



UNIVERSITAT OBERTA DE CATALUNYA (UOC)

DATA SCIENCE MSc

## MASTER'S THESIS

AREA: 3

# Analysis of reinforcement learning techniques applied to honeypot systems

---

Author: Oriol Navarro Ferrer

Tutor: Blas Torregrossa Garcia

Professor: Ferran Prados Carrasco

---

Barcelona, January 3, 2021



# Copyright



This work is subject to Attribution-NonCommercial-NoDerivatives 4.0 International ([CC BY-NC-ND 4.0](#)).



# Thesis Datasheet

Title:	Analysis of reinforcement learning techniques applied to honeypots
Author:	Oriol Navarro Ferrer
Tutor:	Blas Torregrosa Garcia
Responsible Professor:	Ferran Prados Carrasco
Date (mm/aaaa):	01/2020
Program:	MSc Data Science
Master's Thesis area:	Area 3, Cybersecurity
Language:	English
Keywords	Reinforcement learning, honeypot systems, self-adaptive honeypot, threat intelligence, reward functions



”The most beauty aspect of Computer Science, compared to other domains, is that we share information. This is not as common as it could be, and it will change the World”.

*Gerardo García Peña, RootedCon 2012*





# Acknowledgements

To my family, friends, colleagues and acquaintances who have been and will be a pillar on this and other endeavours.



# Abstract

The study of cybersecurity threats is an increasingly relevant element for public and private organizations, due to the increasing number of cyber attacks and their impact on the organizations assets and their reputation. Collecting detailed information that allows to determine how future attacks will be is key to anticipate the organizations' defenses. Tactics, techniques and procedures used by threat actors can be collected using several approaches, one being honeypot systems. The effectiveness of these attack information collection targets depend significantly on their ability to present a realistic environment that can lure attackers to reveal their techniques. This project presents a study of designs and implementations of adaptive honeypots, focused on the use of reinforcement learning, to reach more realistic interactions between honeypots and attackers, and an analysis of the existing techniques and performance metrics.

**Keywords:** Reinforcement learning, honeypot systems, self-adaptive honeypot, threat intelligence, reward functions.



# Contents

<b>Abstract</b>	<b>ix</b>
<b>Contents</b>	<b>xi</b>
<b>List of Figures</b>	<b>xiii</b>
<b>List of Tables</b>	<b>1</b>
<b>1 Introduction</b>	<b>3</b>
1.1 Introduction . . . . .	3
1.1.1 Project description . . . . .	4
1.1.2 Motivation . . . . .	4
1.1.3 Goals . . . . .	4
1.1.4 Methodology . . . . .	5
1.1.5 Planning . . . . .	5
<b>2 State of the Art</b>	<b>7</b>
2.1 State of the Art . . . . .	7
2.1.1 Taxonomy of honeypot systems . . . . .	7
2.1.2 Types of adaptive honeypots . . . . .	8
2.2 Reinforcement learning . . . . .	9
2.2.1 SARSA . . . . .	10
2.2.2 Q-learning . . . . .	10
2.2.3 Deep Q-learning . . . . .	11
2.3 Reinforcement learning usage on the analyzed honeypots . . . . .	11
2.3.1 Heliza . . . . .	11
2.3.2 RASSH . . . . .	12
2.3.3 RLHPot/HARM . . . . .	12
2.3.4 QRASSH/IRASSH . . . . .	13

<b>3</b>	<b>Analysis of the impact of reinforcement learning on adaptive honeypots</b>	<b>17</b>
3.1	Analysis of the impact of reinforcement learning on adaptive honeypots . . . . .	17
3.1.1	Parameterization options on QRASSH/IRASSH . . . . .	17
3.1.2	Reward function parameterization on IRASSH . . . . .	18
3.2	Cowrie honeypot data analytics . . . . .	19
3.2.1	Cowrie testing environment . . . . .	19
3.2.2	Logs format . . . . .	19
3.2.3	Analytics . . . . .	20
3.3	Metrics and evaluation of adaptive honeypots . . . . .	27
3.3.1	Development of deployment environment . . . . .	27
3.3.2	Proposed metrics on honeypots evaluation . . . . .	28
3.3.3	IRASSH parameterizations from Cowrie analysis . . . . .	30
3.3.4	Evaluation of the different adaptive honeypot parameterizations . . . . .	31
<b>4</b>	<b>Conclusions and future lines of work</b>	<b>37</b>
4.1	Conclusions . . . . .	37
4.1.1	Limitations . . . . .	37
4.1.2	Results . . . . .	38
4.2	Future lines of work . . . . .	39
4.2.1	Adaptive modification of honeypots configurations . . . . .	39
4.2.2	Graph analysis on honeypot data . . . . .	40
4.2.3	Reward functions additional parameterizations . . . . .	40
4.2.4	Deployment automation and data analysis . . . . .	40
<b>5</b>	<b>Annex 1</b>	<b>43</b>
5.1	Code listings . . . . .	43
5.1.1	QRASSH/IRASSH honeypot.py . . . . .	43
5.1.2	QRASSH/IRASSH nn.py . . . . .	45
	<b>References</b>	<b>45</b>

# List of Figures

1.1	Project planning. . . . .	6
2.1	Markov decision process diagram. . . . .	9
2.2	Shape of QRASSH/IRASSH neural network. . . . .	16
3.1	Kibana Cowrie dashboard on T-Pot. . . . .	20
3.2	Treemap of top recorded commands in Cowrie. . . . .	22
3.3	Treemap most common session lengths for compound commands on Cowrie. . .	22
3.4	Cowrie session lengths recorded per each day (anomalies removed). . . . .	24
3.5	Cowrie download commands recorded per each session and day. . . . .	25
3.6	Sample graph from 30 Cowrie commands analysis. . . . .	27
3.7	Complete graph from Cowrie commands analysis. . . . .	28
3.8	Comparison of command lengths for each session. . . . .	32
3.9	Comparison of download commands issued per day averages. . . . .	34
3.10	Comparison of total number of download commands issued per day. . . . .	35





# List of Tables

2.1	<i>Algorithms used on each of the honeypots analyzed.</i>	11
3.1	<i>Top commands recorded by Cowrie by times of appearance.</i>	21
3.2	<i>Download commands recorded in Cowrie.</i>	25
3.3	<i>Cowrie commands with more graph edges.</i>	29
3.4	<i>Most connected commands on each honeypot configuration.</i>	33



# Chapter 1

## Introduction

### 1.1 Introduction

Cybersecurity is a growing concern due to the increasing number of attacks suffered by organizations worldwide and the relevance of these events [10]. Both as direct impacts (data leakage, extortion attacks, etc...), and as potential reputational impact deriving from these attacks.

Preventive measures against cyberattacks generally leverage on cyber threat intelligence: knowledge on cyber threats and threat actors that can be used to prepare against them, or mitigate the impact of their actions. A specific component of this knowledge are the attackers' tactics, techniques and procedures (TTPs) that are developed by threat actors. TTPs consist of the description of attack techniques and specially-crafted tools that are designed to access organizations' assets and perform a desired number of actions. The MITRE ATT&CK framework defines a standardization of these techniques classification [3].

This knowledge can be attained by several potential sources of information. One common approach is the usage of simulated environments, isolated from productive systems, which are prepared to lure attackers and track their activity for analysis purposes. These systems are known as honeypots. The teams managing these simulated platforms gather the telemetry created by the attacker's interactions with the system in order to identify procedures, tools, and practices in use. These systems commonly provide detailed logs of all attackers interactions, from the establishment of a connection to a register of all the actions performed. A trained analyst can use this data to gather knowledge on the attackers current TTPs and design countermeasures and other components to prevent them, or in the worst case, react against them.

Honeypots are commonly limited in effectiveness by being easily identifiable and prone to evasion by attackers with a sufficient knowledge level [5]. This last fact results on a lack of information about potential future attacks, that can leave organizations with insufficient

awareness, prevention and reaction capabilities.

There are different approaches on honeypot design that are able to react on more realistic ways to attackers' interactions. Those are classified as adaptive honeypots. Research in artificial intelligence techniques on honeypot development has been published since before 2010 [19], [1].

Seamus Dowling summarizes a number of projects that achieve different techniques and goals related to enhancing honeypot systems via reinforcement learning [5]. The papers [5] and [15] mention the usage of reinforcement learning techniques with specific algorithms and metrics as rewards. Dowling [5] uses as a success metric the fact that the attacker has a longer interaction with the honeypot. Pauna [15] prioritizes potential files downloads as an attempt to obtain malware samples used by attackers.

This information is further analyzed in the chapter "State of the Art" 2.

### 1.1.1 Project description

The observations and metrics presented on previous research describe different approaches to complementary goals. This Master's Thesis will analyze the data obtained by using different adaptive honeypot parameterizations and explore potential alternatives, or even improvements, to the analysis performed on the aforementioned projects.

### 1.1.2 Motivation

The usage of honeypots and other threat intelligence gathering systems is a relevant asset for new attack patterns (TTPs) investigation. Strategies to protect organizations' information systems leverage this knowledge in order to prepare defenses for those identified threats. Data science can be applied to this process from various perspectives [16].

An analysis of several publicly available projects, such as Cowrie [12], reveal that those are normally based on automated responses to attackers actions, and described via policies. Honeypots based on static configurations or policies are limited by the ability of attackers to detect or even evade them, interrupting the interaction. Knowledge derived from the resulting information gathered is therefore limited. Adaptive honeypots improve the number of commands and information issued by attackers in the system, capturing more complex TTPs [5].

This project will attempt to further study the details of adaptive honeypot gains compared to non-adaptive systems.

### 1.1.3 Goals

The main goals for this project are the following:

- Analyze the state of the art on data science techniques applied on adaptive honeypots, focusing on reinforcement learning.
- Study the algorithms and metrics used on reinforcement learning agents, identifying those components with a higher relevance on the overall results.
- Analyze the captured honeypots logs with data exploration and visualization approaches.
- Study potential improvements on current research in order to obtain honeypot systems that allow for better knowledge gathering.

The following additional goals will be approached during the project:

- Modify existing projects' parameterization of reinforcement learning implementations, aiming for changes on the reward functions outcomes.
- Develop an automated deployment of testing environments for the honeypots analyzed.

