

Asguard: Adaptive Self-guarded Honeypot

Sereysethy Touch^a and Jean-Noël Colin^b

InfoSec Research Group, NaDI Research Institute, University of Namur, Rue de Bruxelles 61, 5000 Namur, Belgium

Keywords: Adaptive Honeypot, Multiple-objective Honeypot, Reinforcement Learning, Intelligent Honeypot.

Abstract: Cybersecurity is of critical importance to any organisations on the Internet, with attackers exploiting any security loopholes to attack them. To combat cyber threats, a honeypot, a decoy system, has been an effective tool used since 1991 to deceive and lure attackers to reveal their attacks. However, these tools become increasingly easy to detect, which diminishes their usefulness. Recently, adaptive honeypots, which can change their behaviour in response to attackers, have emerged: despite their promise, however, they still have some shortcomings of their own. In this paper we survey conventional and adaptive honeypots and discuss their limitations. We introduce an approach for adaptive honeypots that uses Q -learning, a reinforcement learning algorithm, to effectively achieve two objectives at the same time: (1) learn to engage with attacker to collect their attack tools and (2) guard against being compromised by combining state environment and action to form a new reward function.

1 INTRODUCTION


As computer systems become more open and more complex, security is still a growing challenge faced by every organisation on the Internet. Considerable effort has been exerted and deception techniques (Cohen, 2006) have been used to collect and analyse security logs to study the attacker's behaviour in hopes of understanding how an attack unfolds, and to build a threat modeling (Schneier, 1999). Profiling attacker's behaviour requires the examination of actions taken by attackers and the sequence in which they were executed (Ramsbrock et al., 2007). One of the techniques that has existed since early 1991 is to use a decoy computer system known as a *honeypot* which poses as a vulnerable machine to capture the attack and also helps to detect an ongoing attack. Since then, several critical problems have arisen that dramatically reduce the effectiveness of existing honeypots.


Firstly, during a probe, their presence can be easily fingerprinted by automated tools or bots due to their recognisable responses and predictable behaviours as pointed out in (Vetterl and Clayton, 2018; Morishita et al., 2019; Surnin et al., 2019). These authors demonstrated that by studying their responses and signatures, it is certain that the system behind an IP address is a honeypot. The detection has been car-

ried out on the most widely adopted protocols such as SSH, Telnet and HTTP. There are also websites¹ that scan the Internet and classify whether an IP address is a honeypot. This problem is very critical, as this undermines the usage of the honeypot and limits adversely how attack data can be collected when attackers know that they are being watched.

Some systems avoid detection by providing the expected response, but this can only fool amateur hackers; experienced ones can still sense that they are interacting with dummy systems by inspecting certain aspects of its environment such as the default file system, short response time, available tools, limited connectivity, unusual number of open ports and running services. Attackers can also use an already controlled system to send out attacks to other computers and check whether any communications have been altered or have failed to reach their target. Conventional honeypots try to defer or substitute outgoing messages to contain them or reduce their effectiveness.

Given these limitations, honeypots deserve more attention and need more improvement in order to address those shortcomings. For that purpose, we propose a smart honeypot called Asguard that balances the goals of maintaining the connection with the attacker alive and collecting as much information about the attack as possible, while avoiding being compromised. Our contribution is two-fold:

^a  <https://orcid.org/0000-0003-4657-7131>

^b  <https://orcid.org/0000-0003-4754-7671>

¹<https://honeyscore.shodan.io/>, www.zoomeye.org

- A new adaptive self-guarded honeypot using SSH protocol, that leverages a reinforcement learning algorithm by combining state and action to form a new reward function. This allows for a trade-off between two contradictory objectives: (1) capture attacker's tools and (2) guard against malicious attacks.
- A prototype implementation of the proposed approach by using a fake botnet attack simulator using real attack data. The experimental results show that the honeypot can effectively learn the intended behaviour.

The paper is organised as follows: first, we survey the honeypot landscape, highlighting the strengths and weaknesses of the various systems; then, we present our approach and learning model. Section 4 presents the experimental setup, its overall architecture and the results obtained. Section 5 discusses the obtained results. Finally, we provide a conclusion and future work.

2 BACKGROUND AND RELATED WORKS

A honeypot is “a security resource whose value lies in being probed, attacked, or compromised” (Spitzner, 2003). Different honeypots have been designed and created for different purposes. There are works in (Seifert et al., 2006; Mokube and Adams, 2007; Ng et al., 2018; Nawrocki et al., 2016) that have been done extensively to cover different types of honeypots and propose different classifications of these honeypots. One of the classifications that is usually found in the literature is based on their interaction levels. They are *low-interaction honeypot* (LiHP), *high-interaction honeypot* (HiHP) and *medium-interaction honeypot* (MiHP).

LiHP refers to a honeypot which uses an emulator to emulate some known vulnerabilities of a system, or mimics a limited number of functionalities of a service or a system. One of the historical and popular systems is Honeyd, which can emulate virtual hosts and network services by emulating their network stacks and it can redirect network packets to real systems (Provost, 2003). Napenthes is used to collect malware by emulating some vulnerabilities in a service (Baecher et al., 2006). Another well-known system, meant to be the successor of Napenthes is Dionaea, which can emulate many popular protocols: FTP, HTTP, Memcache, MongoDB, MySQL, MQTT, MSSQL, SMB, etc., (Dionaea, 2015). Conpot is an Industrial Control Systems honeypot (ICS) designed

to be easy to deploy, modify and extend. Its aim is to collect intelligence about attacks on the ICS protocols such as Modbus, IPMI, BACnet, etc., (conpot, 2018).

