Annals of Telecommunications (2019) 74:501–515
https://doi.org/10.1007/s12243-018-0695-7

On the rewards of self-adaptive IoT honeypots

Adrian Pauna[1] · Ion Bica[1] · Florin Pop[2,3] · Aniello Castiglione[4]

Received: 10 July 2018 / Accepted: 3 December 2018 / Published online: 3 January 2019

Abstract
In an era of fully digitally interconnected people and machines, IoT devices become a real target for attackers. Recent incidents such as the well-known Mirai botnet, have shown that the risks incurred are huge and therefore a risk assessment is mandatory. In this paper we present a novel approach on collecting relevant data about IoT attacks. We detail a SSH/Telnet honeypot system that leverages reinforcement learning algorithms in order to interact with the attackers, and we present the results obtained in view of defining optimal reward functions to be used. One of the key issues regarding the performance of such algorithms is the direct dependence on the reward functions used. The main outcome of our study is a full implementation of an IoT honeypot system that leverages Apprenticeship Learning using Inverse Reinforcement Learning, in order to generate best suited reward functions.

Keywords Internet of things · Honeypot systems · Inverse reinforcement learning · Neural network · Self-adaptive honeypot systems · Reinforcement learning

1 Introduction

The Internet of things, or better known as IoT, is a newly created concept which has for the moment a wide spectrum of definitions; however; all have three points in common: communications, devices, and people. One of the best and most concise definitions is in our opinion, the one provided by The Oxford dictionary [1]: "The interconnection via Internet of computing devices embedded in everyday objects enabling them to send and receive data."

IoT devices have become essential for lots of applications, and therefore tend to be ubiquitous, being present almost everywhere: in our houses, in office buildings, on the streets, in factories, in our cars, and so on. According to a newly released Cisco report, there are currently almost 27 billion IoT devices and their number will rise up to 50 billion by 2020 [2]. While this will improve our lives and economy in general, based on the recent developments, IoT also create a huge gap in terms of information security. Symantec [3] recently released a study of 50 smart home devices and results showed that not even basic elements of security were enforced, which makes them easy targets for attackers. Other studies [4, 5] confirm that manufacturers did not consider security controls by design that would block cyberattacks.

One of the common elements of IoT devices, besides being connected to the Internet, is the usage of Telnet service. Y. M. Pa Pa et al. [6] studied the number of attacks on Telnet port 23 and noticed an increase of scans since 2005.

Alaba et al. [7] released a survey on existing security threats and vulnerabilities of the IoT heterogeneous environment. However, a genuine interest in the cyberattacks on IoT devices appeared in the autumn of 2016, when a French OVH Web Host was targeted by a DDoS attack at a throughput of 1 Tbit/s generated by 152,000 IoT devices.

This attack was followed by a huge DDoS flood in the next month when the Dyn DNS service was targeted, which in the end affected hundreds of websites and web services such as: Twitter, Spotify, Reddit, GitHub, and PayPal.

At that moment, it was obvious that such attacks are the result of a botnet, called Mirai that infected hundreds of thousands of IoT devices [8].

To understand the impact of the attack, it is worth mentioning that in a recently published survey of Internet outages, Aceto et al. included Mirai as a top-rated one [9].

✉ Florin Pop
florin.pop@cs.pub.ro; florinpop@ieee.org; florin.pop@ici.ro

[1] Faculty of Military Electronic and Information Systems, Military Technical Academy, Bucharest, Romania

[2] Faculty of Automatic Control and Computers, "Politehnica" University of Bucharest, Bucharest, Romania

[3] National Institute for Research and Development in Informatics (ICI), Bucharest, Romania

[4] Department of Computer Science, University of Salerno, Fisciano, SA, Italy

The initial study of Mirai botnet showed that it took advantage of the main flaws of IoT devices:

- In most cases they used default passwords for Telnet Service, as revealed by Cui et al. [10];
- Hardware constraints do not leave much room for security modules, as revealed in [11, 12];
- Devices should be available anytime and anywhere, which in some situations prevents patching them as often as required [10].

Mirai has the structure of a classical botnet, meaning that it infects IoT devices and thus transforms them into bots. The bots search for other vulnerable systems in order to infect them and wait for commands from a C&C (command-and-control) server in order to perform illegal activities such as: distributed denial of service (DDoS), spamming, click fraud, or phishing attacks. C&C servers are controlled by a botmaster, who decides on the targets and types of attacks to be performed by the bots. It was easy to analyze such botmaster, as the source code was publicly released soon after the attacks [13].

In this context, the use of honeypot systems is a real asset for research. There are several studies and some newly developed honeypots systems such as IoTpot [6] that are used to analyze the behavior of botnets and to collect as much data and information as possible. Most of the honeypot systems developed are emulating Telnet services and give the attackers access to a terminal that usually copies the functionalities of a Linux busybox [14, 15], one of the most widely used Linux distributions in IoT area.

In this paper, we present a new IoT honeypot system that we call self-adaptive, as it is able to interact with the attackers based on a set of actions triggered by a reinforcement learning algorithm. The system is based on a previous work done by Pauna et al. on self-adaptive SSH honeypot systems [16, 17].

The idea of using machine learning is not new and Wagener proposed such a model in 2008 [18], but improvements in SSH honeypot technologies and also in the implementation of RL algorithm, made it possible to develop RASSH [16] and QRASSH [17]. RASSH uses SARSA algorithm in order to generate decisions on what action should be taken, while QRASSH uses Deep Q-learning. Both implementations face the same issue: the direct dependence on the reward function used. This issue is generated by the fact that real hacker behavior is hard to mimic; therefore, a different approach is needed in order to define compliant reward functions.

Our IoT honeypot system uses apprenticeship learning (AL) algorithm to calculate reward functions for specific behaviors [19].

We consider the following relevant contributions of this article. We propose a new paradigm in which a self-adaptive IoT honeypot system using Q-learning is taught what actions to take in order to maximize a long-term reward. Based on the policy learned, an inverse reinforcement learning algorithm calculates the best reward function to be used. We then test the reward function in real-life context. We have proposed several reward functions related to various behaviors in connection to IoT attacks, which we wanted to impregnate to our honeypot system.

In Section 2 we make a summary of the general features of honeypot systems and provide more details on the self-adaptive ones. We briefly present the QRASSH implementation that is the basis on which the apprenticeship learning is developed. We also make a short review of the work done so far in the area of IoT honeypot systems. In Section 3 we explain the IRL algorithm we have used and the way it relates to the basic reinforcement learning algorithms. It should be noted that, as the name suggests, IRL is basically the opposite of RL and its scope is to infer a reward function by observing an agent behavior. In Section 4, we present the implementation of the IoT honeypot IRASSH-T and insist on the workflow that includes three phases: manual training in which the agent is shown an optimal policy, the IRL phase in which the reward function is inferred and the last phase where the reward function derived is tested. We conclude with Section 5 in which we present a set of results obtained and end with Section 6, where conclusions and future work are presented.

2 Self-adaptive SSH honeypot systems

The best definition of a honeypot system was given by Spitzner [20], stating that a honeypot is "a resource whose value lies in being probed, attacked or compromised." Below, we present briefly the evolution of honeypot systems.

2.1 Honeypot systems

Ever since the first honeypot concepts were developed in the 1990s, honeypot systems were seen as a marvelous solution that can be used in order to lure attackers and investigate their actions. Cheswick [21] in his article "An evening with Berferd" first presented his insights on how he and his department colleagues spent an evening setting up traps so that a hacker moved through. From this point on, things evolved and customized systems were designed, emulating specific Internet services, with the only purpose of being attacked and thus enabling to collect valuable data. The history of honeypots went through Gen II, where real operating systems where used to decoy attackers, and high-interaction honeypots were developed. Nowadays, we have reached the point where systems are able to activate different services so as to resemble the surrounding environment and these are called dynamic honeypots [20].

