JAIST Repository
https://dspace.jaist.ac.jp/

Title	Realistic Pentesting Training Framework for Reinforcement Learning Agents
Author(s)	Nguyen, Huynh Phuong Thanh
Citation	
Issue Date	2024-09
Type	Thesis or Dissertation
Text version	author
URL	http://hdl.handle.net/10119/19357
Rights	
Description	Supervisor: BEURAN, Razvan Florin, 先端科学技術研究科, 修士(情報科学)

JAIST
JAPAN
ADVANCED INSTITUTE OF
SCIENCE AND TECHNOLOGY

Japan Advanced Institute of Science and Technology

Master's Thesis

Realistic Pentesting Training Framework for Reinforcement Learning Agents

2210406 NGUYEN Huynh Phuong Thanh

Supervisor: Associate Professor BEURAN Razvan

Graduate School of Advanced Science and Technology
Japan Advanced Institute of Science and Technology
(Information Science)

September, 2024

Acknowledgment

"When you want to give up, remember why you started."

Words cannot express my gratitude to my supervisor, Associate Professor Beuran Razvan, for warmly welcoming me to the Cybersecurity Laboratory and for his invaluable patience and feedback. These have helped me become a better version of myself day by day. I am thankful for his belief in me, the opportunity he provided, and for accepting me as a Master's student even though I had no background in network security and was a recent undergraduate. I appreciate being diligently guided in conducting research, identifying and solving problems, and exploring new insights through scientific publications. His deep commitment to his students and valuable guidance motivates me to persist in my research endeavors, even when faced with difficulties.

I sincerely thank Prof. Tan, my second supervisor, and Prof. Shinoda, Assoc. Prof. Lim, and Assoc. Prof. Uda, who are my defense examiners. I could not undertake this journey without them, as they generously provided valuable expertise and suggestions. I am deeply indebted to my minor research supervisor, Prof. Sakti, for advising me on completing my research in the speech field. Her guidance and support have given me new research directions and helped with the presentation of my research publication.

I had the pleasure of collaborating with KDDI Research, Inc. Their valuable comments and suggestions and their support in writing papers significantly improved and completed my research work.

Many thanks to all sections of JAIST for their support in documentation work and consultancy. I am also grateful for the tuition fee reduction and scholarship for financial support from JAIST. I could not have completed my research work well without their help.

I'd like to acknowledge my previous advisor, Assoc. Prof. Kha Tu Huynh, and my senior, Dr. Duy Tan Le, at IU VNU-HCM University, for introducing me and allowing me to become a Master's student at the Cybersecurity Laboratory. These changes have changed my mind about research work and research environments. I extend my thanks to all lab members and my friends, who were warm and caring throughout my time here. My life in Japan and JAIST would not be memorable and meaningful without them.

Lastly, I would be remiss not to mention my family for their constant motivation and encouragement. Their belief in me has kept my strength high throughout this process. I also thank myself for continuing to move forward and never giving up, even facing challenges, mistakes, and failures during this journey. As this unforgettable Master's journey concludes, a new chapter in my research journey begins.

Abstract

Penetration testing, or pentesting, refers to assessing and enhancing network system security by trying to identify and exploit any existing vulnerabilities. This is one of the critical methods widely used by organizations to strengthen their defenses against malicious attacks. The pentester executed an authorized attack on the network systems to gain administrator permissions, allowing them to evaluate overall security characteristics. However, traditional manual pentesting raises several challenges and becomes ineffective due to its time-consuming nature and the need for technical security skills. With modern network systems becoming more complex and threats increasingly sophisticated, manual pentesting can not adapt slowly.

Applying Artificial Intelligence (AI) techniques to the cybersecurity domain is proposed to solve this problem and enable the automation of pentesting procedures. Reinforcement Learning (RL) was created as an AI optimization technique to produce an attack policy that learns the best strategy through environmental interaction. Due to this learn-via-interaction mechanism, it has recently become an effective method for creating autonomous pentesting agents. These agents are trained to replicate the actions of humans with enhanced speed, scale, precision, and automation. Many proposed approaches involve using simulation environments to train pentesting RL agents. The main advantages of these studies are their speed, low resource consumption, and ease of design, making them a potential solution to shift from manual to automated procedures.

However, due to the logical modeling of attack actions and observations, there is a heavy reliance on predefined constants and probabilistic values for agent actions and environment states. This can lead to inaccuracies in replicating real-world behavior due to unexpected factors, decreasing agent accuracy and performance. Additionally, the simulated network may not accurately represent the configuration and topology of an actual network. Thus, simulation environments for training RL pentesting agents present challenges when deploying them in actual network infrastructure due to the lack of realism in the simulation-trained agents.

We propose PenGym, a framework for training pentesting RL agents in realistic environments, to address this issue. The most significant features of PenGym are its support for real pentesting actions, full automation of the network environment creation, and good execution performance. PenGym

covers network discovery and host-based exploitation actions that are available to train, test, and validate RL agents in an emulated network environment. Compared to typical simulation-based agent training, the main advantage is that PenGym can execute actual actions in a real network environment while providing a reasonable training time. We conducted several experiments to demonstrate the effectiveness of using PenGym as a realistic training environment compared to a simulation approach (NASim).

For the smallest scenario, agents trained in both simulation and emulation environments achieved equivalent results with minor differences. For the mid-size scenario, simulation-related limitations occurred. Although the agents trained in the NASim environment performed well in the simulation environment, they were ineffective when tested in the emulation environment, showing high variation and large attack step counts. In contrast, after fixing logical modeling issues in the simulation to create the revised version called NASim(rev.), testing results were comparable to PenGym in both the simulation and emulation environments.

For the largest scenario, the effectiveness of the PenGym approach is emphasized. Agents trained in the original NASim environment behaved poorly when tested in a realistic environment, having a high failure rate. In contrast, agents trained in PenGym successfully reached the pentesting goal in all our trials. Even though NASim(rev.) was revised with a more accurate model, experiment results with the largest scenario indicated that agents trained in PenGym slightly outperformed and were more stable than those trained in NASim(rev.). Thus, the average number of step differences required to reach the pentesting goal ranged from 1.4 to 8, which is better for PenGym. Consequently, PenGym provides a reliable and realistic training environment for pentesting RL agents, eliminating the need to model agent actions via simulation. This finding justifies the use of realistic environments for creating and training RL agents for pentesting purposes.

Regarding time performance, due to the actual action execution on the cyber range, PenGym requires more training time than simulation environments. However, it provides a reasonable training time in more complex scenarios while preserving realism and feasibility as real networks become more sophisticated. In particular, for the most complicated scenario and most intricate RL algorithm, PenGym training takes around $17,000$ s compared to $14,000$ s in NASim, with a ratio of roughly 1.2.

Future work involves proposing a realistic automatic scenario generator to assist in constructing a realistic pentesting scenario for training RL agents.

Keywords: Penetration Testing, Reinforcement Learning, Agent Training Environment, Realistic Environment, Cyber Range, Cybersecurity

Contents

1 Introduction **1**
- 1.1 Traditional Pentesting 1
- 1.2 Reinforcement Learning for Pentesting 2
- 1.3 Problem Statement 2
- 1.4 Contributions 3
- 1.5 Thesis Structure 4

2 Background **5**
- 2.1 Pentesting Mechanisms and RL Overview 5
- 2.2 Motivation 7

3 Literature Review **9**
- 3.1 Pentesting and Related Tools 11
- 3.2 RL Training Environments 11
  - 3.2.1 Simulation Environments 12
  - 3.2.2 Emulation Environments 13
- 3.3 Cyber Range Creation for Pentesting 14

4 PenGym Framework **15**
- 4.1 PenGym Overview 15
- 4.2 Action Implementation 17
  - 4.2.1 Service Scan 19
  - 4.2.2 OS Scan 21
  - 4.2.3 Subnet Scan 22
  - 4.2.4 Exploit 23
  - 4.2.5 Process Scan 24
  - 4.2.6 Privilege Escalation 25
- 4.3 Action Optimizations 26
  - 4.3.1 Single Action Optimization 27
  - 4.3.2 Training Time Optimization 29
- 4.4 Cyber Range Creation 31

4.4.1 Cyber Range Composition 31
4.4.2 Cyber Range Description Generation 32
4.4.3 Bridge Functionality 33

5 Functionality Validation 35
5.1 Action Implementation Validation 35
5.1.1 Service Scan 35
5.1.2 OS Scan 36
5.1.3 Subnet Scan 37
5.1.4 Exploit 37
5.1.5 Process Scan 37
5.1.6 Privilege Escalation 38
5.2 Cyber Range Creation Validation 38
5.2.1 Configuration Validation 38
5.2.2 Creation Time 39

6 Experiment Results 40
6.1 Experiment Scenarios 40
6.1.1 Tiny Scenario 41
6.1.2 Small Scenario 41
6.1.3 Medium Scenario 42
6.2 Preliminary Experiments 43
6.3 Detailed Experiments 46
6.3.1 Agent Training 46
6.3.2 Agent Testing 48

7 Discussion 53
7.1 Comparative Analysis of Simulation and Emulation Approaches 53
7.1.1 Host Configuration 53
7.1.2 Actions 55
7.1.3 Observations 56
7.2 Simulation Modeling Issues 56
7.2.1 Firewall Functionality Issue 56
7.2.2 Scan Action Issue 57
7.2.3 Remote Action Issue 58

8 Conclusion 59
8.1 Summary 59
8.2 Future Work 60

List of Figures

2.1 Pentesting execution process in practice. 6
2.2 Application of RL in pentesting procedure. 7
2.3 Limitations of using a simulation environment. 8

4.1 Overview of PenGym architecture. 16
4.2 Flowchart of the training time optimization mechanism. 29
4.3 Processing flow of the CyRIS cyber range description file generation module. 33

6.1 Cyber range constructed in PenGym based on the tiny scenario in NASim. 42
6.2 Cyber range constructed in PenGym based on the small-linear scenario in NASim. 43
6.3 Cyber range constructed in PenGym based on the medium-multi-site scenario in NASim. 44
6.4 Average training reward versus episode number for the QLearning and QLearning Replay agents trained in the NASim, NASim(rev.), and PenGym environments for the three representative experiment scenarios. 49
6.5 Average number of attack steps for the trained QLearning and QLearning Replay agents when tested in the NASim* and PenGym environments for the three representative experiment scenarios (NASim* refers to either NASim or NASim(rev.), depending on the simulator variant used for training). 50

7.1 Visualization of the firewall functionality issue. 57
7.2 Visualization of the scan action issue. 57
7.3 Visualization of the remote action issue. 58

List of Tables

3.1 Comparison of related frameworks from abstraction level and environment feature perspectives. 10

4.1 Comparison of action implementation mechanisms between NASim and PenGym environment 19
4.2 Summary of the methods and Metasploit modules used for the PenGym action space implementation. 20
4.3 Summary of the implementation and validation of action optimization techniques. 27
4.4 Description of the parameters used to specify the composition of a cyber range in PenGym. 32

5.1 Action validations between NASim and PenGym 36
5.2 Cyber range creation time for all scenarios 39

6.1 Overview of all the experiment scenarios in PenGym. 41
6.2 Results of preliminary experiments conducted for all the available scenarios (NASim* refers to either NASim or NASim(rev.), according to the simulator variant used for training). 45
6.3 Detailed experiment results for the QLearning and QLearning Replay agents trained and tested in the NASim, NASim(rev.) and PenGym environments using the medium-multi-site scenario. 51

7.1 Summary of the differences between the PenGym and NASim environments. 54

List of Algorithms

1 Service Scan Action 21
2 OS Scan Action 22
3 Subnet Scan Action 23
4 Exploit Action 24
5 Process Scan Action 25
6 Privilege Escalation Action 26
7 Training Time Optimization 30
8 Bridge Functionality 34

Chapter 1

Introduction

The development of the interconnected digital world, where the Internet is central, has changed how businesses, governments, and individuals operate, leading to significant economic and social benefits [1]. This has enhanced communication and created more opportunities for cybercriminals to launch attacks to steal sensitive data. These attacks can range from massive state-sponsored attempts to disrupt events like the US election to simple attacks on individuals to gain passwords or credit card details for monetary gain [2, 3]. As more people rely on globally connected computer systems, securing these systems against malicious attacks is becoming increasingly important [4]. Ensuring the security of network systems and infrastructure is a critical aspect of cybersecurity [5, 6]. An effective method and technologies are needed to secure computer systems against these threats.

1.1 Traditional Pentesting

Penetration testing, or pentesting, is widely considered to be the most effective method for evaluating network security [7]. This approach involves authorized network attacks on computer systems to identify vulnerabilities that could be exploited by an attacker and assess their overall security characteristics [8, 9]. However, traditional pentesting, which is conducted manually, poses several challenges, as it has high demands in terms of time and technique security skills. This high cost will become a big challenge as digital systems grow larger and more complex, causing a demand for security professionals that is not being met immediately.

Given the lack of skilled resources and the necessity for pentesting in network security, several tools have been developed to assist pentesters and improve their efficiency. They include network and vulnerability scanners,

as well as libraries of known security vulnerabilities. One of the most popular tools is the open-source Metasploit framework [10]. The Metasploit framework contains a rich library of known system vulnerability exploits and other useful tools, such as scanners for information gathering on a target. Pentesters can focus on finding vulnerabilities and selecting exploits rather than manually developing them. This enables faster work and makes security assessment more accessible to non-experts [11].

1.2 Reinforcement Learning for Pentesting

Nowadays, due to the inability to conduct manual pentesting in complicated networks, Artificial Intelligence (AI) is used to achieve efficient and reliable pentesting techniques to automate the process. The original concept for this took the form of "attack graphs", which modeled an existing computer network as a graph of connected computers. Attacks can then be simulated on the network using known vulnerabilities and exploits [12]. Attack graphs can effectively identify possible ways to breach a system. However, using these graphs requires complete knowledge of the system, which is unrealistic from a real-world point of view of attackers. Additionally, the manual construction of the attack graph for each system being assessed is required.

Recently, Reinforcement Learning (RL) [13] appeared as a promising solution to bridge this gap by automating pentesting. The use of RL gained lots of attention in recent years with its success in producing World Go champion-beating agents [14]. Although it is not as widely applied as other supervised machine learning approaches, it has been successfully used in several realistic robotics tasks [15]. RL pentesting agents are designed to mimic the actions of human pentesters but with enhanced speed, scale, and precision. This is achieved by enabling the RL agents to navigate complex network environments, detect vulnerabilities, and exploit them to evaluate security risks. Fundamentally, through a process of trial and error, the RL agents learn to optimize their actions by adapting to various environment challenges [16].

1.3 Problem Statement

Regarding the application of RL techniques to enhance the pentesting process, many simulation environments have been introduced to train RL agents to automate these tasks. These environments utilize logical modeling for both the agents and network environments. Simulators provide an in-memory abstraction of processes in real computer networks, making them faster and

easier to use. However, their heavy reliance on predefined constants and probabilistic values for agent actions and environment states leads to potential inaccuracies in replicating real-world behavior due to unexpected factors, decreasing agent accuracy and performance. For example, the authors of CyberBattleSim themselves argue that their framework is too simplistic to be used in the real world [17]. This means that agent performance may suffer when used with real networks due to the differences with the simulated environment. In particular, the translation of simulated actions (e.g., exploits, privilege escalation) to real actions is not trivial. Additionally, the simulated network may not accurately represent the configuration and topology of an actual network. As a result, creating and operating realistic environments for the training of pentesting AI agents is crucial.

1.4 Contributions

In this research, we present an effective and reliable realistic training framework for RL pentesting agents, named PenGym. It enables RL agents to execute actual actions on hosts in a network environment (a.k.a cyber range) dedicated to agent training. Thus, it eliminates the need to model agent actions via execution assumptions and success probability. The applicability of PenGym was shown in several perspectives, as follows: i) Support for various complex pentesting scenarios; (ii) Full automation of cyber range creation by leveraging the tools presented in [18]; (iii) Improved execution performance by applying optimization mechanisms to minimize the execution time for each action, as well as the training time. Moreover, several experiments with more complex and challenging pentesting scenarios were conducted to demonstrate the usefulness and effectiveness of using PenGym compared to simulation, particularly the simulation environment named NASim [19].

The realism of PenGym comes from the use of real actions in the action space, and the use of a real network, including actual hosts created using KVM technology. These hosts contain actual services, processes, operating systems, and vulnerabilities. The actions can be executed on these hosts to obtain interactive shell objects. We aim to achieve a realistic RL training environment with these real characteristics to deploy the trained agents in actual infrastructure.

Using PenGym, security researchers and practitioners can train RL agents to perform pentesting tasks in a safe and controlled environment, thus obtaining more realistic results than simulation but without the risks associated with real network pentesting. By providing the environment for executing actions, the framework assists in evaluating and comparing the effectiveness

of various pentesting RL techniques in real network environments.

Our contributions cover a wide variety of aspects, as summarized below:

1. Discuss the advantages and motivations of using RL techniques for pentesting purposes and a realistic training environment approach for pentesting agents, as opposed to simulation (Chapter 2)

2. Present the design and implementation of PenGym, with a particular focus on the action implementation that represents its key feature, action execution and training time optimization, and automated cyber range creation (Chapter 4)

3. Validate PenGym functionality and cyber range creation in various pentesting scenarios (Chapter 5).

4. Present set of experiments that demonstrate the effectiveness of using PenGym compared to NASim in different complicated pentesting scenarios (Chapter 6)

5. Propose a revision of NASim, named NASim(rev.), that fixes all the unrealistic modeling issues that we identified (Chapter 7)

1.5 Thesis Structure

This thesis is organized as follows. Chapter 2 provides the background of pentesting mechanisms, an overview of RL, and the application of RL in the pentesting area. This chapter also clarifies the motivation for using RL and a realistic environment for pentesting. Chapter 3 covers related works regarding pentesting and assisting tools and training environments for RL, especially simulation and emulation approaches, with a detailed comparison to our proposed approach. Moreover, the cyber range creation process for pentesting preparation in existing studies is included. Chapter 4 introduces our framework, PenGym, with a detailed description of action implementation, performance optimization, and automated cyber range creation. The functionality validation and full-scale experiment results are presented in Chapters 5 and 6, respectively. Chapter 7 provides a comparative analysis between simulation and emulation approaches, pointing out the modeling issues within the simulation environment and presenting our solution. The thesis ends with a conclusion and references.