A HiHP consists of using a full-fledged operating system or a real application with known or unknown vulnerabilities. Among those systems, Sebek, one of the HoneyNet projects, is built as a Linux kernel module used to capture system calls and keystrokes. The project is no longer maintained and updated, but its source code is still available on Github repository as an archive (HoneyNet, 2021). Argos relies on QEMU, a virtual machine environment, to execute operating systems as a guest system to detect attacks. It does so by tracking incoming network data (marked as tainted data) and detecting any vulnerabilities by using dynamic taint analysis. For example, it can detect any invalid use of jump instruction that triggers the execution of code supplied by an attacker (Portokalidis et al., 2006).

In contrast to the two extremes, a MiHP provides more functionalities compared to the LiHP, but is still limited compared to HiHP. Examples of such systems are Kippo (Kippo, 2014) and Cowrie (Oosterhof, 2014) its successor. These systems written in Python can emulate SSH and Telnet protocols that allow attackers to login and execute some Linux commands. They can capture shell interactions performed by the attacker and also save all downloaded files. They also provide a fake file system that allows attackers to execute file system related commands. IoT-Pot emulates Telnet protocol which is primarily used by IoT devices. It relies on a module called Frontend Responder that acts as different IoT devices and it also handles TCP connection requests, banner interactions, authentication, and command interaction based on a device profile (Pa et al., 2015).

These conventional systems are widely used by security researchers and practitioners to collect various attack intelligence and attack statistics. The advantage of the LiHP and MiHP is that their developments are simple, and they are easy to deploy and maintain, but their main disadvantage is that they can be easily identified by attackers due to their simplicity, limited functionalities and predictable behaviour. The HiHP, however, is useful to collect good quality attack data, but generally they are very complex in their design and implementation; for this reason, they are more difficult to develop, maintain and deploy. Another main drawback is that they are prone to be compromised by attackers if they are not properly monitored. Another common problem that is seen in these conventional honeypots is that they do not evolve despite the constant changes of attack landscape.

2.1 Adaptive Honeypots

Facing these constantly evolving attacks, we have seen another new class of honeypots called *adaptive honeypot* which was first introduced in 2011 by (Wagener et al., 2011b). These systems can adaptively modify their behaviour with respect to different attack patterns. To engage with attackers, adaptive honeypots use different machine learning techniques notably a reinforcement learning (RL) (Sutton and Barto, 2018) to learn to interact with their environment (attackers) to achieve their learning objectives.

Heliza uses SSH protocol as an entry point to let ill-intent people attack a remotely vulnerable Linux system (Wagener et al., 2011a). The Linux system was modified so that the attacker's commands can be intercepted and acted upon. It uses four actions: *allow*, *block*, *substitute* and *insult* to control Linux commands executions; for instance, when an attacker submits a command, it can decide whether the command should be allowed to execute or blocked. Command execution can be substituted using pre-determined results. Heliza can also insult the attacker in the event that attacker submits some random commands. To drive this decision making process, it uses SARSA, an RL algorithm (Sutton and Barto, 2018) to learn to engage with attackers and achieve its goal. Heliza can be configured to achieve either (1) to collect attack tools, or (2) to keep the attacker busy. Heliza shows that to collect attack tools, the command `wget`, `sudo` and downloaded custom tools should be allowed, and to keep the attacker busy, the commands such as `wget` and `tar` should be substituted with an error message, while the command `sudo` should be blocked instead.

Other improvements over Heliza are RASSH (Pauna and Bica, 2014) and QRASSH (Pauna et al., 2018), but in place of using a real Linux system, they first used Kippo (Kippo, 2014) and later Cowrie (Oosterhof, 2014) to emulate SSH server and Linux shell. These systems add a new action *delay* to slow down command execution. They respectively use SARSA (Sutton and Barto, 2018) and DQN (Mnih et al., 2013) to decide on actions. The same approach is applied but with a reduced action set to conceal the honeypot from automated tools which are indifferent to insults (Dowling et al., 2018).

IoTcandyjar is a honeypot of IoT devices. It builds the intelligent system IoT-Oracle which learns to map the attackers query and responses obtained by scanning IoT devices on the Internet. To choose the best response to a query, it uses two strategies: firstly, a random one is used to build the initial knowledge of the response selection, and then, it uses Q -learning al-

gorithm (Watkins and Dayan, 1992) to learn to choose among them the best response (Luo et al., 2017).

2.2 Limitations

Despite the promising results of these adaptive systems, they still present some drawbacks.

- Firstly, systems like Heliza are prone to be compromised even though they can use the action *block* to halt the execution of malicious commands. The reason is that it is a HiHP which has a high risk of being compromised, and another reason is that it simply cannot detect that it is being compromised.
- The systems which are based on MiHP such as RASSH, QRASSH and Dowling et al.'s systems can be easily fingerprinted due to their limited number of implemented shell commands. As pointed out by (Surnin et al., 2019) in their detection methods, despite an invalid command input, the system always returns zero as exit status.