Still, not so many systems were developed that should be able to have interactive contact with attackers the way Cheswick did. Lately, a new type of honeypot systems was developed, called self-adaptive. Such systems use machine learning (ML) techniques in order to interact with attackers. Instead of having a human setting up traps for the attackers, the above systems use artificial intelligence (AI) algorithms that decide what actions to be taken.

2.2 Self-adaptive honeypot systems

The first self-adaptive honeypot system was conceived by Wagener et al. in 2010 and was called adaptive honeypot alternative (AHA). AHA was, at its core, a Linux operating system with a modified kernel module that allowed execution of specific actions inside an SSH server console [22]. The author modeled attacker and honeypot games using game theory concepts.

Pauna [23] improved AHA by adding rootkit malware detection capabilities. The next step belongs to the same author, who created a new honeypot system called RASSH, which used an existing medium interaction honeypot system KIPPO and integrated a reinforcement learning module [24]. The RL module used the SARSA algorithm to decide what actions to be taken while an attacker tries to execute a Linux command in the console [25].

2.3 Deep Q-learning honeypot systems

Recently Pauna et al. proposed a new improved system that uses deep Q-learning (DQN) algorithm to fully automate the decision process [17] (Fig. 1).

The above-mentioned systems, as the author also states, face issues related to the subjectivity of the reward functions used by the algorithms. Therefore, there is an obvious need for a way to generate optimal reward functions related to the desired behavior.

2.4 IoT honeypot systems

As mentioned previously, with the increasing prevalence of IoT devices, the cyberattacks followed the trend and, therefore, the need to study such incidents skyrocketed. In the last years, a set of newly developed honeypot systems have appeared, specific to IoT paradigms.

T. Luo suggests a honeypot system that uses machine learning in order to achieve behavioral knowledge of IoT devices [26]. Another relevant study was achieved by Q. D. La et al. [27]. The author aims to define a theoretic game approach of deception involving an attacker and a defender in the IoT context. Y. M. Pa Pa et al. show that Telnet-based attacks targeting IoT devices have an exponential trend and propose a low interaction honeypot system that emulates multiple CPU architectures (ARM, MIPS, and PPC) mainly used by IoT devices [28, 29]. J. Guarnizo et al. depict a new IoT honeypot-based scalable architecture named SIPHON. They basically suggest a system composed of real IoT devices in a setup that enables to collect information on attacks from different geo-locations [30].

A list of currently available and operable IoT honeypot systems comprises the following:

- Phype [31]: that emulates a Telnet service in order to capture IoT malware,
- HoneyThing [32]: emulates a specific IoT protocol: TR-069 (CPE WAN Management Protocol),
- IoTPOT [6]: offers a front-end emulating access to a high interaction environment, called IoTBOX, that virtualizes different CPU architectures commonly used by IoT devices,
- ZigBee honeypot [33]: emulates a ZigBee gateway,
- Multi-purpose IoT honeypot [34]: emulates multiple protocols used by IoT devices such as Telnet, SSH, HTTP, and CWMP.

As can be noted, most of them are low interaction honeypots which, as proven by different research studies [35, 36] are easily detected by attackers. On the other hand,

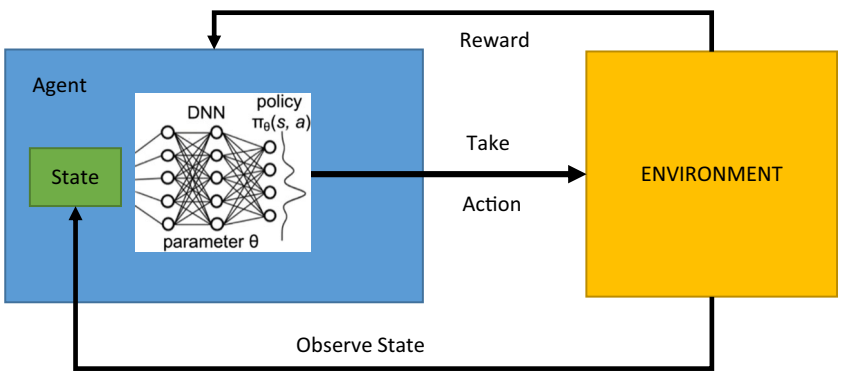

Fig. 1 DQN Self-Adaptive Honeypot paradigm

recent attacks encompass multiple steps and interactions with their targets prior to fully initiating the attack. A good example is the case in which the attacker firstly checks what software and what version of it runs on the IoT device. If they do not match, the attack will not be launched therefore no valuable data will be collected. Such elements make the data collected to be of low quality and quantity [37, 38].

The targeted nature of recent IoT attacks makes it imperative for IoT honeypots to leverage AI algorithms so as to be able to avoid having them revealed and also to be able to interact in a pro-active manner with the attackers. Based on our research, there are some developments as regards IoT honeypot systems that leverage artificial intelligence (AI) [26] but the system we propose is the first fully operable and capable of interacting with attackers by means of machine learning algorithms.

3 Inverse reinforcement learning

In essence, reinforcement learning is the most intuitive form of trial and error learning similar to the way most of the rational beings learn. Just to give an example, when a little child learns to walk for the first time, he moves his legs randomly and then slowly discovers what actions he needs to perform to go forward. This is also known as learning by exploration. The entire process is modeled as a Markov decision process (MDP), meaning that in every state all the actions are available [39]. Based on this principle, the problem is mathematically modeled as a Bellman equation for optimization of long-run discounted rewards.

Better said, the main drive for an agent when choosing actions is the reward function he uses. This takes us to the point where we can say that, based on a particular reward function, an agent will act upon a specific behavior. Taking that into account, we can acknowledge that in order to force the agent to have a specific behavior, we have to modify/exploit the reward structure. And this is where inverse reinforcement learning (IRL) comes into the picture, as IRL helps us to determine the underlying reward structure when given the optimal expert policy (behavior).

IRL has recently become an area of research and one of the well-known related studies belongs to Ng and Russell [40] followed by Abbeel and Ng [19]. Further on, we will present briefly the mathematics and algorithms behind IRL and more specifically apprenticeship learning via inverse reinforcement learning (AL IRL).

In order to have a better view of AL IRL, we will outline the key points regarding reinforcement learning and inverse reinforcement learning.

3.1 Reinforcement learning

RL is a type of unsupervised machine learning that uses an agent to interact with an environment with the purpose of reaching a goal [25]. The simple defined workflow is that the learning agent takes an action that will change the state of its environment. In other words, a state is the configuration of the environment at a given moment and an action is how the agent affects that state (Fig. 2).

The main characteristic of RL is that the algorithm decides on taking actions that do not maximize an immediate reward but take into consideration the future rewards. Better said, the agent tries to maximize future rewards by directing itself to states from which greater reward is gained and these are called states with high value. The *value* can be defined as "how good" it is to be in a specific state related to the expected future reward. The RL algorithms make use of a value function that maps states to values.

The behavior of an agent or more precisely the actions it takes in certain states will generate a policy. A policy usually denoted as π is the mapping of states to probabilities to select actions.

3.2 Deep Q-learning

In order to understand the deep Q-learning (DQN) algorithm, we must first describe the Q-learning algorithm which is the base for it.

Q-learning As mentioned above, RL algorithms make use of value function, which in case of Q-learning is derived from the action-state value function that describes the value of taking an action in a specific state.

