RL-IoT: Reinforcement Learning to Interact with IoT Devices

RL-IoT: Reinforcement Learning to Interact with IoT Devices

Giulia Milan
Politecnico di Torino
Turin, Italy
giulia.milan@polito.it

Luca Vassio
Politecnico di Torino
Turin, Italy
luca.vassio@polito.it

Idilio Drago
University of Turin
Turin, Italy
idilio.drago@unito.it

Marco Mellia
Politecnico di Torino
Turin, Italy
marco.mellia@polito.it

***Abstract*—Our life is getting filled by Internet of Things (IoT) devices. These devices often rely on closed or poorly documented protocols, with unknown formats and semantics. Learning how to interact with such devices in an autonomous manner is the key for interoperability and automatic verification of their capabilities. In this paper, we propose RL-IoT, a system that explores how to automatically interact with possibly unknown IoT devices. We leverage reinforcement learning (RL) to recover the semantics of protocol messages and to take control of the device to reach a given goal, while minimizing the number of interactions. We assume to know only a database of possible IoT protocol messages, whose semantics are however unknown. RL-IoT exchanges messages with the target IoT device, learning those commands that are useful to reach the given goal. Our results show that RL-IoT is able to solve both simple and complex tasks. With properly tuned parameters, RL-IoT learns how to perform actions with the target device, a Yeelight smart bulb in our case study, completing non-trivial patterns with as few as 400 interactions. RL-IoT paves the road for automatic interactions with poorly documented IoT protocols, thus enabling interoperable systems.**

***Keywords*—Reinforcement learning, IoT**

I. Introduction

The popularity of IoT devices keeps growing at a fast pace, with the number of connected devices projected to be around 31 billion units worldwide by 2025. IoT devices are present in many IT systems, from smart homes to drones, from industry 4.0 scenarios to medical systems.[1] These devices rely on multiple standard protocols and technologies [1], such as MQTT, CoAP and XMPP, but often they implement proprietary and not well-documented protocols whose semantics may be obscure.

A general approach for learning how to interact with IoT devices would represent an important step for many applications, including interoperability and cybersecurity. In the literature, this problem lies under the umbrella of protocol reverse engineering, i.e., the process of learning the protocol used by an application, having no or limited access to the protocol specification [2]–[4]. For interoperability purposes, one often faces a simplified version of the problem, in which *some* information about the protocol is indeed available. For instance, protocol messages and syntax may be public, but with little information about *protocol semantics*. Equally, even if some protocol information may be available, finding the precise operations providing a particular functionality may be a hard task due to poor documentation.

In this work, we build a system capable of learning by experience how to interact with IoT devices. In details, given i) a target IoT device, e.g., a smart bulb, ii) a superset of protocol messages (not all of them supported by the target device), iii) a communication network, and iv) a feedback channel, we want to learn the specific sequence of messages that allows us to change the IoT device settings according to a desired sequence of states. At the end, the system shall unveil the semantics of the messages and their possible mutual interactions in the shortest possible time.

To reach our goal, we rely on reinforcement learning (RL) [5]. A *learner* stimulates the device and observes how it reacts, obtaining a positive (negative) reward when the device does (does not) perform the desired action. We assume to receive a feedback from the device, for instance having a side channel to observe how its status changes (e.g., a camera looking at the smart bulb) or a feedback channel directly offered by the IoT protocol. More formally, RL builds an internal state-machine representing a portion of the IoT protocol. The learner's goal is to discover how to navigate the state-machine, finding the best (e.g., shortest) sequence of actions to reach our goal.

We present RL-IoT, a RL-based framework to automatically interact with IoT devices. We focus on a case study of a Yeelight smart bulb, which offers a proprietary protocol, generically documented for all Yeelight devices. We present the design of RL-IoT and offer a thorough set of experiments, comparing different RL methods, tuning parameters, and showing that RL-IoT is effective to control the smart bulb, successfully completing both simple and complicated sequences of actions.