Chapter 2

Background

This chapter discusses the background of pentesting mechanisms and provides an overview of the RL technique. It clarifies the relationship between RL and the pentesting process. This also mentions the advantages and motivation of using a realistic training environment to train RL pentesting agents.

2.1 Pentesting Mechanisms and RL Overview

From a practical perspective, pentesting is a multi-stage process that demands high competence and technical expertise due to the complexity of computer networks [20]. As illustrated in Figure 2.1, a pentester needs to gather information about the target system and determine the type of attack to execute. The target system then responds to its attack action. After that, the pentester analyzes this response to choose an appropriate next step. This continues until testing is completed and a report is created.

As alluded to earlier, pentesting is a process that relies on learning from the current state or observation of an action to decide the next step. Without previous experience, a brute force or random approach is used to explore all possible testing methods for the system. However, this approach can become overwhelming as the network grows in complexity and size, making it impossible for the real infrastructures.

Cybersecurity researchers have recently identified the need for an intelligent pentesting framework to support human experts by handling high-demand tasks, including information gathering, vulnerability assessment, and exploitation [21, 22]. RL is considered a promising approach in this area, as it allows an automated pentesting system to operate and gain skills gradually, similar to a real tester. Moreover, RL algorithms have been proven to work well for cybersecurity [23]. In particular, RL agents can leverage their

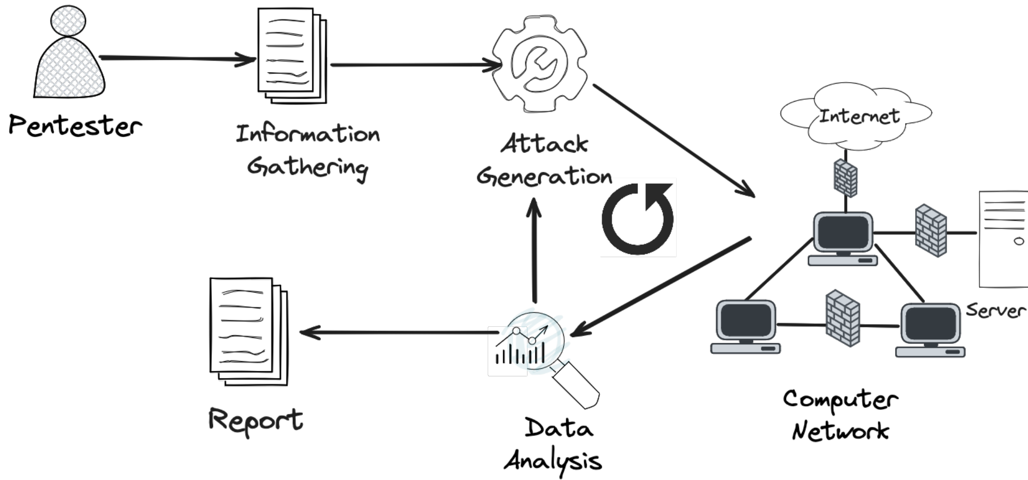


Figure 2.1: Pentesting execution process in practice.

exploration capabilities to discover previously unknown attack scenarios.

Theoretically, an RL agent interacts with its environment by performing actions and observing the results. Each action results in positive or negative feedback, depending on its outcome. Through trial and error, the agent learns an optimal policy to solve sequential decision-making problems [13]. This process starts with an initial, typically random, policy. It then iteratively learns the values of taking certain actions in given states, $Q(s, a)$. This is done by choosing an action based on the current policy, applying that action to the environment, and then updating the state-action value, $Q(s, a)$, based on the received experience.

Figure 2.2 displays the procedure of using RL for pentesting. An RL agent interacts with a training environment over time. At each time step t, the agent receives a state $s(t)$ in a state space S that details the environment, such as network topology, host configuration, etc. It then selects an action $a(t)$ from an action space A, according to a learning policy that maps the state $s(t)$ to an action $a(t)$. After acting, the agent receives a $r(t)$ reward and transitions to the next state $s(t+1)$. The reward $r(t)$ is based on the quality of the results from the action. The agent aims to optimize the learning policy by selecting the most suitable action at each step to achieve its objective, which is for pentesting purposes to gain unauthorized access to the system.

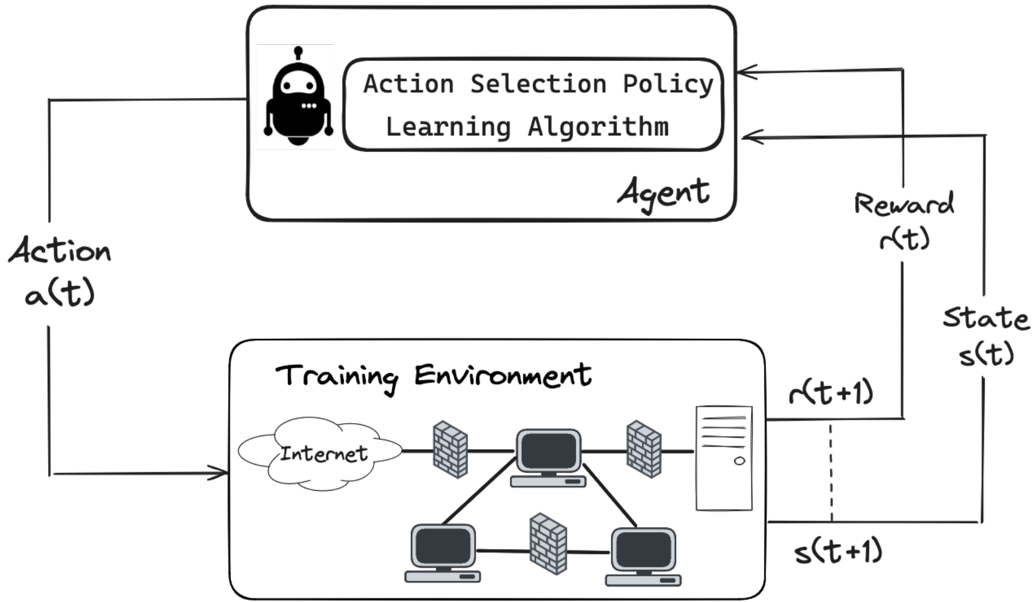


Figure 2.2: Application of RL in pentesting procedure.

2.2 Motivation

Several simulation frameworks have been proposed in recent years to create a training environment for RL pentesting agents. However, they may encounter functional issues caused by the modeling of agent actions and the required environment state. Figure 2.3 visualizes the potential limitations of using a simulation environment for training pentesting RL agents. The simulation environment attempts to model network characteristics and mimic real actions and observations by checking several logical conditions. Training the pentest agent in this environment poses high risks when deploying in a realistic setting. The root cause is the difficulty and complexity of accurately modeling a realistic network and real action behaviors.

The specific issues that we have identified for NASim [19], which is one of the most recent environments of this type, will be described in detail in Chapter 7. These problems indicate that the logic in NASim is incorrect for some actions, and the simulation mechanisms do not cover all situations, such as the lack of firewalls in some attacks. Several niche conditions may arise in more complex pentesting scenarios that reflect real-world networks. Handling these conditions efficiently with a list of condition statements is not always possible. This highlights the need for a realistic environment for training pentesting RL agents where observations are derived from real actions executed in an actual cyber range.

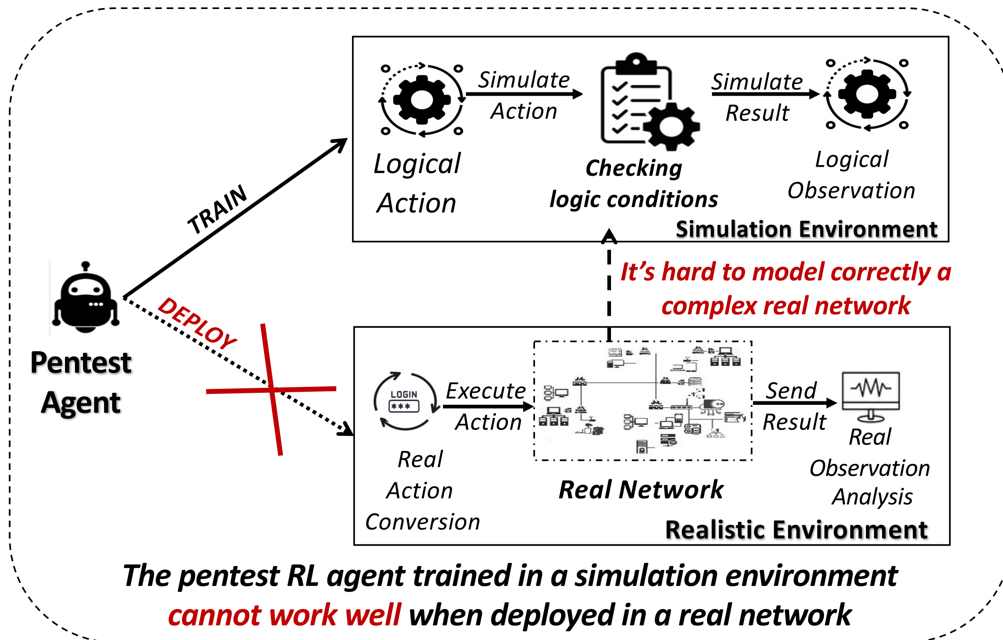


Figure 2.3: Limitations of using a simulation environment.

Furthermore, preparing the training environment in existing frameworks presents challenges due to the manual cyber range creation inside this environment. Creating complex networks involving numerous hosts with multiple services or processes is time-consuming and difficult. This issue motivated us to automate the creation of the cyber range.

Chapter 3

Literature Review

A growing number of research works have been published on automated pentesting topics by creating a training environment for RL pentesting agents. The use of RL to support the development of autonomous intelligent pentesting agents has become increasingly popular and efficient. Moreover, recent studies have explored the design and implementation of cyber range environments for conducting cyber attack simulations and for training RL agents in tasks such as intrusion detection, malware analysis, and penetration testing. Table 3.1 provides a summary of the most representative studies in this context and their characteristics when compared to PenGym.

All approaches are compared based on their abstraction level and execution environment features. Regarding the abstraction level, simulation-based approaches use a simulation environment to execute actions. In these approaches, actions are modeled by checking several required conditions and returning success if all the conditions are met [19]. On the other hand, emulation environments require actual hosts, an actual network topology, and agents that execute real actions on those hosts [24]. When considering execution environment features, the configurable elements are used for comparison, including features such as firewalls and host actions.

This chapter provides an overview of recent studies related to pentesting techniques and associated tools. The related works on RL training environments are discussed in detail, covering both simulation and emulation approaches, along with their advantages and limitations. Moreover, the process of creating a cyber range for pentesting is also included.

Table 3.1: Comparison of related frameworks from abstraction level and environment feature perspectives.

	Small World [25]	BRAWL [26]	GALAXY [27]	Smart Security Audit [28]	CyGIL [24]	Microsoft CBS [17]	CybORG [29]	FARLAND [30]	NASim [19]	NASimEmu [31]	PenGym
Abstraction Level											
Simulation Based						✓	✓	✓	✓	✓	✓
Real Hosts	✓	✓	✓	✓	✓		✓	✓		✓	✓
Real Network Topology	✓	✓	✓		✓		✓	✓		✓	✓
Real Actions	✓	✓	✓	✓	✓		✓	✓		✓	✓
Real Observations	✓	✓	✓	✓	✓		✓	✓		✓	✓
Designed for RL				✓	✓	✓	✓	✓	✓	✓	✓
Host-Based Exploitation	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Network-Based Exploitation	✓	✓	✓		✓			✓	✓	✓	✓
Environment Features											
Firewalls	✓								✓		✓
Network Scanning	✓	✓	✓		✓			✓	✓	✓	✓
Host Scanning (OS Scan, Process Scan, Service Scan)				✓					✓		✓
Exploits	✓	✓	✓	✓	✓	✓	✓		✓	✓	✓
Privilege Escalation	✓		✓	✓	✓	✓	✓		✓	✓	✓

3.1 Pentesting and Related Tools

Pentesting has existed for more than four decades and has become a critical process in the development life cycle from a system security perspective [32]. It involves performing an authorized and controlled attack on the network to evaluate its security posture. The process begins with information gathering, aiming to find vulnerabilities in the network and then exploiting them. This is repeated using the newly gained access until the target is reached [11]. Specifically, the information-gathering step typically uses tools such as port scanning and OS detection to collect system information. It can determine potential vulnerabilities that can be exploited. Then, the pentester executes an exploit action that takes advantage of the discovered vulnerabilities, causing unintended behavior in the system aimed at compromising the target and gaining privileged access to it. Even though the systems and networks evaluated using pentesting can differ significantly, the same general steps are followed in each case. This has allowed the development of various tools and frameworks to make pentesting more efficient.

Network scanner techniques are used to find useful information about the network, with the best-known tool being Nmap [33, 34]. It provides several functions to detect the OS, open ports, and services currently running on the system. The Metasploit framework launches exploits, enabling tactical operations rather than technical tasks. This automation increases efficiency during pentesting. The tools available to pentesters have significantly improved efficiency but still require considerable expertise and execution time.

3.2 RL Training Environments

Several frameworks have been developed for various pentesting purposes. SmallWorld [25] and BRAWL [26] utilize cloud-based infrastructure and virtualization technologies to create realistic scenarios for teaching and learning in security-related areas. Another study focused on designing a pentesting training environment. For instance, GALAXY [27] was proposed as an emulated cybersecurity environment for training red agents to blend with regular user traffic using evolutionary algorithms. However, these frameworks lack RL capabilities, essential for creating intelligent pentesting agents. Although some AI-assisted pentesting training environments have been introduced, such as those in [28, 35], they primarily focus on host-based exploitation and offer a limited range of goals and actions. In 2018, a training environment for network penetration testers based on a Partially Observable Markov Decision Process (POMDP) was proposed [22]. Still, the details about the

environment and RL training were not mentioned.

The research works related to pentesting frameworks include environments for security education (SmallWorld, BRAWL), as well as pentesting training environments using AI assistants and Markov decision processes. However, these frameworks lack detailed information on the RL training environment and have limited actions and scenarios. Simulation and emulation have emerged as the primary approaches for designing training environments where RL-based agents can automatically carry out attacks and enhance their learning algorithms.

3.2.1 Simulation Environments

Recently, RL methods to train automatic pentesting agents have become increasingly popular among researchers. There has been a rise in the simulation of pentesting environments for training and testing RL agents.

Network Attack Simulator (NASim) [19] proposed an RL agent training approach for network-wide penetration tests using the API of OpenAI Gym [36]. Using abstractions modeled with a finite state machine, NASim represents networks and cyber assets, including hosts, devices, subnets, firewalls, services, and applications. The simplified action space includes network and host discovery, service exploitation for each configured service vulnerability, and privilege escalation for each hackable process running in the network. The agent can simulate a simplified kill chain through discovery, privilege escalation, and service exploits across the network. However, NASim assumes that the simulated actions must satisfy various predefined conditions and uses probabilities to determine their success. While NASim contributes a comprehensive perspective on real-world pentesting environments, such as network characteristics and host configuration, it also leads to a *reality gap* due to its assumptions about pentesting actions. The success or failure of the action spaces is determined by various predefined conditions and probability factors.

Microsoft released CyberBattleSim (CBS) [17], an open-source RL agent network training environment built using the OpenAI Gym API. It is designed to train a 'red agent' that focuses on the lateral movement phase of a cyberattack in a simulated fixed network with predefined configured vulnerabilities. However, the authors have noted that due to its highly abstract nature, CBS cannot be directly applied to real-world systems.

Regarding the simulation-based approach, given NASim as an example, highlighting in Chapter 2, the use of simulation logical actions could lead to unexpected issues, resulting in unrealistic results and affecting the applicability of the agent in real-world networks. This is particularly evident when

dealing with complex scenarios that require numerous logical conditions.

3.2.2 Emulation Environments

Researchers have started studying an emulation approach to design a realistic RL-based network environment to overcome the limitations of using a simulation environment for training pentesting RL agents. It enables the ability to execute actual actions on real hosts, and the results are obtained based on how these hosts respond to the actions.

CyGIL [24] is an experimental testbed for emulated RL training that includes both network-level and host-level action space. It uses a stateless environment architecture and executes actions through the MITRE ATT&CK framework. While CyGIL employs real hosts and real actions, it omits the network traffic rules. Furthermore, it skips discovery phases in the pentesting process, such as host scanning. This can result in an unrealistic reflection of real-world conditions, where a service should be explored before executing service-based exploit actions.

Another framework, called CybORG [29], has been developed as a training platform for autonomous RL decision-making agents in adversarial scenarios. It provides simulation and emulation environments for training and testing agents. CybORG primarily focuses on developing an autonomous pentesting agent using RL and host-based exploitation. However, it does not consider network traffic discovery or connections between subnets. Furthermore, only the defense agent uses the RL method to learn attack strategies, while the attack agent follows a predefined attack flow.

The FARLAND framework [30] is another promising approach designed for training agents via simulation and testing agents via emulation. It provides attack and defense agents with traditional and deception actions, which can mislead the opposing agent and reduce its performance. Moreover, it offers functionality such as probabilistic state representations and support for adversarial red agents. However, rather than RL algorithms, red agents determine future actions through a probabilistic program. Moreover, FARLAND emphasizes network-based discovery over host-based exploitation.