Q-learning falls under the temporal difference (TD) algorithms, which means that there are time differences between actions and rewards received [25]. To be more specific, in the case of TD algorithms, we make updates after each episode when an action is taken. In this way, we make a prediction (based on previous experience), take a new action, and, based on the reward we receive, we once again update our prediction. In most of the implementations, we have a table that stores the Q values for every possible state-action pair obtained using the formula:

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_t + \gamma \max Q(S_{t+1}, A_{t+1}) - Q(S_t, A_t)] \quad (1)$$

where:

– α is a parameter $0 \rightarrow 1$ called learning rate and determines the speed at which learning can occur.

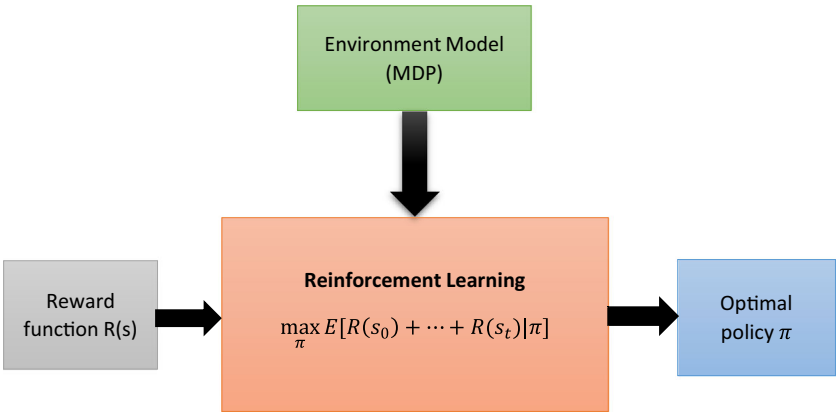

Fig. 2 Reinforcement learning

– γ is a parameter $0 \rightarrow 1$ called discount factor and it determines the way future rewards are used in updating Q values

Our policy π will be the outcome of selecting the action with the highest Q values for the specific state.

Deep Q-learning As in most cases, the state-action space is humongous and complicated to be stored in a table; the correct evolution was to use neural networks (NN) in order to generalize and pattern match between states [41]. In this implementation, NN acts as a function approximator that receives as input a state and an action and will output the value of the state-action pair. Of course, the NN will have a series of parameters associated called weights, therefore the Q function will be $Q(s, a, \theta)$ where θ is a vector of parameters. Basically, instead of updating the values in a table, the algorithm will iteratively update the θ parameters of the NN and thus it will learn better estimates of the state-actions values. As regards the NN implementation, it is a basic one with two feed-forward passes and back.

As depicted in Fig. 3, we pass the state trough the NN and get an estimation of the quality of each action. We then observe the next state and once again pass it through the NN. We calculate the highest Q-value in the output and use a modified Bellman equation to calculate a target value.

$$Q(s_t, a) = \alpha[r_{t+1} + \gamma \max(Q(s_{t+1}, a))] \tag{2}$$

Where:

- $Q(s_t, a)$ is the value (quality) of each action given, the current state and the available actions,
- α is the learning rate,
- r_{t+1} is the reward received after action,
- γ is a discount factor and determines how much each future reward is considered.
- $\max Q(s_{t+1}, a)$ is the highest value (quality) given to the next state.

In the end, we compute the mean squared error between the first forward pass and the target and we back propagate it through the NN. Our expected output vector is called target, is identified with y and its formula is:

$$r_{t+1} + \gamma \max Q(s_{t+1}, a). \tag{3}$$

3.3 Apprenticeship learning via inverse reinforcement learning

As already discussed in reinforcement learning algorithm, the reward function is manually encoded and used in order to generate a (near) optimal policy that maximizes a numerical reward signal.

In the case of AL via RL, we infer a (near) optimal reward function from an optimal policy taught by an expert [19]. In fact, the result is a reward function that tries to explain the expert behavior. Using this function, the algorithm tries to induce a similar behavior (Fig. 4).

The algorithm uses a set of expert trajectories. An expert trajectory is a set of states which the expert has passed through while teaching the behavior.

The reward function is defined by the formula:

$$R(s) = w \cdot \phi(s) \tag{4}$$

where:

- ϕ is the feature vector that contains the state variables, which provide the information about the current configuration of the environment. If the agent is in state s_n, then the features of the state will be $\phi(s_n)$,
- w is the weight vector which determines the reward an agent receives in a particular state. Therefore, for a different weight vector, we have a different reward function.

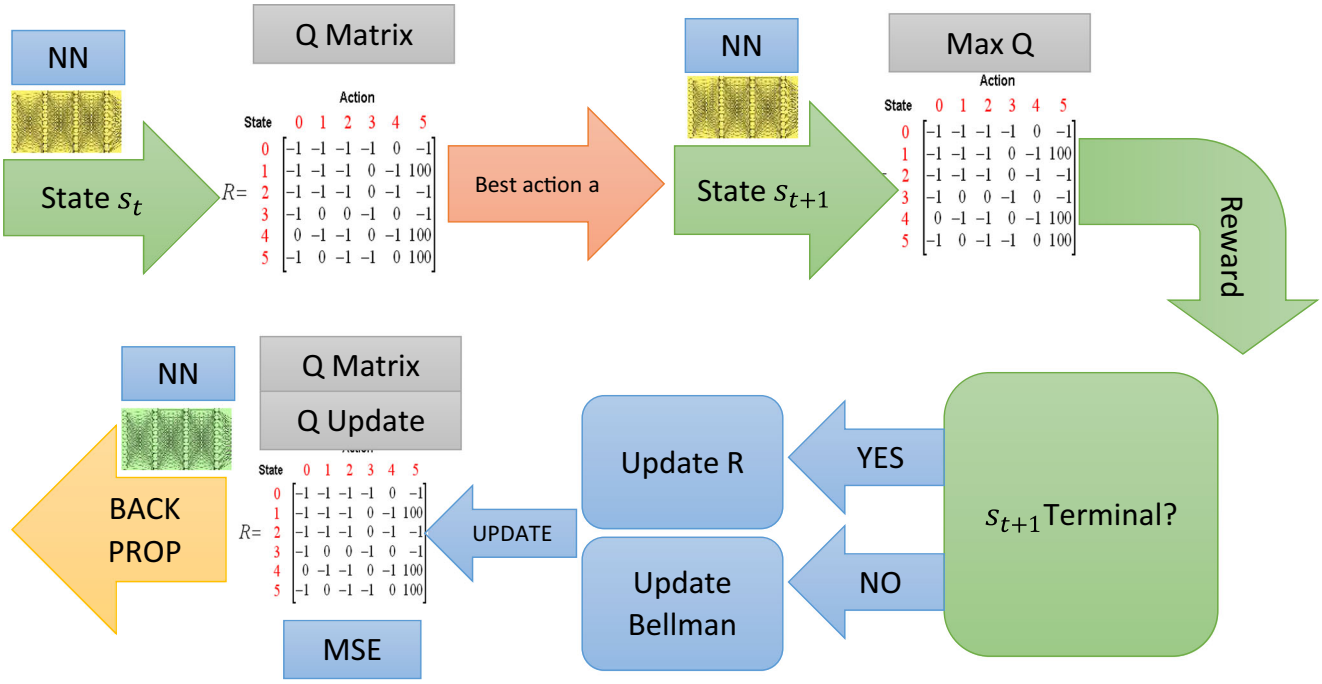

Fig. 3 DQN workflow

The apprenticeship learning algorithm will return an appropriate weight vector that characterizes a reward function which led the expert to behave in the observed way.

An expert trajectory is the sequence of states in the order performed by the expert. The expert is walked through the state space guided by its expert reward function described by the following formula:

$$R^*(s) = w^* \cdot \phi(s) \tag{5}$$

where R^*, the expert's reward, is a function and w^* is the optimal weight vector.

The retrieved weight vector will characterize a reward function that explains the expert trajectories. This is done by using the feature expectations.