3 OUR PROPOSED APPROACH

In this section, we will describe our new honeypot approach that aims at addressing the limitations exposed earlier; it leverages reinforcement learning to learn how to achieve two opposing objectives: engaging with attacker to collect attack data while keeping safe of deep system compromise. To do that, we combine environment's state and action taken by a learning agent in the RL setting to form a reward function.

3.1 Problem Formulation

Our honeypot poses as a vulnerable Linux system that attackers can access through SSH server. The honeypot allows them to authenticate using any usernames and passwords. Once authenticated, they can execute Linux shell commands either by submitting commands to execute without requesting a shell session or by first opening a shell session. But rather than letting the system be freely compromised, we also want it to learn to guard against a deep system compromise, hence the name *adaptive self-guarded honeypot or Asguard*. As in Heliza (Wagener et al., 2011a), our honeypot is a learning agent in the reinforcement learning problem, in which the agent observes its state environment, decides on which action to take and receives a reward signal. We distinguish the learning phase from the operational phase. At the learning phase, the agent uses the received reward signal to

calculate state-action values and the objective of the agent is to find an optimal policy to maximise these values, whereas in the operational phase, the agent selects action from the learned state-action values (Sutton and Barto, 2018).

3.2 Reinforcement Learning

Formally, a reinforcement learning problem is represented by a *Markov Decision Process* (MDP), which is defined by a tuple

$$\langle S, A, P, r \rangle, \quad (1)$$

where S is a set of discrete finite states representing the environment, A is a set of actions that the agent can perform when visiting a state, a probability transition matrix $P(s, a, s')$ of reaching a next state s' from a state s taking an action a , and $r : \langle s, a \rangle \mapsto \mathbb{R}$ is a reward function when the agent is in a state $s \in S$ and taking an action $a \in A$. In RL, we differentiate two groups of learning methods: *model-based* and *model-free* methods. The model-based methods such as dynamic programming and heuristic search require the model of the environment to be completely defined and available to search for an optimal policy, whereas the model-free methods such as Monte Carlo and temporal-difference like SARSA or Q -Learning, rely only on the experiences through interacting with its environment and collecting rewards to find the optimal policy (Sutton and Barto, 2018). In our case, we will use a model-free method because we do not have access to the model of attacks.

3.3 Environment

In our approach, similarly to Heliza (Wagener et al., 2011a), the state of the environment is defined as the attacker's command input. In a typical attack, attackers will input a sequence of these commands, and each command can be mapped to the following set of commands:

- L : a set of shell commands and some other installed programs during the system setup, some example of commands are `cd`, `pwd`, `echo`, `cp`, ...
- D : a set of download commands that are used to download programs from external servers, for instance we have `wget`, `curl`, `ftpget`, etc., Heliza does not separate D from L .
- C a set of custom commands which are the commands that attackers have to download from external servers before they can be executed, all commands of this type will be mapped to an element *custom*, hence $C = \{custom\}$.

- U : a set of other inputs that cannot be mapped to any other above set, it can be an empty string, ENTER keystroke, hence $U = \{unknown\}$.

So the final states of the environment is

$$S = LUDUCUU \quad (2)$$

3.4 Actions

We only select a subset of actions $A = \{allow, block, substitute\}$ from Heliza (Wagener et al., 2011a). That is, *allow* is to execute the command, *block* is to deny its execution, and *substitute* is to fake its execution. The reason is that *block* can make the attacker change their attack behaviour or have recourse to different commands when they are not available on the system, which will result in more command transitions. Another reason is that *block* can protect the honeypot from being compromised by preventing the execution of malicious commands. And *substitute* can also increase command transitions facing new or unknown programs. However, *insult* is not included, because insulting an attacker seems like an obvious way to advertise the honeypot presence to attackers. Furthermore, *delay* from RASSH (Pauna and Bica, 2014) is not included either, since delaying the execution of a simple or known command also raises a suspicion of a honeypot.

3.5 Reward Function

The reward function allows the agent to learn the behaviour in an environment; normally the purpose of the honeypot is to let attackers freely attack the system, to capture the attack intelligence; however, with an additional objective, we also want the honeypot to avoid being deeply compromised and to adaptively learn how to best react to an attack. In this regard, the desired behaviour is **(1) to capture attacker's tools via download commands**, and **(2) to allow the honeypot to guard against the risk of being compromised by preventing the execution of custom commands**, because we assume that attackers will use their downloaded custom commands to compromise the honeypot and use it to attack other systems.