#### 1.1.4 Methodology

The project will be based on the following methodology:

- Analysis of papers and documentation referring to reinforcement learning honeypots, identifying parameterization and metrics in place.
- Analytic approach to modifications on the projects to study behavioural changes, and even potential improvements.
- Definition of relevant metrics for reward functions evaluation on reinforcement learning implementations and their impact on attackers interactions.
- Perform an evaluation on the proposed changes.
- Explore data analytics techniques that will be used to represent and visualize the honeypot captured data

#### 1.1.5 Planning

The planning has been defined with consideration to the project's goals and methodology, the thesis preparation and its defense:

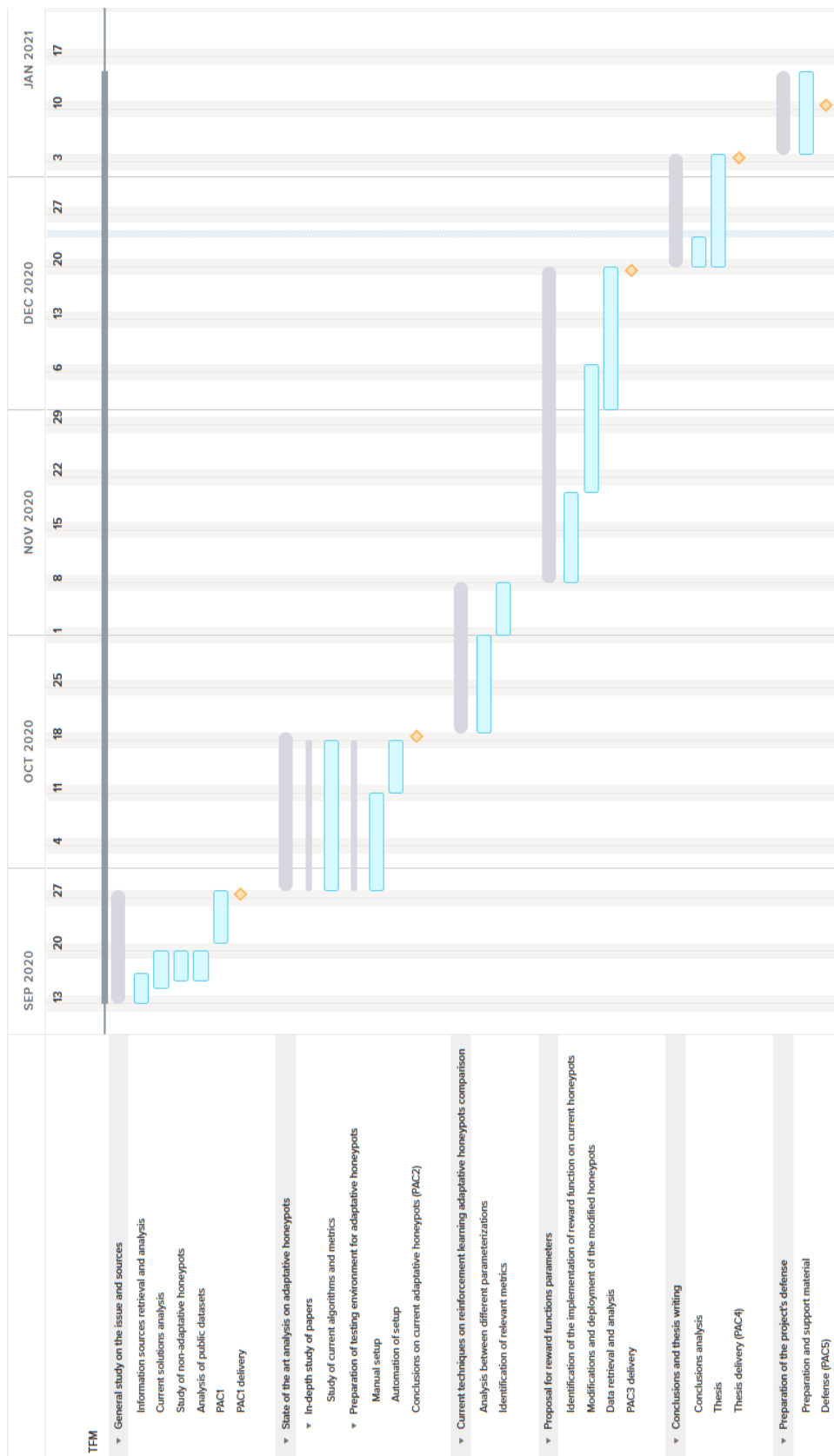


Figure 1.1: Project planning.

# Chapter 2

## State of the Art

### 2.1 State of the Art

Adaptive honeypots have been proposed as an improvement to classic honeypots based on fixed policies, static configuration files and similar approaches. Non-adaptive systems require manual updates and administrators actions to prepare for new techniques used by attackers or malicious software for evading them [5], limiting the capture of up-to-date cyber threat intelligence. This section outlines how new types of honeypots relying on machine learning or deep learning capabilities fit into classic honeypot taxonomies, and describes their main objectives and characteristics.

#### 2.1.1 Taxonomy of honeypot systems

A classic honeypot taxonomy is outlined in Seifert's 2006 paper [17]. This early analysis classifies honeypots under certain criteria, considering their interaction level, data capture capabilities, containment approach, distribution appearance, communications interface and role in multi-tiered architectures.

Dowling, on his paper's conclusions [5] proposes the addition of a new item on the *interaction level* class:

- Adaptive: Learns from attack interaction.

This addition to the taxonomy implies a new design paradigm that allows to achieve additional goals on honeypot development. Honeypots based on static parameterization can collect TTPs from human or automated attackers that are not prepared for evading them. Adaptive honeypots can adjust to new evasion techniques, optimizing threat intelligence collection.

The following types of honeypots are classified considering their interaction level:

- Low interaction honeypots present a simple, sometimes non-functional service, targeting the collection of basic indicators of compromise (IOCs) that can provide an overview of IP addresses, usernames and passwords attempted, etc... being used by threat actors.
- High interaction honeypots present a complete operating system that can emulate the interaction of the attacker with a full-fledged system. They allow the observation of more complex TTPs. The complexity for developing and configuring these systems is also higher, and an attacker can test random operating system functionalities to determine if the system is a honeypot.
- Adaptive honeypots can learn from attacker's behaviour, luring the attacker to reveal its TTPs by adapting the responses by learning from previously observed techniques.

### 2.1.2 Types of adaptive honeypots

Research on adaptive honeypots follows a common goal, obtaining more knowledge on attackers operations by being able to deceive them on thinking that the honeypot is a real system. This project focuses on the usage of reinforcement learning algorithms for enhancing interaction with the potential attackers, and will study several honeypots developed using these techniques.

Wagener [19] presents the use of reinforcement learning in order to increase attackers interactions on the Heliza honeypot. Pauna [13] proposes a similar approach on reinforcement learning with a more complex honeypot implementation with RASSH. Both of these honeypots attempt on improving its interaction with humans by adapting to complex behaviours. Further research from Adrian Pauna presents an IoT-specific honeypot (IRASSH) [15] which focuses on collecting malware samples, and is derived from QRASSH, a honeypot that uses Deep Q-Learning to leverage honeypot interaction [14].

Other research attempts to develop honeypots that can detect automated evasion tools. The RLHPot [5] implements such techniques, moving away from Heliza and RASSH concepts, and optimizing its approach to capture data created by automated agents attacking the honeypot. Seamus Dowling describes more in detail the design of this honeypot [4], also known as HARM in his PhD dissertation.

As a summary, adaptive honeypots which use reinforcement learning are a set of designs that include various algorithms and techniques depending on their approach to the problem. Relevant points in their differences are the usage of different algorithms, and how their reward functions assign different success metrics in these cases, which will drive the evaluation of potential attacks to have a more or less significant impact on honeypots optimization.



## 2.2 Reinforcement learning

As described in [18], reinforcement learning is a machine learning area which studies the learning from interaction by automated agents. It approaches problems starting with an unlabelled set of data and decides on actions based on its current knowledge.

The problem, particularly on the algorithms studied in this project, is modelled as a Markov decision process (MDP). It consists of the following elements:

- Task: An instance of the reinforcement learning problem.
- Reinforcement learning agent: The entity which is acting as a decision-maker for the problem. Takes actions based on the interaction with the environment.
- Environment: The set of different situations that are presented to the agent, modelled as a set of states.
- Steps: Discrete time interval in which the agent and the environment interact.
- State: Representation of the environment at a given step received by the agent.
- Action: The agent selects one of the actions available in the state on a given step.
- Reward: Numerical value received by the agent on the next step, which represents the result from the previous action.
- Policy: Function representing the probability of moving to a given state based on taking an action. The policy maximizes the cumulative reward obtained.

These elements and their interactions are summarized on figure 2.1, reproduced from [18] .

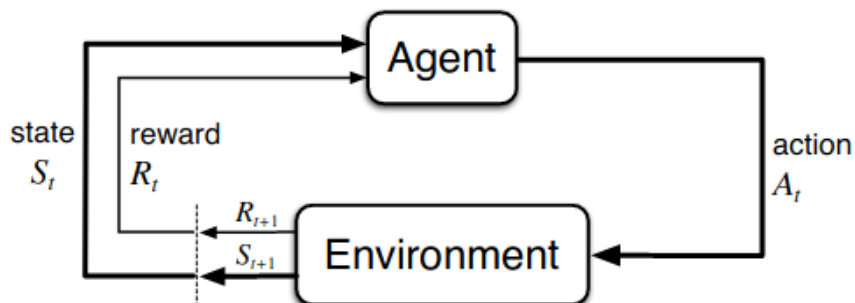


Figure 2.1: Markov decision process diagram.

In these kind of systems, a process representation through states, actions, transition probabilities and reward, is used. Every certain time period the system's agents that are representing

a specific state must perform a transition through one of the actions available for that state, and getting to a new state. When executing an action, a reward is assigned to the agent. The goal for these agents is to get a higher reward over a long-time period by choosing the actions that will maximize it.

This model suits the honeypots problem, since it can work on an environment that can only be described by interacting with it. The reinforcement agent has to interact with other agents (the attackers) that provide the next environment state after an action has been presented to them.

An additional important element on MDP-described reinforcement learning problems are the *Q-values*. They represent the state-action values, which are the coupling of the state  $s$  and the action  $a$ , which result on the state  $s'$ . An MDP-modelled problem will aim for the optimization of these values for choosing the best possible outcome based on its current knowledge.

### 2.2.1 SARSA

Some of the honeypots analyzed during this project implement a State, Action, Reward, State, Action (SARSA) algorithm. It defines the following function for updating *Q-values* [2]:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)]$$

The function for updating *Q-values* depends on the current state of the agent, the action that the agent chooses, the reward for choosing this action, the new state after the action and the next action the agent chooses in the new state. The agent updates the policy (it's an *on-policy* algorithm) after the actions are taken. The *Q-values* represent the possible reward that will be received in the next step for taking the action  $a$  in state  $s$  plus the future one that will be received from the next interaction [18].

### 2.2.2 Q-learning

Q-learning is an algorithm which is similar to SARSA, but which always applies a greedy policy. It will evaluate the maximum values obtained, the best action [18]. Q-learning implementations keep a table (*Q-table*) that contains the values of applying the *Q-function* to the possible states and actions, formalized in the following expression:

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)]$$

### 2.2.3 Deep Q-learning

A variant of Q-learning used in some of the analyzed honeypots is Deep Q-learning. Instead of keeping a *Q-table* with all possible *Q-values*, the design of Deep Q-learning approximates the *Q-values* with a neural network model [9].

The states are the inputs to the neural network, which computes the possible *Q-values* of all possible actions are returned. The algorithm will then choose the maximum value, that corresponds to the maximum reward [11].