Recently, NASimEmu [31] was introduced as the latest realistic framework for training RL pentesting agents and is based on NASim. This allows agents to be trained in simulation and then deployed in the emulator, thereby verifying the realism of the abstraction used. Actual exploits and privilege escalation actions are implemented in NASimEmu. However, as the authors stated, process scanning is still unavailable due to the lack of process configuration of virtual hosts. Furthermore, firewall rules are also a limitation in NASimEmu. Although the results show the ability to verify the realism of

trained agents in simulation by deploying in emulation, the action space is still limited and does not accurately reflect the available actions in the original NASim environment. The absence of a firewall and the limited actions potentially affect the experiment results when dealing with more complex scenarios. The logical conditions may encounter unexpected issues as well.

In summary, all the frameworks mentioned above have their limitations. Simulation environments designed for autonomous RL pentesting agents have a fast execution time and require limited resources. However, they contain modeled elements that may lead to unexpected issues, resulting in unrealistic results and affecting the applicability of a trained agent on actual networks. Other environments use an emulation approach to bridge the *reality gap* of simulation. Some support only host-based exploitation or network-based discovery actions. Moreover, some systems fail to consider the host configuration before attempting an attack or do not support external firewalls.

3.3 Cyber Range Creation for Pentesting

Cyber Range is a crucial component of the training environment. It sets up a network environment with multiple actual hosts connected following a specific scenario. The services, processes, vulnerabilities, and traffic rules from this scenario are configured within the Cyber Range, accurately reflecting the network architecture and host configuration of the training scenario. Previous studies have noted that some frameworks support full virtualization using KVM, but they lack automatic virtual interfaces and bridge setups [27, 29]. Some emulated frameworks require pre-built VMs and vulnerable applications to be built from scratch [25].

In our research, we use CyRIS to enable the automatic creation of the network environment. CyRIS, known for facilitating the automatic creation of network environments using KVM technology, covers both host configuration and network connection. Bridges and virtual interfaces can be created via CyRIS. A generation module is developed to connect CyRIS and PenGym, which provides mechanisms to convert the pentesting scenario into a CyRIS description for cyber range creation.

Considering the limitations of NASim and inspiration from emulation approaches, we developed PenGym as an effective and realistic pentesting training framework for RL agents. It provides network exploration and host-based exploitation actions, all of which are conducted in the real environment. The capability of using realistic environments for training pentesting RL agents is demonstrated through various network scenarios. Moreover, CyRIS was integrated with PenGym to automate the cyber range creation process.

Chapter 4

PenGym Framework

This chapter introduces the architecture and workflow of PenGym, along with a typical cycle of training or testing an RL pentesting agent. We detail the implementation of the entire action space, from both theoretical and technical perspectives. Additionally, we propose optimization mechanisms to minimize the time of action execution and the RL agent training process. The module implementation for automated cyber range creation is also described.

4.1 PenGym Overview

PenGym is a framework for creating and managing realistic environments aimed at training RL AI agents for pentesting purposes. It provides a network environment that supports various pentesting tasks, including scan, exploit, and privilege escalation actions, allowing RL agents to learn action selection strategies through interaction experiences. PenGym uses the same API with the OpenAI Gymnasium (formerly Gym) [36] library, thus making it possible to employ it with all the RL agents that follow those specifications. An overview of PenGym is shown in Figure 4.1, which consists of two main components: a training environment and the Cyber Range. In the training environment, the *Action/State Module* serves as a core element in communication between the agents and the environment. The automatic Cyber Range creation mechanism is integrated into PenGym using CyRIS. In this paper, NASim scenario files are utilized for the training environment and Cyber Range preparation.

The *Action/State Module* provides a range of actions that are operated on actual hosts, reflecting the pentesting in conditions similar to real-world scenarios. It is responsible for handling action execution and observation between agents and the environment, which has two main functionalities:

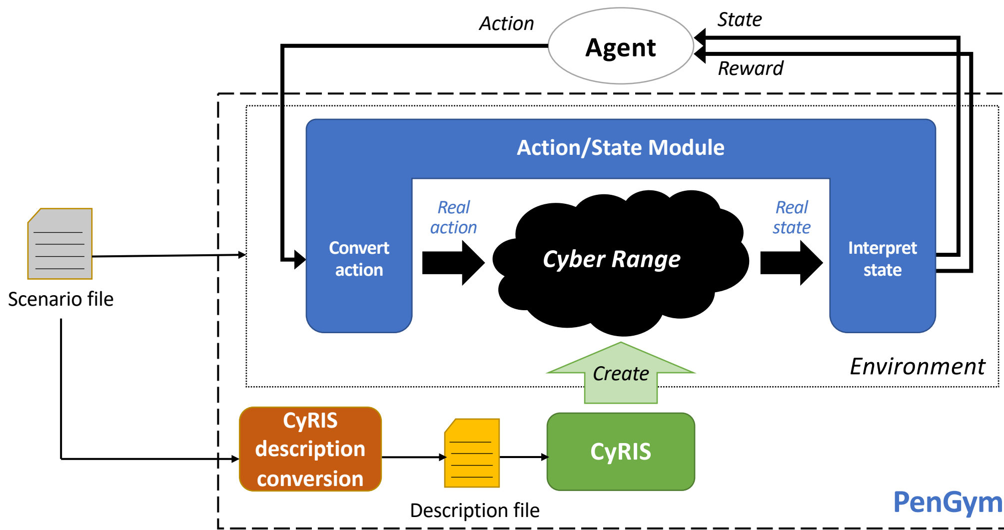


Figure 4.1: Overview of PenGym architecture.

- Convert the logical actions generated by the RL agent into executable realistic actions that are executed in the Cyber Range (an actual network environment used for cybersecurity training purposes).
- Interpret the actual results of the actions, then return the environment state and the reward to the agent, allowing for further learning.

The *Cyber Range* is created by leveraging CyRIS functionality. PenGym converts a scenario file into a CyRIS description file using a conversion module. Then, CyRIS automatically creates the Cyber Range based on this description. The architecture of a Cyber Range is determined based on the content of this scenario file to build an equivalent environment in PenGym.

A typical cycle of training or testing of an autonomous agent includes:

(i) PenGym resets the environment to get the start state

(ii) The RL agent chooses an action from the available space, using the current state and an algorithm suited to its learning objectives

(iii) PenGym converts this logical action from the RL agent into an executable real action, which is then executed in the Cyber Range environment setup using KVM virtual machines

(iv) PenGym interprets the actual observations, such as the status of available services, along with the new state of the environment, and sends back to the agent

(v) The RL agent utilizes the acquired state information and the corresponding reward to refine and enhance the underlying algorithm, ultimately fostering the agent learning and optimization process.

(vi) The agent selects the next appropriate action until the goal is reached or the step limit is exceeded, based on the updated learning policy

This complete cycle would be a single episode, and an agent would then reset the environment and repeat this for as many episodes as desired.

In PenGym, rewards are predefined in the scenario file, including the rewards for target hosts and each action. The reward for each action is determined by the complexity score generated by the Common Vulnerability Scoring System (CVSS). This score reflects the difficulty of finding an exploitable machine, the probability of success, and the skill required to use a given exploit [37]. The total reward is the value of all exploited machines minus the cost of actions performed.

The agents receive the rewards and update their learning algorithm to optimize strategies for the next action. Thus, if no machine is compromised, the reward is simply the cost of the action performed. With these reward rules, the goal of the agent is to compromise all target machines on the network while minimizing the number or cost of actions used. This simulates a real-world malicious attacker whose assumed goal is to retrieve privileged information or gain privileged access to the system.

4.2 Action Implementation

In NASim, the action space is a collection of feasible actions that an agent can execute. This encompasses a range of scan actions, including service scan, OS scan, process scan, and subnet scan, which allow for identifying potential vulnerabilities and access points in the network. The purpose of the scan actions is to imitate the capabilities of the Nmap utility for network discovery, and security auditing [38], such as providing details about the active services on every port of a specified host or operating system information.

Additionally, the action space includes an exploit action for each service and a privilege escalation action for each process. These actions enable the exploitation of vulnerabilities in a running service or process, leading to unauthorized access to the system. In NASim, the success of any exploit action is determined by checking the reachability of the target host, the availability of the target service, the permissions of firewall rules, and the predefined success probability of the action in the description file. The success probabilities of each exploit can be chosen. However, by default, probabilities are

randomly sampled from a distribution based on the attack complexity score distribution of the top 10 most common exploits used in 2017 [39], as stated in the NASim paper. Privilege escalation, on the other hand, simulates root-level access to the target host. Overall, the comprehensive action space in NASim provides a simulation of the tactics and techniques that attackers use to compromise network security.

The limitations of using a simulation environment include potential inaccuracies in replicating real-world behavior due to unexpected factors. Although the result of an action is determined by checking specific conditions in the description file, it is important to recognize that other factors can also impact the success of the action. Moreover, the network environment itself may not be accurately replicated in simulation due to the high complexity of a realistic system, including the configuration and topology of the network. Therefore, while simulation can provide valuable insights into network security, real-world execution is the only way to determine the effectiveness of penetration testing agents.

The *Action/State Module* in PenGym aims to bridge this gap by enabling the actual execution of RL agent actions in the target cyber range. The outcome of each action is determined based on the current status of the host in the network environment, reflecting the realistic conditions of the system. The actions currently implemented in PenGym cover the entire available actions of NASim, divided into six categories:

1. Service Scan
2. Operating System (OS) Scan
3. Subnet Scan
4. Exploit
5. Process Scan
6. Privilege Escalation

The real action implementation for Service Scan, OS Scan, and Subnet Scan is based on the Nmap utility to retrieve information about the services and operating systems running on each host and the accessible subnets. By leveraging this information, agents can better understand the overall system architecture and identify potential vulnerabilities. Specifically, we use the `python-nmap` library [40] to control Nmap execution via the Python programming language. As for Process Scan, it is implemented by using the `ps` command available in Linux operating systems.

The Exploit and Privilege Escalation actions are implemented using the Metasploit pentesting framework [10] to execute the corresponding actions on the real target hosts. They allow agents to interact with the target virtual machine and gain escalated access privileges, enabling them to perform more advanced attacks and compromise the system. In particular, we use the `pymetasploit3` library [41] to control Metasploit execution via Python.

Table 4.1 provides the comparison of action implementation, with differences highlighted in bold font and validation results between NASim and PenGym environments. All actions in NASim are replaced with actual actions utilizing `nmap` and `metasploit` as the main techniques in PenGym.

Table 4.1: Comparison of action implementation mechanisms between NASim and PenGym environment

Action	NASim Implementation	PenGym Implementation
Service Scan	Check if the host is discovered and return its configured services	Check if the host is discovered and **scan the running services using `nmap`**
OS Scan	Check if the host is discovered and return its configured OS type	Check if the host is discovered and **scan the OS type using `nmap`**
Process Scan	Check if the host is compromised and return its configured processes	Check if the host is compromised and **scan the running processes using `ps`**
Subnet Scan	Check if the host is compromised and return the connected hosts within the subnet	Check if the host is compromised and **scan the connected hosts within its subnet using `nmap`**
Exploit	Check if the host is discovered and probabilistically update its state to "compromised"	Check if the host is discovered and **execute a corresponding `metasploit` module to compromise it**
Privilege Escalation	Check if the host is compromised and update the access level of the host to ROOT	Check if the host is compromised and **execute a corresponding `metasploit` module to gain ROOT access**

Table 4.2 summarizes the implementation of all actions, showing for each of them the method used, the action description, the requirement, and the acquired access level for host-exploitation actions (i.e., the available user access permissions after successful exploitation).

4.2.1 Service Scan

Service scan is an essential cybersecurity technique employed to identify the various services running on a host system. It aims to identify the specific software applications running on a given machine and other essential details about each service. For instance, a service scan can help pinpoint which

Table 4.2: Summary of the methods and Metasploit modules used for the PenGym action space implementation.

Actions		Method/ Metasploit module	Description	Access Level
Scan Actions	service_scan	nmap scan	Scan the **running services** of the current machine	N/A
	os_scan	nmap scan	Scan the **OS type** of the current machine	
	process_scan	ps command line	Scan the **running processes** of the current machine	
	subnet_scan	nmap scan	Scan the **connected hosts within** the subnet of the current machine	
Exploit Actions	e_ssh	ssh_login [42]	Execute a **SSH dictionary attack** to get the user shell	USER
	e_ftp	vsftpd_234_backdoor [43]	Send the specific bytes on port 21 to **trigger the** vsf_sysutil_extra() function (CVE-2011-2523 [44])	**ROOT**
	e_http	apache_normalize_path_rce [45]	Exploit an unauthenticated Remote Command Execution vulnerability via the **misconfiguration of the** package (CVE-2021-41773 [46])	USER
	e_samba	is_known_pipename [47]	Exploit the Remote Command Execution vulnerability by **triggering an arbitrary shared library load vulnerability** in Samba (CVE-2017-7494 [48])	**ROOT**
	e_smtp	opensmtpd_mail_from_rce [49]	Exploit a **command injection** in the MAIL FROM field to execute a command as the root user (CVE-2020-7247 [50])	**ROOT**
Privilege Escalation Actions	pe_tomcat	cve_2021_4034_pwnkit_lpe_pkexec [51]	Exploit a **bug in the polkit pkexec binary** that relates to how it processes arguments (CVE-2021-4034 [52])	**ROOT**
	pe_proftpd	proftpd_133c_backdoor [53]	Exploit a **malicious backdoor** that was added to the ProFTPD download archive (CVE-2015-3306 [54])	**ROOT**
	pe_cron	cron_persistence [55]	Create a cron or crontab entry to **execute a payload**	**ROOT**

ports are open on a machine and which services are listening to those ports. Additionally, it can aid in detecting potential vulnerabilities in those services.

To implement the Service Scan action, we use the `nmap`, enabling PenGym to identify a wide range of services. The service scan will proceed once PenGym identifies the target host as reachable. This involves scanning the active services of the target host. PenGym attempts to connect to various services on the machine to gather data. Upon success, the list of running services on the target host is returned. By utilizing `nmap`, PenGym is able to

identify a wide range of services, including web servers, email servers, SSH services, and many others. This allows PenGym to perform a comprehensive analysis of the target system and identify any potential vulnerabilities that may exist. The pseudocode for the implementation is provided in Algorithm 1. Several arguments are used to obtain necessary information and minimize the execution time of Nmap, such as `-Pn`, `-n`, `-T5`, `-sS`, and `-sV` explained details in Section 4.3.

Algorithm 1 Service Scan Action

```
Require: host, nmap, port=False
 1: if port exist then
 2:     result ← nmap.scan(host,port,args ='Pn -n -sS -sV -T5')
 3: else
 4:     result ← nmap.scan(host,args='-Pn -n -sS -sV -T5')
 5: end if
 6: service_list ← list()
 7: for host in result do
 8:     existed_ports ← get_existed_ports(host)
 9:     for port in existed_ports do
10:         if host[port][state] is OPEN then
11:             service_list.append(service_name)
12:         end if
13:     end for
14: end for
15: Return service_list
```

4.2.2 OS Scan

OS scanning is used to identify the operating system running on a target host. It is a crucial component of pentesting as it can inform decisions regarding subsequent steps and tools to be used. This functionality works by sending a series of probes to the target machine and analyzing the responses to determine the characteristics of the operating system. The accuracy of the scan results depends on the response from the target machine and the effectiveness of the probing technique used.

The OS Scan action is also implemented by leveraging the `nmap` utility. This is accomplished in a manner similar to the service scan action. Once it

is confirmed that the host is reachable, PenGym initiates an OS scan of each IP address associated with the target host. The output indicates the success or failure of the action. When the scan is successful, it returns a potential operating system running on the target host; if the scan fails, it returns an appropriate failure message. The pseudocode for the implementation is provided in Algorithm 2. In addition to time-optimization arguments, the argument `-O` is used to activate OS detection.

Algorithm 2 OS Scan Action

Require: `host`, `nmap`, `port=False`

```
1: if port exist then
2:     result ← nmap.scan(host,port,args = '-Pn -n -O -T5'
3: else
4:     result ← nmap.scan(host,args='-Pn -n -O -T5'
5: end if
6: os ← get_os_name(result)
7: Return os
```

4.2.3 Subnet Scan

The subnet scan is used to identify the active hosts within a specified network range and determine the potential targets in that subnet. This scan is useful in scenarios where the attacker wants to identify all the hosts that are connected to the network or subnet. In PenGym, we used `nmap` to implement this action. It sends a `ping` message to each IP address within the specified network range of interconnected subnets to identify active neighboring hosts.

First, it checks the permission of the target host to perform a subnet scan. Once the target host is compromised, PenGym performs an actual scan to retrieve a list of active hosts within the target host subnet. This list includes discovered and newly discovered hosts so that new potential targets can be easily identified. If the execution fails, the system will provide detailed error messages that will help determine the root cause of the issue. After the successful completion of a subnet scan, the state of any newly discovered hosts will be updated to indicate that they are reachable within the network. Additionally, the state of the subnet connection itself will also be updated accordingly. The pseudocode for the implementation is provided in Algorithm 3. In addition to the time-optimization arguments, `--minparallel` and `--maxparallel` are used to enable the parallel probing of the hosts.

