Journal of Cyber Security Technology

ISSN: 2374-2917 (Print) 2374-2925 (Online) Journal homepage: www.tandfonline.com/journals/tsec20

Improving adaptive honeypot functionality with efficient reinforcement learning parameters for automated malware

Seamus Dowling, Michael Schukat & Enda Barrett

To cite this article: Seamus Dowling, Michael Schukat & Enda Barrett (2018) Improving adaptive honeypot functionality with efficient reinforcement learning parameters for automated malware, Journal of Cyber Security Technology, 2:2, 75-91, DOI: 10.1080/23742917.2018.1495375

To link to this article: https://doi.org/10.1080/23742917.2018.1495375

JOURNAL OF CYBER SECURITY TECHNOLOGY
2018, VOL. 2, NO. 2, 75–91
https://doi.org/10.1080/23742917.2018.1495375

Taylor & Francis
Taylor & Francis Group

Improving adaptive honeypot functionality with efficient reinforcement learning parameters for automated malware

Seamus Dowling, Michael Schukat and Enda Barrett

Discipline of IT, College of Engineering & Informatics, NUI Galway, Galway, Ireland

ABSTRACT

This paper presents an intelligent honeypot that uses reinforcement learning to proactively engage with and learn from attacker interactions. It adapts its behaviour for automated malware to optimise the volume of data collected. Malware employs highly automated methods to create a global botnet. These automated methods are used to self-propagate and compromise hosts. Honeypots have been deployed to capture these automated interactions. Machine-learning techniques have previously been employed to retrospectively model botnet interactions. We develop a honeypot that uses reinforcement learning with a specific state action space formalism to interact with automated malware. It compares functionality with similar intelligent honeypots which target human interaction. It also demonstrates that datasets collected from an intelligent honeypot deployment are considerably larger than standard high interaction deployments and existing adaptive honeypots.

ARTICLE HISTORY
Received 1 January 2018
Accepted 24 May 2018

KEYWORDS
Honeypot; reinforcement learning; adaptive; malware; automated

Introduction

Honeypot technologies have been deployed since 1992 [1]. Their purpose is to capture cyber attack information and analyse behaviour. Honeypot design has changed over time as cyber attack methodologies and targets have evolved. For example, with the recent emergence of the Internet of Things (IoT), the opportunity for end device compromise has expanded. To counter this, IoTbot [2] was developed and deployed to target IoT specific attacks. Unlike intrusion detection systems (IDS) and intrusion prevention systems (IPS) honeypots are deployed to capture data for retrospective analysis. They offer no proactive security measures and are only focused on data capture. To this end, honeypots have evolved to respond to attacker commands with the goal of prolonging interactions [3]. This goal results in larger datasets providing better material for behavioural analytics. Machine-learning techniques have been incorporated into honeypot functionality enabling them to learn from attacks. The responses presented to attackers are optimised to prolong interaction. Supervised, unsupervised and REINFORCEMENT learning (RL) techniques have been deployed to retrospectively model behaviour [4] and also to interact proactively with the attacker [5]. During the

CONTACT Seamus Dowling seamus.dowling@gmit.ie Discipline of IT, College of Engineering & Informatics, NUI Galway, Galway, Ireland

infection phase, botnets can propagate and compromise hosts. These automated processes search the web for vulnerable hosts to compromise and then report back to the command and control (C&C). During the next phase the botmaster will instruct the C&C to communicate with the hosts, instructing them to execute further automated commands. Honeypots capture all this automated activity from initial bruteforce attempts, to compromises and subsequent attack instructions. This article describes an RL approach to create an adaptive honeypot suitable for prolonging interactions with automated bots. Previous approaches such as Heliza [3] and RASSH [5] have also used RL to prolong the attack duration. However this is largely for human attackers. Our results show that the recent rise in automated bots renders these techniques inadequate and we propose a novel state action space formalism designed to prolong interactions for automated malware. Thus we propose the following contributions over existing work:

- Whilst the application of RL to this domain is not new, the proposed state action space formalism is new and forms a core contribution of this work.
- We compare the performance of our approach against both RASSH and Heliza and demonstrate empirically how our modified state space improves bot interaction.
- Demonstrates that using an action set specifically for automated malware produces a larger dataset with more command transitions compared to a standard high interaction honeypot.