## 2.3 Reinforcement learning usage on the analyzed honeypots

The projects currently analyzed include the following reinforcement learning algorithms on their designs (table 2.1):

Project	RL Algorithm
Heliza	SARSA
RASSH	SARSA
RLHPot/HARM	SARSA/Q-learning
IRASSH	Deep Q-learning
QRASSH	Deep Q-learning

Table 2.1: *Algorithms used on each of the honeypots analyzed.*

The majority of projects under analysis stem from the lines of research started with Heliza and RASSH, and are therefore using an evolved version of a similar reinforcement learning agent based on SARSA, modifying other honeypot aspects and also their reward functions. The use of neural networks is a promising direction for new research, as outlined by Pauna [15].

The details of the implementation of each of the honeypots analyzed in this project will be extended on the upcoming sections.

### 2.3.1 Heliza

Heliza creators presented a new paradigm in 2010 for the application of reinforcement learning to adaptive honeypots [19]. The problems are modelled as a Markov Decision Process using the SARSA algorithm.

Basically, the system intercepts the commands issued by the attacker and decides which is the best action to be applied in order to maximize the future reward, derived from the scores assigned to each command.

This paper introduces how the problem is approached, which has been followed by other research cited in this project:

- Environment: Corresponds to the attackers connecting to the honeypot.
- States: Each of the commands that can be issued by an attacker plus the honeypot states. "Insult" (the honeypot has answered to the attacker with an insult), "custom" (the attacker issues a custom command) or "empty" (the attacker types a *enter* keystroke with no command).
- Actions: Allows, blocks, substitute the command or insult the attacker. The latter action attempts to perform a reverse Turing Test to determine if the attacker is an automated tool or a human.
- Rewards: Heliza has two different reward sets. Each of the commands recognized by the honeypot has a score assigned, depending on the goals (there is a policy for keeping the attackers as long as possible to collect more complex intelligence, and a policy for prioritizing custom tools downloads by the attackers).
- The state-agent function works with an  $\epsilon$ -greedy method.

### 2.3.2 RASSH

The paper on RASSH [13], by A. Pauna and I. Bica, follows the research presented by G. Wagener, R. State, A. Dulaunoy and T. Engeland on Heliza. RASSH focuses on a new implementation on Python as a fork of Kippo (a classic non-adaptive honeypot that has been succeeded by Cowrie). While its implementation is new (based on Pybrain), the reinforcement learning design and application have been evolved from Heliza's.

It uses the SARSA algorithm, also with an  $\epsilon$ -greedy method, and implements a new action "delay" (it delays the execution of the commands issued by the attacker). It also changes the "substitute command" action for a "fake command" one, which provides a fake output to the attacker.

The results stated on the paper confirm that the use of standard Python libraries makes for a more scalable and extensible honeypot compared to Heliza, while achieving similar results on the same data set.

### 2.3.3 RLHPot/HARM

Seamus Dowling, Michael Schukat and Enda Barrett studied common honeypot evasion mechanisms [5], and summarized previous research on adaptive honeypots. They presented a new

implementation based on Cowrie, which is named RLHpot. In Dowling's PhD dissertation [4] this honeypot is named as HARM.

The aim of RLHpot is to increase command transitions to overcome honeypot detection techniques. It is based on the RASSH framework, although its performance is better as identified on Dowling's dissertation. It also uses SARSA, but the code can be easily modified for switching to Q-learning.

RLHpot distinguishes among different categories of commands (native GNU/Linux commands, custom attack commands, compound commands, unknown commands and others). A significant difference is a reduced set of actions, "allow", "block" and "substitute command". This reduced action set allows for a faster convergence.

### 2.3.4 QRASSH/IRASSH

QRASSH was presented on [15]. It uses the RASSH framework and keeps its action set, but it differs from previous research implementing a Deep Q-learning algorithm.

QRASSH is written as an extension to Cowrie. This consists on a substantial modification of this honeypot in order to perform additional actions whenever a command is received. This extension adds several components to the base Cowrie project. It implements a reinforcement learning agent and the supporting modules around it. The components which are specifically relevant for this project are:

- `src/irassh/shell/honeypot.py`: This modified file for the QRASSH/IRASSH honeypots overrides the default Cowrie behaviour and calls the proxy component. It instantiates the proxy element, which will intercept the commands issued to the honeypot and return which action to take. The `honeypot.py` component will either execute the action or return the defined messages according to the proxy's decision.

The code listing on Annex 1 5.1.1 contains the instantiations of the proxy component, the honeypot call to proxy to determine which action to take:

```
actionValid = validator.validate(raw_cmd, self.protocol.clientIP)
```

And the execution of the command is valid.

```
if pp and actionValid:
    self.protocol.call_command(pp, cmdclass, *cmd_array[0]['rargs'])
```

If the resulting action doesn't allow the command execution, the class *StdOutStdErrEmulationProtocol* is invoked, which implements an interface to return different errors to the attacker interacting with the honeypot.

```
StdOutStdErrEmulationProtocol(self.protocol, cmdclass,
                               cmd['rargs'], None, None)
```

- `src/irassh/actions/proxy.py`: Corresponds to the element that selects which of the actions will be taken by the reinforcement learning agent. It instantiates this agent and interacts with it on each intercepted command. It also implements the different responses that will be given to `honeypot.py` for each possible decision by the learning agent. The different actions have their own implementation. While "allow" and "delay" permit the execution of the command intercepted, the rest of actions return different error messages.

A sample of one of these actions ("allow") can be found on the following listing. The class attribute modifier `setPassed(True)` will instruct the `honeypot.py` component to execute the actual command:

```
class AllowAction(Action):
    def process(self):
        self.setPassed(True)

    def getActionName(self):
        return "Allow"

    def getColor(self):
        return "32"
```

- `src/irassh/rl/learning.py`: Implements the *q-learner* class, which is the abstraction for the reinforcement learning agent, and contains most of its parameterization, including the reward function parameters.

The function that chooses actions first calculates a random float between 0 and 1, if the result is smaller than the configured  $\epsilon$  or the training process is shorter than a configured time-frame, a random action will be chosen. Otherwise, the neural network will be instantiated, returning a set of *Q-values* for each action. The maximum value will be the action performed by the honeypot as a response to the attacker's issued command.

```
def choose_action_and_train(self, state):
    if len(state.shape) == 1:
        state = np.expand_dims(state, axis=0)
    self.t += 1
    self.hacker_cmds += 1

    if random.random() < self.epsilon or self.t < self.observe:
        action = np.random.randint(0, self.number_of_actions)
```

```
    else:
        qval = self.model.predict(state, batch_size=1)
        action = (np.argmax(qval))
    self.lastAction = action
    return action
```

- `src/irassh/rl/nn.py`: Implements the neural network used for this project, as stated in the original QRASSH paper [15]. The neural network shape is rather simple. It uses one input layer, two hidden layers and an output layer, as depicted in figure 2.2.

The *neural\_net* function on `nn.py` implements the neural network, the code listing is included on Annex 1 5.1.2. After developing a new reward function configuration, the input layer needs to be adjusted to the number of commands included in the file.

- `src/irassh/rl/cmd2number_reward.p`: Contains the reward function scores in Pickle format. Further detail on this file format is provided in section 3.1.1, which details the reward function parameterizations on QRASSH/IRASSH.

IRASSH further develops this concept by adding a manual expert policy, which allows for a more convenient way of creating Pickle files with reward values. Other than that, the functional level is similar to QRASSH.

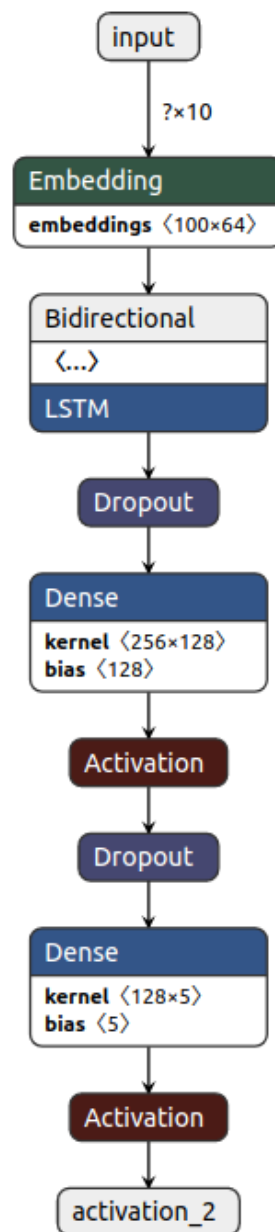


Figure 2.2: Shape of QRASSH/IRASSH neural network.



# Chapter 3

## Analysis of the impact of reinforcement learning on adaptive honeypots

### 3.1 Analysis of the impact of reinforcement learning on adaptive honeypots

Both QRASSH and IRASSH are an extension of the original Cowrie Honeypot (an SSH non-adaptive medium-interaction honeypot) that include a Deep Q-Learning reinforcement learning agent.

This chapter will describe with more detail the impact of reinforcement learning on the honeypot IRASSH comparing the attackers behaviours registered to the ones collected by an also deployed Cowrie instance. IRASSH has been used since it was implemented using newer libraries and functionalities than QRASSH. When not using the additional features that IRASSH introduces, its functional level is equivalent to QRASSH.

The study will apply data analytics and visualization techniques in order to obtain knowledge on the effect of modifications to the reinforcement learning algorithm parameterization. The results of this analysis have been used as an input for reward functions targeting several knowledge gathering purposes identified in previous research, as well as a case that has been developed by analyzing attackers behaviours as graph sequences. The results have been compared using specific metrics on section [3.3](#).

#### 3.1.1 Parameterization options on QRASSH/IRASSH

The aim of adaptive honeypots, as detailed in chapter [2](#), is to apply a more realistic interaction between the attacker and the honeypot based on different possible actions implemented on the honeypot considering the attackers behaviours.

Both RASSH and QRASSH apply a number of reactions to any command issued by the attacker. The honeypots will either "allow", "block", "delay", "answer with fake information" or "insult" (send a message to the attacker to identify if it's an automated agent or a human interacting with the honeypot) as an output from the reinforcement learning agent decision.

Additionally, IRASSH supports the configuration of expert policies. It assists the honeypot administrator by recording specific events in a lab environment and creating a Pickle configuration file that will contain reward function scores tailored for the desired behaviour. This configuration can also be created manually by crafting a Pickle file, albeit this IRASSH functionality is more convenient. However, this feature is not currently working on the last publicly available version of IRASSH. During this project, files have been created manually, using some additional Python scripts to encode and decode Pickle files.

### 3.1.2 Reward function parameterization on IRASSH

In QRASSH/IRASSH the reward function scores are numeric values assigned to each kind of command that is received on the honeypot (including the unknown ones). The reinforcement learning agent will read the current command issued by the attacker and process the best action-state value considering these scores. This enables one of the goals of this project, which is analyzing the behaviours presented by attackers on a non-adaptive honeypot and trying to achieve configurations that will help prioritize the gathering of knowledge on specific cyberattacks.

The modification of the reward function values can be performed by modifying the file `src/irassh/rl/cmd2number_reward.p`.