Algorithm 3 Subnet Scan Action

Require: subnet, nmap, port=False

```
1: host_list ← list()
2: if port exist then
3:    result ← nmap.scan(subnet,port,args = '-Pn -n -sS -T5
      --minparallel 100 --maxparallel 100')
4:    for host in result do
5:       if host['tcp'][port][state] is OPEN then
6:          host_list.append(host)
7:       end if
8:    end for
9: else
10:   result ← nmap.scan(host,args ='-Pn -n -sS -T5
      --minparallel 100 --maxparallel 100')
11:   for host in result do
12:      if host[status][state] is UP then
13:         host_list.append(host)
14:      end if
15:   end for
16: end if
17: Return host_list
```

4.2.4 Exploit

Exploits are techniques for finding and taking advantage of vulnerabilities in software or systems to gain unauthorized access or perform malicious actions. A successful exploit will result in the target machine becoming compromised, and further steps can be performed, such as stealing sensitive data, installing malware, or taking control of the system. Therefore, exploits are critical components of the penetration testing process for advancing toward a target.

In PenGym, the exploit action category contains five actions, each corresponding to a different service. Each action attacks the host based on the vulnerability of its respective service. They are implemented via `pymetasploit3` library. The success of an exploit action is determined by various factors, such as whether the target machine is reachable, whether the service is present, and whether the service is blocked by a firewall. First, PenGym checks the discovered target host and its traffic permissions. If these conditions are not

satisfied, a connection error will be returned. It is important to ensure that the exploit attack is conducted only on the intended target and has all permission for exploitation. Once the target host state is satisfied, the exploit attack uses the Metasploit module to gain user access to the system.

The shell object and the access level (typically `USER`) are returned upon successful completion. This shell object can execute commands to navigate the file system, and the access level is used to determine what actions are allowed or restricted for the current user. According to the implementation presented in Algorithm 4, depending on the specific service, the corresponding module is called with the required parameters to execute the action. For instance, the `e_smtp` action calls the module `smtp/opensmtpd_mail_from_rce` to gain unauthorized access through the vulnerability of the `smtp` service and returns a shell object that allows shell command execution.

Algorithm 4 Exploit Action

```
Require: host, service
 1: msfprc ← get_msfprc_client()
 2: shell ← check_shell_exist()
 3: if shell exist then
 4:     shell ← get_existed_shell_of_host()
 5:     access_level ← get_host_access_level(shell)
 6:     Return shell, access_level
 7: else
 8:     module ← get_exploit_module(service)
 9:     result ← execute_exploit_module(module, host)
10: end if
11: shell ← get_shell(result)
12: access_level ← get_host_access_level(shell)
13: Return shell, access_level
```

4.2.5 Process Scan

Process Scan enables pentesters to conduct a thorough security assessment by identifying the processes running on a target host. This is done by determining which vulnerabilities can potentially be exploited to gain further control of the host, such as via privilege escalation techniques. Note that process scanning requires access to the target host, which is executed after successfully gaining access to it.

The implementation in PenGym uses the shell object obtained through the exploit action. This shell object executes the `ps` command, which gath-

ers data about the processes on a host. A list of processes running on the target host is returned if successful. Before running the process scan, the system verifies that the host is compromised and the necessary access level has been achieved. The current access level should equal or exceed the `USER` level access required for this scan. PenGym generates a comprehensive and detailed list of active processes within the target host by executing a process scan. The results obtained from the process scan function can be leveraged to identify potential vulnerabilities in the host system and determine the best approach for gaining root privilege escalation to the target machine. The pseudocode for the implementation is provided in Algorithm 5.

Algorithm 5 Process Scan Action

```
1: shell ← get_shell_from_target_host()
2: if shell exist then
3:     shell.write('ps')
4:     result ← shell.read()
5:     process_list ← get_process_list(result)
6: end if
7: Return process_list
```

4.2.6 Privilege Escalation

Privilege escalation is an essential step in pentesting, in which one attempts to gain administrator (`ROOT`) access to the target system. By obtaining a higher access level than that of a regular user, a pentester gains complete control of the target system and can perform any actions on it. Privilege escalation is achieved by exploiting specific vulnerabilities or misconfigurations of the system to gain root-level access. Note that privilege escalation is conducted after successfully gaining regular user access to the target host.

In PenGym, the privilege escalation consists of three actions, each corresponding to a different process. Their success depends on several factors, including the level of access already obtained, the security controls, and the techniques used to bypass the controls. The privilege escalation action in PenGym requires checking the required access level and ensuring that the necessary permissions are in place before running the action. Similar to exploit actions, the attack module and the following parameters are changed to be compatible with the specific process. Each module gains unauthorized access to the target through different vulnerabilities, so the configuration of each module is adjusted accordingly. For instance, the `pe_cron` action calls

the `local/cron_persistence` module to attack the target host and gains root access due to a misconfiguration in the `cron` process. A shell object with root access is obtained as the outcome of this action. The pseudocode of the privilege escalation actions is shown in Algorithm 6.

Algorithm 6 Privilege Escalation Action

```
Require: host, process
 1: msfprc ← get_msfprc_client()
 2: root_shell ← check_root_shell_exist()
 3: if root_shell exists then
 4:     root_shell ← get_root_shell()
 5:     access_level ← get_host_access_level(root_shell)
 6:     Return root_shell, access_level
 7: else
 8:     shell ← get_existed_shell_of_host()
 9:     module ← get_exploit_module(process)
10:     result ← execute_pe_module(module, host, shell)
11: end if
12: root_shell ← get_shell(result)
13: access_level ← get_host_access_level(root_shell)
14: Return root_shell, access_level
```

We note that, for the PenGym implementation, some logical NASim actions in the privilege escalation category were replaced with similar actions due to OS limitations. Thus, the `pe_tomcat` action in NASim was revised to use the `pkexec` package instead of the `tomcat` package due to the inability to gain root access directly through a vulnerability in the `tomcat` process in recent OSes. Moreover, the `pe_daclsvc` and `pe_schtask` Window-based actions in NASim are replaced with `pe_proftpd` and `pe_cron`. This is because only Linux OS is currently supported in the PenGym cyber range.

4.3 Action Optimizations

This research implemented two optimization mechanisms to enhance execution performance. The individual action optimization reduces the execution time for a single action, while the training time optimization aims to minimize total training time by avoiding unnecessary repeated actions.

4.3.1 Single Action Optimization

We summarize in Table 4.3 the implementation and validation results of all individual action optimization techniques.

Table 4.3: Summary of the implementation and validation of action optimization techniques.

Actions		Initial Execution Time [s]	Optimized Execution Time [s]	Optimization Techniques
Scan Actions	service_scan	**15.4799**	**6.6132**	Add ports, arguments: -Pn, -n, -T5 --min-parallel --max-parallel
	os_scan	**9.5054**	**4.5023**	
	process_scan	1.0552	1.0552	
	subnet_scan	**55.9613**	**15.3443**	
Exploit Actions	e_ssh	1.3303	1.3303	Disable the AutoCheck attribute in Metasploit Stop Metasploit job as soon as all the necessary information in a shell is obtained
	e_ftp	10.0423	9.1394	
	e_http	**20.6608**	**6.5873**	
	e_samba	10.9031	9.8926	
	e_smtp	8.1215	6.5839	
Privilege Escalation Actions	pe_tomcat	**219.931**	**13.0525**	
	pe_proftpd	**28.8921**	**20.617**	
	pe_cron	75.6347	68.0839	

Scan Actions Several arguments are used to scan only necessary information and optimize the scan process in Nmap to minimize execution time. Additionally, providing the list of available ports in the current scenario improves performance by reducing the time spent scanning unavailable ports. The arguments used in these actions are listed below:

- `-Pn`: Disable the `ping` function to skill the initial host discovery step and proceed to scan all specified targets as if they are active.
- `-n`: Disable the DNS resolution in `nmap` scan function
- `-T5`: Use the most aggressive timing template
- `--min-parallel`: Specifies the minimum number of parallel probes to perform at once
- `--max-parallel`: Specifies the maximum number of parallel probes to perform at once
- `-sS`: Use TCP SYN scanning for `nmap` scan function

- `-sV`: Activate version detection
- `-O`: Activate OS detection

Each scan function uses corresponding arguments to minimize execution time. By determining specific information, such as OS detection, the `nmap` scan function retrieves only the necessary details instead of all information. This helps reduce unnecessary information collection time. Moreover, additional attributes in the `nmap` function are used to optimize the scan mechanisms, such as the timing template.

Exploit and Privilege Escalation Actions Most of the execution time in pentesting comes from these actions due to their complexity. Disabling the `AutoCheck` attribute and applying *stopping conditions* are utilized to optimize the Metasploit module execution. Stopping conditions are determined to finish the Metasploit job earlier for these actions. A job terminates once all shell information is obtained. However, some information is not required for subsequent actions. Stopping the job early can reduce execution time while preserving the functionality of the shell object. Moreover, the Metasploit sessions are cleaned up after an attack sequence ends.

Optimization Results We conducted several tests to exhibit the advantages and efficiency of each optimization technique regarding action execution time. For host-based scanning and exploitation actions, a single host is configured with all services and processes to measure the execution time of these actions. For the subnet scan, which requires more connected hosts and subnets, the test is performed on Host (2,1) in the medium-multi-site scenario (Figure 6.3 in Chapter 6), which has the most connections to subnets among all available networks in PenGym. In the subnet scan, the number of connected subnets can affect the execution time. Therefore, it is essential to consider the condition of connection complexity, as it can clearly show the effectiveness of the optimization.

According to Table 4.3, the time needed to execute most actions was reduced. The highlighted results show significant enhancements in execution time after applying various optimization techniques. For instance, the execution time for the `pe_tomcat` action has been reduced from nearly 220 s to 13 s, representing an impressive 17x improvement. In complex scenarios where this action is executed on multiple hosts, this reduction substantially impacts the overall execution time, leading to much more efficient operations. Note that optimization techniques were not applied for `process_scan` and `e_ssh` due to the simple command-line execution nature of these actions.

4.3.2 Training Time Optimization

Regarding optimizing the total time in the training stage, we utilized a *host map* dictionary and a *boolean flag* to minimize the execution time of redundant actions. Figure 4.2 and Algorithm 7 illustrate the flowchart and pseudocode of this optimization technique, respectively.

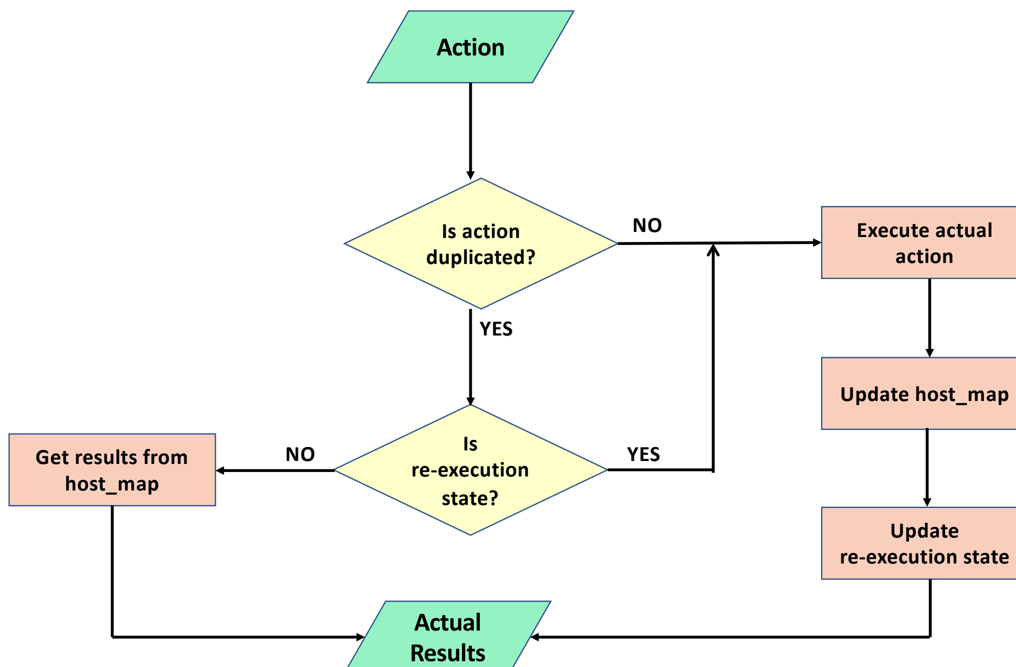


Figure 4.2: Flowchart of the training time optimization mechanism.

The *host map* dictionary stores the actual observation and related information after the first successful execution of an action, which can then be reused for subsequent executions instead of executing the action again. This dictionary is used only for a single pentesting execution period in testing. This period ends when the target hosts are compromised or the step limit is exceeded. Similarly, it is used for a single training time in training, which ends when all the training epochs are finished. The dictionary is reset to an empty state at the beginning of each testing period or training time. This ensures that the optimization strategy does not affect the realism of the overall training process or the evaluation of trained agents during testing. It helps minimize time by avoiding redundant actions that have already been successful in the same period.

The *boolean flag* is used in conjunction with the *host map* dictionary to mark the state of each host. The flag assists in determining if the re-execution

Algorithm 7 Training Time Optimization

```
Require: action, host_map, boolean_flag
 1: actual_action ← False
 2: duplicate ← check_duplicate_action(action)
 3: if duplicate True then
 4:     re_execute ← check_re_execute_state(action,boolean_flag)
 5:     if re_execute True then
 6:         actual_action ← True
 7:     else
 8:         actual_results ← get_results(action,host_map)
 9:     end if
10: else
11:     actual_action ← True
12: end if
13: if actual_action True then
14:     actual_results ← execute_actual_action(action)
15:     update_host_map(host_map)
16:     update_host_map(boolean_flag)
17: end if
18: Return actual_results
```

of a specific action is necessary during the training phase, preventing an actual action from being executed multiple times unnecessarily.

In practice, the success or failure of service scans, OS scans, and exploit actions depend on the traffic rules between each subnet. A service scan between two hosts may fail due to a lack of permitted services, and the *host map* dictionary will reuse this result for any future service scans between them. However, if this service scan is initiated from a different host, the actual service scan must be re-executed. Consequently, the *boolean flag* is used to manage the state for the service scan, OS scan, and exploit actions.

Optimization Results We conducted tests to measure the efficiency of optimization techniques in the training process. we calculated the average training time per episode based on training conducted over 20 episodes in the medium-multi-site scenario for two `ql_replay` agents that were trained before and after applying the optimization techniques. According to our results, the average training time per episode was approximately 2.15 hours before optimization, compared to 0.05 hours after optimization, representing a reduction of about 43 times.

During the initial phase of training, the agent cannot achieve the optimal path, so around 2000 steps are conducted. Without using the *host-map* dictionary and *boolean flag* to control redundant actions, the actual actions are executed multiple times, wasting time during training. In contrast, by using the optimization technique, most of the time comes from the first successful execution of each action, and the results are reused for subsequent actions.

By avoiding redundant actions that were already successful, these optimization methods help minimize time. Moreover, they reflect real-world situations when a pentester does not repeat successful actions. Therefore, using the *host-map* dictionary and *boolean flag* in PenGym optimizes execution time while maintaining the realism of the training and testing stages.

4.4 Cyber Range Creation

Creating a Cyber Range is a process of preparing a realistic training environment. However, manual creation may struggle with complex network environments. Therefore, this research has implemented a module to transform the NASim scenario file into a CyRIS description file, automating the Cyber Range creation process. The main purpose of this module is to create a fully configured network both with regard to network connections and host content, in which all services, processes, and available vulnerabilities are installed automatically, thereby preparing the environment for penetration testing purposes.

CyRIS [18] is designed to facilitate cybersecurity training by automating the creation and management of the required training environment. It uses a text-based representation in YAML format to describe the characteristics of this cyber range, including both environment setup and security content generation. KVM virtualization technology [56] is employed to construct a customized cyber range, including virtual machines with predefined services, processes, and vulnerabilities, all connected following the description file. This aids in preparing and installing the corresponding cyber range instance on a computer and network infrastructure.

4.4.1 Cyber Range Composition

The design of the cyber range environment is defined in a scenario file, which contains network characteristics such as the connection of subnets, the configuration of hosts (e.g., operating system, processes, services, etc.), and firewall rules between subnets. Table 4.4 provides a description of each required parameter for designing a cyber range structure in the scenario.

Table 4.4: Description of the parameters used to specify the composition of a cyber range in PenGym.

Parameter	**Description**
`subnets`	The number and size of each subnet
`topology`	The connection of each subnet
`sensitive_hosts`	The addresses of target hosts and the corresponding rewards
`os`	The available operating system running on any given host
`services`	The available services running on any given host
`processes`	The available processes running on any given host
`exploits`	The available exploit actions in current scenario
`priviledge_escalation`	The available privilege escalation actions in current scenario
`host_configuration`	The configuration and firewall rules of each host
`firewall`	The firewall rules of subnet communication

The topology defines how subnets are connected and controls which subnets can communicate directly with each other and the external network. The firewall function allows certain services to be accessed from machines within a subnet with the correct permissions while blocking access from unwanted entry points. Each firewall is defined by a set of rules that dictate which service traffic is permitted in each direction along a connection between any two subnets or from the external network. Note that although all machines within a subnet can communicate fully, communication between machines on different subnets follows the firewall rules between their subnets.

4.4.2 Cyber Range Description Generation

In this research, we implemented a PenGym module that transforms the NASim training scenario file into a CyRIS cyber range description file, thereby automating the cyber range creation process. The purpose is to create a fully configured network at both network and system levels, thereby preparing the training environment for pentesting. Figure 4.3 indicates the processing flow of this module, as follows:

- **Information extraction:** In this step, *configuration information* is first extracted from the NASim scenario file, including details about firewalls, services, processes, and host-based exploitation actions.

  Then, as needed for the last processing step, *network information* is extracted, defining the services or processes available for each host and the subnet connections.

- **Script generation:** The configuration information and the corresponding service installation packages form the input for this step. The *Service Installation Package DB* is an external component that needs to be prepared ahead, containing all the service packages and installation requirements for each service and process. For instance, the `opensmtpd-6.6.1` package is prepared in advance for `smtp` service setup and includes a list of dependencies required for configuration.

