with some exceptions as the authors point out), but the environment can be changed to take agents from a simulation, to an emulation. The emulations run on virtual machines, which are orchestrated via Vagrant. These are configurable Linux and Windows machines, based on Metasploitable3 images, with pre-defined vulnerable services to choose from. The authors map the `Exploit` and `PrivEsc` actions already existing in NASim to predefined Metasploit modules. While describing their work, they are also clear about some of the limitations. The emulation does not fully correspond to the simulated environment due to complications implementing some of the features. They do however reason that these limitations can be overcome with future work.

3.2 Autonomous Penetration Testing Agents

This section contains the bulk of the work that has been proposed. We qualify autonomous penetration testing agents as agents that make use of RL, DRL, or related forms such as Imitation Learning (IL) in order to make decisions in an environment, for which the goal is to find vulnerabilities in the different components that make up the environment.

3.2.1 Tabular Methods. Tabular methods refer to RL algorithms in their simplest form: that in which the state and action spaces are small enough for the approximate value functions to be represented as arrays, or tables [33]. One of the first works in this field of autonomous pentesting was done by Jonathan Schwartz [29] back in 2018. In his Bachelor Thesis titled *Autonomous Penetration Testing using Reinforcement Learning*, he trained two Q-Learning agents employing different exploration strategies on his Network Attack Simulator. The work could be considered basic nowadays, but nevertheless, it paved the way for a lot of future research, especially since the simulator has been made publicly available, and is used extensively by other work. While there are other usages of tabular methods than the one described above (i.e. for attack planning), and they certainly have their place for smaller problems, we disregard the rest of them. The reason being, that they pre-date the DRL methods, and aren't able to converge for the type of problem described by this paper, due to the missing generalization across the large state and action space. To close this sub-section, we argue that all the main ideas can be found back in the more advanced works employing DRL or IL.

3.2.2 Deep Reinforcement Learning. The agents presented in this section all make use of DRL, and have been classified according to the environment they run in: simulated, or real-world. The motivation for this choice lies in the similar objectives and challenges that these agents try to pursue.

Simulated Environment. The work that has been proposed using a simulated environment is more concerned with the theoretical side of the problem and it's implications, such as being able to handle larger networks which implies larger state and action spaces, partial observability, and experimenting with multi-agent architectures.

Zhou et al. propose an algorithm they call NDSPI-DQN [41]. This name results from the improvements they brought forward to the standard DQN algorithm, namely: Noisy Nets, Duelling network architectures, Prioritized Experience Replay (PER), Soft Q-Learning, and Intrinsic curiosity module. Further, they decouple the attack vector, which means that the neural network (NN) is split to be able to decide separately on the target, and the operation to perform on the target. This permits the agent to handle larger environments, since the growing action space is handled more easily. As for the reward structure, the agent receives a reward for the compromised host(s), minus the cost associated to the executed exploit. The experiments are done using the NASim. Tests are performed on four different scenarios with a growing number of network elements, but leave the same number of valuable hosts, to simulate sparse rewards. The results show that the agent successfully handles such scenarios. Tran et al. focus on the large action space problem, and to this end, proposed an action decomposition strategy to reduce the large discrete action space into manageable sets for DRL agents, which stabilizes the learning and enhances convergence properties [36]. This strategy will be discussed in more detail in Section 4.2. The evaluation is done using the CybORG simulator. To test the efficacy of the strategy, they simulate several network scenarios with up to 100 hosts. The findings demonstrate their capability to manage an action space comprising over 4600 actions. Nguyen et al. [20] are concerned with the same problem and introduce a double agent architecture (DAA), which aims to decompose the problem into two separate parts. First is to understand the network topology, and second, choosing appropriate exploitable services to attack a host. Both agents are trained using the A2C algorithm with a basic NN structure. The training and testing are executed within an altered NASim version, to make the process more difficult and realistic. Overall, they are able to handle an action space of about 4100 actions, but the algorithm suffers from sample inefficiency.
Yang et al. consider another approach of representing the problem through Multi-Objective Reinforcement Learning (MORL) and using a Multi-Objective MDP (MOMDP) [40]. This, according to the authors, represents a more practical setting for modelling penetration testing. Another problem they're concerned with is the growing amount of actions the agents are faced with during the training phase. To this end, they propose a coverage masking mechanism to mimic human pentesters, who pay more attention on recently uncovered information. Their proposed PPO agent with Random Network Distillation (RND), to account for sparse rewards, is tested on two different simulators: NASim and CAGE. The scenarios use 23 hosts at most and the performance is compared to previous work in the field (NDSPI-DQN [41] and HA-DRL [36]). The results show that they achieve similar performance, but are unable to outperform previous work.
Li et al. [16] propose EPPTA (Efficient POMDP-Driven Penetration Testing Agent), built on an asynchronous RL framework, which is designed to be compatible with most on-policy actor-critic RL algorithms. Specifically, in their research, they adopt the PPO algorithm to train their agent. Since the environment is modelled as a POMDP, the agent can't observe the true state of the target network. Therefore, it has to rely on a probability distribution over states, referred to as the belief state. To handle this problem, the authors introduce a belief module into their framework, to efficiently capture the belief state of the agent. In addition, they adapt their implementation to be compatible with the Sample Factory library [22],

which increases the convergence capabilities of the agent through better computational performance. The agent is tested using the NASim benchmark scenarios, and compared to previous work such as HA-DRL [36] and NDSPI-DQN [41], as well as EPPTA without the usage of the Sample Factory. The results show that they are capable of outperforming the previous work and of handling the largest benchmark scenario of 95 hosts and over 3500 actions.