The following example contains a sample of the policy in text format, which is then transformed to a Pickle file with the `src/bin/lines2pickle.py` file. A decode Python program has been created for this project in order to analyze the default included Pickle files (`src/bin/decode-pickle.py`)

```
{
  "unknown": [0, 0],
  "mkdir": [1, 200],
  "wget": [2, 500],
  "uname": [3, 0],
  "nc": [4, 500]
}
```

## 3.2 Cowrie honeypot data analytics

Cowrie, as a medium-interaction honeypot without reinforcement learning, just presents the attackers connecting to it with a standard GNU/Linux bash interface. It intercepts the commands that the attacker issues and substitutes them for controlled environment implementations. The purpose is leading the attacker to think that he performed an action on the system, but the honeypot keeps him in its sandbox.

Since Cowrie is a traditional non-adaptive honeypot, it has been a useful benchmark which the adaptive-honeypot captured data can be compared to.

### 3.2.1 Cowrie testing environment

The deployment of Cowrie has been performed using the open-source project T-Pot (developed by Deutsche Telekom employees [8]). It includes a number of honeypot systems running on Docker containers, Cowrie being the most interesting for this project, since it presents an SSH (and Telnet) interface to a sandboxed GNU/Linux system, which has been forked in the adaptive honeypot analyzed projects. The Cowrie honeypot using T-Pot has been deployed on an Amazon Web Services instance running for approximately two months.

The logs have not been continuously recorded for several days due to instability issues on the Cowrie deployment. Since Cowrie is not modifying its behaviour due to the previously recorded events, this did not affect the relevance of the collected data.

Figure 3.1 shows the Kibana dashboard for the tested environment.

### 3.2.2 Logs format

Cowrie and its derivatives (including QRASSH/IRASSH) use the following log format in json:

```
{"eventid":"cowrie.command.input","input":"free -m | grep Mem |  
  awk '{print $2 , $3, $4, $5, $6, $7}'","message":"CMD: free -m  
  | grep Mem | awk '{print $2 , $3, $4, $5, $6, $7}'","sensor":"1  
cb98149b5ab","timestamp":"2020-10-21T17:39:08.951656Z","src_ip  
":"134.73.5.5","session":"7c25724b1616"}
```

The *eventid* field identifies the type of event that is recorded in Cowrie. Those include the establishment of a connection, login attempts (and its results), and more interestingly the category *cowrie.command.input* that contains the recorded commands issued by the attacker. In this event type, the *input* field contains the actual commands as typed by the attacker. In the previous listing, the commands shown are multiple, using pipes for the chain of commands to be executed. This adds a layer of complexity to log parsing, since each of these command chains

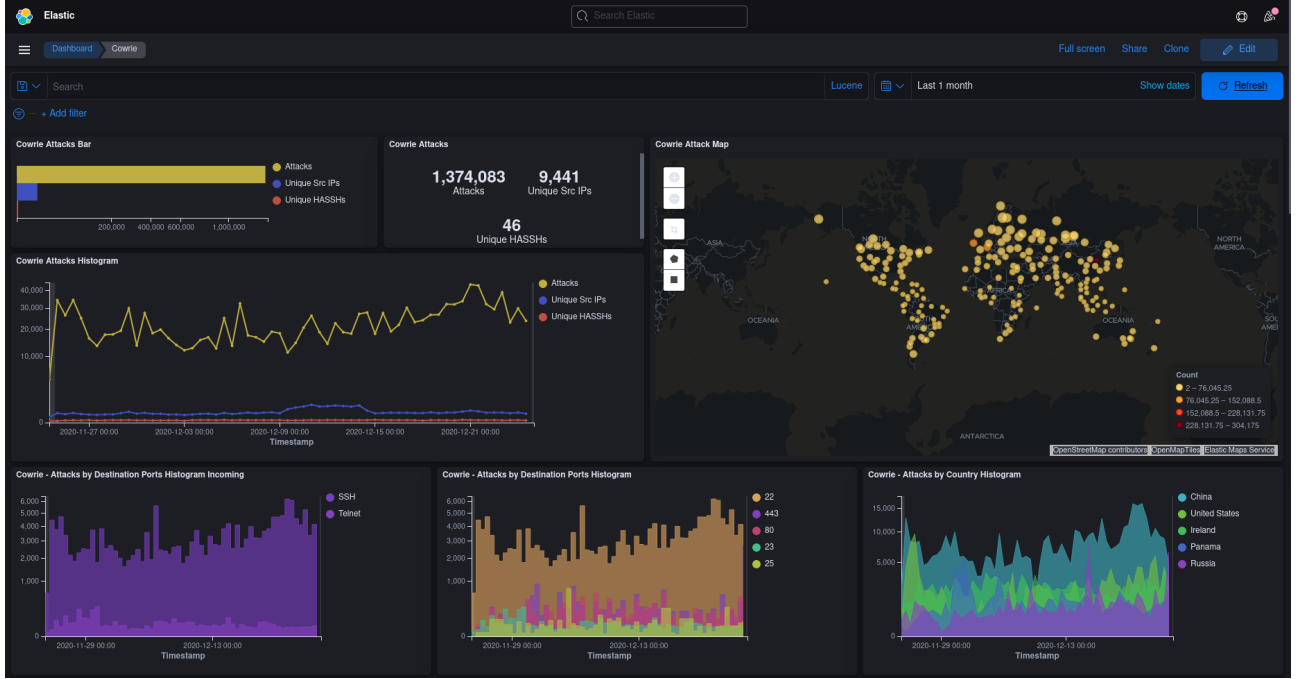


Figure 3.1: Kibana Cowrie dashboard on T-Pot.

are processed by the IRASSH honeypot as independent commands. Different analysis have been performed considering compound commands, while others have taken into account a list of expanded command lines resulting on different data rows inputs per each of the commands in a compound one.

These events have been the primary source of information for the following section analysis. They have been loaded into Pandas DataFrames to enable several analytics cases.

### 3.2.3 Analytics

The data captured shows a number of interesting insights from a cyber intelligence perspective. The analysis have been developed in Jupyter Notebooks using Python Pandas and are available on the public GitHub repository [6].

The honeypot, at the time of creating this thesis has captured:

- 825010 interactions (the Cowrie honeypot records all the SSH protocol connections, commands issued, login and logoff events, etc...).
- 2895 unique source IP addresses.
- 172606 commands captured in the honeypot. The command lines with more than one command have been split in order to reflect with more detail the total number of com-

mands issued. Without expanding compound command inputs, there have been 80982 command lines captured. 46,8% of the command inputs have been compound ones.

### 3.2.3.1 Commands logged on the honeypot

A list of the most common received commands has been obtained processing the *cowrie.command.input* events, which are presented on table 3.1:

Command	Number of times recorded
grep	29663
cat	15386
uname	14826
echo	12632
cd	9957
wc	9866
awk	9863
rm	5037
passwd	4992
ls	4984
head	4982
bash	4956
chmod	4946
free	4929
crontab	4928

Table 3.1: *Top commands recorded by Cowrie by times of appearance.*

Almost all of the commands shown in the most common list are used to query or search for information about the honeypot. Some of them are more intrusive, trying to change the users passwords or attempting to validate the users permissions, which can correspond to an automated tool or attacker intention of checking the privileges obtained in the honeypot. All of those commands are commonly used to identify if the system is a honeypot. Figure 3.2 includes a treemap visualization of the top recorded commands in Cowrie ordered by their absolute number of appearances.

Another relevant point observed on raw Cowrie data has been the presence of compound commands, that is the presence of more than one single command on the Bash input. For instance, using *ls -l ; cat /etc/passwd* would execute the *ls -l* command, and subsequently *cat /etc/passwd*. There are other separators and logical operators that have been considered on the data preparation. For general analysis, and following how QRASSH and IRASSH work, compound commands have been expanded as separate rows in a DataFrame, replicating the rest of information (source IP address, timestamp...etc).

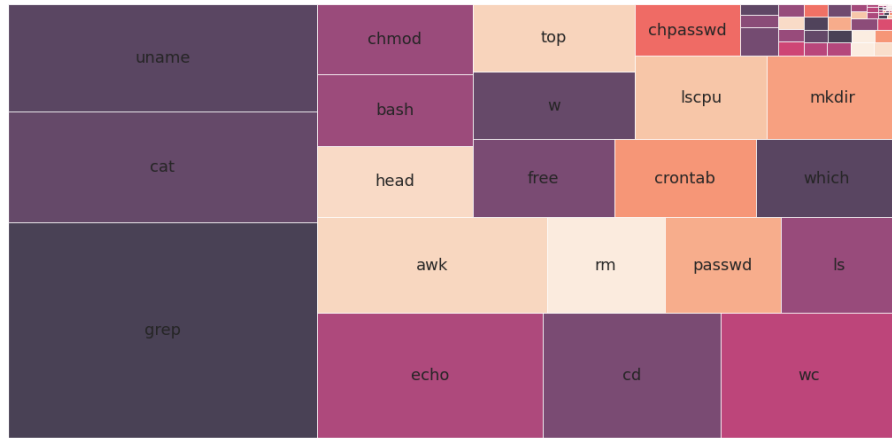


Figure 3.2: Treemap of top recorded commands in Cowrie.

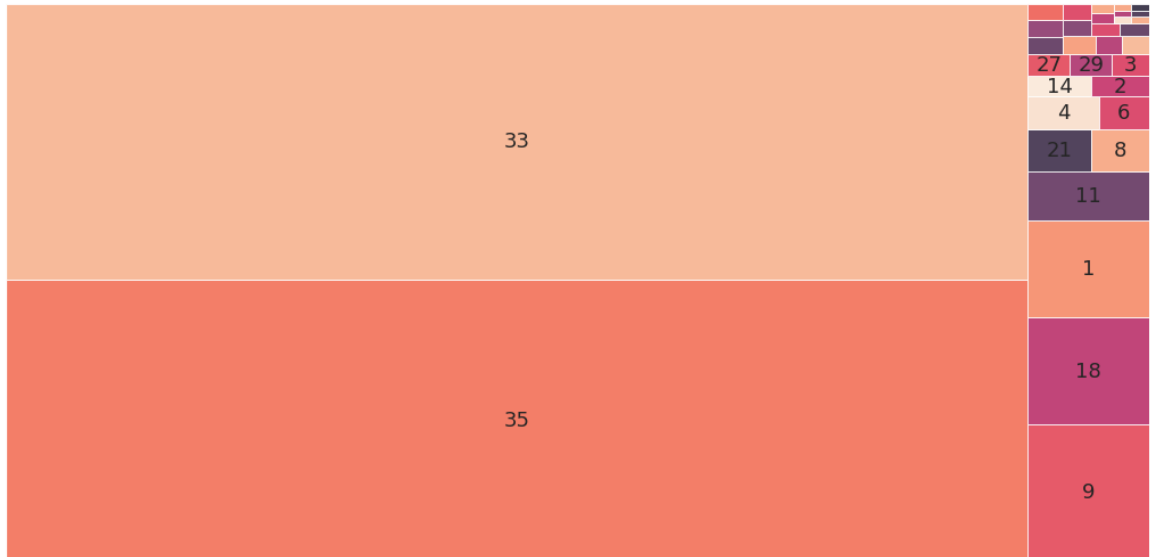


Figure 3.3: Treemap most common session lengths for compound commands on Cowrie.