  Based on the existing services or processes in a scenario, the corresponding packages and requirements are used to generate a list of *host configuration installation* shell scripts. These scripts include firewall configuration, service and process installation, and vulnerability settings. The shell scripts are subsequently executed to automate the installation and configuration procedure (e.g., `./configure`, `make install`) within each host.

- **CyRIS description generation:** After preparing the installation scripts, they are combined with the network information from the first step. This information is used to generate a *CyRIS description* file. All cyber range information is output in this phase according to the CyRIS description file structure.

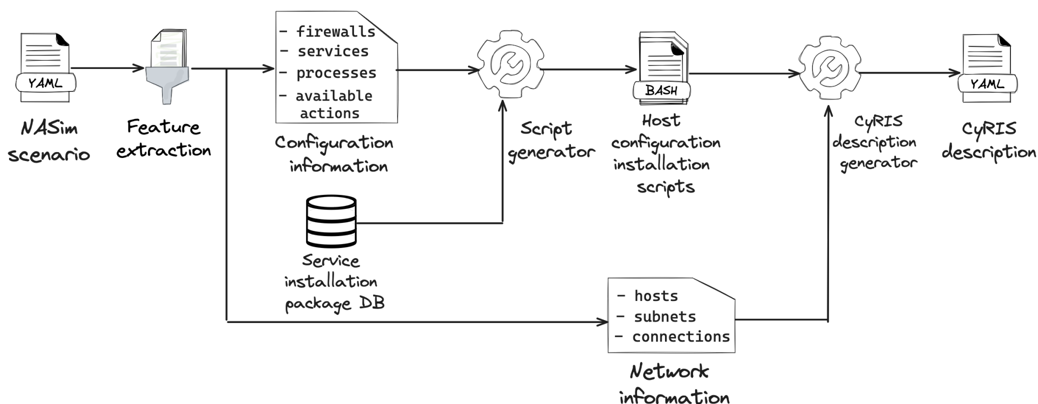


Figure 4.3: Processing flow of the CyRIS cyber range description file generation module.

4.4.3 Bridge Functionality

Network bridges are used to improve the realism of subnet communication and the effectiveness of the cyber range preparation. They control the net-

work connection and enable interaction between the main server that is running PenGym and the cyber range. PenGym is installed on that server, and the cyber range is also created on it by using CyRIS. All actions executed on the virtual machine originate from the main server. Instead of setting up necessary tools like Metasploit or Nmap on each virtual host, these actions can be executed from the main server using bridge functionality.

Bridges have two main states: on and off. Initially, most bridges are turned off, except for the bridge connecting the cyber range subnets to the virtual Internet cloud. If any host in the cyber range is compromised, the corresponding bridge is turned on. This allows the interaction between the main server and the affected host, as well as connections between its subnet and other subnets. The bridge functionality maintains the simple nature of the cyber range by not installing any external pentesting framework while also reflecting the realism of network communication. The pseudocode of this functionality is shown in Algorithm 8.

Algorithm 8 Bridge Functionality

```
Require: bridge_map
 1: turn_off_all_bridges()
 2: reset_bridge_map(bridge_map)
 3: subnets ← get_internet_connected_subnets()
 4: for subnet in subnets do
 5:     bridge ← get_bridge(subnet)
 6:     turn_on_bridge(bridge)
 7:     update_bridge_map(bridge)      ▷ use to control the state of bridges
 8: end for
 9: /* Executing actions */
10: if is_new_host_compromised() True then
11:     host ← get_new_host_compromised()
12:     subnet ← get_subnet(host)
13:     bridge ← get_bridge(subnet)
14:     state ← get_bridge_state(bridge, bridge_map)
15:     if state OFF then
16:         turn_on_bridge(bridge)
17:         update_bridge_map(bridge)
18:     end if
19: end if
```

Chapter 5

Functionality Validation

This chapter discusses the functionality validation of PenGym via detailed results and observations from actions performed in both NASim and PenGym. It also includes validating cyber range creation regarding execution time and environment configuration.

5.1 Action Implementation Validation

Although we conducted validation experiments for the action space across all available scenarios, we selected the medium-multi-site scenario (see Figure 6.3) for explanation purposes. This is because it is a complex scenario in which each host contains multiple services and processes, reflecting the complexity of real-world conditions to demonstrate detailed comparison results. Table 5.1 summarizes validation results between NASim and PenGym. According to this table, the observations of each action in PenGym and NASim environments are equivalent. We also show the action execution times for comparison. Note that the time in PenGym is only for the first execution of an action since we cache the output data to speed up subsequent executions, as explained in detail in Section 4.3.

5.1.1 Service Scan

The target host address must be provided for the service scan action, while the port value is optional. However, by providing all available ports in this scenario, the execution time is minimized by avoiding scanning unusual ports. In NASim, the list of services corresponding to the current scenario is returned, with `1` and `0` representing `True` and `False`, respectively. Table 5.1 shows Host (2,1) results, in which only the `smtp` service receives a `1` value

Table 5.1: Action validations between NASim and PenGym

Actions		Observation		Execution Time [s]	
		NASim	PenGym	NASim	PenGym
Scan Actions	service_scan	‘ssh’: 0, ‘http’: 0, ‘ftp’: 0, ‘smtp’: 1, ‘samba’: 0	[‘smtp’]	0.000002	0.517642
	os_scan	‘linux’: 1	[‘linux’]	0.000015	4.433648
	process_scan	‘tomcat’: 0, ‘proftpd’: 0, ‘cron’: 1	[‘cron’]	0.000006	1.026706
	subnet_scan	(1, 0): True, (1, 1): True... (6, 0): True, (6, 1): True	[‘44.1.5.2’, ‘44.1.5.3’... ‘44.1.10.4’, ‘44.1.10.5’]	0.000067	15.060795
Exploit Actions	e_ssh	access=USER		0.000024	1.365891
	e_http			0.000035	6.973643
	e_ftp	access=ROOT		0.000045	10.561138
	e_samba			0.000043	9.935159
	e_smtp			0.000027	6.661961
Privilege Escalation Actions	pe_tomcat	access=ROOT		0.000029	17.72336
	pe_proftpd			0.00004	18.250733
	pe_cron			0.00004	68.051982

since it is the only service running on the target host. An equivalent result is produced in PenGym, which returns a list of available services.

Regarding execution time, PenGym takes more time to run an action than NASim. As mentioned in Table 4.1, the action in NASim is executed by several checking conditions, in contrast to executing an actual action by using `nmap` to connect to the host and receive the real result. Due to the package transmission in `nmap` mechanisms, these actions require more time to establish a connection and communicate.

5.1.2 OS Scan

The process of OS scanning in `nmap`, which detects OS information, is similar to the service scan action. The parameters used in an OS scan action are the target host and all available ports to optimize execution time. In NASim, the list of OSes corresponding to the current scenario is returned, with `1` and `0` representing `True` and `False`, respectively. In this research, only Linux

OS is supported, so no `0` value is received. Table 5.1 shows the results for Host (2,1), where the `linux` receives a `1`. An equivalent result is produced in PenGym, which returns the detected OS.

Regarding execution time, due to the complicated `nmap` connection establishment process, PenGym takes more time to run an action than NASim.

5.1.3 Subnet Scan

PenGym returns the list of IP addresses for all hosts within each interconnected subnet of the target host. In NASim, all hosts are returned with a value of `True` for those discovered and `False` for those not yet discovered. We can validate the equivalence between NASim and PenGym observations by mapping these hosts with their associated addresses. For example, a subnet scan on Host (2,1) leads to discovering all the hosts in the network, as that host is connected to all subnets in the scenario.

Similar to the above scan actions, executing a subnet scan in PenGym takes more time than in NASim due to the actual process in the `nmap` function. The time required is much greater than previous scans because a subnet scan scans a range of connected subnets to obtain all available hosts. This requires scanning all hosts to determine the active ones. Although it takes more time than the simulation approach, the realism of the action is preserved, and the time is acceptable.

5.1.4 Exploit

In exploit actions, an access level, either `USER` or `HOST`, is obtained upon successfully gaining unauthorized access to the target host. In NASim, the predefined access level associated with the specific action is returned. Meanwhile, PenGym obtains a shell object by executing the actual action using the `metasploit`. This shell can be used for subsequent actions, such as process scanning and privilege escalation, to investigate other critical potential vulnerabilities. The access value is then identified by executing the `whoami` command directly through this shell. Table 5.1 shows the equivalence between the NASim and PenGym observations of exploit actions.

5.1.5 Process Scan

The process scan action is performed using the `ps` command executed via the shell previously obtained on the targeted host through an exploit action. The observations in NASim and PenGym environments are equivalent, functioning similarly to service scan and OS scan actions. In PenGym, the list of

running processes is returned, and a dictionary indicating existing processes with a value of 1 is obtained in NASim.

5.1.6 Privilege Escalation

The mechanisms of privilege escalation actions work similarly to exploit actions in both NASim and PenGym environments. The access levels for both environments are equivalent, which are the `ROOT` access. Additionally, the shell object with administrator permission is retrieved in PenGym.

Regarding the execution time of host-based exploitation actions containing exploit and privilege escalation actions, it varies based on the specific services or processes and their corresponding `metasploit` modules. The time complexity arises from the complexity of the exploit module and the kind of payload. As a result, compared to the simulation approach, an actual shell is obtained, which can interact and assess the access level of the target. In practice, the success of exploitation actions depends on various factors such as firewalls, vulnerabilities, the architecture of the payload, etc. Although there is a time trade-off between PenGym and NASim, it reflects the realism and difficulty lacking in the simulation.

5.2 Cyber Range Creation Validation

Automated cyber range creation is an important new aspect of this research. It uses a predefined scenario to create the corresponding cyber range automatically. The duration for creating all scenarios was measured in Table 5.2 to demonstrate the efficiency of integrating this component into PenGym.

5.2.1 Configuration Validation

The cyber range configurations are verified on each host by running a shell command to check the currently active services or processes. The firewall rules on each host are validated using the `iptables` command line. For instance, the services and processes on Host (5, 0) are validated for the medium-multi-site scenario by executing the command "`ps -aux | grep -e ftp -e proftpd -e cron`." The output showed the actual running services and processes on this host, such as the `ftp` service, along with the `proftpd` and `cron` processes. The other services and processes are also checked as part of the validation procedure to confirm no undesired services are running on a host. The different services and processes are also checked as part of the validation procedure to ensure no undesired services run on a host. In

the context of medium-multi-site scenario, the similar command is executed with `ssh`, `smtp`, `samba`, `http` services and `tomcat` process to make sure the correctness installation in Host (5, 0).

The same procedure was conducted for all cyber ranges, achieving compatible results between the actual environment and the scenario. The validation process assessed the correct creation of the cyber range concerning the predefined scenario, with wholly automated host and network configuration.

5.2.2 Creation Time

According to the results, the creation time for each cyber range increases as the complexity of the scenario increases. It takes a few minutes to create a simple scenario, but it can take a few hours for a more complex one.

Table 5.2: Cyber range creation time for all scenarios

Network Size	Scenario	Cyber Range Creation Time [s]
Tiny	tiny	577.81
	tiny-hard	871.34
	tiny-small	1432.21
Small	small-honeypot	2396.06
	small-linear	2145.32
Medium	medium	4666.14
	medium-single-site	4830.39
	medium-multi-site	5390.20

Specifically, the creation for medium-multi-site, which is the largest and most complex network scenario in our experiments, took approximately 5390 s. Although this may seem long, note that a cyber range is created only once during experimentation, and around 50% of creation time comes from the package installation process. Compared to the manual creation approach, it is considered as a reasonable time. Due to the complexity of package installation, creating and configuring manually for the entire cyber range could take more time and is more prone to mistakes because of the large number of hosts and steps involved.

Chapter 6

Experiment Results

We conducted a series of experiments to demonstrate the feasibility and effectiveness of the PenGym in terms of time performance, stability, and adaptability of trained agents when dealing with various complex pentesting scenarios. We selected NASim for comparison to highlight the key differences between PenGym as an emulation approach and NASim as a simulation approach. In addition, we used two versions of the NASim. The first one, NASim, is the original NASim environment, and the second one, *NASim(rev.)*, is the revised version of NASim for which we corrected some logical errors in the original version, as explained in Chapter 7. Thus, the RL agents were trained and tested using NASim, NASim(rev.), and PenGym.

6.1 Experiment Scenarios

We covered all eight scenarios available in NASim that are categorized based on the network size as *tiny*, *small*, and *medium*. Table 6.1 summarized the attributes of all scenarios, including the number of subnets, number of hosts, action space size, and optimal step count to reach the pentesting goal.

In this section, tiny, small-linear, and medium-multi-site scenarios are selected for visualization and detailed experiments. These scenarios represent each kind of network size with varying levels and complexities. The reasons for these choices are discussed in the next section. All the experiments were conducted on a dual 12-core 2.2 GHz Intel(R) Xeon(R) Silver 4214 CPU server with 64 GB RAM.

Table 6.1: Overview of all the experiment scenarios in PenGym.

Network Size	Scenario	Number of subnets	Number of hosts	Action space size	Optimal step count
Tiny	**tiny***	**3**	**3**	**18**	**6**
	tiny-hard	3	3	27	5
	tiny-small	4	5	45	7
Small	small-honeypot	4	8	72	7
	small-linear*	**6**	**8**	**72**	**12**
Medium	medium	5	16	192	7
	medium-single-site	1	16	192	4
	medium-multi-site*	**6**	**16**	**192**	**7**

* These scenarios were selected for detailed experiments.

6.1.1 Tiny Scenario

We selected `tiny`, the smallest scenario for the tiny network size, as illustrated in Figure 6.1. This scenario consists of 3 hosts divided into 3 subnets, with hosts (2,0) and (3,0) being the pentesting goals. Subnet(1) is directly connected to the Internet, while the other subnets are internally connected. Each host has the same basic configuration, with a Linux OS, `ssh` service, and `tomcat` process. Firewall rules are enforced for secure subnet communication, allowing only `ssh` communication. Specifically, Subnet(1) is accessible from the Internet via `ssh` but cannot connect externally. Additionally, there is a firewall rule blocking `ssh` service from host (3,0) to host (1,0), even though communication is allowed between Subnet(1) and Subnet(3). These restrictions were implemented using the `iptables` Linux firewall configuration utility. This scenario contains 18 available actions that can be executed, which are the lowest values compared to others.

6.1.2 Small Scenario

Figure 6.2 shows the design of the small-linear scenario, which belongs to a small network size. It includes 8 hosts, categorized into 6 subnets. Each subnet contains up to 2 hosts. Hosts (3,0) and (4,0) are the pentesting

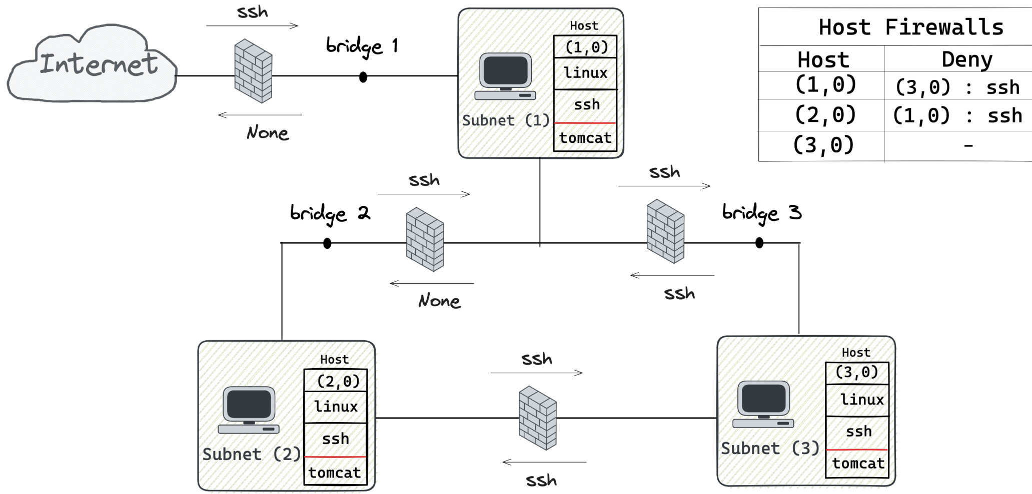


Figure 6.1: Cyber range constructed in PenGym based on the tiny scenario in NASim.

goals in this scenario. This scenario highlights the complexity from a subnet perspective by increasing the number of subnets while keeping the number of hosts in each subnet simple. Multiple services are used for some hosts, while some hosts do not run any processes. This scenario contains 72 actions, but the optimal path is 12, the largest. It indicates the complexity of exploring the path to achieve the goal. Similar to the previous scenario, `iptables` is used to set up the firewall rules.

6.1.3 Medium Scenario

For the medium scenario, medium-multi-site is selected as the representation, which is the largest scenario. Figure 6.3 visualizes the cyber range architecture in this scenario. It comprises 16 hosts divided into 6 subnets, with hosts (2,1) and (3,4) being the pentesting goals. Each host contains one or multiple services and processes. Some hosts may lack either element, increasing the complexity of the network. All actions are available in this scenario. Network traffic restrictions between each subnet were implemented using the `iptables` Linux firewall configuration utility. In this scenario, there are no specific firewall rules between any hosts. From the statistical value in Table 6.1, the total actions in this scenario are 192 compared to 18 in the smallest scenario. This emphasizes the high complexity of this scenario.