Results show that not only RL-IoT is able to find the optimal sequence of commands to control the device, but also discover multiple solutions, combining commands that at a first sight are not useful to reach the goal. For example, it finds out that a command for changing the brightness of a smart bulb can also be used to switch the light off. Among the different RL algorithms tested, Q-learning presents the best performance. With tuned parameters, it learns the optimal sequence of

[1] https://www.statista.com/statistics/1101442/iot-number-of-connected-devices-worldwide

commands after few hundreds interactions, exploring the state space of the smart bulb, which in turn has millions of states.

RL-IoT demonstrates how RL solutions can be successfully exploited to support semantic interoperability, opening to possible automated solutions to discover the semantics of poorly documented IoT systems. RL-IoT is open source and freely available to the community.[2]

II. Related Work

The work most similar to ours is [6] where the authors propose the use of the Q-learning algorithm to facilitate the interoperability of IoT systems. However, the authors only discuss the applicability of the RL-approach to a REST-based protocol, without introducing a general system or validating the approach. Here we demonstrate the potentiality of the idea without assuming a specific protocol. We also demonstrate the feasibility of RL-IoT in practice and contribute the software to the community.

Considering the use of RL for learning protocols, most previous work targets security applications, such as honeypots. Authors of [7] develop a honeypot capable of learning commands from direct interaction with attackers. Their self-adaptive honeypot emulates a SSH server and uses the SARSA RL algorithm to interact with attackers. Later, the same authors propose an improved version based on Deep Q-learning [8]. The authors of [9] design another adaptive honeypot, modelling the attacker as a Semi-Markov Decision Process (SMDP) and applying RL to learn the optimal policy.

The authors of [10] present adaptive honeypots for studying the security of IoT devices. They propose to use RL to automatically obtain knowledge about the behaviour of attackers, building an "intelligent-interaction" honeypot that could engage attackers. Authors of [11] study IoT attacks too. The authors argue that the diversity of protocols, software and hardware of IoT devices, together with dynamic changes in attacking strategies calls for automatic ways to recognize the attacks. They use RL techniques to search for the best way to answer attackers' commands.

All these efforts share the RL-based approach with our RL-IoT framework. We however target the interoperability scenario, where we want to learn how to interact with IoT devices that may be poorly documented.

III. Methodology

A. Reinforcement learning algorithms

In reinforcement learning, learning is achieved by interacting with the environment and it is based on rewards and punishments [5]. Formally, an agent is in a state $s \in S$ defined in function of the environment. The agent may change state following an action $a \in A$ taken at discrete time steps. At time t, the agent decides which action a_t to take given its current state s_t and, as a consequence, it moves to s_{t+1}. The action then causes a change to the system state and the agent possibly receives a reward r_{t+1}.

[2]https://github.com/SmartData-Polito/RL-IoT

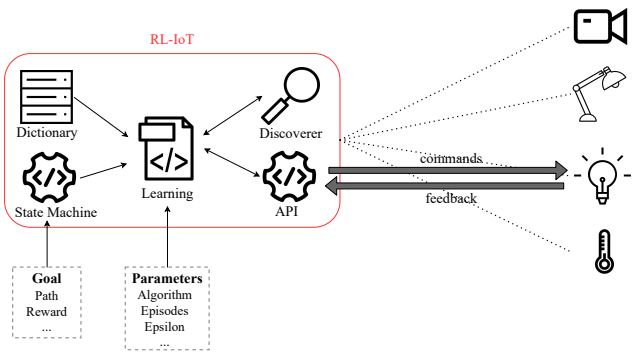

Fig. 1: RL-IoT framework overview.