The work presented here is empirically evaluated on a live deployment which demonstrates the efficacy our approach yielding improved decision-making and increasing command transitions when compared to previous implementations

Previous contributions

Honeypots

Honeypot functionality has improved since their initial deployments. Provos [6] in 2003 presented Honeyd, an easy to deploy, low-risk honeypot. It details how to deploy virtual honeypots with different IPs safely. Honeyd acted as a catalyst for the development of further low interaction honeypots. Faster networking and virtualisation technologies presented honeypot developers with a means of deploying high interaction honeypots and honeynets with low risk, isolating attack traffic from connected hardware and networks. Nepenthes [7] and Argos [8] became very popular global honeypot tools. High interaction honeypots provide backend databases to collect all activity such as IP addresses, timestamps, attempts, interactions, commands, downloads, and executions. Global honeynet projects provided large datasets, on which cumulative analysis and modelling can be performed [9,10]. This analysis provided the means to model attacker interactions [11]. After an attacker has compromised a honeypot, it will attempt to interact in a structured manner. The initial engagement for an attack, post compromise, is to examine the hardware and software to determine if progression is relevant. On a live production system, this will return the underlying architecture, CPU, uptime, operating system, user-privileges and further relevant information. Hardware and software properties are checked, to determine if the compromised host has further potential, or if the host is a virtualised environment [12]. An attacker may then modify the host system,

including passwords. On a honeypot, engaging the attack sequence at this point prolongs activity. An attack then attempts to download, install and run malware to complete the compromise. Emerging technologies often present new threats with reduced security functionality. IoT deployments attract bots and malicious code targeting IoT end devices [2]. These devices are required to collect, collate and transmit information. Their ability to do so depends on the available resources such as CPU, memory and power. Devices that are physically constrained will not be able to implement security that replicates existing mainstream IT infrastructure security measures. This increases the number of devices that can be interconnected and more importantly, be compromised and used in a cyber attack. For malicious code developers, this provides a fertile ground for exploitation and compromise. Intermediate network devices such as routers and Internet facing end devices are accessible through SSH. This makes it a popular attack vector for malware. SSH honeypots produce large datasets for analysis as they capture predominately automated traffic [13]. In a desire to gather as much information as possible on attacker behaviour, a honeypot could allow the execution of malicious code [14]. The honeypot developer could be liable if their honeypot inadvertently becomes involved in further attacks. Entrapment could be a mitigating factor when it comes to the prosecution of an attacker. This highlights the need for creating honeypots that prolong attacker interaction for an optimum time period, without breaching legal or ethical responsibilities.

Reinforcement learning

RL is a machine-learning technique in which a learning agent learns directly from its environment, through trial and error interactions without any prior knowledge. Rather than being instructed as to which action it should take given a specific set of inputs, it instead learns based on previous experiences as to which action it should take in the current circumstance.

Markov decision processes

RL problems can generally be modelled using Markov decision processes (MDPs). In fact, RL methods facilitate solutions to MDPs in the absence of a complete environmental model. This is particularly useful when dealing with real-world problems such as honeypots, as the model can often be unknown or difficult to approximate. MDPs are a particular mathematical framework suited to modelling decision-making under uncertainty. A MDP can typically be represented as a four tuple consisting of states, actions, transition probabilities and rewards.