The sessions captured on Cowrie average a mean of 31,6 commands, and a median of 33. Lengths with 33 or 35 commands are vastly more numerous than other compound length sessions. The following treemap diagram 3.3 shows this data visualization. This fact can be considered an anomaly on the captured data, but these commands have been manually validated to discard potential data capture errors. In a deeper analysis, randomly searching two different sessions with length 33 consist on the commands:

```
Session: c905b8d75cb4
Session Length: 33
Command sequence:
cat /proc/cpuinfo | grep name | wc -l
```

```

echo "root:2c6z9x4hmM2S"|chpasswd|bash
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}'
cat /proc/cpuinfo | grep model | grep name | wc -l
lscpu | grep Model
cd ~ && rm -rf .ssh && mkdir .ssh && echo "ssh-rsa AAAAB3NzaC1yc2EAAA
BJQAAAQEARdp4cun2lhr4KUHBGE7VvAcwdli2a8dbnrT0rbMz1+5073fcB0x8NVbUT0bUa
nUV9tJ2/9p7+vD0EpZ3Tz/+0kX34uAx1RV/75GV0mNx+9EuW0nvNoaJe0QXxziIg9eLBHp
gLmuakb5+BgTFB+rKJAw9u9FSTDengvS8hX1kNFS4Mjux0hJOK8rvEmPecjdySYmb66ny
lAKGwCEE6WEQHmd1mUPgHwGQ0hWCwsQk13yCGPK5w6hYp5zYkFnlC8hGmd4Ww+u97k6pf
TGTUbJk14ujvcD9iUKQTTWYYjIIu5PmUux5bsZ0R4WFwdIe6+i6rBLAsPKgAySVKPRK+oR
w== mdrfckr">>.ssh/authorized_keys && chmod -R go= ~/.ssh && cd ~

```

Session: d53418516393

Session Length: 33

Command sequence: cat /proc/cpuinfo | grep name | wc -l

```

echo "root:fvKcvtzyPqPv"|chpasswd|bash
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}'
cat /proc/cpuinfo | grep model | grep name | wc -l
lscpu | grep Model
cd ~ && rm -rf .ssh && mkdir .ssh && echo "ssh-rsa AAAAB3NzaC1yc2EAAA
ABJQAAAQEARdp4cun2lhr4KUHBGE7VvAcwdli2a8dbnrT0rbMz1+5073fcB0x8NVbUT0b
UanUV9tJ2/9p7+vD0EpZ3Tz/+0kX34uAx1RV/75GV0mNx+9EuW0nvNoaJe0QXxziIg9eL
BHpgLMuakb5+BgTFB+rKJAw9u9FSTDengvS8hX1kNFS4Mjux0hJOK8rvEmPecjdySYmb
66nylAKGwCEE6WEQHmd1mUPgHwGQ0hWCwsQk13yCGPK5w6hYp5zYkFnlC8hGmd4Ww+u9
7k6pfTGTUbJk14ujvcD9iUKQTTWYYjIIu5PmUux5bsZ0R4WFwdIe6+i6rBLAsPKgAySVK
PRK+oRw== mdrfckr">>.ssh/authorized_keys && chmod -R go= ~/.ssh && cd ~

```

Not only the command sequence is almost identical, but also they share a clear indicator of the attacker being the same agent: an identical SSH key to be inserted on the SSH authorized keys file. This situation repeats in the sessions with a length of 33 or 35 commands, with the same SSH key, and with small differences on the issued commands. Therefore, these noisy events are real attacks that can be attributed to the same threat actor. Similar events have also been identified, where a single actor creates a number of sessions that distort other captured

data in the honeypot. These events have been removed for some of the analysis, particularly in the sessions length visualizations. However, observing this level of noise can be relevant from a cybersecurity perspective, it allows to identify a single actor targeting our system with insistence, which could imply a higher risk of attack. For these reasons, the data has been kept for some analytics cases.

An important measure for honeypot comparison, also stated in the Heliza paper has been the length of command sessions (number of commands registered in a session the attacker has initiated on the honeypot). It is stated on the paper as "keeping attackers busy". In order to reproduce this analysis with the logs recorded by Cowrie, the following visualization has been created with Pandas time series plots: 3.4. The noisy attackers identified have been removed to avoid the data anomaly created. This removal, and the malfunction of the deployment for several days on mid-November 2020, are the cause for the empty spaces in the plot.

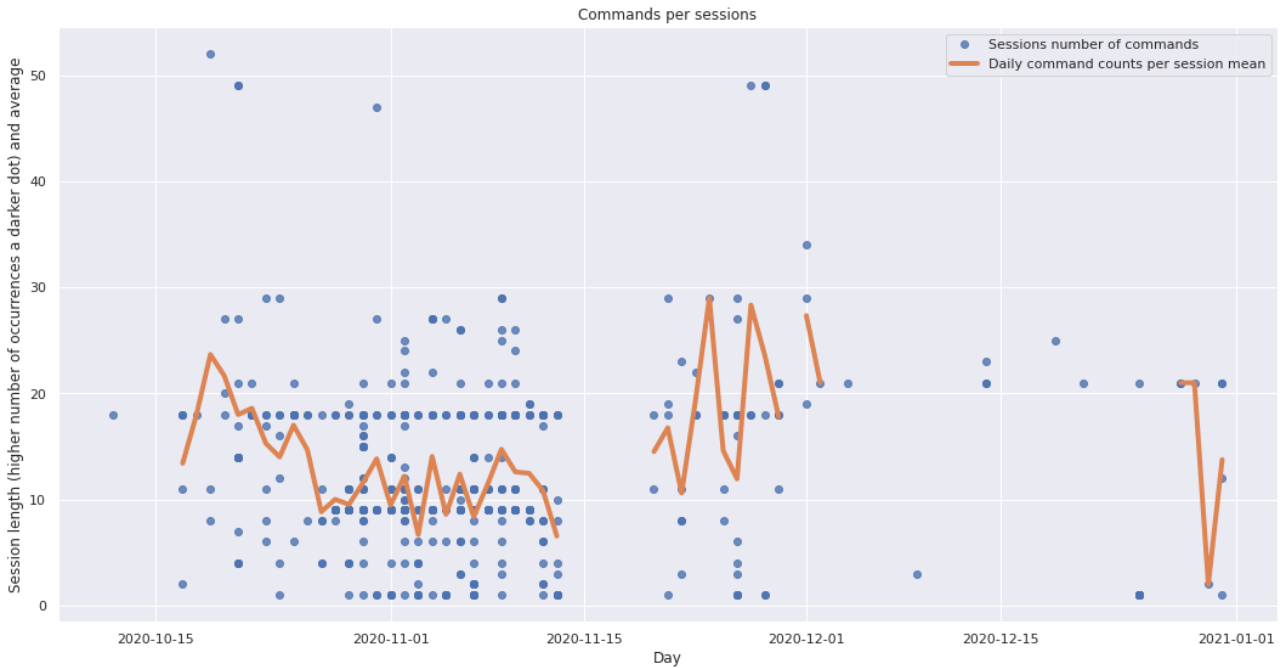


Figure 3.4: Cowrie session lengths recorded per each day (anomalies removed).

### 3.2.3.2 Download commands

Another interesting feature for this project are the number of appearances of download commands. The original Heliza paper [19] calls this activity "Collecting attacker related information". The data, with the logs used for this project, can be queried for the number of appearances in the DataFrame (a new DataFrame expanding compound command lines into



different rows with one command for each has been used), as shown in table 3.2.

Command	Number of times recorded
wget	188
tftp	168
curl	31
scp	7
nc	5
ftpget	5

Table 3.2: *Download commands recorded in Cowrie.*

It is already visible that they account for a small part of the overall number of commands. More specifically, the wget command lines recorded constitute the 0.00109% of the total commands recorded. In comparison, grep (the most commonly recorded command) accounts for 0.17185% of the total.

Also, the original author's papers have stated the value of recording and prioritizing download commands in the honeypots due to their relevance on attackers activity and techniques analyzed. A visualization has been created to perceive the effectiveness of the honeypot on this regard by considering the average number of download commands per each attacker session by days: 3.5.



Figure 3.5: Cowrie download commands recorded per each session and day.

### 3.2.3.3 Graph analysis

A different approach to data analysis of honeypot logs was also studied. Using graph analysis with the Python Networkx library, commands which are related were modelled in a DataFrame. This data was shaped as a *previous* and *next* commands. To have an edge representing this fact, two commands must belong to the same session and be executed one after the other.

As an example, for a given commands Data Frame:

session name	commands
-----	-----
session1	cd home; cat file.txt; wget 1.1.1.1
session2	echo 'a'
session2	rm *

The resulting DataFrame to be modelled as a graph would result on:

previous	next
-----	----
cd	cat
cat	wget
echo	rm

A sample graph from the first 30 commands of Cowrie registered data has the following shape 3.6:

The graph including the overall data set is not as legible as the sample, due to the huge number of commands related to each other 3.7:

However, the graph is useful to identify those commands with more edges, which appear more frequently in attack sequences. Those have been considered the "most significant commands" because they can be interpreted as the commands that will more often lead to other commands executions. Using the following Python command, the list on table 3.3 was obtained.

```
sorted(dict(G.degree()).items(), key = lambda x : x[1], reverse = True)[:15]
```

As a conclusion from the analysis, it is visible that the Cowrie honeypot records a small number of download commands on each session, although most sessions contain at least one download command. The visualization used shows that the tendency does not significantly improve or worsen over time.

The goals for the adaptive honeypots configurations developed for this project consider these insights and try to improve the intelligence gathering of the honeypot by collecting more valuable information.

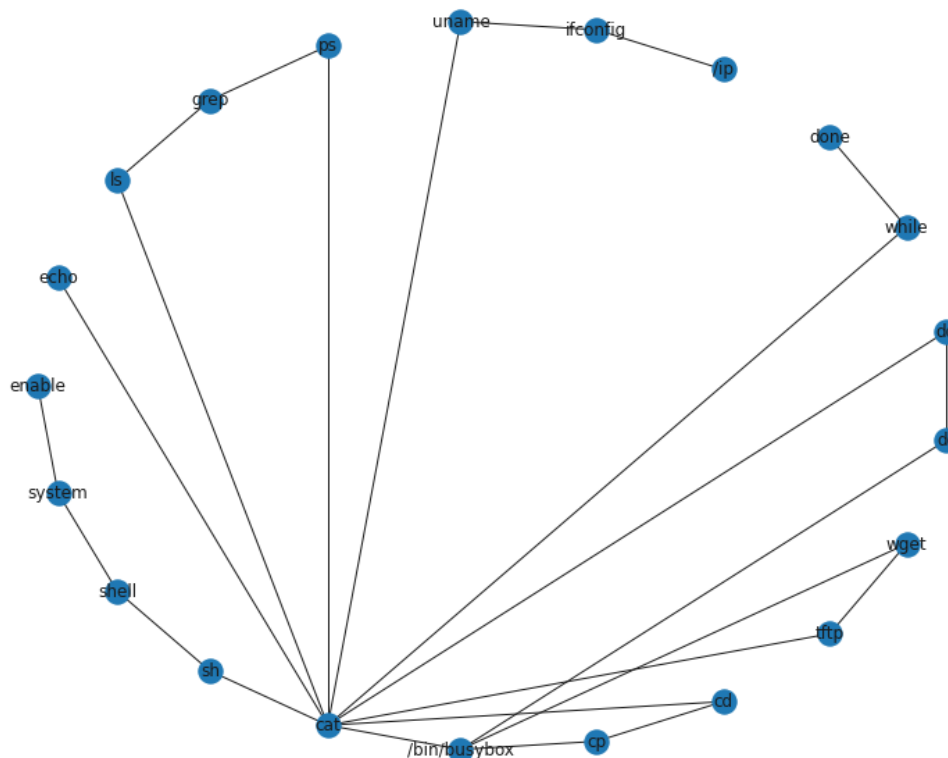


Figure 3.6: Sample graph from 30 Cowrie commands analysis.