Considering the above setup, a policy π determines the action a to be taken by the agent when in a particular state s. The task of an RL algorithm is thus to determine a policy that maximises a function of the received reward. Here we consider well-established algorithms that operate based on a value function $V(s)$, which represents the expected accumulated reward when starting from a particular state s and following a policy π. We include algorithms belonging to two categories:

- *Temporal-Difference (TD) learning*: The agent updates $V(s)$ after every time step as:

$$V(s_t) \leftarrow V(s_t) + \alpha[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)] \quad (1)$$

  The parameter α is the learning rate and γ is a discount factor that weights the importance of the destination state $V(s_{t+1})$.
- *TD(λ) learning*: The agent takes n time steps before updating $V(s_t)$. As such, TD(λ) algorithms must memorize visited states to update them later. The parameter λ controls how the n future states influence $V(s_t)$.

SARSA and Q-learning are popular TD algorithms [5], while SARSA(λ) and Q(λ) are the most common TD(λ) algorithms. Both TD and TD(λ) algorithms need a strategy to select the current policy. The most used strategy is called ϵ-greedy, with a trade-off between exploitation and exploration. Exploration is obtained by randomly selecting actions, with a probability ϵ.

B. IoT reinforcement learning framework

Figure 1 summarizes the core RL-IoT framework. It receives as input a *Goal* that the RL Module should learn how to achieve. The *Goal* represents a sequence of settings the device should follow, i.e., paths on the device state-machine. This goal is device-specific, and we will detail it when discussing our case study with the Yeelight smart bulb.

RL-IoT leverages an internal *Message Dictionary* containing a list of IoT protocol messages that can be used to interact with devices. This dictionary can be built from protocol specifications, via automatic reverse engineering solutions or by traffic sniffing. It can contain a mix of messages from different IoT protocols, vendors, versions, etc.

RL-IoT employs state-of-the-art RL algorithms, where the *Learning* module builds and updates the internal *State Machine*. The Learning module supports the previously cited RL algorithms – Q-Learning, Q-Learning(λ) SARSA and SARSA(λ) [5], each with its parameters. It explores which of the several messages in the Dictionary can be used to change the state of the IoT device towards the given Goal. RL algorithms exploit a reward function (custom to each path) to evaluate the benefits of each action taken by the learner in a given state.

The Learning module interacts with two other modules. Firstly, the *Discoverer* module is responsible for scanning the local network in the search for IoT devices. It employs classic scanning approaches (e.g., `nmap`[3]) for searching online devices and performing an initial fingerprint to determine open ports. At last, the *Socket API* module abstracts all the mechanisms to communicate with the target IoT device. Beside sending commands, it may also support the reception of feedback obtained directly from the IoT device, if available. For instance, it can support parsing messages that return the device state.

C. Environment definition

In general, the state of a device can be represented as the powerset of all the current properties of the device, which describes its behaviour and settings – e.g., whether it is on/off and the combination of all the values of its configurable parameters. We define the state-machine of a protocol as a graph containing nodes for states and edges for commands that let the device move from one state to another. A collection of ordered states linked by commands is a *path*. Commands stored in the Message Dictionary could change the IoT device settings, i.e., the current state. With states and commands we can define a state-action value function for the RL algorithms, described by the value-function matrix Q.

The reward associated with the state-machine and the desired path can be provided to RL-IoT as input, and it is used by the RL agent at each time step. RL-IoT runs this procedure many times, i.e., for many episodes. An episode ends when the RL agent reaches the terminal state(s), or after a maximum number of iterations. During each step in an episode, RL-IoT accumulates reward. With such reward, the RL agent updates the state-action matrix Q according to Equation 1, and uses it to select which next command to send, trying to maximize the total reward.

D. Case study: The Yeelight bulb

We use a Yeelight smart bulb as a case study to demonstrate the feasibility of our approach.[4] We select this device because Yeelight provides a generic protocol documentation valid for all their IoT devices.[5] Knowing the protocol allows us to understand and validate what RL-IoT can learn. The protocol offers 37 commands, and only about half of them work with the selected smart bulb, with multiple commands that could generate the same action. For instance, one could set a color via a set_rgb, set_scene, or $adjust\_prop$ message.