The value of a policy π can be written as follows:

$$E_{s_0 \sim D}[V^{\pi}(s_0)] = E\left[\sum_{t=0}^{\infty} \gamma^t R(s_t) | \pi\right] \tag{6}$$

$$= E\left[\sum_{t=0}^{\infty} \gamma^t w \cdot \phi(s_t) | \pi\right] \tag{7}$$

$$= w \cdot E\left[\sum_{t=0}^{\infty} \gamma^t \phi(s_t) | \pi\right] \tag{8}$$

Where s_0 is drawn from D, denoted as the initial-state distribution, the set of starting states. Equation (6) may be rewritten as:

$$E_{s_0 \sim D}[V^{\pi}(s_0)] = w \cdot \mu(\pi) \tag{9}$$

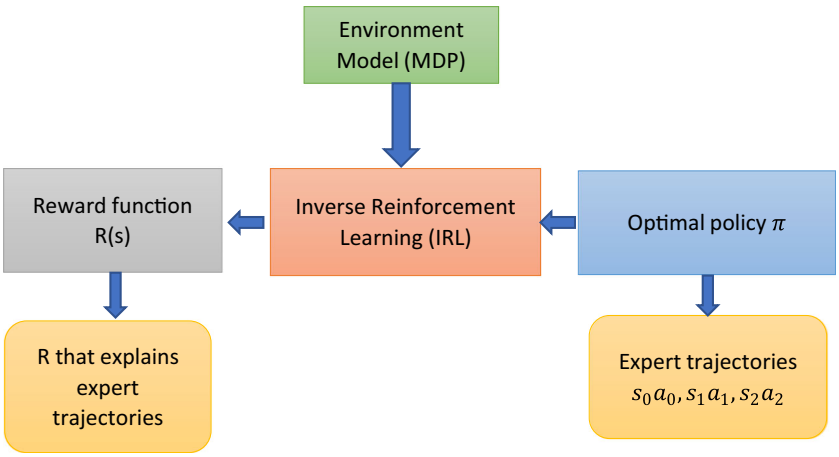

Fig. 4 IRL schema

where

$$\mu(\pi) = E\left[\sum_{t=0}^{\infty} \gamma^t \phi(s_t)|\pi\right] \tag{10}$$

$\mu(\pi)$ is the vector of features expectations of a particular policy.

It is defined as the expected discounted accumulated feature value vector. It represents a discounted sum of the feature vector which is expected to be seen when following a particular policy.

The apprenticeship learning algorithm is based on knowing the expert's feature expectations, that is, knowing μ_E. In practice, an estimate for the expert's feature expectations, $\hat{\mu}_E$ is found empirically; this can be done since we assume access to expert trajectories. An expert trajectory is the expert's path through the state space: $\{s_0, s_1, \ldots\}$. Suppose there are m trajectories through the space, then, we have: $\{s_0, s_1, \ldots\}_{i=1}^{m.}$

The empirical estimate for μ_E is given by the equation:

$$\hat{\mu}_E = \frac{1}{m}\sum_{i=1}^{m}\sum_{t=0}^{\infty}\gamma^t \phi\left(s_t^{(i)}\right) \tag{11}$$

To sum up, the AL via IRL algorithm flow is as follows:

1. We maintain a list of the policy feature expectations obtained after each iteration
2. We initially have π^1 as random policy feature expectations
3. The IRL agent tries to find the reward function that would produce the same feature expectations as the expert policy μ_E generated manually and different from a candidate policy at iteration i, $\mu_{\pi(i)}$. At first, this is generated randomly. A list of policies is maintained within the IRL agent, which are used to find out the appropriate weights for the reward function. The condition for minimization is the following:

$$\min\|w\|_2^2 \tag{12}$$

such that,

$$w^T\mu_E \geq 1 - w^T\mu_{\pi(i)} \geq 1 \tag{13}$$

These weights are used to define a reward function which is used to train a RL agent, and that should create a model that aims to maximize the reward function. From this agent, a new policy can be generated through the same process by which the expert policy was generated but with the RL agent as the expert. This process is repeated until:

$$w^T\left(\mu_E - \mu_{\pi(i)}\right) \leq \epsilon \tag{14}$$

where ϵ is a parameter of the IRL agent.

4. Once we have the weights that generate the reward function, we will compute the optimal policy $\pi^{(1)}$ for the MDP using $R = (w^{(1)})^T\phi$. In order to obtain the policy aiming to maximize the reward function obtained, we will train the RL algorithm with this new reward function until the Q values converge.
5. With the new policy learned, we test it online in order to get the feature expectations corresponding to the new policy.
6. We add such new feature expectations to our list of feature expectations and carry on with IRL next iteration until convergence.

As explained above, the algorithm aims to find a policy so that the agent's performance under it should be close to that of the expert under the unknown reward $R^* = w^* \cdot \phi$. And the condition is that policy $\tilde{\pi}$ will comply with $\|\mu(\tilde{\pi}) - \mu_E\|_2 \leq \epsilon$ where ϵ is an arbitrarily small positive number. In other words, the feature expectations of the expert should be close to the feature expectations induced by such a policy.

4 IRASSH-T implementation

As presented above, the AL IRL algorithm uses an RL agent as a basis and adds the necessary functionalities in order to calculate expert trajectories, expected features and finally the reward function. For this purpose and in line with the scope of our research, we have used the Self-adaptive honeypot system QRASSH [17].

QRASSH is a honeypot system that uses the source code of an existing project called Cowrie [42]. Cowrie is a honeypot system developed by Osterhof as a branch of a previous open source project Kippo [24]. It basically emulates SSH and Telnet services enabling access to a set of Linux commands and features making it a really good decoy for attackers. It also provides the ability to impersonate specific Linux distributions. For our research, in order to lure the attacker, we have used busybox Linux and activated both SSH and Telnet services.

QRASSH adds specific functionalities to Cowrie in order to make it act as an RL agent. It implements a set of five actions: allow command execution, block command execution, delay command execution, fake the command output, and insult the intruders. Such actions are triggered by the deep Q-learning module implemented according to the paradigm in which we consider the environment as the set of commands implemented in the honeypot. In order to have the Q-learning algorithm working, we have defined a reward function with the goal of luring the attackers to download as much malicious

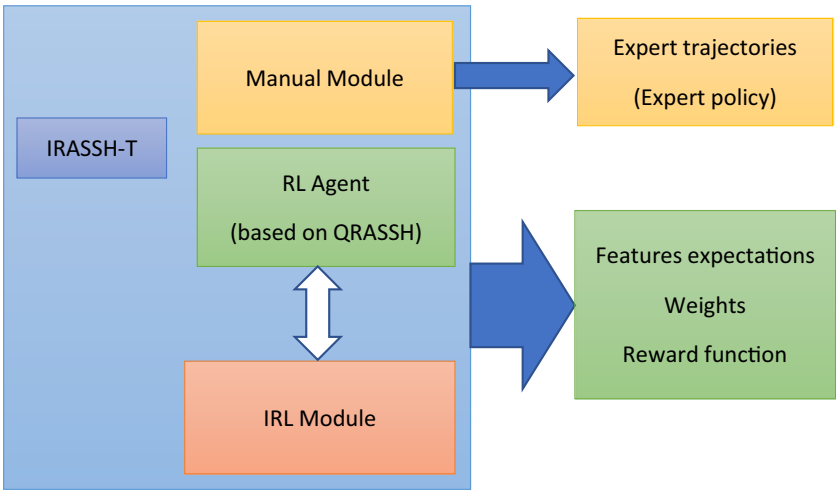

Fig. 5 IRASSH-T block schema

code as possible on the honeypot. Tests have proven that manually designing a reward function is a difficult task, considering that the attackers' behavior is hard to define. Therefore, our research took us to the point of using a method enabling us to estimate quasi-optimal rewards functions that characterize specific behaviors (Fig. 5).