Real Environment. Testing in the real-world is very important, as this is where the agents are ultimately intended to run. Since the application to the real-world is already more complicated and intricate, most work that was realised focuses on a specific subdiscipline of penetration testing. Specifically, we found that all these papers focus on post-exploitation. In post exploitation, we've already gained a foothold on the network and the focus shifts to lateral movement and privilege escalation, while also performing further information gathering.
Deep Exploit [34] is the name of a project realised by Isao Takaesu, and if we look at a chronological order of events, it might actually have been the very first work making use of DRL for autonomous penetration testing as well as running in the real-world. Deep Exploit is a DRL agent capable of exploiting servers on the perimeter and inside networks. It uses the Asynchronous Advantage Actor-Critic (A3C) algorithm, a predecessor of A2C, to train an agent on how to make use of the Metasploit Framework (via the RPC API) [24] while interacting with several different training servers. While the agent interacts with the network, it collects information about the target hosts such as the OS Type, Service Name, and Version through Nmap[2] and other tools, to be able to select the right Metasploit modules. The capabilities are showcased in different scenarios where hosts are directly, or only indirectly reachable.
Kujanpää et al. [14] focus on privilege escalation for Windows 7 hosts. To achieve this, they use model-free DRL in a partially observable environment, they define a set of high-level actions for the agent to take, such as gathering information about running services, tasks, etc. and exploits to target them. The agent initially starts out with no information, and through exploration, the agent state is updated with the observations that are the outputs of executed commands. To select a new action, the gathered information is fed into the actor-critic network. The agent was trained in a simulated Windows 7 environment, which contains elements related to the task at hand. After training the agent is transferred and tested on a Windows 7 VM, configured with some known vulnerabilities.
Maeda et al. [17] on the other hand, make use of the capabilities of PowerShell Empire. The agent is taught to select one of the many modules as actions to be performed on the environment, with the goal of gaining domain admin privileges on the Domain Controller (DC). They also maintain an agent state to account for the information gathered by the agent. The agent was trained in a synthetic environment: Small noise was added to the transition probabilities, which were also adjusted according to different difficulty levels. The motivation for this synthetic setup lies in the unavailability of any other training environments at that time [17]. To test their agent, they deployed a very small AD network, using one Domain Controller and two PCs.

[2]https://nmap.org/

Pham et al. develop a framework called Raijū [23], capable of performing post-exploitation on real targets, through the usage of the Metasploit framework [24]. The agent is given a set of Metasploit modules as possible actions and learns how to use them in a real environment through interaction and updating it's agent state. In particular, they implement, test and compare two agents: A2C and PPO. The environment consists of vulnerable Windows 7 and Metasploitable2 Linux machines, whereas one machine serves as the stepping stone into the small network of three to five hosts. Results show that A2C outperforms PPO according to the author's metrics.

3.2.3 *Imitation Learning.*

IL consists of agents that learn to mimic the behaviour of humans in a specific task, mostly from pre-existing example behaviours, and not from the ground up, like it would be with traditional RL. Wang et al. [38] present an intelligent PT framework named DQfD-AIPT, which collects expert knowledge and uses the Deep Q-Learning from Demonstration (DQfD) algorithm to make use of it. The expert knowledge here refers to a human that provides specific transitions/actions to be performed in a given network scenario and state. These are then encoded and injected into the experience database for the agent to use them. The agent is deployed in multiple different CyberBattleSim environments that have been configured with honeypots, and tested against a simple DQN agent as a benchmark. The goal of their experiment is to show that the DQfD agent is capable of ignoring the honeypot much faster than the DQN agent without prior knowledge. Interacting with the honeypot is undesirable as it gives the agent a large negative return and ends the episode.
Around the same time, a similar paper was published by a different group of researchers, Chen et al. [3]. In this instance, the algorithm the authors used is GAIL (Generative Adversarial Imitation Learning) and they call their framework GAIL-PT. One of the big differences between the two works is the collection of expert knowledge. Whereas DQfD-AIPT makes use of human experts, GAIL-PT relies on pre-trained models as the expert, namely DeepExploit [34], for the collection of data related to Metasploitable2 machines. The expert knowledge base was constructed by collecting the state-action pairs with the highest reward value. Another difference is the usage of actor-critic models compared to Q-Learning models. While GAIL-PT is also tested in a network scenario with honeypots, the authors go one step further and assess the capabilities of gaining elevated privileges in a real-world single host scenario.