Yeelight devices connect to the network using Wi-Fi. After the initial setup, the device periodically broadcasts its presence using *advertisement* UDP messages. It is thus easy for the Discoverer module to find bulbs in the LAN. Once RL-IoT identifies the device IP address, it starts interacting with it sending messages from the Dictionary. Yeelight offers control protocols running on top of both HTTP and raw TCP sockets. The latter relies on *JSON* messages that carry commands.

The commands can have some parameters to set. While these parameters usually belong to finite sets, for some commands the number of admissible values can be huge (like for integer or string parameters). Indeed the combinations of commands and their parameters result into more than 10^9 distinct combinations that the RL agent could send to a Yeelight device. For our case study we simplify the definition of our environment according to our goal. To reduce the action space, we consider the action as only one command, with its parameters that we randomly choose in valid ranges.

Using these protocol specifications, we extract commands and parameters and use them to build our Dictionary, which can be found in our repository.

E. Case study: Definition of goals

For testing RL-IoT, we build and study two scenarios with different state-machines of increasing complexity.

In the first scenario, given a switched-on bulb, our Goal 1 is to learn how to change the color and the brightness of the bulb, in whatever order. In Figure 2 we report the state-machine for this first scenario. Each state considers different attribute values: power p, color c and brightness b. Hence the state is defined by the values of the tuple $\{p, c, b\}$. We disregard the other attributes of the light configuration. Here, we have two final states, where an episode will successfully end: either we reach our goal ($\{p_0 = \text{on}, c_1 \neq c_0, b_1 \neq b_0\}$) or we fail, i.e., we turn off the bulb too early without setting the color and/or the brightness ($\{p_1 = \text{off}, c_*, b_*\}$).

We perform a transition from one state to another inside the state-machine when a command modifies one or more of these attributes. With this strategy, we are able to condense multiple settings into a single state. In Figure 2 we draw possible transitions (arrows) only if a command exists in the protocol to change such property. The actions (commands and their parameters) are not specified in the picture since there might be multiple commands that could produce the same transition. Similarly, there exist a lot of commands that do not change the state, represented as self-transition states (with a looped arrow). Note that it is even possible to get back to a previous state (e.g., setting back the original color c_0).

There are two optimal paths highlighted with green arrows, i.e., the shortest sequences of state changes we want to learn. The optimal policy for Goal 1 visits 3 states with 2 actions, i.e., requiring 2 time steps. Here, we assign the rewards as

[3]https://nmap.org/

[4]For all experiments, we use Yeelight LED Smart Bulb 1S Color (8.5W-E27-YLDP13YL) devices.

[5]https://www.yeelight.com/download/Yeelight_Inter-Operation_Spec.pdf

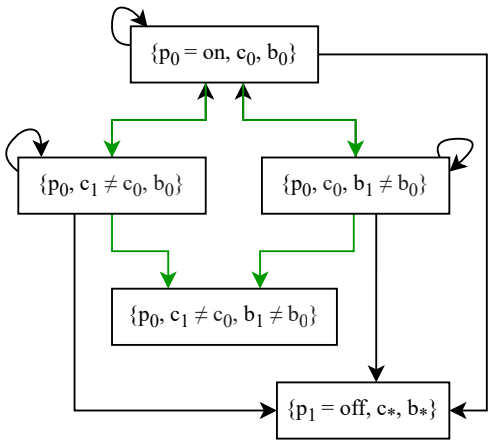

Fig. 2: Goal 1. Simple state-machine where we want to learn how to change the color and the brightness of the bulb, in whatever order. The "*" refers to whatever value.

follows: (i) each new issued command has a small additional negative reward (-1), since we want to reach the goal in as few steps as possible; (ii) we give higher negative reward (-10) when the command produces an error and the state does not change; (iii) we assign no reward when we reach the final state without completing the path $\{p_1 = \text{off}, c_*, b_*\}$; (iv) we give large positive reward ($+205$) when we reach the desired final state $\{p_0, c_1 \neq c_0, b_1 \neq b_0\}$. Hence, with these assigned rewards, the optimal paths will reach a total reward of 203 (i.e., 205 minus 2 steps).