The main components of the IRASSH-T [43] as depicted in Fig. 2 are threefold:

- RL agent that uses QRASSH as basis with specific adaptations, such as the fact that the reward function is now a linear combination of features.
- Manual training module: is a module by which we can teach the agent the expert optimal policy.
- The IRL module: is a module by which we make calculations based on the AL IRL algorithm in order to derive the near optimal reward function.

4.1 RL agent

The implementation uses the following:

- Agent: is considered the honeypot system that is able to interact with the attacker and log all their command and other useful information such as: SSH clients used, Telnet sessions, source IP addresses, GeoIp information
- Environment: the Linux commands inserted by the attacker are considered to be the environment. We have three groups of commands: Linux commands emulated by the honeypot system which are not download commands, Linux commands emulated by the honeypot system which allow download (scp, wget, sftp, etc.), Linux commands which are not emulated in the honeypot system.
- Actions: the five actions previously presented: allow, block, delay, fake, and insult

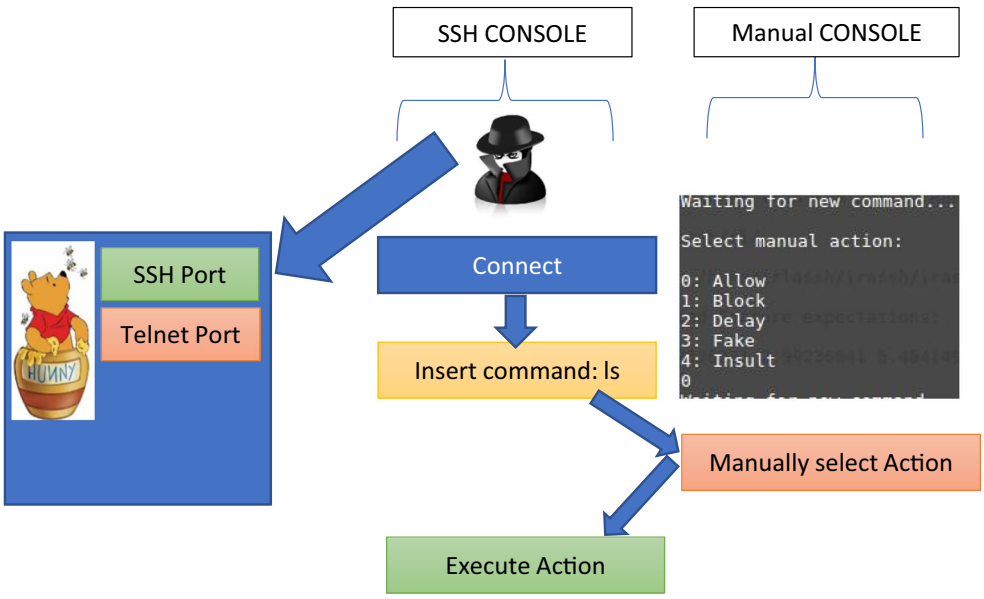

Fig. 6 Manual module

Fig. 7 Output of feature expectation calculated

```
(irlassh-env)/home/irlassh/irassh/irassh/rl $ python
/home/irlassh/irassh/irassh/rl/manual.py
/home/irlassh/irassh/input/cmd_100.p
/home/irlassh/irassh/manual/expertFE_download.p 1 -p
Resulted Feature Expectations:
[[2.27336321, 2.99236041, 0.1233456]]
```

- Reward function: which positively compensates any download command and negatively compensates the exit command (attacker leaving the honeypot)
- The deep Q-learning algorithm implementation, which decides the actions to be taken after each command input.

4.2 Manual training module

This module installs an add-on to the existing honeypot implementation and allows actions to be taken manually in a separate console as presented in Fig. 6.

The module output is a pickle file that will contain the sequence of Linux commands executed by the attacker. As described in the algorithm, this is the way we obtain the expert policy. We then run the manual.py script that will extract the feature expectations related to the expert policy (Fig. 7).

4.3 IRL module

As the main scope of our research was neither to test nor to improve the existing AL IRL algorithm, we have used the code developed by Jangir [44] for the implementation of the IRL module. We have adapted it to our issue, as presented below.

The module uses as input the expert feature expectations calculated using the manual mode and determines a random policy in order to initialize.

For the expert features expectations, we have used as a reference two types of expert behavior: download and retain.

As you can see in Fig. 8, the download behavior results from the expert choosing actions that will "force" the attacker to insert commands meaning downloading as much malicious code as possible. The scenario we had in mind is that an attacker will try to check if gcc compiler is present and therefore download a code in C programing language. But when he tries to locally compile the code, the command will be blocked and, therefore, he will try to download a code in Java programing language with the same functionality.

In the case of retain behavior, we select delay actions as much as possible, which will make the attacker spend more time on the honeypot system. In this way we will have more time to evaluate its origin using other tools than honeypots.

For both behaviors above we calculate expected features.

The expected features are derived from a set of observable features that define the state space. The observable features are

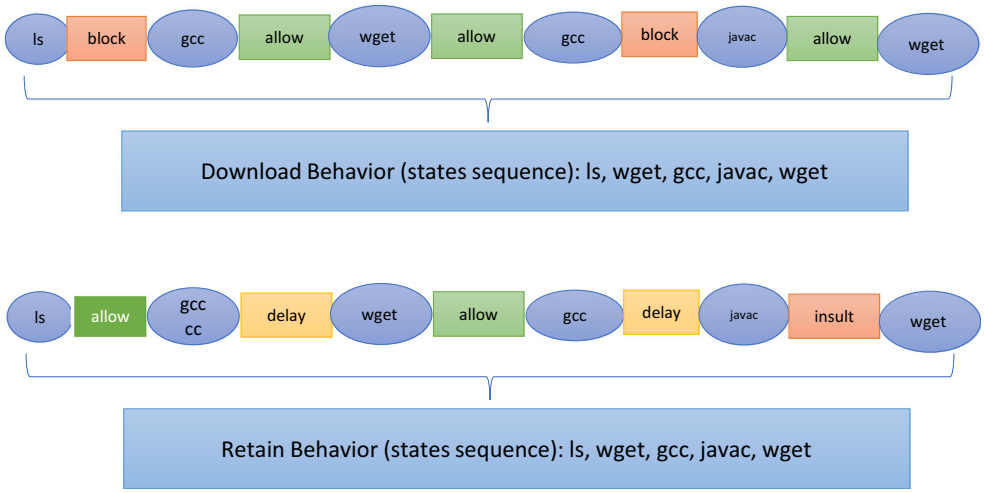

Fig. 8 Download and retain behaviors

Fig. 9 IRLagent parameters

```
NUM_STATES = 8
BEHAVIOR = 'download' # download/retain/exit
FRAMES = 100000 # number of RL training frames per
iteration of IRL
```

read using a set of "sensors." The sensors are basically methods that evaluate the commands from different perspectives as follows:

1. It is a download command "sensor 1,"
2. It is a Linux commands "sensor 2,"
3. It is a hacking command "sensor 3,"
4. There is a delay "sensor 4,"
5. It is the exit command "sensor 5."

Afterwards, we call the *irlAgent* and pass the desired parameters and select the type of expert behavior. We also set the number of FRAMES in which we want the RL algorithm to run (Fig. 9).

Then we create the *irlAgent* class, which takes in the random and expert behaviors, and other important parameters as shown (Fig. 10).