Contrarily to the other systems (Wagener et al., 2011a; Pauna and Bica, 2014; Dowling et al., 2018) which can only have one objective and whose reward functions only depend on the state environment which is the shell command from the attacker. For our case, **not only do we use the state environment, which is as well the shell command from the attacker, but also the action taken by the agent to form a new reward function**. So when an attacker transitions to

a download command D , and the taken action is *allow*, the agent is then rewarded 1, but when the attacker makes a transition to a custom command C and the chosen action is *allow*, it gets a penalty of -1 . Hence, the reward function r_a at a time-step t is given as follows:

$$r_a(s_t, a_t) = \begin{cases} 1 & \text{if } s_t \in D \text{ and } a_t \in \{allow\} \\ -1 & \text{if } s_t \in C \text{ and } a_t \in \{allow\} \\ 0 & \text{otherwise} \end{cases} \quad (3)$$

3.6 Learning Algorithm

To decide which action to take in a particular state, the agent will estimate a state-action value function $q : \langle s, a \rangle \mapsto \mathbb{R}$. To learn this q -function, we use an algorithm based on a temporal-difference called Q -learning (Watkins and Dayan, 1992). This algorithm is an *off-policy* algorithm because it uses experiences derived from a different policy to learn and evaluate its own policy; it is also a *model-free* algorithm because it only relies on its interaction with its environment without knowing the real dynamics of the environment (Sutton and Barto, 2018). The update of q -function of a state s taking an action a , observing a reward r and reaching a state s' by taking an action a' is given as follows

$$q(s, a) = q(s, a) + \alpha [r + \gamma \max_{a'} q(s', a') - q(s, a)] \quad (4)$$

where $\alpha \in [0, 1]$ is a learning rate and $\gamma \in [0, 1]$ is a discount factor. It has been proved that if the agent keeps visiting all state-actions indefinitely, it will converge to an optimal policy. To help the agent learns, we use an ϵ -greedy policy that will balance between *exploration* and *exploitation*. That is, the agent will choose to (1) explore by randomly selecting action for a probability of ϵ , and (2) exploit its learned policy for $1 - \epsilon$ probability. The ϵ should be gradually decayed to allow the agent to favor its learned q -values. The algorithm 1 gives a pseudo code of the Q -learning.

4 EXPERIMENT

In this section, we detail the experimental setup used to validate our approach. The attacker is simulated by a simple botnet simulator using real attack data. The purpose of this experiment is to demonstrate a proof-of-concept of our proposed honeypot that can learn the correct behaviour policy using the learning algorithm in order to reach its two objectives as defined by the reward function in equation 3.

Algorithm 1: Q -Learning algorithm (Sutton and Barto, 2018; Watkins and Dayan, 1992).

```

Initialise  $q(s, a)$  for all states  $s$  and actions  $a$ ;
foreach episode do
  Initialise state  $s$ ;
  repeat
    Choose  $a$  from  $s$  using  $\epsilon$ -greedy policy derived from  $q$ ;
    Take action  $a$ , observe  $r, s'$ ;
     $q(s, a) = q(s, a) + \alpha [r + \gamma \max_{a'} q(s', a') - q(s, a)]$ ;
    Replace  $s$  with  $s'$ ;
  until  $s$  is terminal;

```

4.1 Architecture

The system is built using a proxy as shown in the figure below.

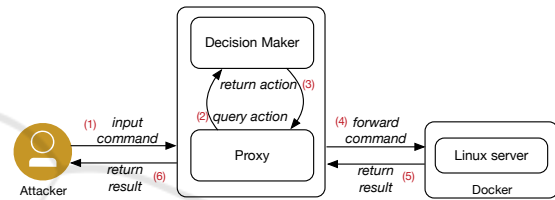


Figure 1: The architecture of proxy based honeypots.

The proxy plays the role of SSH server to the attacker and also as a SSH client to a real Linux server running as a Docker container. The proxy receives the command from the attacker, and uses it to query action from the decision making module which implements the learning algorithm as described in the Algorithm 1. If the action is *allow*, then the command will be forwarded for execution in the Linux container system. After the command is executed, its result is sent back to the proxy which will return it to the attacker. Alternatively, if the action is *block*, the proxy will return a “command not found” message, and finally if the action is *substitute*, the command will not be executed but the proxy will return an existing response for that command.

Even though the proxy has been used in honeypots before in the HTTP protocol to detect web attacks, or to defend web applications by using deceptive techniques (Han et al., 2017; Ishikawa and Sakurai, 2017; Fraunholz et al., 2018; Papalitsas et al., 2018), it has never been used in SSH protocol to intercept its traffic to control it. This proposed architecture thus comes with several advantages over the existing systems. Firstly, the proxy allows an independence between our honeypot and the platform that we want to use as a honeypot. It also solves some problems seen in the HiHP, as it does not require the modification of the platform, and avoids the need of using em-

ulators in LiHP and MiHP. Running systems or services in a sandbox and a controlled environment such as Docker can contain the risk of malicious attacks. Another advantage is that the proposed architecture can be easily applied to other protocols.

4.2 Implementation

The system is built by adding the proxy module with the decision making module to the standard Cowrie honeypot (Oosterhof, 2014). In fact, we can use any programming languages/frameworks² to implement this proxy, but for convenience reasons, we used Cowrie as it already has a usable authentication service for SSH server using the Python Twisted Conch³. The decision making module is implemented in Python3 and Numpy.

4.3 Data Collection

The data used in the experiment were *sporadically collected* using the standard Cowrie (Oosterhof, 2014) that emulates an OpenSSH server. Cowrie was setup to run in a Docker container on a host running Linux Debian 9, and listened on port 22. We modified its default configuration such as *hostname*, *OpenSSH version*, and *kernel information* to match that of the host system, to avoid detection. The system was deployed on the public part of the university network for the period between 12/2018 and 01/2021. In total, we logged approximately 81 millions of raw records, which contain the information related to the attacks such as network connection, session number, key exchanges algorithms, SSH client version, connection time, username, password, input commands, connection duration, downloaded files, etc. We then group the records of the same session number into a single collection called *episode* by combining all the commands entered during the attack. We have 9 millions episodes but only 217K episodes were related to Linux commands, others were ports forwarding traffics. The 217K episodes were randomly split respectively into a training (70%) and an evaluation (30%) datasets using the *train_test_split* helper function from Scikit-Learn.