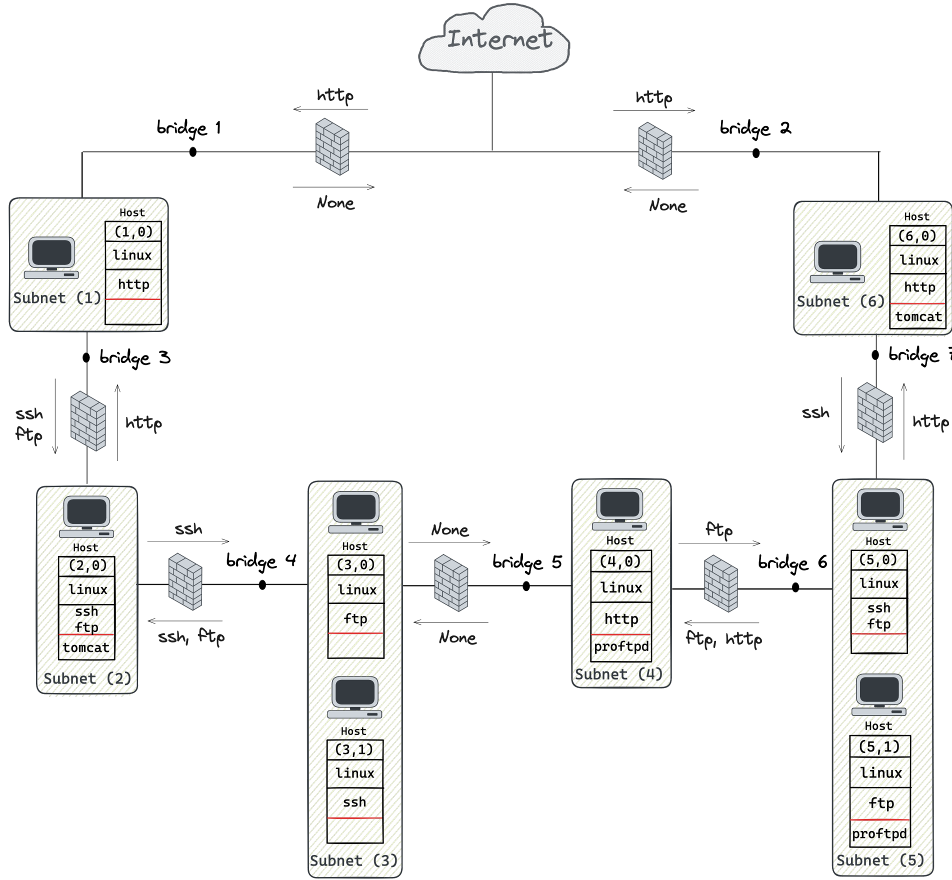


Figure 6.2: Cyber range constructed in PenGym based on the small-linear scenario in NASim.

6.2 Preliminary Experiments

We conducted a set of preliminary experiments in all scenarios to demonstrate the overall capability of agents trained in the PenGym, NASim, and NASim(rev.) environments. The results will aid in deciding the representative scenario for each network level to conduct the detailed experiments.

Experiment Conditions To decide the algorithm agents used for this experiment, in the medium-multi-site network scenario, we used the tabular, epsilon-greedy Q-learning with experience replay (`ql_replay`) [57], Q-learning (`ql`) [58], and DQN [59] algorithms agents, available in NASim, for training. These agents were trained once in the PenGym environment. Our initial results showed that the `ql_replay` agent took about 6 hours to converge within 300 episodes, while the `ql` agent achieved stability within

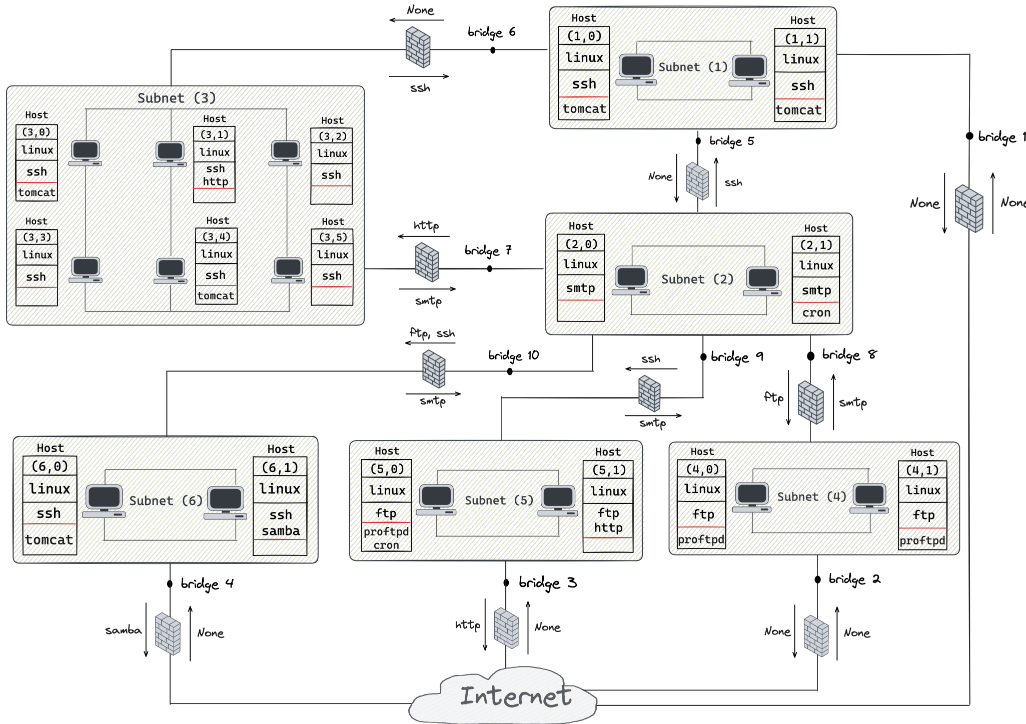


Figure 6.3: Cyber range constructed in PenGym based on the medium-multi-site scenario in NASim.

4000 episodes in approximately 2 hours. Despite requiring more episodes, the training time of the `ql` agent was reasonably efficient compared to the `ql_replay` agent. Moreover, the DQN agent could not stabilize due to its simple architecture. We selected the `ql` agent for the preliminary training experiments across all scenarios based on these results.

We independently trained 5 `ql` agents with 4000 episodes and tested them across all scenarios in PenGym and NASim (rev.) environments. Moreover, we selected medium-multi-site, small-linear, and tiny scenarios for conducting the same experiments with the NASim environment.

Experiment Results Table 6.2 summarizes the results of the preliminary experiments. The training time in all scenarios was measured, and the average ratio between the PenGym and NASim(rev.) environments was calculated based on the network type. The results indicated that the training time ratio decreases as network complexity increases. It reduced from approximately 6.0 to 1.7 as the network grew. The training time in both environments tends to be comparable when dealing with complex scenarios. This leads to a reasonable time, given the complexity of real-life networks.

Table 6.2: Results of preliminary experiments conducted for all the available scenarios (NASim* refers to either NASim or NASim(rev.), according to the simulator variant used for training).

Network	Scenario	Training Results			Testing Steps		
		Environment	Training Time [s]	Ratio†	NASim* environment	PenGym environment	Step difference
Tiny	tiny	NASim	33.72	5.9	**6.6**	**7**	0.4
		NASim(rev.)	34.53		7.4	7	0.4
		PenGym	159.93		7.6	7	0.6
	tiny-hard	NASim(rev.)	34.53		6.6	7.8	1.2
		PenGym	246.85		5.4	5.4	0
	tiny-small	NASim(rev.)	77.67		12	8.8	3.2
		PenGym	473.37		8.8	9.5	0.7
Small	small-honeypot	NASim(rev.)	151.10	4.5	8.8	8.8	0
		PenGym	1019.35		10.4	11.8	1.4
	small-linear	NASim	223.23		**21.2**	**213.8**	192.6
		NASim(rev.)	373.26		15.2	14.8	0.4
		PenGym	877.15		14.8	15.8	1
Medium	medium	NASim(rev.)	3668.87	1.7	12.8	16.6	3.8
		PenGym	6438.76		14.2	14	0.2
	medium-single-site	NASim(rev.)	2630.69		6.6	6.4	0.2
		PenGym	4158.42		6.6	6.2	0.4
	medium-multi-site	NASim	4890.49		**24.6**	**2000**	1975.4
		NASim(rev.)	4348.47		26	24.2	1.8
		PenGym	7518.84		24.6	24	0.6

† Average ratio of training time values in PenGym versus NASim(rev.) for a given network type.

Regarding adaptability, the agents trained in NASim performed well in the simulation environment (NASim). However, they were inapplicable in the realistic environment (PenGym). For the smallest scenario tiny, they achieved good results when tested in a simulation environment. However, in more complex scenarios, such as small-linear, an increase in the number of steps indicated a significant gap in reaching the goal between both environments. Although these agents reached the pentesting goal, it required more steps, approximately 10 times compared to the simulation environment. Particularly in the medium-multi-site, which is the largest scenario, the agents completely failed in PenGym environment by reaching the imposed step limit of 2000 without achieving the goal.

Moreover, the NASim(rev.) and PenGym produced comparable results, with a small gap in the testing steps across all scenarios. However, the step differences pointed out that agents trained in PenGym had more stable performance than those trained in the NASim(rev.). The step difference for

PenGym-trained agents ranges from 0.4 to 1, whereas it ranges from 0 to 3.8 for agents trained in NASim(rev.) environment.

6.3 Detailed Experiments

We then carried out a series of detailed experiments to showcase the effectiveness of PenGym. The preliminary results in Section 6.2 showed that the complexity of the scenario impacted the training performance and convergence of RL agents. In these experiments, we chose three scenarios, tiny, small-linear, medium-multi-site, representing three different complexity levels for detail training and testing purposes. As before, we trained and tested the RL agents using NASim, NASim(rev.), and PenGym environments.

6.3.1 Agent Training

First, we will discuss the agent training procedure, starting with the experimental conditions and then the results we obtained.

Experiment Conditions For each scenario, we trained 10 agents independently for the NASim, NASim(rev.), and PenGym environments. Specifically, we trained 5 agents using the `ql_replay` and 5 agents using the `ql` algorithms that are mentioned in Section 6.2. These algorithms were used due to their ability to converge and maintain stability during the training phase, as shown in our initial experiments. The DQN algorithm was excluded due to the inability to converge the learning strategy of the agent. For the `ql_replay` algorithm, we trained each agent with 300 episodes, and for the `ql` algorithm, we used 4,000 episodes to achieve the convergence for each agent. The episode gap between the two algorithms arises from their complexity. While the `ql_replay` agent employs experience replay to accelerate convergence, the `ql` agent uses a simple Q-learning algorithm.

The modifications and revisions were made to all scenarios to ensure compatibility across all environments, as detailed in Chapter 7. Thus, the probability attribute of the exploit actions was set to 0.999999. This adjustment creates a scenario that more closely resembles our cyber range, where the exploit does not fail (note that we chose this value because 1.0 could not be assigned due to a quirk in NASim implementation). The success or failure of actions depends on the actual action being executed by the real machine. Hence, we adjusted the probability attribute to enhance compatibility between the NASim and PenGym environments.

Experiment Results Figure 6.4 shows the average rewards obtained during the training of 5 `ql` agents and 5 `ql_replay` agents across three scenarios for the NASim, NASim(rev.) and PenGym environments. According to the results, the PenGym framework has demonstrated comparable performance in training agents regarding rewards compared to NASim and NASim(rev.), using both types of agents. However, in a complex scenario such as medium-multi-site, the training results still oscillated more in both NASim and NASim(rev.) environments. Although the training results from the NASim environment perform better than others in the medium-multi-site scenarios, this could be due to an incorrect logical model that enables the agent to reach the goal in fewer steps, ignoring some permission rules.

Regarding the training time, the difference decreased as the network and RL algorithms became more complex. Notably, in the most complicated scenario, medium-multi-site, PenGym averaged nearly $17,000$ s compared to $14,000$ s for NASim(rev.), making the training time comparable between two environments.

Training Time Composition Analysis Based on experiments of training a QLearning Replay agent for 300 episodes when using the largest scenario in both NASim(rev.) and PenGym environments, we calculated the proportion of the time needed to reset the environment, update the agent policy, and execute actions in the environment compared to the total training time. Our results showed that for the NASim(rev.) environment, 98% of the training time came from agent policy updating, while the rest of the time was used for environment resetting and action execution. Consequently, the action execution time in the simulation environment is minimal, with each action taking only a few milliseconds due to simple condition checks, and most of the training time is consumed to update the agent learning strategies.

On the other hand, in the PenGym environment, agent policy updating accounted for only 58% of the training time, whereas action execution time was 40%, and environment resetting took 2% of the time. Even though the proportion of agent policy updating time compared to total training time differs between the two environments, the ratio between PenGym and NASim(rev.) was comparable, around 1.1. This is because the policy update time depends on the number of attack steps. When the training performance of the agents is equivalent between the two environments, the time for updating their policies is also similar.

We conclude that even though the training time in PenGym is slightly longer than in NASim(rev.), it remains within a reasonable limit and is comparable in complex scenarios. The difference in total training time between

the PenGym and NASim(rev.) environments can be attributed to the longer action execution times in PenGym. This is caused by the fact that the actions are executed in an actual environment, and it leads to a different ratio of policy update time to total training time, as more time is spent exploring the environment in PenGym compared to NASim(rev.)

6.3.2 Agent Testing

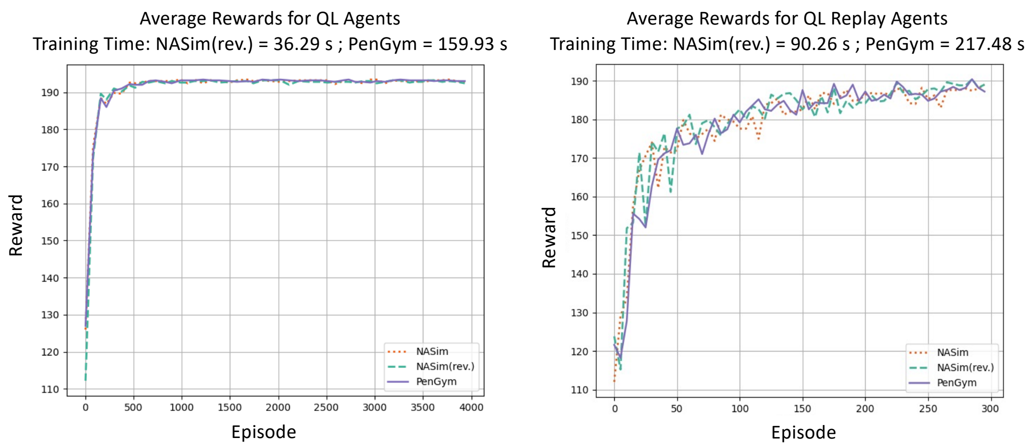
Next, we evaluated the performance of all agents in the NASim, NASim(rev.), and PenGym environments.

Experiment Conditions We conducted 10 testing experiments in each scenario within either the NASim or NASim(rev.) and PenGym environments. To show the advantages of using the RL approach in pentesting, the available brute force agent in NASim was included. It represents the *"try all possible actions to reach the goal"* approach [60] and was executed in 10 trials as well. While both random and brute force agents are available for conducting experiments, we only chose to use the brute force agent since it outperforms the random agent in our trial tests.

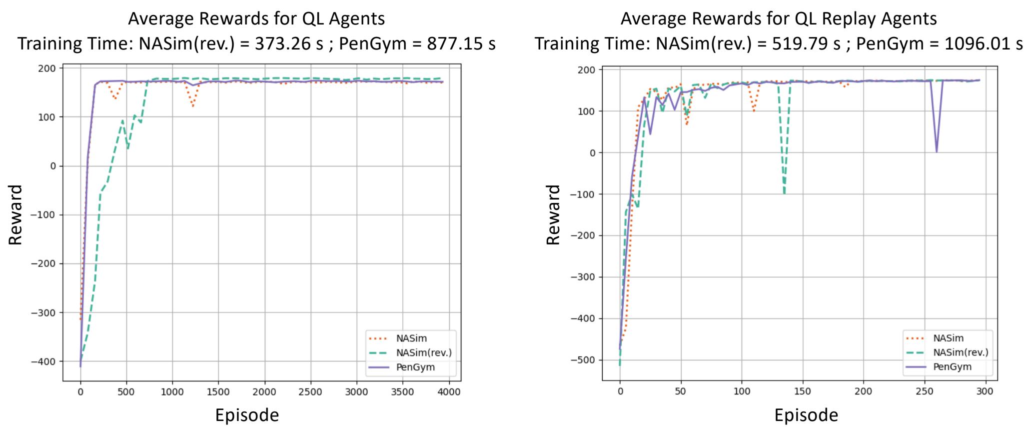
Result Overview Figure 6.5 shows the average attack steps of both `ql` and `ql_replay` agents, trained on NASim, NASim(rev.), and PenGym then tested in these environments. Figures 6.5(a), 6.5(b), and 6.5(c) visualize the testing results using the tiny, small-linear, and medium-multi-site scenarios, respectively. The results of the 10 trials were averaged to derive the final performance values for the agents in each environment. The error bars were also added to visualize the standard deviation values for assessing the performance consistency and stability of the agents across different trials.

The results demonstrate that both `ql` and `ql_replay` agents, trained in PenGym, successfully achieved the pentesting goals across three scenarios. They showed stable performance with minor variation in both simulations (NASim and NASim(rev.)) and emulation (PenGym) environments.