The *getRLAgentFE* function uses the *IRL_helper* from the reinforcement learner to train a new model and get feature

Fig. 10 IRLagent class

```
class irlAgent:
 def __init__(self, params, randomFE, expertFE,
epsilon=0.1, num_frames=5000, behavior="DefaultPolicy",
play_commands=2000, cmd2props = None, oneHot = False):
        self.play_commands = play_commands
        self.params = params
        self.oneHot = oneHot
        if isinstance(params["cmd2number_reward"], str):
#if string load from file
            extention =
params["cmd2number_reward"].split(".")[-1]
            if extention == "xlsx":
                self.cmd2number_reward =
get_cmd2reward(params["cmd2number_reward"])
            else:
                self.cmd2number_reward =
pickle.load(open(params["cmd2number_reward"],"rb"))
        else:               # if dictionary
            self.cmd2number_reward =
                                params["cmd2number_reward"]
        self.sequence_length = params["sequence_length"]
        self.number_of_actions =
                                params["number_of_actions"]
        self.randomPolicy = randomFE
        self.expertPolicy = expertFE
        self.num_frames = num_frames
       if cmd == "exit":
          print(self.cmd2props[cmd_num])
          self.number_of_props = len(cmd2props[cmd])
       if 0 not in self.cmd2props:
          if 0 in cmd2props:
              self.cmd2props [0] = cmd2props[0]
          else:
              self.cmd2props [0] = np.zeros
                                      (self.number_of_props)
       else:
            self.cmd2props = None
       self.behavior = behavior
       self.epsilon = epsilon # termination when t<0.1
       self.randomT = np.linalg.norm(
       np.asarray(self.expertPolicy) -
                           np.asarray(self.randomPolicy))
                    # norm of the diff in expert and random
        self.policiesFE = {
            self.randomT: self.randomPolicy} # storing the
policies and their respective t values in a dictionary
        print("Expert - Random at the Start (t):: ",
self.randomT)
        self.currentT = self.randomT
        self.minimumT = self.randomT
        self.weights_found = False
        self.optimalWeightFinderStart()
```

expectations by playing that model for 2000 iterations. It basically returns the feature expectations for every set of weights (W) it gets.

We update the dictionary in which we keep our obtained policies and their respective t values.

Where:

$t = (\text{weights.tanspose})*(\text{expert}-\text{newPolicy}).$

We implement the convex optimization to update the weights upon receiving a new policy. That means we assign + 1 label to expert policy and − 1 label to all the other policies and optimize for the weights under the mentioned constraints [44, 45] (Fig. 11).

We create an irlAgent and pass the desired parameters, select between the type of expert behavior you wish to learn the weights for and then run the *optimalWeightFinder()* function, as we have already obtained the feature expectations for download, retain and exit behaviors. After the algorithm terminates, we will obtain a list of weights in a pickle file, with the respective selected BEHAVIOR. In order to select the best possible behavior from all the obtained weights, we play the saved. We will get different weights for different iterations. We first play the models to find out which one performs best, then note the iteration number of that model, and the weights obtained corresponding to this iteration number are the weights that get you closest to the expert behavior.

5 Testing results

For testing purposes, we have used a set of logged Linux commands collected with an IoT honeypot system exposed over the Internet.

Dulaunoy et al. present a practical analysis of Internet of things malware by inspecting traffic from a blackhole. We have used the collected dataset in order to generate the desired behavior for our honeypot system [46].

The dataset offers us a bunch of insights from which we have derived the behavior we have taught to our agent. The dataset spans over a period of 16 weeks during which plenty of logged-in sessions were analyzed.

The results show that the owners of Mirai botnet have instructed the infected systems in a less consistent manner during the said time period. This is proven by the heterogeneous variation of commands initiated during the 16 weeks period. To sustain the above, our research shows that:

- The most used set of commands issued in the following order is: “enable, system, shell, sh, /bin/busybox MIRAI”. It shows that after initiating a full session, the botnet tries to gather information on the infected system
- The second most used suite of commands is: “*cd /var/tmp;cd /tmp;rm -f *;tftp -l 7up -r 7up -g 89.33.64.118;chmod a+x 7up;./7up*”. It indicates that after successful login, the botnet tries to download the malware necessary to fully infect the system. It searches for writeable area, wiping everything inside it, downloading the infection binary and launching it in the end.
- The variety of tools used to fetch the malware spreads from *TFTP* to *curl* and *wget* as most of them are simple bash scripts used to download the same binary from different locations.

The previously presented behaviors, we have defined, fit the above analysis, as it is obvious that our honeypot system will directly interact with the *botnet* herders. In the case of the *download* behavior we can force them to download malware from different locations and thus we may lure them to expose much more of the systems they use. Also, the *retain* behavior

Fig. 11 IRLagent convex optimization implementation

```
def optimization(self):  # implement the convex
optimization, posed as an SVM problem
        m = len(self.expertPolicy)
        P = matrix(2.0 * np.eye(m), tc='d')  # min ||w||
        q = matrix(np.zeros(m), tc='d')
        policyList = [self.expertPolicy]
        h_list = [1]
        for i in self.policiesFE.keys():
            policyList.append(self.policiesFE[i])
            h_list.append(1)
        policyMat = np.matrix(policyList)
        policyMat[0] = -1 * policyMat[0]
        G = matrix(policyMat, tc='d')
        h = matrix(-np.array(h_list), tc='d')
        sol = solvers.qp(P, q, G, h)
        weights = np.squeeze(np.asarray(sol['x']))
        norm = np.linalg.norm(weights)
        weights = weights / norm
        return weights  # return the normalized weights
```

fits our need, as in this way we can track the bot herders on a longer period.

In order to test the above, we have followed the three phases presented below and came up with the results expected (Fig. 12).

5.1 Workflow phases

Phase 1 a) We have configured IRASSH-T in manual mode by setting up the manual parameter in honeypot.py with the true value

b) We have run the manual console in parallel with the SSH console and selected actions based on the download behavior scenario

c) We have obtained the cmd.p pickle file and used the command:

python manual.py cmd.p expertFE_download.p –p and obtained the expert features expectations:

[[2.27336321, 2.99236041, 0.1233456]]

Phase 2 a) We have configured IRASSH-T in IRL mode by setting up the use_irl parameter in honeypot.py with the true value

b) We have run *irassh start* and input commands until the irl agent is trained

c) The results is a pickle file named using the behavior attribute "download-optimal_wight.p"

Phase 3 a) We have run the irassh/rl/policy2reward.py with the output from the previous phase and thus obtained the reward function in a pickle file that has the calculated weight as depicted in Table 1.

d) We have configured IRASSH-T in training mode by setting up both parameters use_irl and manual with the value false

Table 1 Calculated weights based on behavior

Weights	Download behavior	Retain behavior
Senzor 1 (download_commad)	0.8802	0.2816
Senzor 2 (linux_command)	−0.0624	−0.5547
Senzor 3 (hacking_command)	0.1123	−0.5922
Senzor 4 (is_delay)	−0.1234	0.7865
Senzor 5 (exit command)	−0.2897	−0.2201

b) We have run IRASSH-T with the obtained previously reward set up accordingly in *actions/proxy.py*, *cmd2number_reward* parameter

5.2 Interpretation of the calculated weights

We admit it is very difficult to infer meaningful patterns in the weights as the distance between feature weights is very ambiguous. Still, a set of results can be defined:

- Apparently the weights for the *download_command* and *is_delay* sensors are related to the expert behavior and have high positive values;
- A high negative value is assigned to the weight that belongs to *exit_command* feature, as in both behaviors the intention is to have the attacker longer inside the console.

5.3 Results summary

We have created an irlAgent and passed him the type of expert behavior for which we wish to learn the weights: in our case the download behavior.

The agent learns a set of weights for the download behavior based on the already learned, from expert, feature expectations.