With similar considerations, we draw and implement also another state-machine that we call Goal 2, not shown for brevity, but available in our technical report [12]. The specific goal we want to learn is, in this specific order: (i) turn the bulb on, (ii) change the device name, (iii) change brightness, and (iv) turn the bulb off. Here our goal is more complex since we want to learn how to move through a specific sequence of states. Since we add the $name$ attribute among those we want to change, the state definition becomes {*power, color, brightness, name*}. We also require the bulb color to remain constant, and thus the color is still considered as part of the state definition. We assign a large positive reward (+222) at the final state if we pass through the desired states in the right order. If we arrive to the same final state, but in a different sequence of the same intermediate states, we assign a positive, but smaller reward (+200). Negative rewards are similar to Goal 1. Here the optimal path is unique, with an optimal length of 4 time steps, generating the maximum total reward of 218 (i.e., 222 minus 4 steps).

F. Performance metrics

We consider three metrics for evaluating results and comparing the performance of the algorithms.

We assume that the sets of states S and actions A are finite sets. If not, there exist methods which combine standard RL algorithms with function approximation techniques, such as neural networks [13], [14]. Having finite sets the Q value function $Q(s, a)$ can be represented as a matrix. In our scenarios, a terminal state always exists. We call this $T(E)$, i.e., the number of time steps used in a single episode E to reach the terminal state. We force $T(E) < T_{\max}$, $T_{\max} = 100$. We compute the total reward $R(E)$ obtained during episode E:

$$R(E) = \sum_{t=1}^{T(E)} r_t(E) \text{ for } E \in \{1, ..., N_E\},$$

being N_E the total number of episodes we let RL-IoT run.

These metrics can be averaged over multiple executions - which we call *runs* - of the learning process. Similarly, we compute the moving average for a specified window size w. Average and moving average help to appreciate the learning curve which is affected by the randomness present in each run due to exploration.

Finally, we compute the cumulative reward $C(n_a)$ from the beginning of the learning process over the number of actions performed n_a:

$$C(n_a) = \sum_{E=1}^{E_{n_a}} \sum_{t=1}^{T_{n_a}(E)} r_t(E) \text{ for } n_a \in \{1, ..., N_a\}$$

This metric takes into account not only the reward reached within an episode E, but also how much reward cumulatively was obtained until that episode. To compare different algorithms, we compute the average among different runs, as for $T(E)$ and $R(E)$. Here, the difference is that we "consume" the same number of actions n_a after a different number of episodes E_{n_a} in different runs.

IV. Results

In this section we summarize the results. For all experiments, RL-IoT runs on a x86-64 PC with 4GB of RAM and two cores, connected to the same Wi-Fi network as the Yeelight bulb. In operation settings of the system, RL-IoT can also run on edge devices with similar or limited computing capabilities.

A. Learning capability

We start focusing on whether RL-IoT can learn how to reach the desired goals. We apply the Q-learning algorithm while observing the reward evolution over episodes, and the number of time steps needed to arrive to the target state at each episode. In order to provide an intuition of how RL-IoT interacts with the smart bulb while exploring possible commands, we share a video of one run at https://tinyurl.com/yws6m7ec.

Figure 3 reports the total reward $R(E)$ (left plot) and the number of time steps $T(E)$ (right plot) of each learning episode for Goal 1. Dotted gray line details a single Q-learning run; solid black line reports the average of 10 runs; red line shows the moving average over the 10-run, taking into account a window w of 10 episodes.

Q-learning initially cannot reach the desired state. Missing the large positive rewards, it accumulates a negative reward on average. After few episodes, $R(E)$ grows to the maximum

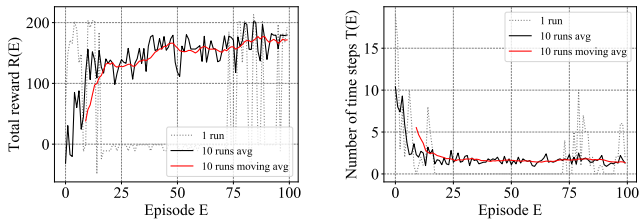

Fig. 3: Q-learning for Goal 1. $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$.