## 3.3 Metrics and evaluation of adaptive honeypots

### 3.3.1 Development of deployment environment

The IRASSH and QRASSH honeypots have been deployed and analyzed as part of this project. Since both share their core functionalities, only IRASSH was used for the honeypot metrics because it's a newer development that can be run with newer Python libraries and a more stable dependencies environment. Both are an extension of the original Cowrie honeypot. This fact allows to analyze logs with similar techniques, as the logs format is almost equal.

The original project source code was published by their authors. It requires a number of dependencies (other Python libraries) and additional components to run (a MySQL database, operating system components), which require a significant effort for every single deployment.

A docker-compose environment that deploys containers for both the database and the instance running the honeypot, and also automates dependencies management, was developed as part of this project and released publicly [7]. This approach has been convenient for fast testing deployment and parameterization and has reduced the number of potential errors due to manual interaction.

The honeypot instances have been set up on Amazon Web Services servers using Terraform



Command	Number of edges
echo	37
cat	32
rm	24
cd	21
wget	21
grep	21
chmod	21
bash	19
sh	18
/bin/busybox	18
ls	13
uname	12
tftp	11
head	11
awk	11

Table 3.3: *Cowrie commands with more graph edges.*

the results with the benchmark data captured by Cowrie, and observing the behaviour of the honeypot system in its productive role.

The tests performed can be subject to a lot of variables that can affect the analysis (such as different attackers interacting with each of the different instances). But the goal of this analysis is to observe trends and visible improvements on the knowledge collected by placing the honeypot in a real environment.

Using Cowrie data, the following metrics were identified as enabling an analysis on the effectiveness of the different parameterizations of the IRASSH adaptive honeypot placed on a productive environment and registering attackers interactions. The first two are implementations of Heliza creators proposals [19]), the third being developed using data exploration from graph analysis as described on section 3.2.3:

- Sessions length: The target of this metric is evaluating the honeypot capability to lure attackers to issue a large number of commands on each session they establish. This number can identify the possibility that attackers were deceived by the honeypot into thinking that the system is a real device, and issued a number of commands without closing the connection. A common attacker behaviour is to immediately leave the honeypot on the suspicion that the system is a trap. On longer interactions, more data can potentially be identified about the attackers TTPs. The metric has been evaluated as the mean of total command counts per session per day.
- Number of download commands: Another target for honeypot evaluation can be the

number of download commands detected on each session. These commands imply that the attacker will download a file on the honeypot, which can be especially interesting for knowledge gathering on attackers tactics, techniques and procedures. The downloaded scripts or binary files provide a more valuable source of information about threat actors than standard commands captured. It also confirms that the attacker has not identified the system as a honeypot, confirming that the adaptive behaviour of the honeypot has succeeded on deceiving attackers. The metric has been evaluated as the mean of download command counts per session per day.

- Most significant commands (from graph analysis): By analyzing which are the most connected commands in a graph-modelled set of interactions, it can be determined if the instance of the honeypot has identified offensive or download commands among the most connected. This fact can indicate that the attackers registered have performed more direct attacks to the honeypot instead of interactions which have been launched as a recon effort and aborted the session when identifying the system as a honeypot. The metric has been evaluated by obtaining the list of most connected commands and checking if offensive or download commands are present.

### 3.3.3 IRASSH parameterizations from Cowrie analysis

The following settings on IRASSH reward function have been developed, both using the default configuration provided by the honeypot developer, and the ones derived from analyzing Cowrie data:

- IRASSH default reinforcement learning: This set of parameters (present on the public GitHub repository for IRASSH and QRASSH) prioritizes download commands (assigning a 500 score), and also tries to create states on which the sessions use known "hacking" commands (assigning a 200 score to each of them).
- Download behaviour configuration: Assigns a higher score for download commands (using a 500 score per download command and 0 to others).
- Longest sessions configuration: To achieve a configuration with the goal of maximizing sessions with more commands on each of them, a list of the top commands appearing on the Cowrie registered data was obtained, and manually transformed into a Pickle file. The policy assigns a 200 weight to those, and 0 for the rest of commands.
- Most significant commands configuration: The commands seen as most connected in the Cowrie data analysis have been set on a Pickle configuration file with a score of 500

each. This configuration shall attract attackers to keep using the most typical commands on the attacks, probably providing longer sessions and increased numbers of download commands issued, compared to Cowrie data.

The detailed configurations can be checked on the public repository being used for this project [7]. The configuration text files (and derived Pickle files) used are stored under the path *irassh/src/irassh/rl/*.

### 3.3.4 Evaluation of the different adaptive honeypot parameterizations

The honeypots have been evaluated considering the metrics presented in the previous sections. The observations from data obtained using Cowrie, the IRASSH default configuration, a behaviour prioritizing download commands, a behaviour setting that prioritizes long sessions, and one favouring the most connected nodes from Cowrie analysis; are compared on the following points.

- Sessions length: Cowrie (with the aforementioned anomalies removed) appears as quite successful on capturing long sessions. This fact could also be explained by the fact that some of the actions taken by the IRASSH counterparts (fake information, delays) can lead the attackers to identify an abnormal situation than what they would expect on a productive system. Comparing the IRASSH configurations, the behaviour that lures the attackers into introducing the commands that appear on longer sessions appears to regularly have a higher command count on each session, although with some exceptions and not by a far distance. Figure 3.8 shows this comparison. This visualization suffers from the lack of a longer observation period, since a surge on the IRASSH default parameterization also leads to think it is more optimal, while the trend is later reduced.
- Number of download commands: The download commands have been plotted on two different graphs: Figure 3.9 presents the daily number of commands mean per session, whereas figure 3.10 shows the total number of download commands received. The average graph shows that the IRASSH configuration that prioritizes download commands is having a larger number of download commands per session than the rest (*wget* appeared using this configuration as 0,05938% of the total commands, as opposed to 0.00109% in Cowrie), despite the longest sessions behaviour configuration having a large number of download commands too. Observing the total number of commands graph on figure 3.10, it reveals that this honeypot configuration received more download commands than the rest of

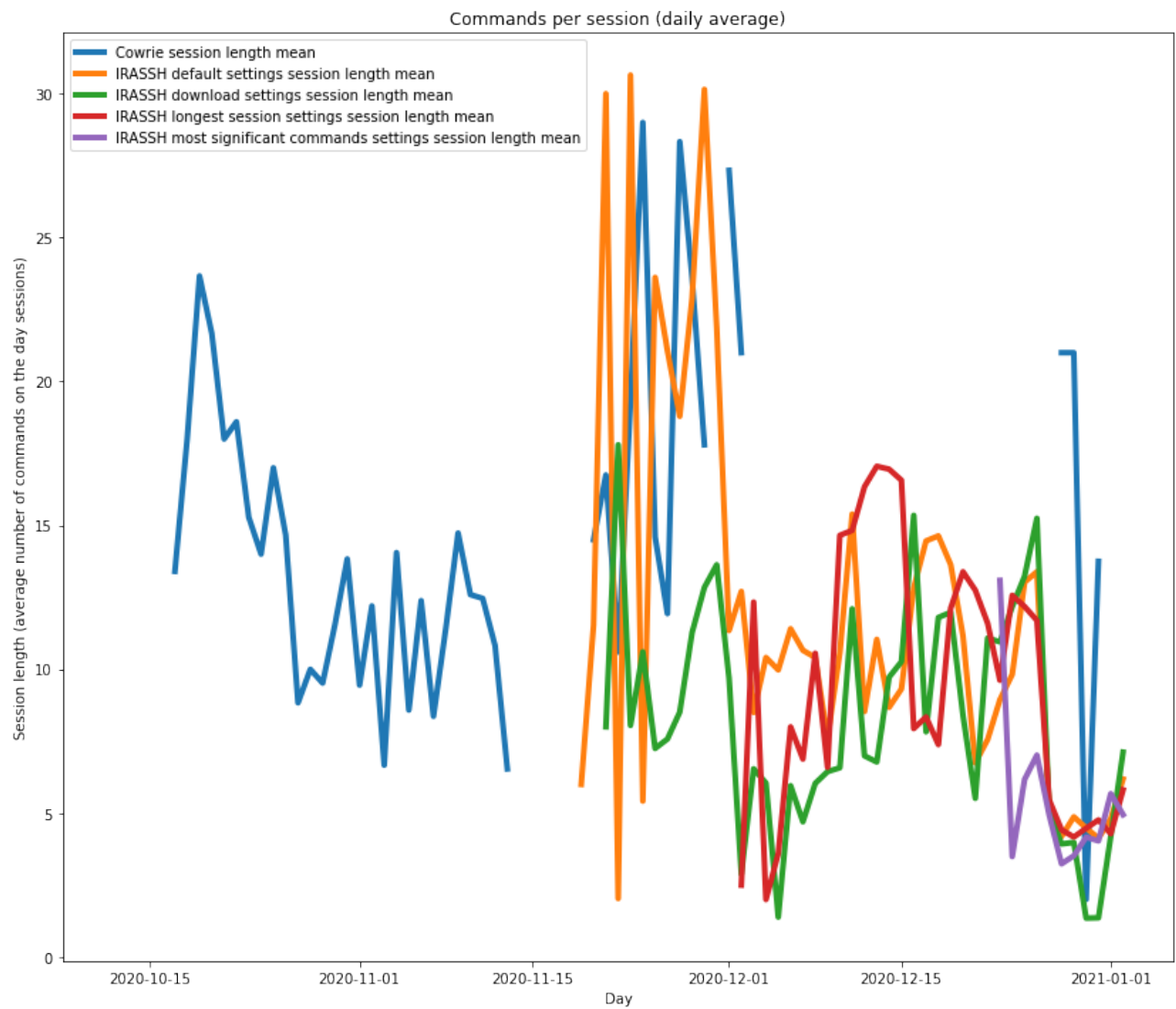


Figure 3.8: Comparison of command lengths for each session.



honeypots combined. The most connected configuration also shows a surge on these commands over the short period it has been running. The configuration includes the *wget* and *tftp* commands.

- Most significant commands: A table summarizing the most connected commands of each of the configurations is visible on 3.4, which is the result of getting the commands that have more edges in the graph representation. The commands and command counts are similar on all the honeypots, but the honeypot configured to capture more download commands registers values for *wget* and *curl* that are higher than the other honeypots, as it could be expected. The longest sessions configuration and the most connected commands configurations do not show higher number counts as it should, although these honeypots have been running for a shorter period. These observations are further developed on chapter 4.