However, agents trained in the NASim environment performed poorly when tested in the PenGym environment. They only achieved the goals in simple scenarios, like tiny, and became inapplicable when faced with more complex scenarios, such as small-linear and medium-multi-site. Although these agents achieved the pentesting goal in some trials, they required more steps than the optimal listed in Table 6.1, demonstrating a significant disparity in the number of required steps. Notably, the agents mostly failed in the largest scenario, exceeding the 2000 step limit. Moreover, the error bars in



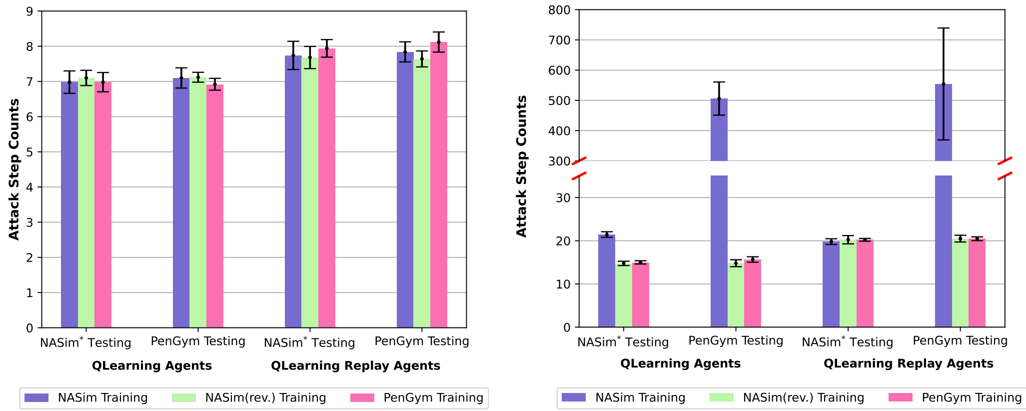
(a) Experiment results for the tiny scenario.



(b) Experiment results for the small-linear scenario.

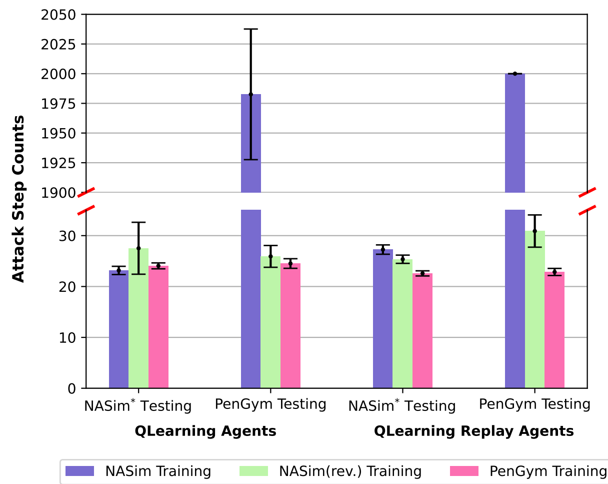
(c) Experiment results for the medium-multi-site scenario.

Figure 6.4: Average training reward versus episode number for the QLearning and QLearning Replay agents trained in the NASim, NASim(rev.), and PenGym environments for the three representative experiment scenarios.



(a) Experiment results for the tiny scenario.

(b) Experiment results for the small-linear scenario.



(c) Experiment results for the medium-multi-site scenario.

Figure 6.5: Average number of attack steps for the trained QLearning and QLearning Replay agents when tested in the NASim* and PenGym environments for the three representative experiment scenarios (NASim* refers to either NASim or NASim(rev.), depending on the simulator variant used for training).

Figures 6.5(b) and 6.5(c) indicate the instability of NASim-trained agents, as suggested by the higher standard deviation. Because the agents trained in NASim mostly failed and had a low success rate in reaching the goal.

On the other hand, since NASim(rev.) corrects the logical modeling issues in NASim, the performance of agents trained in PenGym and the NASim(rev.) is similar in the tiny and small-linear scenarios, with only a minor difference in their averages. This suggests that agents trained in a correctly logical simulation environment can achieve results comparable to those in a realistic environment.

Table 6.3: Detailed experiment results for the QLearning and QLearning Replay agents trained and tested in the NASim, NASim(rev.) and PenGym environments using the medium-multi-site scenario.

Scenario	Algorithm	Training	Testing	Success Rate	Avg. Steps	Avg. Exec. Time [s]	Std. Dev.
medium-multi-site (Medium Network)	**QLearning**	NASim	NASim	10/10	23.16	0.023	0.810
			PenGym	**0.2/10**	**1982.60**	173.735	55.024
		NASim(rev.)	NASim(rev.)	10/10	27.52	0.033	5.104
			PenGym	10/10	25.92	470.216	2.142
		PenGym	NASim(rev.)	10/10	24.06	0.025	0.582
			PenGym	10/10	24.52	416.788	0.944
	QLearning Replay	NASim	NASim	10/10	27.26	0.026	0.919
			PenGym	**0/10**	**2000.00**	141.467	0.000
		NASim(rev.)	NASim(rev.)	10/10	25.36	0.026	0.810
			PenGym	10/10	30.90	356.510	3.169
		PenGym	NASim(rev.)	10/10	22.58	0.023	0.503
			PenGym	10/10	22.86	356.684	0.699
	Brute force	N/A	NASim(rev.)	10/10	681.00	0.0517	0.000
			PenGym	10/10	681.00	2545.606	0.000

Detailed Results Table 6.3 details the testing results in medium-multi-site. This table shows that most agents trained in NASim cannot be applied successfully in PenGym, as evidenced by the low success rates of `ql` and `ql_replay` agents, at 0.2/10 and 0/10 respectively. Despite the agents trained in NASim(rev.) achieving performance comparable to those trained in PenGym, the latter slightly outperformed the former. This is proved when testing `ql_replay` agents trained in NASim(rev.) and PenGym using a realistic environment; the average step gap is around 8 (22.86 steps compared to 30.9 steps). This discrepancy could be due to yet-undiscovered *niche* issues in the NASim(rev.) that affect the performance of agents. Thus, while sim-

ulation environments for training pentesting agents have shown promising results in recent years, these results may not be reliable enough for deployment in real-world infrastructure.

Additionally, the results showed that using brute force, a traditional method in pentesting, was less effective than using RL-based agents. From the experiments, the number of steps required significantly increased as the scenario became more complicated. Specifically, it took 681 steps to reach the goal in the medium-multi-site scenario.

The main advantage of PenGym lies in its ability to facilitate training with actual actions in a real environment, eliminating the need for modeling actions using execution assumptions and success probabilities. Action execution without probabilistic factors enables agents trained in PenGym to improve their algorithm based on realistic results. In NASim, the success of an exploit action is determined by a random value that may fail at unexpected times, which does not accurately reflect realistic actions and leads to non-deterministic effects on agent learning. As the system grows, the action space becomes larger, and these probabilistic factors may affect the learning process of the agents in unknown ways. Therefore, our approach ensures a more realistic representation of the security dynamics in the network.

One conceivable drawback of PenGym is its longer execution time compared to the simulation environment, as actions are physically executed in the network. However, the training time tends to be comparable in complex scenarios. So, we consider the gap in a single pentesting time as a reasonable trade-off for the reliability of agents trained in a realistic environment. Moreover, the entire automation ensures that the execution time is lower than the human-based pentesting process, which takes weeks or even months [24].

Chapter 7

Discussion

This chapter discusses the differences between NASim as a simulation environment and PenGym as an emulation environment regarding host configuration, actions, observation, and functionality issues, summarized in Table 7.1. Several changes were made to make both environments comparable, and the locations of these changes are also included.

7.1 Comparative Analysis of Simulation and Emulation Approaches

In PenGym, a cyber range is created, composed of several virtual machines (VM) that run actual operating systems and services. Compared to NASim, PenGym currently supports only Linux OSs and not Windows OS. So, we changed the host setup, action definition, and action space to adapt to Linux while preserving the comparable characteristics of both environments.

7.1.1 Host Configuration

From the host configuration perspective, the OS and process values are modified. As mentioned, only Linux is available in PenGym, compared to both Windows and Linux in NASim. This leads to differences in processes. NASim includes the `daclsvc` and `schtask` processes, which are available only for Windows. In PenGym, due to the incompatibility of the `daclsvc` process in Linux, we chose `proftpd` as a replacement. Moreover, `cron` is used to replace `schtask` since `cron` is used for task scheduling in Linux.

For cyber range creation, `proftpd` and `cron` are installed directly on the VM hosts. In the NASim scenario, we changed the Windows values to Linux and replaced `daclsvc` and `schtask` with `proftpd` and `cron`, respectively, to

Table 7.1: Summary of the differences between the PenGym and NASim environments.

Category	Criteria	NASim	PenGym	Change Location
Host Configuration	OS	Windows, Linux	Linux	PenGym scenario
	Process	`daclsvc`	`proftpd`	
		`schtask`	`cron`	
Acion	Probability	Not fixed	`0.999999` (exploit actions)	
	`e_smtp`	Access Level: **USER**	Access Level: **ROOT**	
	`pe_tomcat`	Simulate `tomcat` based action	Replace by `pe_pkexec` action	PenGym scenario; PenGym code
	`pe_daclsvc`	Simulate `daclsvc` based action	Replace by `pe_proftpd` action	
	`pe_schtask`	Simulate `schtask` based action	Replace by `pe_cron` action	
Observation	Exploit actions	The services and OS information is included in the observation without prior scan actions	Only the explored services and OS information is included in the observation	PenGym code
Functionality Issues	The firewall does not affect the exploit action from the Internet to the connected subnet	Exploits based on services not allowed by the firewall can be executed from the Internet to hosts within connected subnets	Check the permission services between the Internet and connected subnets	Original NASim code
	The Service Scan and OS Scan do not take firewall into account	Services or OSes not allowed by the firewall can be detected between source and destination subnets in service scan	Filter the permission between source and destination subnets	PenGym code (NASim mode)
	The Remote Action (Exploit) can be executed on its target host	The remote action can be executed on host itself	Check remotely of the source address when executing remote actions	Original NASim code

become the PenGym scenario. These changes ensure consistency between both environments, as both were created using the same scenario.

7.1.2 Actions

The action space has two main modifications: the probability factor and action implementation mechanisms.

Probability In the simulation approach, probability is a factor in determining the success of actions. The probability values vary based on each exploit action, such as 0.9 for `e_ssh` and 0.6 for `e_ftp`. In contrast, all actions in PenGym are executed without relying on the probability attribute. Therefore, we changed this value to 0.999999, which is close to 1, for all exploit actions in the PenGym scenario. Due to the NASim implementation quirk to change this value to 1, as a temporary solution, we chose this value to reduce the impact of the probability factor.

Linux-Based Actions We obtained different access levels for `smtp`-based exploit action. PenGym gained ROOT level, while NASims achieved USER level by attacking a vulnerability in the `smtp` service. This can be explained by the dissimilar vulnerability issues between `smtp` in Linux and Windows. The access level of `e_smtp` was changed to ROOT in the PenGym scenario.

Regarding `tomcat`-based privilege escalation action, to the best of our knowledge, it is impossible to gain root access directly by attacking vulnerabilities in the `tomcat` process, at least in recent OSes. Instead, it can be used as an intermediate step to obtain a Meterpreter shell, and then ROOT access can be gained by exploiting local operating system vulnerabilities through this shell. Such an implementation would eliminate the need to exploit the host first to get access to it so that it would change the built-in sequence of actions in NASim. To address this issue, in PenGym, we used a new method for privilege escalation instead, which uses a vulnerability in the `pkexec` package to gain ROOT access to the target hosts. This modification enables PenGym to follow a realistic attack scenario that still follows NASIM assumptions, which was essential to make their comparison possible.

Furthermore, since the `daclsvc` and `schtask` processes were changed to `proftpd` and `cron`, their corresponding privilege escalation actions were replaced with compatible actions. In PenGym, `pe_proftpd` and `pe_cron` are implemented using `proftpd` and `cron` processes vulnerabilities, replacing `pe_daclsvc` and `pe_schtask` in NASim.

We changed the names of privilege escalation actions in the PenGym scenario and the implementation in the PenGym source code. All these changes do not affect the functionality of NASim since the results of actions are obtained from predefined values. These changes help make both environments comparable for a fair comparison.

7.1.3 Observations

The observation of exploit actions between NASim and PenGym are different. In NASim, if the exploit action is successful, the observation returns the current state of the network, including all services and OS information, even without prior scan actions. This creates inconsistency from a realistic perspective. The exploit action does not send packages to scan the services or OSes, so this information cannot be obtained without scan actions. To make the observation consistent between the two environments, in PenGym, we assumed that when the exploit action is successful and returns a shell object, the services and OS information are scanned using this shell.

7.2 Simulation Modeling Issues

Regarding modeling functionality issues in the simulation environment, we discovered some unexpected logical errors in NASim. We corrected them to be *NASim(rev.)*, ensuring the behavior is comparable to the PenGym environment. They came from uncovered situations when modeling the action, mainly when dealing with complex scenarios that require complicated conditional logic. This complexity of scenarios increased the chances of encountering conditions we could not manually cover. We also opened all issues in the NASim repository [61] for author notification.

7.2.1 Firewall Functionality Issue

The first issue relates to the firewall function, which does not prevent blocked service-based exploit actions from the Internet to the connected subnet. Specifically, in the NASim environment, an exploit action from the Internet to a connected host will succeed even if no traffic is allowed between them. However, this action cannot occur in real life if the firewall blocks all transactions. For example, the `e_ftp()` exploit action from the Internet to Host (4,1) in the medium-multi-site scenario would succeed in the NASim environment, as simplified in Figure 7.1. However, it would fail in the PenGym environment because the exploit module cannot connect to the `ftp` port due

to the blocking rule of the firewall. In PenGym, we devised the solution by checking the permission services between the Internet and connected subnets before executing the action. It was implemented by overriding the NASim function. We opened this issue on the NASim repository via [62].

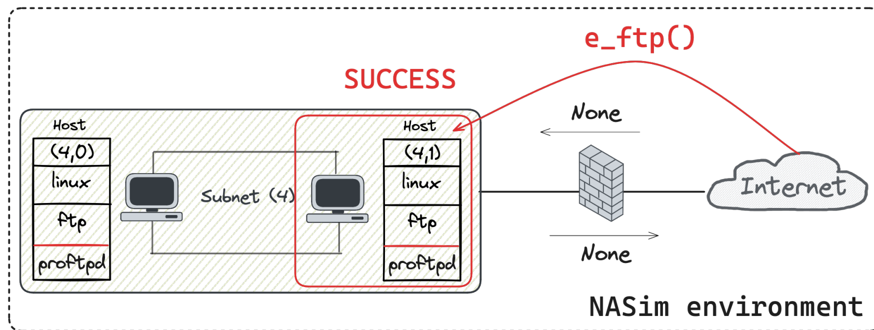


Figure 7.1: Visualization of the firewall functionality issue.

This problem can impact the effectiveness of trained agents due to incorrect logical conditions. These conditions lead the agents to explore the incorrect pentesting path, which is inapplicable in a realistic environment.

7.2.2 Scan Action Issue

The second issue is that the service and OS scans do not consider the firewall. These actions succeed in the NASim environment, even when the service is blocked by a firewall. Figure 7.2 visualize an example of executing `service_scan()` on Host (5, 1), which allows only `http` service communication from the Internet. However, as a result, both `ftp` and `http` services are obtained without restriction from the firewall.

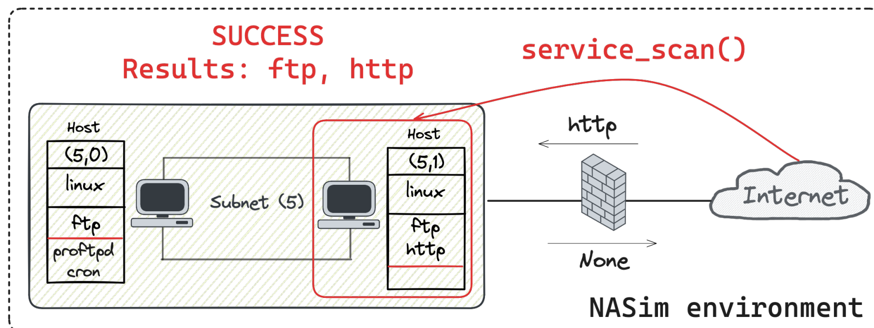


Figure 7.2: Visualization of the scan action issue.

This creates an unrealistic outcome since the Nmap scan function will

send the TCP packets to the system and determine the running services based on the returned packets. Therefore, if a service is blocked, there should be no packet transmission. Hence, the results from the simulation environment conflict with the real-world conditions, making the trained agents ineffective when deployed in real-world infrastructure. PenGym implemented a solution that filters allowed services and checks communication permissions between source and destination subnets during scan actions. These changes were applied to the PenGym source code in simulation mode. The issue was posted on the NASim repository via [63].

Although scan actions are not included in the optimal path, the policy of agents may be affected when deployed in a real network. Agents may encounter difficulties when they face different observations after executing a scan action, especially if the host blocks all transactions and they receive a failure result in a realistic environment that was not previously learned.

7.2.3 Remote Action Issue

Another issue concerns the execution of a remote action on its target host. Theoretically, a remote action, such as an exploit, is typically executed from a different host to the target host. However, in NASim, this property appears to be ignored, which does not make sense in this context. As shown in Figure 7.3, the `e_ftp` action can be executed on Host (5, 1) itself.

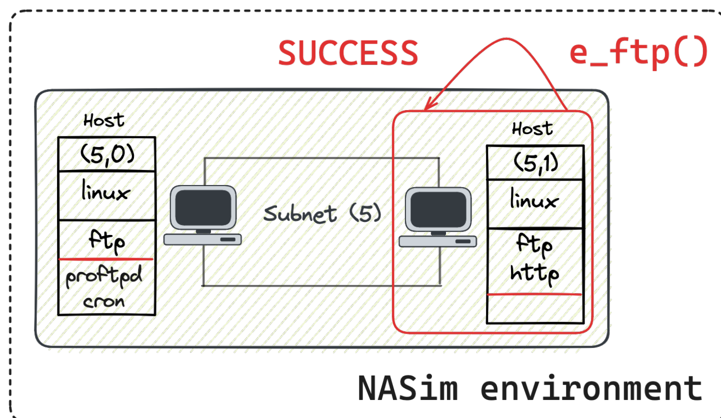


Figure 7.3: Visualization of the remote action issue.