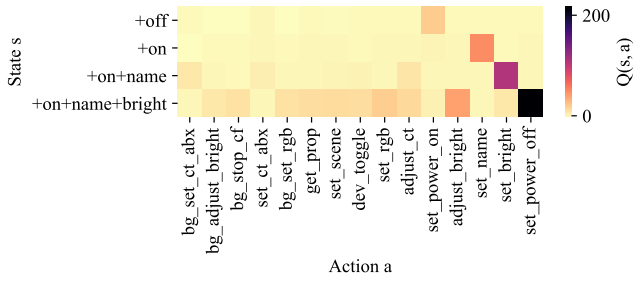

Fig. 4: Part of the final action-value Q matrix for Goal 2. Darker colors show commands (columns) that result in higher expected rewards for the states (rows). $\epsilon = 0.2$, $\alpha = 0.1$, $\gamma = 0.55$.

value that could be observed (203 here). However, comparing the line for a single run to the average over 10 runs we observe a lot of variability. This can be explained by the random exploration component (controlled by ϵ) in the Q-learning algorithm. This exploration phase may penalise the single episode with low final reward, even if the system has already discovered the target goal before. The right plot in Figure 3 shows that Q-learning finds how to reach the desired state with very few actions. After around 15 training episodes, on average, it finds policies composed by 2 or 3 steps, thus the average reward gets closer to the maximum. Recalling that for the trivial Goal 1 scenario the optimal path is composed by 2 steps, we conclude that Q-learning has already found the best path to the goal after 15–20 training episodes.

The results are qualitatively similar for Goal 2, but with slower learning, given the higher complexity of the goal [12]. However, also in this case the learning phase is still able to discover paths with positive reward after around 20 episodes. Given the large state space to explore, the algorithm is still improving its performance even after 100 episodes.

To give the intuition of the learning process achieved by RL-IoT, we depict in Figure 4 a portion of the Q matrix for Goal 2 obtained after 100 episodes. Rows represent only the states belonging to the desired optimal path in the second Goal. Columns represent the top 15 commands (actions) that achieve the highest values for the Q matrix. The darker is the color, the higher is the chance to select that command in that state. Observing the cells with darker colors, we see that Q-learning has indeed learned the expected sequence of

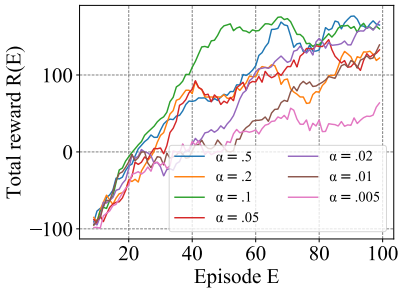

Fig. 5: Tuning α in Q-learning for Goal 2. $\epsilon = 0.2$, $\gamma = 0.95$.

commands to follow the given goal: when off - turn on the lamp, then set the name, the brightness, and at last turn the lamp off. Interestingly, RL-IoT has also identified alternative valid commands to move to the desired state. For example, the algorithm is able to identify several ways to turn the lamp off when in the *+on+name+bright* state – besides the set_power_off command. For instance $adjust\_bright$ to 0, or set_rgb to 0. In other words, RL-IoT discovers multiple ways to perform the same task from its interactions with the environment.

This shows the potential of RL-IoT in supporting the discovery of the semantics of IoT messages. With our use case we can easily verify the actual command semantics. Yet in the general case this could not be easy, e.g., when the protocol uses binary format.

B. Algorithms comparison and training costs

To compare different RL algorithms and parameter impact, we tune parameters to find the best configuration for each algorithm. We only report results on Goal 2, since it is more complex.