- S represents the environmental state space;
- A represents the total action space;
- $p(.|s, a)$ defines a probability distribution governing state transitions
- $s_{t+1} \sim p(.|s_t, a_t)$;
- $q(.|s, a)$ defines a probability distribution governing the rewards received
- $R(s_t, a_t) \sim q(.|s_t, a_t)$;

S is the set of all possible states represents the agent's observable world. At the end of each time period, t is the agent occupies state $s_t \in S$. The agent must then choose an action $a_t \in A(s_t)$, where $A(s_t)$ is the set of all possible actions within state s_t. The execution of the chosen action results in a state transition to s_{t+1} and an immediate numerical reward $R(s_t, a_t)$. The following equation represents the reward function, defining the environmental distribution of rewards:

$$R_a\ s,\ s' =\ E\ \{r_{t+1}|s_t = s,\ a_t = a,\ s_{t+1} = s'\} \tag{1}$$

The learning agent's objective is to optimise its expected long-term discounted reward. The state transition probability $p(s_{t+1}|s_t, a_t)$ governs the likelihood that the agent will transition to state s_{t+1} as a result of choosing a_t in s_t:

$$P_a s, s' = P\ r\ \{s_{t+1} = s'|s_t = s,\ a_t = a\} \tag{2}$$

The numerical reward received upon arrival at the next state is governed by a probability distribution $q(s_{t+1}|s_t, a_t)$ and is indicative as to the benefit of choosing a_t whilst in s_t. In the specific case, where a complete environmental model is known, i.e. (S, A, p, q) are fully observable, the problem reduces to a planning problem and can be solved using traditional dynamic programming techniques such as value iteration. However if there is no complete model available, then RL methods have proven efficacy in solving MDPs.

SARSA learning

During the RL process, the agent chooses an action using a policy (π) given an environmental state signal. The goal of the RL agent is to learn the optimal policy (π^*), the optimal mapping from states to actions. Figure 1 models a basic RL interaction process.

We have to consider how the model in Figure 1 applies to adaptive honeypot development. Throughout its deployment, the honeypot is considered to be an environment with RL integrated into its machinations. We are using SSH as an access point and a simulated Linux server as a vulnerable environment. Within this environment, the server has states that are examined and changed with bash scripts. Examples are *iptables, wget, sudo*, etc. The RL agent can perform actions on these states such as to *allow* or *block* the execution of the scripts. The environment issues a reward to the agent for performing that action. The agent learns from this process and over time, learns the optimum policy π^*, denoting the best action to take for a given state. A simple example of an adaptive honeypot process is shown in Figure 2.

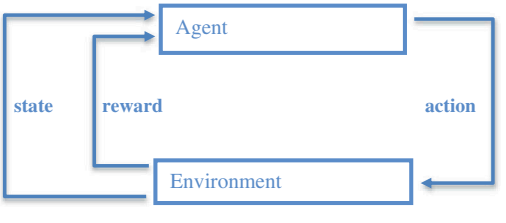

Figure 1. Reinforcement learning model.

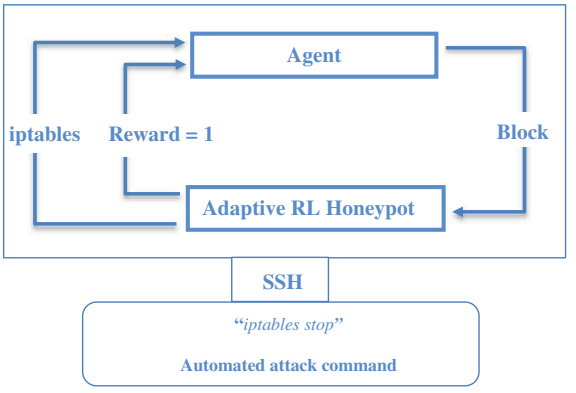

Figure 2. Adaptive honeypot with reinforcement learning process.