3.2.4 *Synthesis.*

We've presented some of the most important characteristics about the different pentesting agents. Table 1 gives an overview of the agents from Section 3.2.2 and 3.2.3. For each paper, we've listed in which simulator the implementation runs, or what sort of real-world environment is used, in case it's an agent that targets a real-world application. GAIL-PT is the only work that was tested (using different versions, under different conditions) in a simulated and a real-world environment. For real-world environments we also distinguish between metasploitable (ms-able) machines, and self-defined environments. Metasploitable machines are readily available and easy to hack VMs. Self-defined environments have been designed and configured by the authors themselves to fit their needs. Next we also list the algorithm that was used. Some papers test several algorithms and compare their performance for the given

Paper	Simulator			Real-World Environment		Algorithm	Model
	NASim	CybORG	CBS	ms-able	self-defined		
NDSPI-DQN [41]	•					DQN	MDP
EPPTA [16]	•					PPO	POMDP
Kujanpää et al. [14]					•	A2C	Model-free
HA-DRL [36]		•				DQN	MDP
Maeda et al. [17]					•	A2C	Model-free
CLAP [40]	•	○				PPO	MOMDP
Raijū[23]				•	•	A2C	Model-free
DQfD-AIPT [38]			•			DQfD	MDP
DAA [20]	○					A2C	MDP
GAIL-PT [3]	•			•		DPPO	MDP
Deep Exploit [34]					•	A3C	Model-free

Table 1: Overview of current Autonomous Penetration Testing agents. If a paper has used and compared different algorithms for training their agent, we take the best performing one. With ○ we represent that some modifications had been made to the simulator, or that a predecessor was used (in the case of CLAP).

scenario. We only list the best performing algorithm. Finally, the table shows the choice that was made by the corresponding authors for modelling the environment, whether the authors modelled it as an MDP, POMDP, or did not specify a model.

3.3 Related Autonomous Agents

Only considering autonomous penetration testing agents results in a very specific and small scope. There are closely related agents that also make use of RL, but only focus on one specific vulnerability, instead of penetration testing as a whole. Other work changes the primitive and instead of DRL, makes use of Large Language Models (LLM).

LLMs. In related work, Deng et al. showcase the implementation of a tool called `PentestGPT` [4]. Their results show that `PentestGPT` is capable of guiding a human through a penetration test of a single machine with reasonable difficulty. The architecture includes three modules: a Reasoning Module for strategy planning, a Generation Module translating sub-tasks into commands, and a Parsing Module handling natural language information. Despite using separate LLM instances for each module to address context loss, some tasks exceed the GPT-4 context window of 32k tokens. Limitations include potential overemphasis on recent information, leading to neglect of older findings and failed penetration test attempts.

Vulnerability Scanners. Compared to the goal of performing penetration testing, and assessing a whole range of different vulnerabilities, we also find works in the literature that make use of RL or DRL to scan for one specific vulnerability [1, 7, 9, 15]. Foley et al. developed a tool called `HAXSS` [9] capable to finding and exploiting XSS vulnerabilities. The architecture consists of two DQN agents that play two different games but work together. One agent learns to build payloads that escape the current context, while the other learns to alter the payloads such that they bypass sanitisation checks. Additionally, each agent uses several worker agents which sample actions using the global policy and after the agents have executed their actions, the policy is updated. In another work from the same lab, Wahaibi et al. present `SQIRL` [1], a grey-box scanner for finding SQL injection (SQLi). They also use several worker agents in a federated learning fashion to train a global policy. The process of altering a SQL statement until injection is reached, is modelled via three different games that the agent repeatedly plays.

Since the agents are also confronted with a growing amount of choice related to the actions, the DQN architecture is altered to compute a Q-value one action at a time by converting them into a set of action representations.

4 RESEARCH CHALLENGES

We're now going to present the challenges associated to the design, development and deployment of autonomous penetration testing agents. Since the goal is to have agents that run in the real-world, some of the challenges we identify here are well-known. An overview of real-world RL challenges has already been provided by Dulac-Arnold et al. [6]. We nevertheless argue that a comparative study of how the challenges are handled specifically by penetration testing agents is important to further describe the current state of the art and identify future research directions.

One of the first challenges to mention is not about technical aspects of RL, but rather it's about reproducibility and comparing existing approaches. While performing our literature study, one of the greatest problems we encountered was to be able to actually compare the proposed implementations. This is due to a range of different factors: algorithms, execution environments, modelling of the environment, different goals, reward structures, implementation details, and most importantly, missing code. In Table 1 we see that some use a common simulator, but then there are different choices regarding the scenario, or modifications have been made to the underlying simulator [20]. Since comparing different work is so difficult and intricate, it is also complicated to get a real, objective overview of what works well, and what doesn't. Some of authors compare their work to other existing implementations [16, 40], but under different circumstances and scenarios, which renders the results questionable. Being hardly able to compare existing work is especially detrimental to evaluating the progress that has been made thus far in addressing the different challenges that follow.