The figure 2 shows an example of an attack episode in which the attacker was connected to our SSH server using `libssh2.1.8.0` as SSH client library. Once the connection was established, they authenticated as the user `root` with the password `!Q@W#E`. During the connection, the attacker input a list of commands in which they wanted to iden-

²<https://www.paramiko.org>

³<https://twistedmatrix.com/>

```
{
  "sensor": "redacted",
  "session": "302422f0962b",
  "cowrie_session_connect": {
    "dst_port": 22,
    "src_port": 33550,
    "protocol": "ssh",
    "src_ip": "redacted",
    "dst_ip": "redacted",
    "time": { "$date": "2019-03-22T15:57:04.703Z" }
  },
  "cowrie_client_version": "b'SSH-2.0-libssh2_1.8.0'",
  "cowrie_login_success": {
    "username": "root",
    "password": "!Q@W#E"
  },
  "cowrie_command_input": ["uname -a ; unset HISTORY
  HISTFILE HISTSAVE HISTZONE HISTORY HISTLOG WATCH ;
  history -n ; export HISTFILE=/dev/null ; export
  HISTSIZE=0 ; export HISTFILESIZE=0 ;
  killall -9 perl ; cd /tmp ; wget -q redacted/
  wp-admin/images/yc || curl -s -O -f redacted/
  wp-admin/images/yc ; perl yc ; rm -rf yc* ; "],
  "duration": 2.0553934574127197
  ...
}
```

Figure 2: An example of an attack episode as a JSON object shows the SSH connection information, username and password, commands input, attack duration, etc.

tify the system by executing the command `uname -a`, then to make sure that no execution trace was kept, they unset the shell environment variables which control commands history. After that, the command `killall` is used to terminate all the processes `perl` before a file named `yc` was downloaded from a compromised web server of a Wordpress application by using the command `wget` or `curl`. The file was then executed by using `perl` before it was deleted by the command `rm`.

The below table shows the number of total episodes, the total numbers of commands and the total of unique episodes.

Table 1: The number of total (Tot.) episodes (Eps.), total commands (Cmd.) and total unique episodes in the datasets.

Dataset	Tot. Eps.	Tot. Cmd.	Tot. unique Eps.
Training	145,973	2,134,143	56,760
Evaluation	71,898	1,045,914	28,231

Tab. 2 displays the command lengths in each dataset in which some have zero command due to the syntax error in the command input. The average number of commands is 14.62 which consists in general of the commands that try to fingerprint the honeypots,

followed by the commands to download files from the Internet, before they were executed and then removed from the server. Tab. 3 shows the information related to different attack duration. By examining the data, many attackers were connected to our system and only executed a single command to identify the system by executing the command `uname`, check the uptime or run the command `echo`, and then were disconnected. There are some sessions which took very long time to finish because there are some commands that require the user interaction and Cowrie did not implement those commands properly, as a result the connection was kept open for a long period of time.

Table 2: The minimum (Min.), maximum (Max.) and average (Avg.) of number of commands (#cmd.) in the datasets.

Dataset	Min. #cmd	Max. #cmd	Avg. #cmd.
Training	0	89	14.62
Evaluation	0	89	14.54

Table 3: The minimum (Min.), the maximum (Max.) and the average (Avg.) of attack duration in seconds in the datasets.

Dataset	Min. duration	Max. duration	Avg. duration
Training	0.11	41619.46	66.03
Evaluation	0.12	184279.77	61.44

The figure below shows the distribution of command execution patterns that we manually identified in the training dataset and it is shown in the log-scale. The same distribution is also observed in the evaluation dataset.

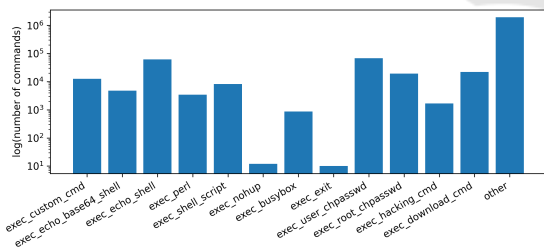


Figure 3: The distribution of command execution patterns in the training dataset.