The optimum policy π^* is learned over time as the honeypot is continuously attacked. The learning process will eventually converge as the honeypot is rewarded for each attack episode. This temporal difference method for on-policy learning uses the transition from one state/action pair to the next state/action pair, to learn the environmental dynamics. Rewards achieved upon transition are used to determine the value of action *a* whilst in state *s*. State, Action, Reward, State, Action also known as SARSA, is a common implementation of on-policy RL (3). The policy is a combination of the *Q*-values and the action selection strategy. The action selection strategy we use in this work is epsilon greedy, which generally chooses the action which results in the greatest reward save for a small percentage of the time when it chooses a random action. Choosing a non-greedy action is considered an exploratory move and it is hoped that by choosing a suboptimal action some of time, better policies may be found in the long run. The following equation describes the update rule for SARSA:

$$Q(s_t,\ a_t) \leftarrow Q(s_t,\ a_t) + \ a[r_{t+1} + \ \gamma Q(s_{t+1},\ a_{t+1}) - \ Q(s_t,\ a_t)] \tag{3}$$

A learning rate parameter is also applied (α) which controls the amount of new knowledge we update per trial. Both RASSH and Heliza use SARSA for on-policy learning. It is important to note that our implementations also use SARSA. This allows for a direct comparison of performance with previous implementations.

Adaptive honeypots

Machine-learning techniques have previously been applied to honeypots. These techniques have analysed attacker behaviour retrospectively on a captured dataset. Supervised and unsupervised methods are used to model malware interaction and to classify attacks [4,15–17]. This analysis is a valuable source of information for security groups. However, by proactively engaging with an attacker, a honeypot can prolong interaction and capture larger datasets potentially leading to better analysis. To this end, the creation of intelligent, adaptive honeypots has been explored. Wagener [3] uses RL to extract as much information as possible from the intruder, to prolong interaction. A honeypot called Heliza was developed to use RL when engaging an attacker. The honeypot implemented behavioural strategies such as blocking commands, returning error messages and issuing insults. Previously, he had

proposed the use of game theory [18] to define the reactive action of a honeypot towards an attacker's behaviour. A hierarchical probabilistic automaton is presented with the purpose of making a honeypot adaptive and autonomous. Taking Figure 1 as the model, Heliza defines the environment as the interactions between a successful SSH compromise and the attacker exiting the honeypot. The underlying finite *state* space is the list of commands entered during the attack on the environment. The *actions* of the honeypot are as follows: allow, block, substitute and insult. The *reward* function has two objectives. The first is to collect attacker related information, which includes transitions to attacker tools, system tools and insulting commands. The second is to extend the duration of an attacker's interaction expressed as the delay between attacker commands suggesting human cognitive functionality. Wagener presents results for the two reward functions. For the first reward he selects two states of *sudo* and *wget* as state examples and shows the learning values calculated for each action. The second reward compares the cumulative number of transitions captured on the honeypot, with a standard high interaction honeypot. The honeypot was developed using a User Mode Linux (UML) framework. Pauna [5] also presents an adaptive honeypot using the same reinforced learning algorithms and parameters as Heliza. He proposes a honeypot named RASSH that provides improvements to scalability, localisation and learning capabilities. It does this by using newer libraries and a high interaction honeypot called Kippo [19]. RASSH compares its functionality with Heliza demonstrating similar behaviour. Both Heliza and RASSH use PyBrain [20] for their implementation of RL and both use actions that are targeting human interaction. As each attack is considered an episode, the learning policy is evaluated at the end of each of these episodes. This on-policy method of learning is implemented on both honeypots using SARSA (3) with the following parameters:

- ε-greedy policy
- The honeypot environment is unknown to the learning agent. We want it to learn and eventually converge. To this end, we set our explorer to ε-greedy.
- Discount factor $\gamma = 1$
- γ is applied when a future reward is estimated. For our honeypot no discount factor is applied, as attack episodes are readily defined commands between SSH open and close events. This allows for retrospective reward calculations.
- Step size $\alpha = 0.05$
- PyBrain also allows for setting parameters for the creation of the entire state/action space. This is relevant for the creation of honeypots, for the tasks in the next section.