4.1 Partial Observability

In the literature we find multiple approaches of modelling the environment for penetration testing (cf. Table 1). In general, most approaches that run in a simulated environment, formulate pentesting as an MDP. This formulation is generally ill-suited [25, 26], especially in a real-world scenario [6]. The reasons are that MDPs do not represent the partial observability of the problem at hand. Penetration testing is inherently about the search and discovery of vulnerabilities. It makes little sense to provide observations that already contains all the vulnerabilities that can be found. This is also known to the simulator developers. NASim, CBS, and NASimEmu propose options to set the simulated environment to only partially observable. In contrast, the proposed agents running in a real environment, make use of model-free RL algorithms [14, 17, 23, 34]. In model-free learning, a model of the environment is not available and the agent learns directly from experience [2]. A third method is to take into account the partial observability by modelling the environment as a POMDP [16].
The key challenge with partial observability is that the agent's observations are now different from the actual state of the environment [6]. The states become *non-markovian.* Therefore, methodologies

have to be employed that can somehow represent and take into consideration the agent's history, to still select the best actions.

Model-free approaches for autonomous pentesting typically employ an agent state to address the problem. In the work of Kujanpää et al., the agent obtains observations through executing actions, which are then converted into the agent state. Further, it is the agent state s_t that is used as the input for the policy $\pi(a_t|s_t)$ and state value function $V(s_t)$. Previous actions are also included into the state of the agent to make it as close to Markov as possible, which makes the training easier [14]. Maeda et al. [17] and Pham et al. [23] also employ an agent state, but they provide less details about the specifics.
The authors of EPPTA model the penetration testing environment as a POMDP. They account for the partial observability through the introduction of a belief module, which incorporates transient belief states $\tilde{b}_{t+1}$ to capture the current belief state more efficiently. These are used to calculate the expected probability of reaching state s_{t+1} based on the current belief state b_t and action s_{a+1}. This innovation enhances the interaction between the agent's NN and the environment, resulting in more accurate probability distribution approximations in partial observable environments according to the authors [16].
While we've mentioned that some simulators offer a model with partial observability, it is difficult to verify whether the authors of the papers using them have actually used this functionality. On one side because it is not explicitly mentioned, on the other, because there is no source code available to independently verify this.

4.2 Large Action Space

When starting to reason about the problem of autonomous pentesting, one is quickly confronted with the problem of large action spaces. This is simply a consequence of the discipline. At any given time, while executing a penetration test, a human faces a large amount of different possible actions that can be executed. Taking the scanning phase as an example, there is the question of which type of scan to perform, against which target, using which parameters, and so on. In the current literature, several empirical studies have been conducted to try address the problem, but the results remain varied. Large action spaces on their own might not be a significant problem, but most often we also encounter large state spaces. The challenge lies in the ability to be able to generalize over such a large set of actions and states and still be able to learn a useful policy [5]. In the following we present how the different pentesting agents have tried to address this challenge.

Attack Vector Decoupling. Zhou et al. chose to decouple the attack vector. This means that the NN is split into two streams, to be able to choose separately on the target, in the form of the value of the host, and the operation/attack to perform. This choice reduces the original action space from $O(MN)$ to $O(M + N)$ [41]. Their experiments show that making use of decoupling really helps the agent learn in large scale environments. Without decoupling, the agent is barely able to handle an environment with 80 hosts, in contrast to 150 with decoupling.

Coverage Masking. Yang et al. introduce a coverage masking mechanism. It mirrors how human penetration testing experts work in real scenarios, shifting their focus on more recently discovered hosts and subnets upon finding them [40]. To simulate this in training, the authors maintain a coverage set that tracks past actions taken by the agent. This coverage vector remembers where the agent's attention was in previous attempts and adjusts the current iteration's focus. It pushes the actor network to choose actions that concentrate more on recently discovered parts of the network, reducing repetitive actions on already explored subnets or hosts.

Multi Agent Reinforcement Learning. Tran et al. proposed an action decomposition strategy for agents to better handle a large and discrete action space. It separates the action set into smaller, more manageable sets. In particular, separate DQN agents are used for each of the created subsets. Each of those agents is trained to output *primitive* actions, which are then combined using a linear function to build up the final action [36]. Since these are all separate agents, they all have separate networks with their own weights and biases. Each agent also receives the same reward signal to minimise its own loss function. The authors argue that as longs as each agent learns the optimal policy to achieve the highest possible long term reward, together with substantial exploration, the combined or the overall policy will converge to the optimal behaviour. Overall, this proposed strategy stabilizes the learning and enhances convergence properties [36]. But they also hypothesize that in a scenario with sparse reward or an extremely large action space, the optimal behaviour condition may not be practically feasible.
For their double-agent architecture (DAA), Nguyen et al. [20] use two agents as the name suggests, whereas one is only tasked with scanning and gathering information about the network and it's hosts (*the structuring agent*), while the other only learns to execute which exploit on which service of a given host (*the exploiting agent*). The exploiting agent is triggered once the structuring agent deems it has gathered enough information about a given host, and receives the state information about the selected host as input.

Synthesis. While the proposed EPPTA framework is able to handle a large action space [16], we attribute this success more to the powerful hardware that was used and the usage of Sample Factory, rather than a specifically designed approach to handle large action spaces. HA-DRL and DAA both use separate agents, which makes them look quite similar, but they are actually very different approaches. DAA trains agents to execute different tasks, while HA-DRL partitions the action space over the set of configured agents, such that a single agent has fewer actions to learn.