4.4 Fake Botnet Attack Simulator

To train and evaluate our honeypot using the data described in Section 4.3, we built a simulator that simulates a simple fake botnet attack behaviour which is similar to the one from (Dowling et al., 2018). For each episode, the simulator iterates over commands and changes its attack behaviour with respect to the action taken by the honeypot on each command sub-

mitted to the honeypot. More precisely, when the action is *allow* or *substitute*, it will continue to submit the next command, if any. Only when two consecutive actions are *block*, it will then terminate the episode and move to the next one. This behavior can be modified to make it more complex over time, for example, we can have a conditional execution between command executions. Currently, this fake botnet ignores the result of the command execution, and just continues to the next command. The pseudo code of this simple attack behavior is given below

Algorithm 2: Simple Fake Botnet Attack Behaviour.

```

foreach episode do
    state = "continue";
    foreach command do
        Submit the command to the honeypot;
        Get action for the command, taken by the honeypot;
        if action == "allow" or
           action == "substitute" then
            state = "continue";
        if action == "block" then
            if state == "continue" then
                state = "block";
            else
                if state == "block" then
                    state = "terminate";
        if state == "terminate" then
            terminate the episode;

```

4.5 Results

We ran the same experiment 10 times with these hyper-parameters: $\epsilon = 0.5$ and it is decayed by 0.99991 for each new episode until it reaches the minimum value of 0.1, $\alpha = 0.01$, and $\gamma = 0.99$. However we can only show one instance of the experiment result, because the obtained results could not be averaged to a single q -value of each command-action pair. The reason is that the value of q -value depends on the actions randomly selected during the training and as mentioned above, the fake botnet ignores the result of the command execution, though the result of the commands of interest determining the desired behaviour appeared consistently the same for all the experiments. Tab. 4 recapitulates an instance of final q -values of some commands, in which the chosen action for each command corresponds to its highest q -value highlighted in bold. The result clearly indicates that the action *allow* is chosen for the download command `wget` which follows the first objective of getting attack's tools, but the action *block* is selected instead for the command `custom`, which does match

the second objective of protecting the honeypot. For the other commands, the selected actions are less significant due to the randomness of the action selection.

Table 4: Final q -values of some commands of Asguard.

Command	<i>allow</i>	<i>block</i>	<i>substitute</i>
tar	0.1015	0.0042	0.0064
sudo	.0	.0	0.0076
chmod	0.1824	0.1398	0.1482
uname	0.0756	0.0763	0.0846
unknown	0.0992	0.2720	0.1536
custom	-0.9250	0.0594	0.0457
ps	0.0896	0.0154	0.0206
wget	1.7083	0.4956	0.5095
bash	0.1092	0.1080	0.1212

4.6 Evaluation

In this section, we compare the performance of Asguard with a newly developed system called Midgard. Since we cannot compare the proposed solution side-by-side with the other adaptive systems, because to the best of our knowledge, this is the first time that such system can learn to achieve two objectives (Eq. 3), whereas the previous systems can only learn to achieve a single objective. Midgard shares the same objectives, architecture and implementation as Asguard, except that **its reward function only depends on the state environment** like in the existing adaptive systems (Wagener et al., 2011b; Pauna and Bica, 2014; Dowling et al., 2018). That is, when an attacker transitions to a download command D , the agent is rewarded 1 regardless of any actions taken, and when an attacker transitions to a custom command C , it is given -1 for any actions taken. Thus, its reward function at a time-step t is given below

$$r_m(s_t, a_t) = \begin{cases} 1 & \text{if } s_t \in D \text{ and } \forall a_t \in A \\ -1 & \text{if } s_t \in C \text{ and } \forall a_t \in A \\ 0 & \text{otherwise} \end{cases} \quad (5)$$

We conducted the same experiment (Section 4) 10 times for Midgard, and use its result as a baseline to compare it with the Asguard’s result. To evaluate their performances, we can consider our problem as a classification problem in Machine Learning. That is, the command `wget` is allowed, and the command `custom` is either blocked or substituted, they are considered as *True Positive*. If the command `wget` is blocked or substituted, and the command `custom` is allowed, they are considered as *False Positive*. We calculate **average precision** and **recall** over the 10 trained agents by using the same fake botnet simulator on our evaluation dataset on two settings: *exploration* and *no-exploration*. The former means that the agent still

randomly chooses some actions for a small constant probability of $\epsilon = 0.1$, and the latter means that the agent fully exploits its learned policy.

Tab. 5 displays the average precision and recall for the two commands `custom` and `wget` for both Asguard and Midgard. The results show that in the two evaluation settings, Asguard reaches the precision of more than 93% in getting attacker’s tools, and 96% in protecting itself, while Midgard only reaches approximately 68% for the two objectives. The recall results also indicate that Asguard can recover more than 91% of the available commands, compared to less than 50% for Midgard.

Table 5: The average (Avg.) precision and recall of the commands `wget` and `custom` of Asguard and Midgard.

	Exploration		No-Exploration	
	<i>Asguard</i>	<i>Midgard</i>	<i>Asguard</i>	<i>Midgard</i>
	wget			
Avg. Precision	0.9353	0.4832	1	0.5
Avg. Recall	0.9180	0.4632	0.9874	0.4828
	custom			
Avg. Precision	0.9676	0.7866	1	0.8
Avg. Recall	0.9265	0.4321	0.9695	0.4119

The Fig. 4 illustrates an instance of the learning curves of q -value of the two systems, in contrast to Asguard which quickly converges to the correct q -values, Midgard struggles to learn the correct values for the two commands. Tab. 6 shows the average percentage of the total commands across the experiments, that are recovered during the evaluation. Again, Asguard still performed better in recovering the total commands compared to Midgard.