Even with reduced commands and states, performing an exhaustive search for all the combinations of algorithm parameters is unfeasible. That is because RL-IoT needs around 40 minutes to execute 100 episodes, due to rate-limits caused by the Yeelight protocol. We thus perform greedy experiments, in which we vary only one parameter at a time to understand its impact on results, starting from values suggested by [5], [15].

We report in Figure 5 the tuning of the learning rate α with Q-learning for Goal 2. The learning rate affects the performance of the RL algorithm, preventing it to reach the maximum reward. In a nutshell, better to learn fast but not too fast. The tuning of the other parameters give similar results, with slightly higher (ϵ) and lower (γ, λ) impact on the total reward. See [12] for details.

After parameter tuning we obtain $\epsilon = 0.2$ and $\alpha = 0.1$. For SARSA and SARSA(λ) we obtain $\gamma = 0.75$, while for Q-learning and Q(λ) we get $\gamma = 0.55$. Finally, $\lambda = 0.9$ results the best for Q(λ), and $\lambda = 0.5$ for SARSA(λ). Notice that in Section IV-A, we already used the tuned parameters here described.

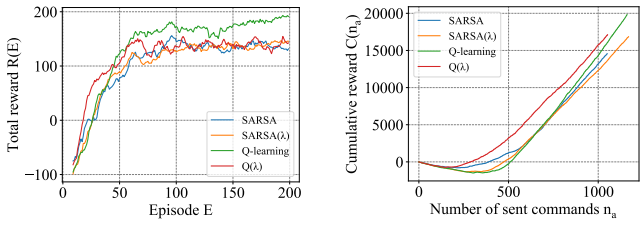

Fig. 6: Algorithm comparison for Goal 2 in terms of reward over episodes (left) and cumulative reward as a function of the number of commands (right) over 200 episodes.

With these values, in Figure 6 we compare the best configurations for the four algorithms, in Goal 2. The left plot shows the moving average ($w = 10$) of the total reward $R(E)$, while the right plot shows the cumulative reward $C(n_a)$ over the number of commands sent to the IoT devices.

Looking at the left plot, we conclude that Q(λ) obtains the highest rewards during the initial 50 episodes. In other words, the algorithm learns faster than others. Yet, from episode 50 onward Q-learning wins, reaching the maximum values at around 200 episodes.

We evaluate the costs of training the RL algorithms in terms of number of commands sent to the IoT devices. Since the Yeelight protocol has a rate limit on requests, we need to pace RL-IoT to avoid passing these limits and triggering the device protections. Therefore, RL-IoT needs to minimize the number of commands to achieve satisfactory learning in real scenarios. The right plot in Figure 6 depicts the cumulative reward $C(n_a)$ obtained by each algorithm as a function of the number of commands n_a sent to the device.[6] We see that all algorithms start with a negative accumulated reward. Some algorithms (e.g., Q-learning) need to send around 400 commands before starting accumulating a positive reward. In line with results shown in the left plot, Q(λ) is the fastest to reach positive reward, needing around 250 commands. Whereas Q-learning is the last one to see positive numbers, its accumulated reward grows faster than others after sending around 600 commands, again confirming results seen in the left plot.

All in all, we conclude that Q(λ) is able to learn solutions leading to positive rewards faster for Goal 2. Standard Q-learning, while requiring more commands than others, is the algorithm able to accumulate more reward. SARSA and SARSA(λ) show figures in between the alternatives.

V. Conclusions

We proposed RL-IoT, a system based on reinforcement learning that learns how to automatically interact with IoT devices. Given a dictionary of possible messages, the system learns which ones to send to the device to achieve a given goal. We showed the effectiveness of RL-IoT in a case study with a Yeelight smart bulb. We were able to learn non-trivial patterns with as few as 400 interactions while also discovering alternative solutions. RL-IoT opens the opportunity to use RL to automatically explore the state machine of unknown protocols, thus assisting on the interoperability of IoT devices.

As future work, we will extend our experiments to make RL-IoT interact with devices of multiple vendors. In this way, we will verify that RL-IoT can learn the different commands to achieve a single goal on multiple devices, hopefully demonstrating interoperability in practical cases.