4.3 Reward Structure

Having a good reward structure is of very high importance, since the agent learns through experience while interacting with the environment, where the rewards serve as a way of letting the agent know, that it has performed the right sequence of actions[3]. The reward is a way of communicating to the agent *what* to achieve, but may not be used to tell it *how* to do it [33]. Therefore, setting the right reward structure is a challenge not to be overlooked.

[3]This is a simplification, as we're omitting problems such as credit assignment here.

Throughout the literature study, we have encountered a range of different reward structures. Some just use a very simple and basic structure, such as giving the agent no reward while the goal was not reached and a reward of one upon reaching it [14], or higher, based on the assigned value of the host [3, 17]. Or the agent just gets a small negative reward, for every time step that the task is not solved, to represent a sense of urgency and communicate to the agent to find the solution as quickly as possible [1, 23]. Some also set a very large negative for something that the agent shall avoid at all costs (i.e. interacting with a honey pot [3, 38]). What we encountered most, are reward structures that associate a cost to actions, but also assign values to specific hosts [20]. This philosophy is followed by NASim [30], and thus, many of the implementations that make use of them, employ it as well [38, 40, 41].
Others choose a more complex reward structure. Since Nguyen et al. [20] proposed the usage of two agents, they also need to account for the reward of the *structuring* agent, whose actions do not directly generate a big reward, since it only performs discovery actions. They solve this by feeding the reward generated by the *executing* agents to the *structuring* agent as well.
Contrary to most prior work, Yang et al. [40] chose a MOMDP to model the environment. In this scenario, the reward is a vector with n individual rewards, each corresponding to an objective. Since their simulator originally only gave a single reward, they modify it to give out rewards for exploiting vulnerabilities, and privilege escalation. As multiple conflicting objectives is not a rare sight in MORL, the authors make use of Chebyshev Critic Scalarization to convert them into a single objective function.

We'd like to argue that not all proposed reward structures are of good use. For instance, assigning large negative rewards upon interaction with a honeypot [3, 38] is not something that is transferable to new environments. Especially in partial observability. It might also be interesting to experiment with a reward related to the agent state, such as the gathered accesses for instance, not the value of hosts. Just giving a reward for hacking one or more specific hosts would devalue finding vulnerabilities which were not known about in other systems. This is also one of the reasons why multi-objective rewards are quite interesting as tested by Yang at al., and also argued by Vamplew et al. [37]

4.4 Adapting to the Real-World

Developing agents to run in a simulated environment is relatively easy. In the best case, there already exist a bunch of simulators and all that's left is to choose the algorithm and to train the agent with the right hyper-parameters. Adapting to the real-world is a lot more difficult. The simulator that took care of all the necessary abstractions is no longer there. In addition, we're interacting with real machines now, that exhibit their own behaviour, which has to be accounted for, and that can even break. Another problem is being able to train the agent, such that it has accumulated enough knowledge to be useful. This is one of the challenges we cover here. Second, there is also the challenge of using the right design approach that helps the agent switch between training and execution in the real-world.

Training Environment. Before being able to launch the agent in a real environment, it needs to learn in a training environment. While real machines could in principle be used as a training ground, this is not feasible in most cases, since resetting the machines alone would increase the training time considerably [14, 23]. Therefore, Kujanpää et al. have implemented a small simulator to train the agent specifically for the task of privilege escalation [14]. In contrast, Pham et al. store the values of the environments and results of exploits in CSV files, to be used in the training phase [23]. In this way, the values could be loaded and given to the agent, instead of directly interacting with the environment and needing to reset machines.

Action Space Representation. Through the chosen high-level action design, which enables modifying the low-level implementation of the actions, Kujanpää et al. [14] are able to take their agent from the simulated environment, into the real-world. Some tinkering had be done to be able to execute all the actions on a real-environment, but the choice of using high-level actions proved to be effective, and the agent was able to successfully exploit all vulnerabilities is scope. In contrast, Maeda et al. [17] directly mapped the 204 different PowerShell Empire modules as actions for the agent to take. The communication between the two components was established through PowerShell Empire's restful API. Similarly, Pham et al. [23] map 99 different Metasploit modules as actions for the agent to perform post-exploitation on Linux and Windows machines. Although the agent selects the action, they make use of an *exploiter* to actually execute the action. This same design of using a proxy to actually execute the actions is also chosen by [1, 9].

There is a clear gap between where the research wants to be, and where it currently stands. We argue that it might be better to take different approaches. Instead of using general-purpose simulators that offer different scenarios and maybe even have an emulation component, it might be more interesting to use synthetic environments like Maeda et al., or to directly develop a simulator for the specific task that we want to address, such as Kujanpää et al. But this might only work for small environments. In the end, the agent can only learn what the environment allows the agent to learn. More general-purpose simulators are still interesting to benchmark different algorithms (*cf.* 3.2.2), but it doesn't serve any good use if the knowledge can't be transferred to the real-world afterwards.