Design and implementation of reinforcement learning honeypot

Our goal is to improve honeypot functionality by increasing attacker interactions. Our methodology modifies previous implementations with RL parameters that are more effective for automated malware. Our reward function is simplified creating a new state action space formalism. This is implemented on our adaptive honeypot and the following three tasks are pursued for this proposal:

(1) Direct comparison with previous research using a reduced action set in a simulated environment.
(2) Deployment of a live honeypot with a reduced action set targeting known automated malware.
(3) Comparison of the number of command transitions on an adaptive honeypot with reduced action set with previous research and a live high interaction honeypot.

Automated malware

Dictionary attacks, botnet propagation and DDoS flood requests are examples of highly automated malware. Botnets can provide a mechanism for global self-propagation of cyber attack infection and control. They are defined as large networks of compromised machines used to carry out further attacks [21]. These botnets are under the control of a single C&C. Previous research has shown that 99.6% of SSH attacks captured on a honeypot are automated [13].

Figure 3 demonstrates a diurnal pattern of attack traffic on a honeypot dataset. The temporal graph of all attacks overlaid on the geo-location of the attack sources, presents an insight into malware propagation. These are everyday compromised machines that are inadvertently participating in an attack. It represents the diurnal characteristic of end users turning on their machines in the morning and turning them off in the evening. All this traffic is highly automated. Human social engineering may play a part in how the end devices are compromised. Botmasters communicate with C&Cs to send instructions to the compromised hosts. But to generate and communicate with a global botnet of compromised hosts, malware employs highly automated methods.

Reward function

As stated, this paper incorporates RL into a honeypot to enable it to adapt to attack methodologies in real time. Previous contributions related to this concept used action

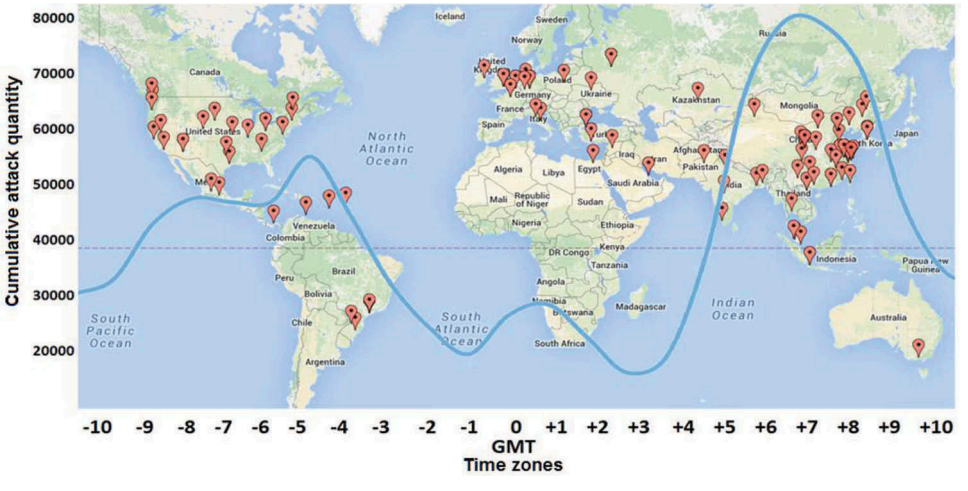

Figure 3. Diurnal pattern of attacks on a honeypot.

sets in the RL model provisioning human interaction. Actions such as insulting the attacker and delaying the response were proposed and implemented into prototype honeypots. Two reward functions corresponding to the two goals of the honeypots were implemented:

(1) Collect attacker related commands and transitional information (r_t)
(2) Extend the duration of an attacker's interaction (r_d)