5 DISCUSSION

The experimental results show that what Asguard learned matches the two learning objectives: allowing download commands to execute will result in getting attacker’s tools, while blocking or substituting their executions will protect the honeypot from being fully compromised. However, systematically preventing the execution of all attacker’s tools also restricts our ability to understand a more sophisticated attack behaviour, or Advanced Persistent Threat (APT) which can take place in different time periods. Furthermore, preventing the execution of the attacker’s downloaded tools is not the only way that keeps the honeypot safe because there are other methods that the attacker can use to compromise it.

Using a proxy to the real system or service can easily fool attackers into thinking that they are attack-

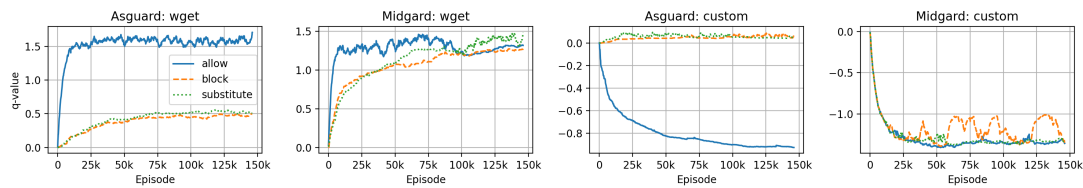


Figure 4: The learning curves of q-values for the commands `wget` and `custom` for Asguard and Midgard.

Table 6: The average (Avg.) percentage (per.) of the total commands (cmd.) recovered for Asguard and Midgard.

	Exploration		No-exploration	
	<i>Asguard</i>	<i>Midgard</i>	<i>Asguard</i>	<i>Midgard</i>
Avg. per. total cmd. recovered	97.64%	82.97%	99.84%	98.65%

ing a real one; as shown in the evaluation result, the agent can successfully “engage” with attackers by recovering more than 97% of the total commands, but this experiment is limited by the fact that we only used the attack simulator that exhibits a simple attack pattern and currently, it completely ignores the return of the command executions. Facing with real human attackers or clever attackers, their behaviours may be different and/or unpredictable, therefore, the agent may take longer to learn the intended behaviour, or it can fail entirely to learn, especially at the early stage of the learning phase, as the agent relies on a random policy to choose the actions and only receives a reward to update its policy when reaching the goal states. For example the attacker can try to execute some basic shell commands like `cd`, `ls` or `echo` which are supposedly available on all Linux systems, but the agent can randomly decide to block them by returning the error message of `command not found` instead, this can be used to fingerprint the system, which will considerably reduce the command transitions, hence, the agent will never have the chance to interact longer with attackers to learn their behaviour.

Another difficulty arisen from using the proxy to intercept the command input to make decision is that the attacker can hide all their commands in a shell script and then execute it; in this case, the proxy may not be able to intercept these hidden commands, the only thing that it can catch is the script file. To remedy this problem, the proxy will have to go one step further by detecting the presence of the shell script and analysing its content, which could require additional development. But this is a trade-off between using the proxy to intercept the attacker input to deceive attackers to some extent, and diving deeper into a kernel space to intercept system calls to make decision as in the case of Heliza (Wagener et al., 2011a).

6 CONCLUSION AND FUTURE WORK

In this paper, we proposed a new adaptive honeypot for the SSH protocol, that can find a trade-off between two learning opposing objectives: engage with attacker to collect information on attacks, and also guard against the risk of being compromised seen in high-interaction honeypots, as a result of the new reward function that combines the state environment and the action of the agent. This combination allows us to define more complex learning objectives that help the agent to learn faster, as opposed to the reward function that only depends on the state. We also proposed an implementation of our proposed approach using proxy that makes it possible to separate between our honeypot and the platform that we want to use as a honeypot. This can avoid the need of modifying the platform, a major problem of high-interaction honeypots, and using emulators, another limitation of low- and medium-interaction honeypots.

As our next steps, we will deploy this system in real environment and evaluate its performance and quality in terms of data being collected. Another future direction that we want to investigate is to consider a more complex attack behaviour and a complex state observation, that will see the state as a combination of command and its arguments, and the honeypot properties such as resources (CPU, memory) consumption, network connections. . . We will also extend the notion of risk of being compromised by defining it as different risk levels that allow the agent to adapt accordingly, which was discussed in the above section.

REFERENCES

- Baecher, P., Koetter, M., Holz, T., Dornseif, M., and Freiling, F. (2006). The nepenthes platform: An efficient approach to collect malware. In *International Workshop on Recent Advances in Intrusion Detection*, pages 165–184. Springer.
- Cohen, F. (2006). The use of deception techniques: Honeypots and decoys. *Handbook of Information Security*, 3(1):646–655.