4.5 Comparison

As we've established in the introduction to this Section 4, it is very difficult to qualitatively assess the proposed agents. Here, we perform an indicative evaluation, represented in Table 2. This comparison aims to showcase which methods are able to handle partial observability, while also displaying the largest action space they have been tested with. The numbers have either been taken directly from the paper, or calculated with the provided formula and scenario description. For DQfD-AIPT and Deep Exploit, neither were the case, and thus we were unable to reliably make the count. We'd like to highlight that EPPTA is the only agent running in a simulator that also accounts for partial observability. Whilst there are other agents running in simulations that do achieve to handle a larger

Paper	Challenges	
	PO	LAS
NDSPI-DQN [41]	-	2700
EPPTA [16]	+	3515
Kujanpää et al. [14]	+	38
HA-DRL [36]	-	4646
Maeda et al. [17]	+	204
CLAP [40]	-	322
Raijū[23]	+	99
DQfD-AIPT [38]	-	?
DAA [20]	-	4136
GAIL-PT [3]	-	322
Deep Exploit [34]	+	?

Table 2: Comparison of proposed agents for Partial Observability (PO) and Large Action Space (LAS)

action space, they use MDPs for the environment representation, and thus don't have to account for the added computational difficulties that POMDPs bring along [19]. On the other hand, Maeda et al.'s agent has the largest action space out of the real-world agents, but the agent does not make use of the entire action set. It has to search through 204 PowerShell Empire modules, but for the given task of lateral movement and privilege escalation, only a handful of them are really useful.

5 FUTURE RESEARCH DIRECTIONS

Through our study and analysis of the literature and existing methods, we are able to identify a plethora of different research directions and future work still needed to go beyond prototypes. Starting with simulators. The main limitation is being able to deploy an agent that has been trained in a simulated environment, into the real-world. One promising approach would be to have the agent learn high level actions, but the actual action is executed by an underlying component. The feasibility has already been demonstrated by Kujanpää et al., but their simulator had been created specifically for the problem they intended to solve [14]. Continuing, a more practical comparison between existing methods would be interesting, where they would be tested in the same environment, under the same constraints, to have a better overview of the capabilities of the algorithm. The inclusion of LLMs could also be a very interesting and promising research directions. They could be used to provide an overall strategy or write the commands to be executed, like we have seen demonstrated by Deng et al. [4].

More experimentation with reward structures is also necessary. Structures that are able to represent rewards from which the agent can learn the most, that give the best feedback, while also being applicable to the real-world. We've seen some work already making use of MARL [20, 36], and this is certainly a very interesting avenue for future work, having already been pointed out by Schwartz [29]. The vision could be to have one agent that learns the overall penetration testing strategy and have multiple agents specialised in the different sub-tasks. It has already been shown that agents can be very efficient at one given task, and even find new CVEs [1, 9]. Going further in the usage of IL is certainly also an important direction. It has already been shown that agents making use of IL are able to converge faster [3, 38].

Finally, the challenges such as large action spaces, and acting under partial observability are not exclusive to autonomous penetration testing agents, but fundamental challenges for the applicability of RL to the real-world. These also require a lot of research to be addressed. For a more complete overview we refer to Dulac-Arnold et al. [6].

6 CONCLUSION

In this work we've analysed and compared the different autonomous penetration testing agents and the proposed environments for them to be trained in. We've also looked at some closely related works like vulnerability scanners and LLMs for pentesting. Through this comparison we have identified several research challenges: handling large action spaces, partial observability, creating adequate reward structures, and taking the agents from a simulated environment into the real world. While good efforts have been made regarding each of the challenges, we're still far away from going beyond prototypes. Solving or partially addressing these challenges is necessary to be able to have trained agents that can truly serve a purpose in the real world cyber security domain. Lastly, we've pointed out several different promising research directions that could drive forward the field.

ACKNOWLEDGMENTS

We would like to thank Arbia Riahi for her valuable feedback.

REFERENCES

[1] Salim Al Wahaibi, Myles Foley, and Sergio Maffeis. 2023. {SQIRL}:{Grey-Box} Detection of {SQL} Injection Vulnerabilities Using Reinforcement Learning. In *32nd USENIX Security Symposium (USENIX Security 23)*. 6097–6114.

[2] Steven L. Brunton and J. Nathan Kutz. 2022. *Data-Driven Science and Engineering: Machine Learning, Dynamical Systems, and Control* (2 ed.). Cambridge University Press.

[3] Jinyin Chen, Shulong Hu, Haibin Zheng, Changyou Xing, and Guomin Zhang. 2023. GAIL-PT: An intelligent penetration testing framework with generative adversarial imitation learning. *Computers & Security* 126 (March 2023), 103055. https://doi.org/10.1016/j.cose.2022.103055

[4] Gelei Deng, Yi Liu, Víctor Mayoral-Vilches, Peng Liu, Yuekang Li, Yuan Xu, Tianwei Zhang, Yang Liu, Martin Pinzger, and Stefan Rass. 2023. PentestGPT: An LLM-empowered Automatic Penetration Testing Tool. http://arxiv.org/abs/2308.06782 arXiv:2308.06782 [cs].

[5] Gabriel Dulac-Arnold, Richard Evans, Hado van Hasselt, Peter Sunehag, Timothy Lillicrap, Jonathan Hunt, Timothy Mann, Theophane Weber, Thomas Degris, and Ben Coppin. 2016. Deep Reinforcement Learning in Large Discrete Action Spaces. https://doi.org/10.48550/arXiv.1512.07679 arXiv:1512.07679 [cs, stat].