We point out that attack traffic is automated. Using actions such as *insult* and *delay* artificially contributes to the duration of an attack episode and provides no further insight into the machinations of automated malware. Therefore, there is no requirement to implement the second reward (r_d). Our implementation pursues the goal of increasing the number of attacker commands entered on our honeypot (r_t). We propose that adaptive honeypots deployed on an SSH attack vector should implement RL with a reduced action set aimed at automated malware. Due to the fact that we removed insult as a realistic response to automated malware attacks, we need to modify the reward function for the RL implementation on the honeypot. RASSH compared itself to Heliza by charting the learning evolution for collecting attacker information (transitions), on state *wget*. The formula presented is as follows:

$$r_t(s_i, a_j) = \begin{cases} 1 \text{ if } i \in C \\ \min\limits_{x \in Y}(l_d(i,x)) \text{ if } i \in I \\ 0 \text{ otherwise} \end{cases} \tag{4}$$

where $Y = C \cup L$

The formula for transition reward r_t, based on state/action (s_i, a_j), is as follows:

- If the input string i is a customised attacker command C, then reward $r_t(s_i,a_j) = 1$
- If the input string i is a Linux bash command L, then reward $r_t(s_i,a_j) = 0$
- If the input string i is determined to be an insult I, then the minimum normalised Levenstein [22] distance l_d relative to Y or ENTER is calculated.

The insult set I is considered to anything other than Y and ENTER. The reward for this is obtained by calculating the Levenstein distance between the attack command and {Y, ENTER}. As stated, we propose that insult is not a relevant action item for automated attacks. We propose reducing the action set to {allow, block, substitute}. *Allow* and *Block* are realistic responses to malware behaviour. Botnets can use complex if-else structures to determine next steps during compromise [13]. Using *Substitute* to return an alternative response to an attack command potentially increases the number of attack transitions to newer commands. The reduced action set coupled with state set Y, creates a discrete state/action space. A simplified reward function will help the honeypot learn and converge quicker. The revised formula for reward is as follows:

$$r_t(s_i, a_j) = \begin{cases} 1 \text{ if } i \in Y \\ 0 \text{ otherwise} \end{cases} \tag{5}$$

where $Y = C \cup L$

Honeypot elements

We created honeypots using RASSH as the software framework and the data presented by Heliza. Wagener states that the data 'is quite valuable for a honeypot operator who does not want to setup Heliza but rather simply wants to install static fake services' [3,p.228]. RASSH implemented an improved version of Heliza using Kippo and observed similar behaviour with the dataset. Kippo is no longer under active development but has been extended and released as Cowrie [23]. To ensure that the extensions in Cowrie did not affect comparisons of honeypot functionality, a controlled simulation was created. Both Kippo and Cowrie honeypots were installed in an Eclipse development environment. The RL algorithm is implemented using PyBrain, which is similar to RASSH and allows for strict comparisons between the two. Both honeypots behaved similarly with the same attacking dataset. The SARSA (3) update rule calculates *Q*-values by incorporating the reward achieved from choosing action *a*, the current value of the state action pair and the discounted value of the next state action pair. Applying a discount factor in this way allows the agent to discount the value of future rewards, as they may not turn out to have that value once you get there. The current attack command is parsed as the current state and passed to the PyBrain SARSA module. PyBrain selects the action that will return the maximal value for a given state. The *Q*-values are returned to the honeypot to be stored in a lookup table. The honeypot continues by interacting with the attack command depending on the action selected by the SARSA learning agent. The adaptive honeypot has the following elements:

- Modified honeypot. Cowrie is a widely used honeypot distribution that logs all SSH interactions in a MySQL database. This has been modified to generate the parameters to pass to the learning agent. Depending on the action selected by the learning agent, the honeypot will allow, block or substitute attack commands
- SARSA agent. This module receives the required parameters from the adaptive honeypot and calculates $Q(s_t,\ a_t)$ as per Equation (3). It determines the responses chosen by the adaptive honeypot and learns over time which actions yield the greatest amount of reward.