The agent, iteratively, calculates a set of weights. We have played the model for each iteration and, thus, we have

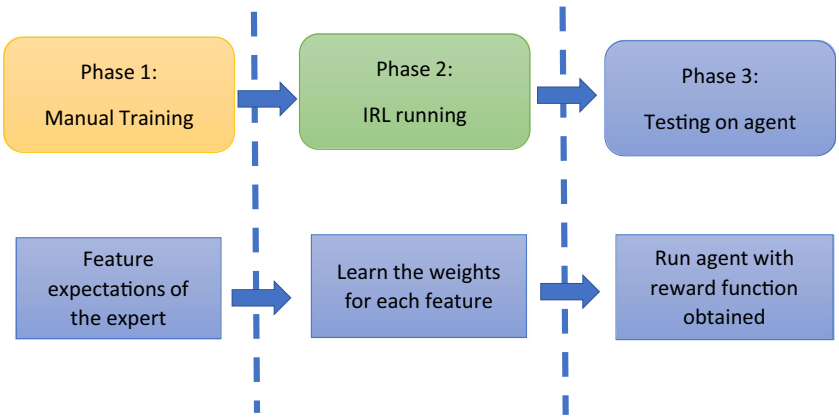

Fig. 12 Phases of honeypot AL IRL workflow

```
(irlassh-env) irlassh@adrianp-PHD ~/irassh/bin $ ./playlog
../log/tty/20180319-104107-0ad857a84e1f-0i.log
BusyBox v1.14.1 (2018-06-12 12:06:47 EET) built in shell
(msh)
Enter 'help' for a list of built-in commands.
# ls
ALLOW
#ls
DELAY
#pwd
This is an insult msg
INSULT
#wget
ALLOW
wget: missing URL
Usage: wget[OPTION].. [URL]..
Try 'wget - -help' for more options
#sftp
ALLOW
#curl
ALLOW
Curl: try 'curl -help' or 'curl -manual' for more
information
#scp
ALLOW
```

Fig. 13 Output of obtained reward function

determined the set of weights that approximate the expert behavior. Based on the calculated optimal set of weights we have derived an optimal reward function.

We then tested the optimal derived reward function by configuring it on the RL agent. The RL agent, our honeypot, interacts with the attackers and confirms the download behavior as in Fig. 13, in the snapshot taken from the SSH interface.

As can be seen in the above, the learned reward function acts as we expected, by allowing all the download commands.

6 Conclusions and future work

At this moment, the presence of IoT in our lives is obvious through the smart home applications based on IoT, and the adoption trend is exponential [47, 48].

Due to their low computational design which does not allow to take into account too many security controls that would mitigate risks, IoT systems are vulnerable. There are studies that try to compensate this issue by various means such as the usage of lightweight block ciphers for low-resources devices [49], but there is a long way until we have a practicable solution. Therefore, the study of cyberattacks related to IoT is a must and the usage of self-adaptive honeypot systems is already a success story.

The paper presents a fully-fledged IoT honeypot system capable of learning optimal reward functions to be used in real-life implementation. We have tested two specific behaviors that partially map the Mirai botnet. As shown by the visual results presented, the learned reward function helps the agent to act as desired behavior. Our evaluation indicates that our system improves the reward functions used by self-adaptive honeypots capable of interacting with attackers and, in this way, will capture more attacks and lure them to download more malware. The usage of self–adaptive honeypot system makes it possible to collect IoT target malware because of their capability to directly interact with the attackers. With the usage of inverse reinforcement learning algorithm, we have proved that it is possible to overcome one of the key issues of self-adaptive honeypots, namely the subjectivity of the reward functions. For future developments we consider relevant testing of other behaviors, such trying to decoy the attackers to reveal their origins. IoT honeypots are used at large scale and there is a genuine interest in developing ways to pattern attackers' behavior [50–52]. The results are promising but still there is room for improvement, for example considering novel encryption schemes [53, 54]. We also think that different IRL algorithms [55] can be used in order to better calculate reward functions.

Acknowledgments We express our thanks to all reviewers for their valuable comments and remarks.

Funding information The research presented in this paper is supported by the following projects: ATLAS (PN-III-P1-1.2-PCCDI-2017-0272) and ROBIN (PN-III-P1-1.2-PCCDI-2017-0734).

References

1. Online dictionaries. (2017) Definition: "Internet of things (IoT)". https://en.oxforddictionaries.com/definition/internet_of_things. Accessed on November 2018