- conpot (2018). conpot. <https://github.com/mushorg/conpot>. [Online; accessed 13-July-2021].
- Dionaea (2015). dionaea. <https://github.com/DinoTools/dionaea>. [Online; accessed 13-July-2021].
- Dowling, S., Schukat, M., and Barrett, E. (2018). Improving adaptive honeypot functionality with efficient reinforcement learning parameters for automated malware. *Journal of Cyber Security Technology*, 2(2):75–91.
- Fraunholz, D., Reti, D., Duque Anton, S., and Schotten, H. D. (2018). Cloxy: A context-aware deception-as-a-service reverse proxy for web services. In *Proceedings of the 5th ACM Workshop on Moving Target Defense*, pages 40–47.
- Han, X., Kheir, N., and Balzarotti, D. (2017). Evaluation of deception-based web attacks detection. In *Proceedings of the 2017 Workshop on Moving Target Defense*, pages 65–73.
- HoneyNet (2021). Sebek: kernel module based data capture. [Online; accessed 13-July-2021].
- Ishikawa, T. and Sakurai, K. (2017). Parameter manipulation attack prevention and detection by using web application deception proxy. In *Proceedings of the 11th International Conference on Ubiquitous Information Management and Communication*, pages 1–9.
- Kippo (2014). Kippo. <https://github.com/desaster/kippo>. [Online; accessed 13-July-2021].
- Luo, T., Xu, Z., Jin, X., Jia, Y., and Ouyang, X. (2017). Iot-candyjar: Towards an intelligent-interaction honeypot for iot devices. *Black Hat*.
- Mnih, V., Kavukcuoglu, K., Silver, D., Graves, A., Antonoglou, I., Wierstra, D., and Riedmiller, M. (2013). Playing atari with deep reinforcement learning. *arXiv preprint arXiv:1312.5602*.
- Mokube, I. and Adams, M. (2007). Honeypots: concepts, approaches, and challenges. In *Proceedings of the 45th annual southeast regional conference*, pages 321–326. ACM.
- Morishita, S., Hoizumi, T., Ueno, W., Tanabe, R., Gañán, C., van Eeten, M. J., Yoshioka, K., and Matsumoto, T. (2019). Detect me if you... oh wait. an internet-wide view of self-revealing honeypots. In *2019 IFIP/IEEE Symposium on Integrated Network and Service Management (IM)*, pages 134–143.
- Nawrocki, M., Wählich, M., Schmidt, T. C., Keil, C., and Schönfelder, J. (2016). A survey on honeypot software and data analysis. *arXiv preprint arXiv:1608.06249*.
- Ng, C. K., Pan, L., and Xiang, Y. (2018). *Honeypot Frameworks and Their Applications: A New Framework*. Springer.
- Oosterhof, M. (2014). Cowrie. [Online; accessed 13-July-2021].
- Pa, Y. M. P., Suzuki, S., Yoshioka, K., Matsumoto, T., Kasama, T., and Rossow, C. (2015). Iotpot: analysing the rise of iot compromises. In *9th {USENIX} Workshop on Offensive Technologies ({WOOT} 15)*.
- Papalitsas, J., Rauti, S., Tammi, J., and Leppänen, V. (2018). A honeypot proxy framework for deceiving attackers with fabricated content. In *Cyber Threat Intelligence*, pages 239–258. Springer.
- Pauna, A. and Bica, I. (2014). Rassh-reinforced adaptive ssh honeypot. In *Communications (COMM), 2014 10th International Conference on*, pages 1–6. IEEE.
- Pauna, A., Iacob, A.-C., and Bica, I. (2018). Qrassha a self-adaptive ssh honeypot driven by q-learning. In *2018 international conference on communications (COMM)*, pages 441–446. IEEE.
- Portokalidis, G., Slowinska, A., and Bos, H. (2006). Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *ACM SIGOPS Operating Systems Review*, 40(4):15–27.
- Provos, N. (2003). Honeyd-a virtual honeypot daemon. In *10th DFN-CERT Workshop, Hamburg, Germany*, volume 2, page 4.
- Ramsbrock, D., Berthier, R., and Cukier, M. (2007). Profiling attacker behavior following ssh compromises. In *Dependable Systems and Networks, 2007. DSN'07. 37th Annual IEEE/IFIP International Conference on*, pages 119–124. IEEE.
- Schneier, B. (1999). Attack trees. *Dr. Dobbs's journal*, 24(12):21–29.
- Seifert, C., Welch, I., and Komisarczuk, P. (2006). Taxonomy of honeypots. Technical report cs-tr-06/12, Victoria University of Wellington, School of Mathematical and Computer Sciences.
- Spitzner, L. (2003). *Honeypots: Tracking Hackers*, volume 1. Spitzner, Lance.
- Surnin, O., Hussain, F., Hussain, R., Ostrovskaya, S., Polovinkin, A., Lee, J., and Fernando, X. (2019). Probabilistic estimation of honeypot detection in internet of things environment. In *2019 International Conference on Computing, Networking and Communications (ICNC)*, pages 191–196. IEEE.
- Sutton, R. S. and Barto, A. G. (2018). *Reinforcement learning: An introduction*. MIT press.
- Vetterl, A. and Clayton, R. (2018). Bitter harvest: Systematically fingerprinting low-and medium-interaction honeypots at internet scale. In *12th {USENIX} Workshop on Offensive Technologies ({WOOT} 18)*.
- Wagener, G., Dulaunoy, A., Engel, T., et al. (2011a). Heliza: talking dirty to the attackers. *Journal in computer virology*, 7(3):221–232.
- Wagener, G., State, R., Engel, T., and Dulaunoy, A. (2011b). Adaptive and self-configurable honeypots. In *12th IFIP/IEEE International Symposium on Integrated Network Management (IM 2011) and Workshops*, pages 345–352. IEEE.
- Watkins, C. J. and Dayan, P. (1992). Q-learning. *Machine learning*, 8(3-4):279–292.