Cowrie	IRASSH default	IRASSH download	IRASSH longest sessions	IRASSH most connected
echo: 37	chmod: 29	cat: 41	cat: 30	rm: 14
cat: 32	cat: 29	chmod: 36	cd: 26	cat: 11
rm: 24	rm: 28	wget: 34	echo: 21	grep: 11
cd: 21	wget: 25	curl: 29	grep: 20	chmod: 11
wget: 21	cd: 25	rm: 27	rm: 19	wget: 11
grep: 21	uname: 25	uname: 23	chmod: 18	uname: 10
chmod: 21	grep: 23	cd: 21	ls: 17	cd: 10
bash: 19	echo: 21	echo: 20	uname: 15	sh: 9
sh: 18	curl: 21	grep: 20	wget: 15	tftp: 9
/bin/busybox: 18	history: 15	history: 17	sh: 13	history: 8
ls: 13	bash: 15	sh: 14	tftp: 13	perl: 7
uname: 12	ls: 13	perl: 13	passwd: 11	echo: 6
tftp: 11	sh: 13	ls: 12	bash: 11	ls: 6
head: 11	get: 13	tftp: 12	head: 11	ftpget: 6
awk: 11	m: 12	awk: 11	awk: 11	ps: 5

Table 3.4: *Most connected commands on each honeypot configuration.*

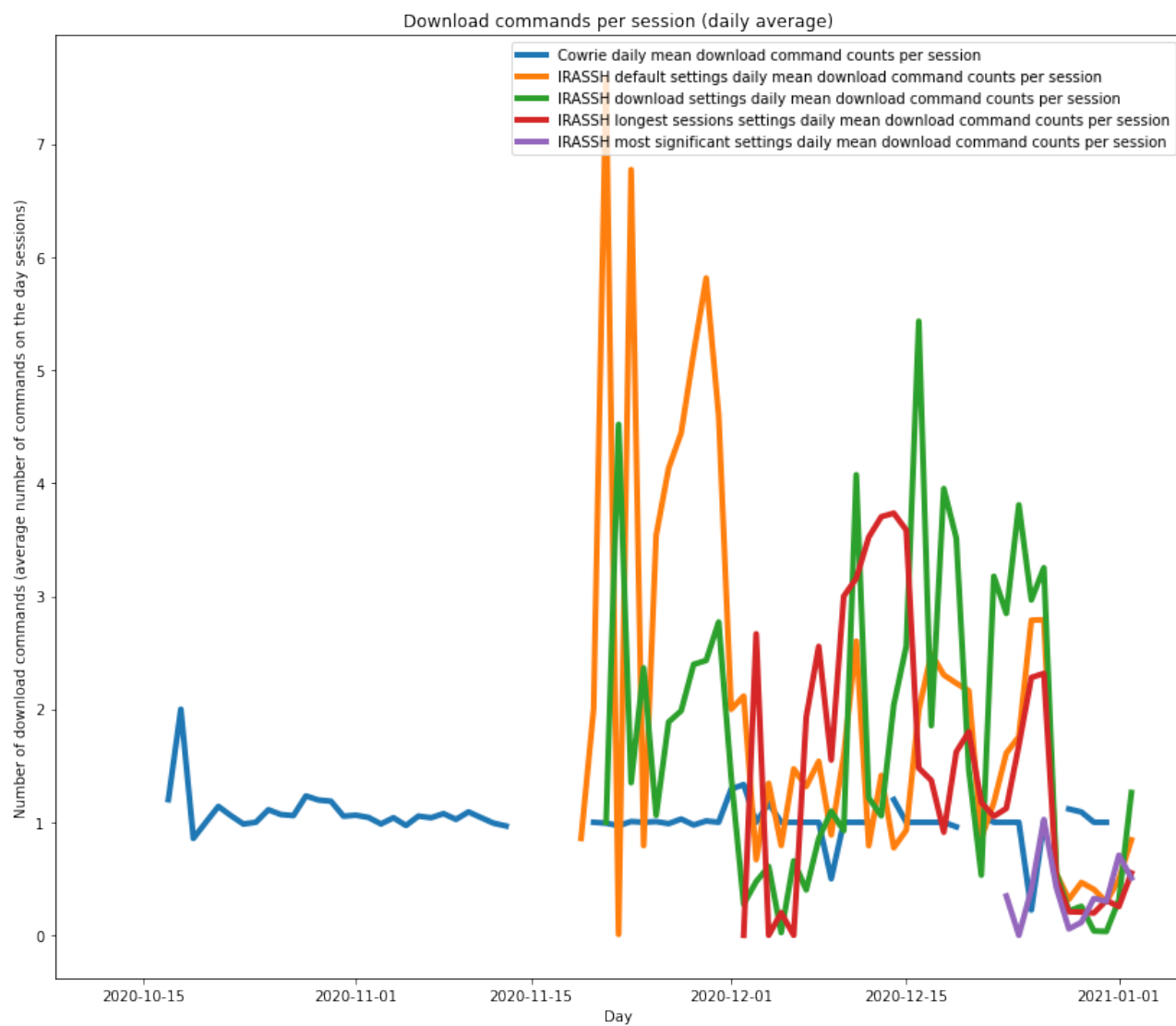


Figure 3.9: Comparison of download commands issued per day averages.

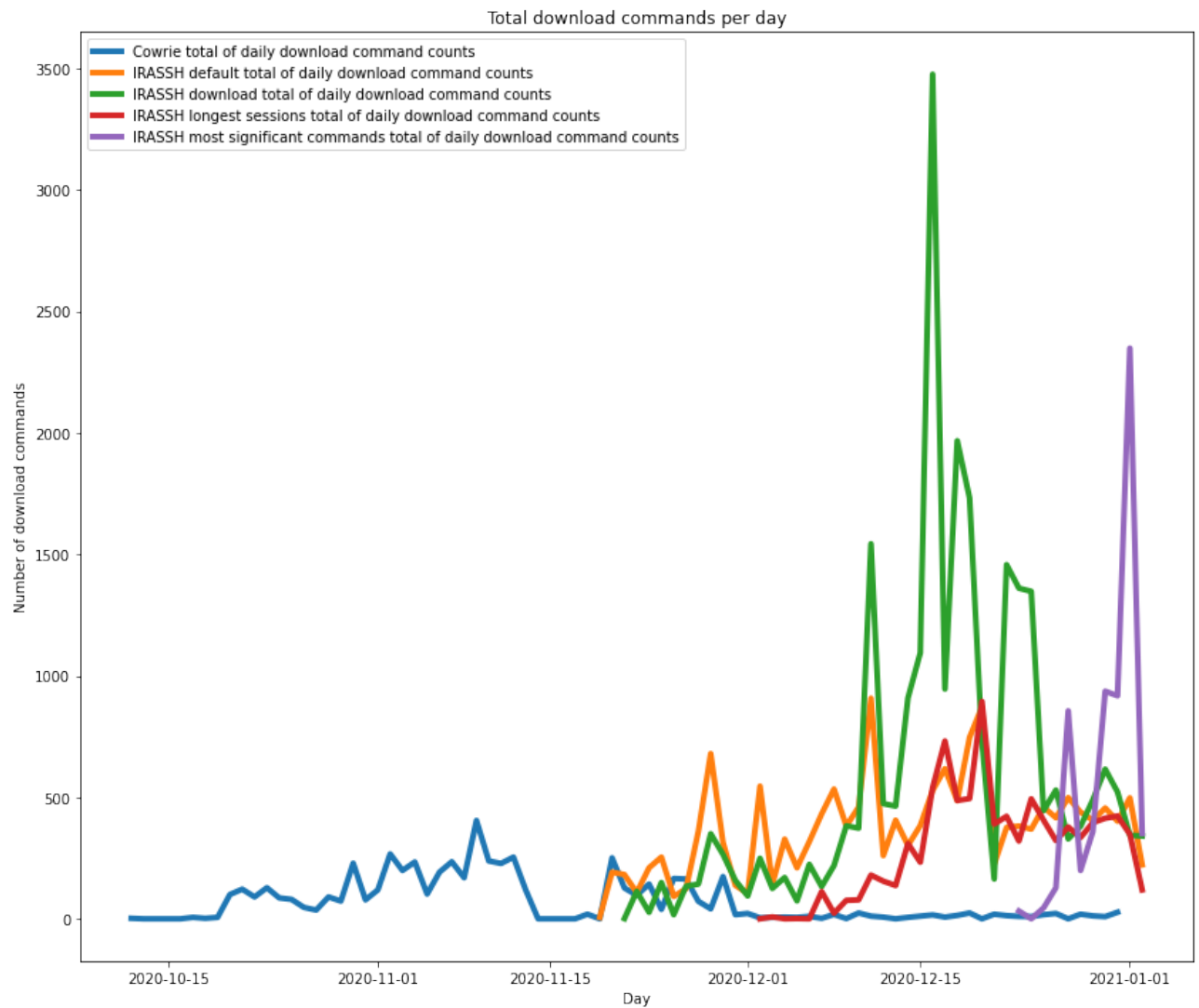


Figure 3.10: Comparison of total number of download commands issued per day.



# Chapter 4

## Conclusions and future lines of work

### 4.1 Conclusions

A number of observations and reflections have arisen during this project. This section summarizes the most relevant information and outcomes.

#### 4.1.1 Limitations

First and foremost, the results presented on this thesis are limited by the own nature of the field experiments that have been conducted. The experiments are bound to the nature of the universe of observations being used: Actual attackers present on the Internet and looking for misconfigured, vulnerable or unmaintained systems. Despite this fact, the collected data is relevant from both a data science and cybersecurity point of view. Reinforcement learning agents based on a Markov Decision Process are adequate for problems where the environment is unknown and can be only analyzed through interaction with it. The honeypot optimization problem fits into this category. Honeypots are, by principle, a tool to capture the interactions with attackers that come across them. Any observation performed on these systems on the Internet will be affected by the attackers (either human or automated) that can potentially change their behaviour depending on the honeypot characteristics. Therefore, collected data can be considered valid in the context of a limited and not deterministic experiment.

An anomaly on the data collected on Cowrie (most sessions have a similar length of commands issued) is not a data collection error, but an event related to cybersecurity. As previously mentioned, it is possible that a specific actor or group targeted the deployed Cowrie honeypot more intensively. The data has been explored and the captured data shows identical commands and other identifiable information related to a single attacker for sessions with specific lengths.

A longer period of observation would substantially improve the relevance of the performed

experiments. As a future effort to contribute with these experiments, the test honeypots deployed on the context of this project will be kept running and the resulting data will be publicly shared.

Another important limiting factor on this project development were the issues detected on the public honeypot projects used for analysis and technical implementations. These projects included several bugs, and lacked a configured environment. Deployments required some additional development and analysis which had not been accounted for in the original planning. These contributions have been open-sourced for anyone to be able to deploy the projects with less effort.

### 4.1.2 Results

The results presented, even considering the aforementioned limitations, provide some trends that are aligned with the hypothesis of previous research and the goals pursued by this project.