[6] Gabriel Dulac-Arnold, Nir Levine, Daniel J Mankowitz, Jerry Li, Cosmin Paduraru, Sven Gowal, and Todd Hester. 2021. Challenges of real-world reinforcement learning: definitions, benchmarks and analysis. *Machine Learning* 110, 9 (2021), 2419–2468.

[7] László Erdődi, Åvald Åslaugson Sommervoll, and Fabio Massimo Zennaro. 2021. Simulating SQL injection vulnerability exploitation using Q-learning reinforcement learning agents. *Journal of Information Security and Applications* 61 (Sept. 2021), 102903. https://doi.org/10.1016/j.jisa.2021.102903

[8] Jonathan Esteban. 2022. Simulating Network Lateral Movements through the CyberBattleSim Web Platform. https://dspace.mit.edu/handle/1721.1/143191 Accepted: 2022-06-15T13:02:28Z.

[9] Myles Foley and Sergio Maffeis. 2022. Haxss: Hierarchical Reinforcement Learning for XSS Payload Generation. In *2022 IEEE International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*. IEEE, Wuhan, China, 147–158. https://doi.org/10.1109/TrustCom56396.2022.00031

[10] Hado van Hasselt, Arthur Guez, and David Silver. 2016. Deep Reinforcement Learning with Double Q-Learning. *Proceedings of the AAAI Conference on Artificial Intelligence* 30, 1 (March 2016). https://doi.org/10.1609/aaai.v30i1.10295 Number: 1.