[6]Different algorithms have a variable maximum number of commands n_a because they might use a different number of commands to reach the end of the episodes.

Acknowledgments

The research leading to these results has been funded by the Huawei R&D Center (France) and the SmartData@PoliTO center for Big Data technologies.

References

[1] P. Sethi and S. Sarangi, "Internet of Things: Architectures, Protocols, and Applications," *Journal of Electrical and Computer Engineering*, vol. 2017, pp. 1–25, 2017.

[2] I. Bermudez, A. Tongaonkar, M. Iliofotou, M. Mellia, and M. M. Munafò, "Towards Automatic Protocol Field Inference," *Comput. Commun.*, vol. 84, no. C, p. 40–51, 2016.

[3] J. Caballero, H. Yin, Z. Liang, and D. Song, "Polyglot: Automatic extraction of protocol message format using dynamic binary analysis," in *Proceedings of the ACM Conference on Computer and Communications Security*, pp. 317–329, 2007.

[4] Z. Lin, X. Jiang, D. Xu, and X. Zhang, "Automatic Protocol Format Reverse Engineering through Context-Aware Monitored Execution," in *15th Symposium on Network and Distributed System Security (NDSS)*, 2008.

[5] R. S. Sutton and A. G. Barto, *Reinforcement Learning: An Introduction*. A Bradford Book, 2018.

[6] S. Kotstein and C. Decker, "Reinforcement learning for IoT interoperability," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*, pp. 11–18, IEEE, 2019.

[7] A. Pauna and I. Bica, "RASSH - Reinforced Adaptive SSH Honeypot," in *10th International Conference on Communications (COMM 2014)*, pp. 1–6, 2014.

[8] A. Pauna, I. Andrei C., and I. Bica, "QRASSH - A self-adaptive SSH Honeypot driven by Q-Learning," in *12th International Conference on Communications (COMM 2018)*, pp. 441–446, 2018.

[9] L. Huang and Q. Zhu, "Adaptive Honeypot Engagement Through Reinforcement Learning of Semi-Markov Decision Processes," *Decision and Game Theory for Security*, pp. 196–216, 2019.

[10] T. Luo, Z. Xu, X. Jin, Y. Jia, and X. Ouyang, "IoTCandyJar: Towards an Intelligent-Interaction Honeypot for IoT Device," *Black Hat*, pp. 1–11, 2017.

[11] T. Gu, A. Abhishek, H. Fu, H. Zhang, D. Basu, and P. Mohapatra, "Towards Learning-automation IoT Attack Detection through Reinforcement Learning," in *2020 IEEE 21st International Symposium on "A World of Wireless, Mobile and Multimedia Networks" (WoWMoM)*, pp. 88–97, 2020.

[12] G. Milan, L. Vassio, I. Drago, and M. Mellia, "RL-IoT: Reinforcement Learning to Interact with IoT Devices." https://arxiv.org/abs/2105.00884. arXiv:2105.0088, 2021.

[13] V. Mnih, K. Kavukcuoglu, D. Silver, A. Rusu, J. Veness, M. Bellemare, A. Graves, M. Riedmiller, A. Fidjeland, G. Ostrovski, S. Petersen, C. Beattie, A. Sadik, I. Antonoglou, H. King, D. Kumaran, D. Wierstra, S. Legg, and D. Hassabis, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, pp. 529–33, 2015.

[14] H. van Hasselt, "Reinforcement Learning in Continuous State and Action Spaces," in *Reinforcement Learning: State-of-the-Art* (M. Wiering and M. van Otterlo, eds.), pp. 207–251, Springer Berlin Heidelberg, 2012.

[15] V. Kumar, "Reinforcement learning: Temporal-Difference, SARSA, Q-Learning & Expected SARSA in python." https://towardsdatascience.com/reinforcement-learning-temporal-difference-sarsa-q-learning-expected-sarsa-on-python-9fecfda7467e, 2019.