- Adaptive honeypots collect better knowledge (cyber threat intelligence) than traditional medium-interaction honeypots: During the analysis of different parameterizations of honeypots on chapter 3, IRASSH honeypots configured with specific target reward function parameters obtained better results than Cowrie and their counterparts with less-specific configurations, or those aiming for other target behaviours. One exception being the sessions length metrics, which have been affected by the anomaly of data captured in Cowrie.
- The studied reinforcement learning honeypots have been analyzed, and relevant configurations have been identified and modified to implement the field experiments based on different policies.
- Honeypot data analytics: A number of analytics cases have been developed in order to estimate the trends presented by each configuration. Those have permitted a comparison among different honeypots.
- Deployment environment: A deployment environment has been developed that serves as an improvement on the convenience of setting up test environments.

## 4.2 Future lines of work

### 4.2.1 Adaptive modification of honeypots configurations

The proposed reinforcement learning honeypots approach the reinforcement learning problem on a similar fashion. They change the reactions that the honeypots present after each different command issued to them. The set of actions differs, and also how reward functions assign different actions based on their parameters.

However, further action sets could be considered. For instance, how the system is presented could altered and learnt over time. The states of the honeypot, instead of representing commands received, could represent different configurations applied to the system and accessed by the attackers. A small list of examples would be:

- **System name:** The honeypot name is now fixed as a static string. On production IT systems the name can represent the role of the system (for instance, FWPN01 could be guessed as a firewall, from the first two characters). Furthermore, home Internet routers can have names that allow for guessing the provider that packaged it to the customer. Modifying this system name to a set of different possible strings when the attacker issues the *uname* command could lead to different attacks being issued.
- **Technical specifications:** The CPU type and their number, architecture, amount of memory, etc... can lead an attacker to guess a system role or assume more compute capability by the honeypot system. Returning fake information on response to commands that query this information and learnt over time as providing better rewards.
- **Commands available:** Some tools available on the system could extend some sessions that are aborted when the attacker detects their absence. Providing different information on commands listing or execution could help collect more knowledge.
- **Users configured on the honeypot:** All RASSH derivatives include a user named "Richard" by default. Deploying the honeypot with the default configuration leads to an easy identification by attackers. Using a dynamic list of configured users could difficult the honeypot detection.

The states could be grouped into several system roles (generic server - GS, home router - HR, industrial firewall - IF, etc...) that would pack these features in a grouped basis. Furthermore, the new state set including this parameter changes in addition to Linux commands and Other commands, as the RASSH-based honeypots currently do. The states would be defined as:

$$S = \{L \cup O \cup GS \cup HR \cup IF\}$$

Alternatively, this type of honeypot could have a state set with only these parameterization changes:

$$S = \{GS \cup HR \cup IF\}$$

The viability of implementing these states sets in a new adaptive medium-interaction policy should be studied further, although this approach is similar to the state sets used in Heliza.

#### 4.2.2 Graph analysis on honeypot data

The usage of graphs for modelling commands issued by attackers presents as a potential source for getting more knowledge on their attacks, and also to include more reward functions parameterizations.

The analysis performed on this project used unweighted edges on non-directed graphs. Potential improvements on this analysis could include:

- Directed graphs: Using directed graphs provides more knowledge on the attack sequences, and allow for identifying similar attack techniques, or allow to compare their similarity.
- Weighted edges: More relevant commands could have a higher weight on a transition to them, or in the transitions that they use. Commands that are previous or lead to known hacking commands or download commands could be prioritized on reward functions using this analysis.

#### 4.2.3 Reward functions additional parameterizations

This project consisted on the evaluation of different parameterization sets based on the default Cowrie honeypot analysis. With the further insights collected on this project these policies could be improved to consider other types of goals. For instance, studying the commands that are not implemented on the honeypot invoked frequently by attackers could be presented with negative scores to discard them.

The opportunities presented by analyzing the honeypot data as graphs can provide more fine-grained knowledge that can allow for more refined configurations targeting specific types of attacks.

#### 4.2.4 Deployment automation and data analysis

A number of potential automation features on honeypots deployment could have been developed to further enable administrators to have an easier, faster and less prone to errors process.



Configurations change (on the system names, on several static configuration files) could also be randomized on this step to disguise the honeypot and making it more unique.

The automated deployment system developed in this project included the IRASSH honeypot and its supporting MySQL database. A plan for automating data collection was also considered, but time constraints prevented this integration. Additional Docker containers with a data analytics platform, such as an ELK stack (Elasticsearch, Logstash and Kibana) could be added to the automated components, and data on the honeypot could be collected by it. Kibana allows for a number of dashboard analysis to be compared in real-time, easing cyber intelligence collection or anomalies identification.



# Chapter 5

## Annex 1

### 5.1 Code listings

#### 5.1.1 QRASSH/IRASSH honeypot.py

```
lastpp = None
actionValid = None
for index, cmd in reversed(list(enumerate(cmd_array))):

    cmdclass = self.protocol.getCommand(cmd['command'], environ['PATH'].split(':'))

    if cmdclass:
        # log command to database
        raw_cmd = cmd['command']
        rl_state.current_command = raw_cmd
        proxy.ActionPersister().save(self.actionState, raw_cmd)

        # generate action
        actionListener = proxy.ActionListener(self.actionState)

        if manual:
            generator = proxy.FileActionGenerator()
        elif use_irl:
            generator = proxy.IrlActionGenerator()
        else:
            generator = proxy.RlActionGenerator()
        actionFactory = proxy.ActionFactory(self.protocol.terminal.
                                            write, actionListener,
                                            generator)
```

```

        validator = proxy.ActionValidator(actionFactory)
        actionValid = validator.validate(raw_cmd, self.protocol.
                                         clientIP)

        actionName = validator.getActionName()
        actionColor = validator.getActionColor()

        log.msg(eventid='irassh.command.action.success', action=
                actionName, input=cmd[
                'command'] + " " + ' '.
                .join(cmd['rargs']),
                format='Command found:
                %(input)s')

        if index == len(cmd_array)-1:
            lastpp = StdOutStdErrEmulationProtocol(self.protocol,
                                                    cmdclass, cmd['
                                                    rargs'], None,
                                                    None)

            pp = lastpp
        else:
            pp = StdOutStdErrEmulationProtocol(self.protocol,
                                                    cmdclass, cmd['
                                                    rargs'], None,
                                                    lastpp)

            lastpp = pp

        if self.ttylogEnabled:
            ttyAction = '\033[1;' + actionColor + 'm' + actionName.
                upper() + '\033[1;
                m\n'

            ttylog.ttylog_write(self.protocol.terminal.ttylogFile,
                                len(ttyAction),
                                ttylog.TYPE_OUTPUT
                                , time.time(),
                                ttyAction)

        else:
            log.msg(eventid='irassh.command.failed', input=' '.join(cmd2
                ), format='Command not
                found: %(input)s')

            self.protocol.terminal.write('bash: {}: command not found\n'
                .format(cmd['command']
                ))

        runOrPrompt()

```

```
if pp and actionValid:
    self.protocol.call_command(pp, cmdclass, *cmd_array[0]['rargs'])
```

### 5.1.2 QRASSH/IRASSH nn.py

```
def neural_net(input_length, number_of_actions, params, load=''):
    model = Sequential()

    # First layer.
    model.add(Embedding(250, 32, input_length=input_length))
    if input_length==1:
        model.add(Dense(params[0], init='lecun_uniform'))
        model.add(Activation('relu'))
    else:
        model.add(Bidirectional(LSTM(params[0])))
    model.add(Dropout(0.2))

    # Second layer.
    model.add(Dense(params[1], init='lecun_uniform'))
    model.add(Activation('relu'))
    model.add(Dropout(0.2))

    # Output layer.
    model.add(Dense(number_of_actions, init='lecun_uniform'))
    model.add(Activation('linear'))

    optimizer = Adam()
    model.compile(loss='mse', optimizer=optimizer)

    if load:
        model.load_weights(load)

    return model
```



# Bibliography

- [1] W. Z. Ansiry Zakaria and M. L. Mat Kiah. *A Review on Artificial Intelligence Techniques for Developing Intelligent Honeypot. 2012 8th International Conference on Computing Technology and Information Management (NCM and ICNIT)*, 2012.
- [2] G. Bonaccorso. *Mastering Machine Learning Algorithms - Second Edition*. Packt Publishing, 2020.
- [3] The MITRE Corporation. *MITRE ATT&CK framework*, 2020. <https://attack.mitre.org/>.
- [4] S. Dowling. *A new framework for adaptive and agile honeypots*. PhD thesis, National University of Ireland Galway, Ireland, 2020.
- [5] S. Dowling, M. Schukat, and E. Barrett. *Using Reinforcement Learning to Conceal Honeypot Functionality. ECML PKDD 2018: Machine Learning and Knowledge Discovery in Databases*, 2018.
- [6] O. Navarro Ferrer. *Honeypot data analytics GitHub public repository*, 2020. <https://github.com/n3if/honeypot-data-analytics>.
- [7] O. Navarro Ferrer (Forked from A. Pauna repository). *IRASSH project fork*, 2020. <https://github.com/n3if/irassh>.
- [8] Deutsche Telekom Security GmbH. *T-Pot GitHub page*, 2020. <https://github.com/telekom-security/tpotce>.
- [9] Y. Li. *Deep Reinforcement Learning. arXiv.org*, 2018.
- [10] D. McCandless and T. Evans. *World's Biggest Data Breaches and Hacks*, 2020. <https://www.informationisbeautiful.net/visualizations/worlds-biggest-data-breaches-hacks/>.

- 
- [11] V. Mnih, K. Kavukcuoglu, and D. Silver. *Playing Atari with Deep Reinforcement Learning*. *NIPS Deep Learning Workshop 2013*, 2013.
  - [12] M. Oosterhof. *Cowrie Honeypot*, 2020. <https://github.com/cowrie/cowrie>.
  - [13] A. Pauna and I. Bica. *RASSH Reinforced Adaptive SSH Honeypot*. *2014 10th International Conference on Communications (COMM)*, 2014.
  - [14] A. Pauna, I. Bica, F. Pop, and A. Castiglione. *On the rewards of self-adaptive IoT honeypots*. *Annals of Telecommunications*, 74, 2019.
  - [15] A. Pauna, A. Iacob, and I. Bica. *QRASSH A self-adaptive SSH Honeypot driven by Q-Learning*. *2018 International Conference on Communications (COMM)*, 2019.
  - [16] I. H. Sarker, A. S. M. Kayes, S. Badsha, H. Alqahtani, and P. Watters. *Cybersecurity data science: an overview from machine learning perspective*. *Journal of Big Data*, 7(41), 2020.
  - [17] C. Seifert, I. Welch, and P. Komisarczuk. *Taxonomy of honeypots*. Citeseer, 2006.
  - [18] R. S. Sutton and A. G. Barto. *Reinforcement Learning: An Introduction*. The MIT Press, 2014, 2015.
  - [19] G. Wagener, R. State, A. Dulaunoy, and T. Engel. *Heliza: talking dirty to the attackers*. *Journal in Computer Virology*, 7, 2010.