2. Alper Erdal, "Cisco internet of things," October 2015. Available: https://www.slideshare.net/Panduit/cisco-internet-of-things. Accessed on November 2018
3. Barcena, Mario Ballano, and Candid Wueest. Insecurity in the internet of things. Security response, symantec (2015)
4. Markowsky, Linda, and George Markowsky (2015) Scanning for vulnerable devices in the internet of things. In Intelligent data acquisition and advanced computing systems: technology and applications (IDAACS), 2015 IEEE 8th international conference on, vol. 1, pp. 463–467. IEEE
5. NSA (2016) The next wave - the internet of things: it's a wonderfully integrated life, vol. 21, no. 2
6. Pa Y, Pa M, Suzuki S, Yoshioka K, Matsumoto T, Kasama T, Rossow C (2016) Iotpot: a novel honeypot for revealing current iot threats. J Inf Proc 24(3):522–533
7. Alaba FA, Othman M, Hashem IAT, Alotaibi F (2017) Internet of things security: a survey. J Netw Comput Appl 88:10–28, ISSN 1084–8045. https://doi.org/10.1016/j.jnca.2017.04.002
8. Antonakakis M, April T, Bailey M, Bernhard M, Bursztein E, Cochran J, Durumeric Z et al. (2017) Understanding the mirai botnet. In USENIX security symposium, pp. 1092–1110
9. Aceto G, Botta A, Marchetta P, Persico V, Pescapé A (2018) A comprehensive survey on internet outages. J Netw Comput Appl 113:36–63, ISSN 1084-8045. https://doi.org/10.1016/j.jnca.2018.03.026
10. Cui A, and Stolfo SJ (2010) A quantitative analysis of the insecurity of embedded network devices: results of a wide-area scan. In Proceedings of the 26th annual computer security applications conference, pp. 97–106. ACM. https://doi.org/10.1145/1920261.1920276
11. He H, Maple C, Watson T, Tiwari A, Mehnen J, Jin Y, and Gabrys B (2016) The security challenges in the IoT enabled cyber-physical systems and opportunities for evolutionary computing & other computational intelligence. In Evolutionary computation (CEC), 2016 IEEE congress on, pp. 1015–1021. IEEE
12. Kumar P, Kunwar RS, and Sachan A (2016) A survey report on: security & challenges in internet of things. In Proc National Conference on ICT & IoT, pp. 35–39
13. Senpai A (2016) Mirai-source-code. Available: https://github.com/jgamblin/Mirai-Source-Code. Accessed on November 2018
14. Vlasenko D (2008) Busybox: The swiss army knife of embedded linux. Available: https://www.busybox.net/about.html, Accessed on November 2018
15. Catuogno L, Castiglione A, Palmieri F (2015) A honeypot system with honeyword-driven fake interactive sessions. In: 2015 international conference on high performance computing and simulation (HPCS), pp. 187–194. IEEE, July
16. Pauna A, and Bica I (2014) RASSH-reinforced adaptive SSH honeypot. In Communications (COMM), 2014 10th international conference on, pp. 1–6. IEEE. https://doi.org/10.1109/ICComm.2014.6866707
17. Pauna A, Iacob A-C, and Bica I (2018) QRASSH-A self-adaptive SSH honeypot driven by Q-learning. In 2018 international conference on communications (COMM), pp. 441–446. IEEE. https://doi.org/10.1109/ICComm.2018.8484261
18. Wagener G, State R, Engel T, and Dulaunoy A (2011) Adaptive and self-configurable honeypots. In 12th IFIP/IEEE international symposium on integrated network management (IM 2011) and workshops, pp. 345–352. IEEE. https://doi.org/10.1109/INM.2011.5990710
19. Abbeel P, and Ng AY (2004) Apprenticeship learning via inverse reinforcement learning. In Proceedings of the twenty-first international conference on machine learning, p. 1. ACM
20. Spitzner L: https://www.symantec.com/connect/articles/dynamic-honeypots, Accessed on November 2018
21. Cheswick B (1992) An evening with Berferd in which a cracker is lured, endured, and studied. In Proc. winter USENIX conference, San Francisco, pp. 20–24
22. Wagener G Thesis: self-adaptive honeypots coercing and assessing attacker behaviour http://hdl.handle.net/10993/15673. Accessed on November 2018
23. Pauna A (2012) Improved self adaptive honeypots capable of detecting rootkit malware. In Communications (COMM), 2012 9th international conference on, pp. 281–284. IEEE, https://doi.org/10.1109/ICComm.2012.6262612
24. Kippo: https://github.com/desaster/kippo, Accessed on November 2018
25. Sutton RS, Barto AG (2018) Reinforcement learning: An introduction. MIT press. https://www.amazon.com/Reinforcement-Learning-Introduction-Adaptive-Computation/dp/0262193981. Accessed 19 Dec 2018
26. Luo T, Xu Z, Jin X, Jia Y, and Ouyang X (2017) Iotcandyjar: towards an intelligent-interaction honeypot for iot devices. Black Hat
27. La QD, Quek TQS, and Lee J (2016) A game theoretic model for enabling honeypots in IoT networks. In Communications (ICC), 2016 IEEE international conference on, pp. 1–6. IEEE. https://doi.org/10.1109/ICC.2016.7510833
28. Yin Minn Pa Pa, Suzuki S, Yoshioka K, Matsumoto T, Kasama T, and Rossow C (2015) IoTPOT: analysing the rise of IoT compromises. In Proceedings of the 9th USENIX conference on offensive technologies (WOOT'15), Aurélien Francillon and Thomas Ptacek (Eds.). USENIX Association, Berkeley, CA, USA, 9–9
29. Bhangwar NH, Halepoto IA, Khokhar S, Laghari AA (2017) On routing protocols for high performance. Stud Inf Control 26(4):441–448
30. Guarnizo JD, Tambe A, Bhunia SS, Ochoa M, Tippenhauer NO, Shabtai A, and Elovici Y (2017) SIPHON: towards scalable high-interaction physical honeypots. In Proceedings of the 3rd ACM workshop on cyber-physical system security (CPSS '17). ACM, New York, NY, USA, 57–68. https://doi.org/10.1145/3055186.3055192
31. Phype. Telnet IoT honeypot. https://github.com/Phype/telnet-iot-honeypot. Accessed on November 2018
32. Honeything. https://github.com/omererdem/honeything. Accessed on November 2018
33. Dowling S, Schukat M and Melvin H (2017) A ZigBee honeypot to assess IoT cyberattack behaviour, 28th Irish Signals and Systems Conference (ISSC), Killarney, 2017, pp. 1–6. https://doi.org/10.1109/ISSC.2017.7983603
34. Krishnaprasad P (2017) "Capturing attacks on IoT devices with a multi-purpose IoT honeypot". PhD thesis, Indian Institute of Technology Kanpur
35. Krawetz N (2004) Anti-honeypot technology, in IEEE Security & Privacy, vol. 2, no. 1, pp. 76–79. https://doi.org/10.1109/MSECP.2004.1264861
36. T. Holz and F. Raynal (2005) Detecting honeypots and other suspicious environments, Proceedings from the Sixth Annual IEEE SMC Information Assurance Workshop, West Point, NY, USA, pp. 29–36. https://doi.org/10.1109/IAW.2005.1495930
37. Fioriti V, Chinnici M (2017) Node seniority ranking in networks. Stud Inf Control 26(4):397–402
38. Năstase L, Sandu IE, Popescu N (2017) An experimental evaluation of application layer protocols for the internet of things. Stud Inf Control 26(4):403–412
39. Bellman R (1957) A Markovian decision process. J Math Mech 6:679–684. https://www.jstor.org/stable/24900506?seq=1#page_scan_tab_contents. Accessed 19 Dec 2018
40. Ng AY, Russell SJ (2000) Algorithms for inverse reinforcement learning. In: Langley P (ed) Proceedings of the seventeenth

international conference on machine learning (ICML '00). Morgan Kaufmann Publishers Inc., San Francisco, CA, pp 663–670
41. Li H, Wei T, Ren AO, Qi Z, and Wang Y (2017) Deep reinforcement learning: framework, applications, and embedded implementations. In Computer-aided design (ICCAD), 2017 IEEE/ACM international conference on, pp. 847–854. IEEE. https://doi.org/10.1109/ICCAD.2017.8203866
42. Cowrie: https://github.com/micheloosterhof/cowrie, Accessed on November 2018
43. IRASSH-T: https://github.com/adpauna/irassh, Accessed on November 2018
44. Miguel Sousa Lobo, Lieven Vandenberghe, Stephen Boyd, Hervé Lebret, Applications of second-order cone programming, Linear Algebra Appl, volume 284, issues 1–3,1998, Pages 193–228, ISSN 0024-3795, https://doi.org/10.1016/S0024-3795(98)10032-0
45. Quadratic Programming in Python https://scaron.info/blog/quadratic-programming-in-python.html. Accessed on November 2018
46. Dulaunoy A, Wagener G, Mokaddem S, and Wagner C (2017) An extended analysis of an IoT malware from a blackhole network. TNC17
47. Alaa M, Zaidan AA, Zaidan BB, Talal M, Kiah MLM (2017) A review of smart home applications based on internet of things. J Netw Comput Appl 97:48–65. https://doi.org/10.1016/j.jnca.2017.08.017
48. Popa D, Pop F, Serbanescu C, and Castiglione A (2019) Deep learning model for home automation and energy reduction in a smart home environment platform. Neural Comput Applic (in press)
49. Mohd BJ, Hayajneh T, Vasilakos AV (2015) A survey on lightweight block ciphers for low-resource devices: comparative study and open issues. J Netw Comput Appl 58:73–93, ISSN 1084-8045. https://doi.org/10.1016/j.jnca.2015.09.00
50. Ishitaki T, Obukata R, Oda T, and Barolli L (2017) Application of deep recurrent neural networks for prediction of user behavior in Tor networks. In Advanced information networking and applications workshops (WAINA), 2017 31st international conference on, pp. 238–243. IEEE, https://doi.org/10.1109/WAINA.2017.63
51. Chifor B-C, Bica I, Patriciu V-V, Pop F (2018) A security authorization scheme for smart home internet of things devices. Futur Gener Comput Syst 86:740–749
52. Esposito C, Castiglione A, Palmieri F, Ficco M, Dobre C, Iordache GV, Pop F (2018) Event-based sensor data exchange and fusion in the internet of things environments. J Parallel Distrib Comput 118:328–343
53. Zhimin Yu, Chong-zhi Gao, Zhengjun Jing, Brij Bhooshan Gupta, Qiuru Cai. A practical public key encryption scheme based on learning parity with noise. IEEE Access, 2018. https://doi.org/10.1109/ACCESS.2018.2840119, 6, 31918, 31923
54. Yang L, Han Z, Huang Z, Ma J (2018) A remotely keyed file encryption scheme under mobile cloud computing. J Netw Comput Appl 106:90–99
55. Tan C, Li Y, and Cheng Y (2017) An inverse reinforcement learning algorithm for semi-Markov decision processes. In Computational intelligence (SSCI), 2017 IEEE symposium series on, pp. 1–6. IEEE. https://doi.org/10.1109/SSCI.2017.8280816