We implemented the solution to ensure consistency between both environments by overriding the NASim function in PenGym. When executing remote actions, such as exploit actions, the identical host conditions between the source host and the target host are checked. This issue was presented in the NASim repository via [64].

Chapter 8

Conclusion

This chapter summarizes our research, highlighting the key findings and contributions made throughout this thesis. Moreover, it presents the final conclusions drawn from our study, emphasizing the significance and implications of the results. Additionally, we will discuss potential future work related to this research, which further direction could enhance the current works.

8.1 Summary

This research aims to bridge the reality gap in the simulation approach for training pentesting RL agents. We focus on demonstrating the effectiveness of RL techniques and an emulation environment approach compared to traditional and simulation methods. PenGym is proposed as our solution for creating realistic environments to support training RL pentesting agents and accommodating diverse complexity network scenarios. By enabling the execution of actual actions with real observations, PenGym eliminates the need for the probabilistic modeling of actions, resulting in a more accurate representation of security dynamics compared to simulation-based environments.

Regarding the realism and reliability of our approach, PenGym can create successful cyber ranges that reflect the pentesting scenario, with actual hosts connecting to form a real network. Moreover, actual actions can be executed on these hosts to get information about running services and processes or to exploit existing vulnerabilities to compromise the hosts. The experimental results also indicated that agents trained in PenGym performed well, reaching pentesting goals across all scenarios when tested in a realistic environment. Therefore, this approach provides a realistic and reliable training environment to enhance the applicability of the trained agents when deployed in real-world infrastructure, while maintaining a reasonable training time.

We presented a set of experiments to emphasize the effectiveness of our approach. For the smallest scenario, agents trained in NASim, NASim(rev.), and PenGym environments achieved equivalent results when testing in both simulation and emulation environments. For the mid-size scenario, simulation-related issues occurred. Although the agents trained in the NASim environment performed well in the simulation environment, they were ineffective when testing in the emulation environment, showing a high variation and large attack step counts. In contrast, with a more accurate logical model, the NASim(rev.) environment yielded testing results comparable to PenGym in both the simulation and emulation environments. For the largest scenario, agents trained in NASim mostly failed when testing in PenGym environments, with very low success rates of the `ql` and `ql_replay` agents. The effectiveness of PenGym was clearly shown in these experiments. Although agents trained in NASim(rev.) reached the pentesting goal with reasonable attack steps across all test trials, their performance was inconsistent compared to PenGym, which had higher stability. The results also indicate that agents trained in PenGym slightly outperformed those trained in NASim(rev.) for all test trials in both simulation and emulation environments, with average step differences ranging from 1.4 to 8 steps. Thus, using PenGym for a realistic training environment enhances the applicability of the trained agents when deployed in a real-world infrastructure.

Due to its cyber range execution, PenGym requires more training time than NASim(rev.). However, it maintains a reasonable training duration, especially in more complex scenarios. For the largest scenario and most intricate RL algorithm, PenGym training takes around $17,000$ s compared to $14,000$ s in NASim, with a ratio of roughly 1.2.

We envision that PenGym could be integrated into current cybersecurity practices since RL agents trained via PenGym can realistically mimic attacker strategies and actions. This integration would make possible the automation of pentesting in the future, as RL agents trained via PenGym could assist pentesters in their work, further improving the effectiveness of cybersecurity assessment. The PenGym framework was released on GitHub (`https://github.com/cyb3rlab/PenGym`) as open source with full support for all the available scenarios, even the more complex ones.

8.2 Future Work

This research is a preliminary study into using a realistic environment for training automated RL pentesting agents and aiding in their application to real infrastructure. More scalable and efficient RL algorithms need to

be researched. These algorithms should be able to face the large scale of networks while handling thousands of possible actions.

Furthermore, we aim to propose a realistic automatic scenario generator for future research. Currently, the scenarios used in this study are manually created based on user purposes obtained from NASim work. Therefore, a method that can automatically generate scenarios resembling real-world systems is necessary to enhance the adaptability of pentesting RL agents.

Publications

1. **H. P. T. Nguyen.**, K. Hasegawa., K. Fukushima., and R. Beuran., "PenGym: Realistic training environment for reinforcement learning pentesting agents," *Computers & Security*, 2024. (Under Reviewed).

2. **H. P. T. Nguyen.**, Z. Chen., K. Hasegawa., K. Fukushima., and R. Beuran., "PenGym: Pentesting training framework for reinforcement learning agents," in *Proceedings of the 10th International Conference on Information Systems Security and Privacy - ICISSP*, (Rome, Italia), pp. 498–509, INSTICC, SciTePress, 2024.

3. **H. P. T. Nguyen** and S. Sakti, "Multilingual self-supervised visually grounded speech models," in *Proceedings of the 3rd Annual Meeting of the Special Interest Group on Under-resourced Languages @ LREC-COLING 2024* (M. Melero, S. Sakti, and C. Soria, eds.), (Torino, Italia), pp. 237–243, ELRA and ICCL, May 2024.

4. T. D. Le, **H. P. T. Nguyen**, K.-T. Huynh, and R. Beuran, "Smart grid cyber-attack analysis and countermeasures," in *2022 RIVF International Conference on Computing and Communication Technologies (RIVF)*, pp. 590–595, 2022.

References

[1] O. S. Thomas J. Holt and Y. T. Chua, "Exploring and estimating the revenues and profits of participants in stolen data markets," *Deviant Behavior*, vol. 37, no. 4, pp. 353–367, 2016.

[2] K. Chandrasekar, G. Cleary, O. Cox, H. Lau, B. Nahorney, B. O. Gorman, D. O'Brien, S. Wallace, P. Wood, and C. Wueest, "Internet security threat report," *Symantec Corp*, vol. 22, p. 38, 2017.

[3] A. C. S. Center, "ACSC threat report," *Australian Government*, 2017.

[4] R. Von Solms and J. Van Niekerk, "From information security to cyber security," *Computers & Security*, vol. 38, pp. 97–102, 2013. Cybercrime in the Digital Economy.

[5] F. M. Zennaro and L. Erdődi, "Modelling penetration testing with reinforcement learning using capture-the-flag challenges: Trade-offs between model-free learning and a priori knowledge," *IET Information Security*, vol. 17, no. 3, pp. 441–457, 2023.

[6] Q. Li, M. Hu, H. Hao, M. Zhang, and Y. Li, "INNES: An intelligent network penetration testing model based on deep reinforcement learning," *Applied Intelligence*, vol. 53, no. 22, pp. 27110–27127, 2023.

[7] Y. Stefinko, A. Piskozub, and R. Banakh, "Manual and automated penetration testing. benefits and drawbacks. modern tendency," in *2016 13th international conference on modern problems of radio engineering, telecommunications and computer science (TCSET)*, pp. 488–491, IEEE, 2016.

[8] B. Arkin, S. Stender, and G. McGraw, "Software penetration testing," *IEEE Security & Privacy*, vol. 3, no. 1, pp. 84–87, 2005.

[9] I. Arce and G. McGraw, "Guest editors' introduction: Why attacking systems is a good idea," *IEEE Security & Privacy*, vol. 2, pp. 17–19, July 2004.

[10] D. Maynor, *Metasploit toolkit for penetration testing, exploit development, and vulnerability research.* Syngess Publishing, Elsevier, 2011.

[11] C. Sarraute, "Automated attack planning," *arXiv preprint arXiv:1307.7808*, 2013.

[12] C. Phillips and L. P. Swiler, "A graph-based system for network-vulnerability analysis," in *Proceedings of the 1998 workshop on New security paradigms*, pp. 71–79, 1998.

[13] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[14] D. Silver, J. Schrittwieser, K. Simonyan, I. Antonoglou, A. Huang, A. Guez, T. Hubert, L. Baker, M. Lai, A. Bolton, *et al.*, "Mastering the game of go without human knowledge," *nature*, vol. 550, no. 7676, pp. 354–359, 2017.

[15] J. Kober, J. A. Bagnell, and J. Peters, "Reinforcement learning in robotics: A survey," *The International Journal of Robotics Research*, vol. 32, no. 11, pp. 1238–1274, 2013.

[16] S. Zhou, J. Liu, D. Hou, X. Zhong, and Y. Zhang, "Autonomous Penetration Testing Based on Improved Deep Q-Network," *Applied Sciences*, vol. 11, no. 19, 2021.

[17] Microsoft Defender Research Team, "CyberBattleSim." `https://github.com/microsoft/cyberbattlesim`, 2021. 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.

[18] R. Beuran, D. Tang, C. Pham, K. Chinen, Y. Tan, and Y. Shinoda, "Integrated framework for hands-on cybersecurity training: CyTrONE," *Computers & Security*, vol. 78C, pp. 43–59, 2018.

[19] J. Schwartz and H. Kurniawati, "Autonomous penetration testing using reinforcement learning," *arXiv:1905.05965*, 2019.

[20] M. C. Ghanem and T. M. Chen, "Reinforcement learning for efficient network penetration testing," *Information*, vol. 11, no. 1, p. 6, 2019.

[21] C. Sarraute, O. Buffet, and J. Hoffmann, "POMDPs make better hackers: Accounting for uncertainty in penetration testing," in *Proceedings*

of the AAAI Conference on Artificial Intelligence, vol. 26(1), pp. 1816–1824, 2012.

[22] M. C. Ghanem and T. M. Chen, "Reinforcement learning for intelligent penetration testing," in *2018 Second World Conference on Smart Trends in Systems, Security and Sustainability (WorldS4)*, pp. 185–192, IEEE, 2018.

[23] T. T. Nguyen and V. J. Reddi, "Deep reinforcement learning for cyber security," *IEEE Transactions on Neural Networks and Learning Systems*, vol. 34, no. 8, pp. 3779–3795, 2021.

[24] L. Li, J.-P. S. El Rami, A. Taylor, J. H. Rao, and T. Kunz, "Enabling A Network AI Gym for Autonomous Cyber Agents," in *2022 International Conference on Computational Science and Computational Intelligence (CSCI)*, pp. 172–177, IEEE, 2022.

[25] A. Furfaro, A. Piccolo, A. Parise, L. Argento, and D. Sacca, "A cloud-based platform for the emulation of complex cybersecurity scenarios," *Future Generation Computer Systems*, vol. 89, pp. 791–803, 2018.

[26] The MITRE Corporation, "BRAWL." `https://github.com/mitre/brawl-public-game-001`, 2018.

[27] K. Schoonover, E. Michalak, S. Harris, A. Gausmann, H. Reinbolt, D. R. Tauritz, C. Rawlings, and A. S. Pope, "Galaxy: a network emulation framework for cybersecurity," in *11th USENIX Workshop on Cyber Security Experimentation and Test (CSET 18)*, pp. 1–8, 2018.

[28] K. Pozdniakov, E. Alonso, V. Stankovic, K. Tam, and K. Jones, "Smart Security Audit: Reinforcement Learning with a Deep Neural Network Approximator," in *2020 International Conference on Cyber Situational Awareness, Data Analytics and Assessment (CyberSA)*, pp. 1–8, 2020.

[29] M. Standen, M. Lucas, D. Bowman, T. J. Richer, J. Kim, and D. Marriott, "CybORG: A gym for the development of autonomous cyber agents," in *Proceedings of the 1st International Workshop on Adaptive Cyber Defense*, pp. 1–7, August 2021.

[30] A. Molina-Markham, C. Miniter, B. Powell, and A. Ridley, "Network environment design for autonomous cyberdefense," *arXiv:2103.07583*, 2021.

[31] J. Janisch, T. Pevný, and V. Lisý, "NASimEmu: Network attack simulator & emulator for training agents generalizing to novel scenarios," in *Computer Security. ESORICS 2023 International Workshops*, (Cham), pp. 589–608, Springer Nature Switzerland, 2024.

[32] R. R. Linde, "Operating system penetration," in *Proceedings of the May 19-22, 1975, national computer conference and exposition*, pp. 361–368, 1975.

[33] F. Holik, J. Horalek, O. Marik, S. Neradova, and S. Zitta, "Effective penetration testing with metasploit framework and methodologies," in *2014 IEEE 15th International Symposium on Computational Intelligence and Informatics (CINTI)*, pp. 237–242, IEEE, 2014.

[34] G. F. Lyon, *Nmap network scanning: The official Nmap project guide to network discovery and security scanning*. Insecure, 2009.

[35] S. Chaudhary, A. O'Brien, and S. Xu, "Automated post-breach penetration testing through reinforcement learning," in *2020 IEEE Conference on Communications and Network Security (CNS)*, pp. 1–2, IEEE, 2020.

[36] G. Brockman, V. Cheung, L. Pettersson, J. Schneider, J. Schulman, J. Tang, and W. Zaremba, "OpenAI Gym," *arXiv:1606.01540*, 2016.

[37] P. Mell, K. Scarfone, and S. Romanosky, "Common vulnerability scoring system," *IEEE Security & Privacy*, vol. 4, no. 6, pp. 85–89, 2006.

[38] G. Lyon, "Nmap security scanner." https://nmap.org/, 2014.

[39] S. Donnelly, "Soft target: The top 10 vulnerabilities used by cybercriminals," *Recorded Future*, 2018.

[40] A. Norman, "Python-Nmap." https://pypi.org/project/python-nmap/, 2021.

[41] D. McInerney, "Pymetasploit3." https://pypi.org/project/pymetasploit3/, 2020.

[42] Rapid7 Vulnerability & Exploit Database , "SSH Logic Check Scanner." `https://www.rapid7.com/db/modules/auxiliary/scanner/ssh/ssh_login/`, 2018. Accessed: July 23, 2024.

[43] Rapid7 Vulnerability & Exploit Database , "VSFTPD v2.3.4 Backdoor Command Execution ." `https://www.rapid7.com/db/modules/exploit/unix/ftp/vsftpd_234_backdoor/`, 2018. Accessed: July 23, 2024.

[44] National Vulnerability Database, "CVE-2011-2523." https://nvd.nist.gov/vuln/detail/CVE-2011-2523, 2021. Accessed: July 23, 2024.

[45] Rapid7 Vulnerability & Exploit Database , "Apache 2.4.49/2.4.50 Traversal RCE." https://www.rapid7.com/db/modules/exploit/multi/http/apache_normalize_path_rce/, 2021. Accessed: July 23, 2024.

[46] National Vulnerability Database, "CVE-2021-41773." https://nvd.nist.gov/vuln/detail/CVE-2021-41773, 2023. Accessed: July 23, 2024.

[47] Rapid7 Vulnerability & Exploit Database , "Samba is_known_pipename() Arbitrary Module Load." https://www.rapid7.com/db/modules/exploit/linux/samba/is_known_pipename/, 2018. Accessed: July 23, 2024.

[48] National Vulnerability Database, "CVE-2017-7494." https://nvd.nist.gov/vuln/detail/CVE-2017-7494, 2022. Accessed: July 23, 2024.

[49] Rapid7 Vulnerability & Exploit Database , "OpenSMTPD MAIL FROM Remote Code Execution." https://www.rapid7.com/db/modules/exploit/unix/smtp/opensmtpd_mail_from_rce/, 2020. Accessed: July 23, 2024.

[50] National Vulnerability Database, "CVE-2020-7247." https://nvd.nist.gov/vuln/detail/CVE-2020-7247, 2024. Accessed: May 2, 2023.

[51] Rapid7 Vulnerability & Exploit Database , "Local Privilege Escalation in polkits pkexec." https://www.rapid7.com/db/modules/exploit/linux/local/cve_2021_4034_pwnkit_lpe_pkexec/, 2022. Accessed: July 23, 2024.

[52] National Vulnerability Database, "CVE-2021-4034." https://nvd.nist.gov/vuln/detail/CVE-2021-4034, 2021. Accessed: May 2, 2023.

[53] Rapid7 Vulnerability & Exploit Database , "ProFTPD-1.3.3c Backdoor Command Execution." https://www.rapid7.com/db/modules/exploit/unix/ftp/proftpd_133c_backdoor/, 2018. Accessed: July 23, 2024.

[54] National Vulnerability Database, "CVE-2015-3306." https://nvd.nist.gov/vuln/detail/CVE-2015-3306, 2021. Accessed: May 2, 2023.

[55] Rapid7 Vulnerability & Exploit Database , "Cron Persistence." https://www.rapid7.com/db/modules/exploit/linux/local/cron_persistence/, 2018. Accessed: July 23, 2024.

[56] I. Habib, "Virtualization with KVM," *Linux Journal*, vol. 2008, no. 166, p. 8, 2008.

[57] R. S. Sutton and A. G. Barto, *Reinforcement learning: An introduction.* MIT press, 2018.

[58] C. J. Watkins and P. Dayan, "Q-learning," *Machine learning*, vol. 8, pp. 279–292, 1992.

[59] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski, *et al.*, "Human-level control through deep reinforcement learning," *nature*, vol. 518, no. 7540, pp. 529–533, 2015.

[60] M. Sutton, A. Greene, and P. Amini, *Fuzzing: brute force vulnerability discovery.* Pearson Education, 2007.

[61] J. Schwartz and H. Kurniawatti, "NetworkAttackSimulator." https://github.com/Jjschwartz/NetworkAttackSimulator/tree/master, 2023. Accessed: July 23, 2024.

[62] H. P. T. Nguyen, "The firewall does not affect the exploit action from the Internet to the connected subnet." https://github.com/Jjschwartz/NetworkAttackSimulator/issues/45, 2023. Accessed: July 23, 2024.

[63] H. P. T. Nguyen, "Service Scan does not take firewall into account." https://github.com/Jjschwartz/NetworkAttackSimulator/issues/46, 2023. Accessed: July 23, 2024.

[64] H. P. T. Nguyen, "The Remote Action (Exploit action) can be executed on its target host." https://github.com/Jjschwartz/NetworkAttackSimulator/issues/47, 2023. Accessed: July 23, 2024.