[11] Matthew Hausknecht and Peter Stone. 2017. Deep Recurrent Q-Learning for Partially Observable MDPs. http://arxiv.org/abs/1507.06527 arXiv:1507.06527 [cs].
[12] Jaromír Janisch, Tomáš Pevný, and Viliam Lisý. 2023. NASimEmu: Network Attack Simulator & Emulator for Training Agents Generalizing to Novel Scenarios. https://doi.org/10.48550/arXiv.2305.17246 arXiv:2305.17246 [cs].
[13] L. P. Kaelbling, M. L. Littman, and A. W. Moore. 1996. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research* 4 (May 1996), 237–285. https://doi.org/10.1613/jair.301
[14] Kalle Kujanpää, Willie Victor, and Alexander Ilin. 2021. Automating Privilege Escalation with Deep Reinforcement Learning. In *Proceedings of the 14th ACM Workshop on Artificial Intelligence and Security*. 157–168. https://doi.org/10.1145/3474369.3486877 arXiv:2110.01362 [cs].
[15] Soyoung Lee, Seongil Wi, and Sooel Son. 2022. Link: Black-Box Detection of Cross-Site Scripting Vulnerabilities Using Reinforcement Learning. In *Proceedings of the ACM Web Conference 2022*. ACM, Virtual Event, Lyon France, 743–754. https://doi.org/10.1145/3485447.3512234
[16] Zegang Li, Qian Zhang, and Guangwen Yang. 2023. *EPPTA: Efficient Partially Observable Reinforcement Learning Agent for Penetration testing Applications*. preprint. Preprints. https://doi.org/10.22541/au.169406476.64066230/v1
[17] Ryusei Maeda and Mamoru Mimura. 2021. Automating post-exploitation with deep reinforcement learning. *Computers & Security* 100 (Jan. 2021), 102108. https://doi.org/10.1016/j.cose.2020.102108
[18] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.
[19] George E. Monahan. 1982. State of the Art—A Survey of Partially Observable Markov Decision Processes: Theory, Models, and Algorithms. *Management Science* 28, 1 (Jan. 1982), 1–16. https://doi.org/10.1287/mnsc.28.1.1
[20] Hoang Nguyen, Songpon Teerakanok, Atsuo Inomata, and Tetsutaro Uehara. 2021. The Proposal of Double Agent Architecture using Actor-critic Algorithm for Penetration Testing:. In *Proceedings of the 7th International Conference on Information Systems Security and Privacy*. SCITEPRESS - Science and Technology Publications, Online Streaming, — Select a Country —, 440–449. https://doi.org/10.5220/0010232504400449
[21] Thanh Thi Nguyen and Vijay Janapa Reddi. 2023. Deep Reinforcement Learning for Cyber Security. *IEEE Transactions on Neural Networks and Learning Systems* 34, 8 (Aug. 2023), 3779–3795. https://doi.org/10.1109/TNNLS.2021.3121870 Conference Name: IEEE Transactions on Neural Networks and Learning Systems.
[22] Aleksei Petrenko, Zhehui Huang, Tushar Kumar, Gaurav S. Sukhatme, and Vladlen Koltun. 2020. Sample Factory: Egocentric 3D Control from Pixels at 100000 FPS with Asynchronous Reinforcement Learning. In *Proceedings of the 37th International Conference on Machine Learning, ICML 2020, 13-18 July 2020, Virtual Event (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 7652–7662. http://proceedings.mlr.press/v119/petrenko20a.html
[23] Van-Hau Pham, Hien Do Hoang, Phan Thanh Trung, Van Dinh Quoc, Trong-Nghia To, and Phan The Duy. 2023. Raij\=u: Reinforcement Learning-Guided Post-Exploitation for Automating Security Assessment of Network Systems. https://doi.org/10.48550/arXiv.2309.15518 arXiv:2309.15518 [cs].
[24] Rapid7. [n. d.]. Metasploit. https://www.metasploit.com/.
[25] Carlos Sarraute, Olivier Buffet, and Jörg Hoffmann. 2012. POMDPs Make Better Hackers: Accounting for Uncertainty in Penetration Testing. *Proceedings of the AAAI Conference on Artificial Intelligence* 26, 1 (2012), 1816–1824. https://doi.org/10.1609/aaai.v26i1.8363 Number: 1.
[26] Carlos Sarraute, Olivier Buffet, and Joerg Hoffmann. 2013. Penetration Testing == POMDP Solving? http://arxiv.org/abs/1306.4714 arXiv:1306.4714 [cs].
[27] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. 2016. Prioritized Experience Replay. https://doi.org/10.48550/arXiv.1511.05952 arXiv:1511.05952 [cs].
[28] John Schulman, Filip Wolski, Prafulla Dhariwal, Alec Radford, and Oleg Klimov. 2017. Proximal Policy Optimization Algorithms. https://doi.org/10.48550/arXiv.1707.06347 arXiv:1707.06347 [cs].
[29] Jonathon Schwartz and Hanna Kurniawati. 2019. Autonomous penetration testing using reinforcement learning. *arXiv preprint arXiv:1905.05965* (2019).
[30] Jonathon Schwartz and Hanna Kurniawatti. 2019. NASim: Network Attack Simulator. https://networkattacksimulator.readthedocs.io/.
[31] Kamran Shaukat, Suhuai Luo, Vijay Varadharajan, Ibrahim A. Hameed, and Min Xu. 2020. A Survey on Machine Learning Techniques for Cyber Security in the Last Decade. *IEEE Access* 8 (2020), 222310–222354. https://doi.org/10.1109/ACCESS.2020.3041951
[32] Maxwell Standen, Martin Lucas, David Bowman, Toby J. Richer, Junae Kim, and Damian Marriott. 2021. CybORG: A Gym for the Development of Autonomous Cyber Agents. https://doi.org/10.48550/arXiv.2108.09118 arXiv:2108.09118 [cs].
[33] Richard S. Sutton and Andrew G. Barto. 2018. *Reinforcement Learning: An Introduction*. A Bradford Book, Cambridge, MA, USA.
[34] Isao Takaesu. [n. d.]. DeepExploit. https://github.com/13o-bbr-bbq/machine_learning_security/tree/master/DeepExploit.
[35] Microsoft Defender Research Team. 2021. CyberBattleSim. https://github.com/microsoft/cyberbattlesim. Created by Christian Seifert, Michael Betser, William Blum, James Bono, Kate Farris, Emily Goren, Justin Grana, Kristian Holsheimer, Brandon Marken, Joshua Neil, Nicole Nichols, Jugal Parikh, Haoran Wei..
[36] Khuong Tran, Ashlesha Akella, Maxwell Standen, Junae Kim, David Bowman, Toby Richer, and Chin-Teng Lin. 2021. Deep hierarchical reinforcement agents for automated penetration testing. http://arxiv.org/abs/2109.06449 arXiv:2109.06449 [cs].
[37] Peter Vamplew, Benjamin J. Smith, Johan Källström, Gabriel Ramos, Roxana Rădulescu, Diederik M. Roijers, Conor F. Hayes, Fredrik Heintz, Patrick Mannion, Pieter J. K. Libin, Richard Dazeley, and Cameron Foale. 2022. Scalar reward is not enough: a response to Silver, Singh, Precup and Sutton (2021). *Autonomous Agents and Multi-Agent Systems* 36, 2 (July 2022), 41. https://doi.org/10.1007/s10458-022-09575-5
[38] Yongjie Wang, Yang Li, Xinli Xiong, Jingye Zhang, Qian Yao, and Chuanxin Shen. 2023. DQfD-AIPT: An Intelligent Penetration Testing Framework Incorporating Expert Demonstration Data. *Security and Communication Networks* 2023 (May 2023), e5834434. https://doi.org/10.1155/2023/5834434 Publisher: Hindawi.
[39] Christopher J. C. H. Watkins and Peter Dayan. 1992. Q-learning. *Machine Learning* 8, 3 (May 1992), 279–292. https://doi.org/10.1007/BF00992698
[40] Yizhou Yang and Xin Liu. 2022. Behaviour-Diverse Automatic Penetration Testing: A Curiosity-Driven Multi-Objective Deep Reinforcement Learning Approach. http://arxiv.org/abs/2202.10630 arXiv:2202.10630 [cs].
[41] Shicheng Zhou, Jingju Liu, Dongdong Hou, Xiaofeng Zhong, and Yue Zhang. 2021. Autonomous Penetration Testing Based on Improved Deep Q-Network. *Applied Sciences* 11, 19 (Jan. 2021), 8823. https://doi.org/10.3390/app11198823 Number: 19 Publisher: Multidisciplinary Digital Publishing Institute.