Experimental results

Direct comparison with previous research using a reduced action set in a simulated environment

The Cowrie honeypot used PyBrain, MySQL and a state/action module similar to RASSH (modified to formula (5)). The RASSH software framework demonstrated similar learning behaviour to Heliza. They both presented an action-value graph for state *wget*, for transition reward (4). We implemented the same software code with the revised reward formula (5). In our controlled experiment, the honeypot was attacked 350 times. Each *wget* attack command populates the *Q*-values table as described in Section 3.3. We wrote a simple botnet using Linux bash scripts that accessed the honeypot and executed commands provided by Heliza. Development code for the Kippo honeypot is freely available [24]. At the end of the attack sequence, the reward values were extracted from the MySQL database. They produced the graph in Figure 4.

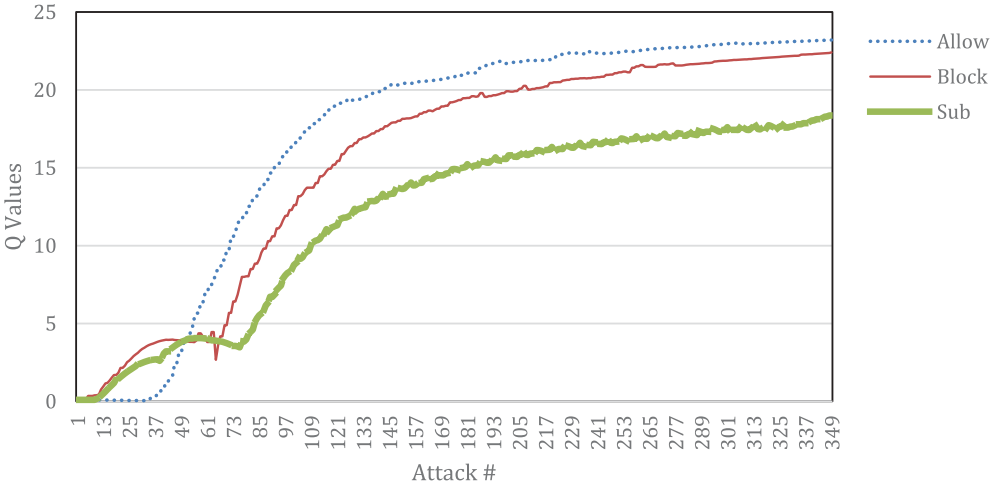

Figure 4. Learning evolution for *wget* with reward (5) in controlled experiment.

The reward evolution in Figure 4 is similar to RASSH and Heliza. However two distinct differences exist.

- The learning evolution converges towards the optimum policy for all actions. Both RASSH and Heliza display a linear evolution.
- A clear separation appears at approximately attack 65 as the honeypot determines the best actions to apply to an attack command. RASSH and Heliza do not definitively determine the best action during the attack sequence.

These two distinctions demonstrate a more efficient learning process when compared to previous implementations. Heliza determined best action only at the end of the 350 attack sequences. The results did not demonstrate a move towards convergence. RASSH replicated those results. With our implementation, the honeypot learned very early that allowing an attack returns the best reward value. This began at attack 35 and returned higher rewards from attack 50 onwards. The honeypot still explored the environment with all actions being selected for the 350 attacks. From approximately attack 120, the honeypot starts to converge towards the optimum policy.

Deployment of a live honeypot with a reduced action set targeting known automated malware

Task 1 deployed an adaptive honeypot in a controlled environment. It examined one state, *wget*, and presented encouraging results. We needed to determine if these results are replicated in a live deployment. Therefore the honeypot from 4.1 was modified to generate rewards on 75 most popular Linux states and was deployed on the Internet. We used Amazon Web Services (AWS) EC2 to facilitate an Internet facing honeypot in the European region. Cowrie, PyBrain, MySQL and other dependencies were installed on the instance. EC2 provides for a convenient method of honeypot deployment. It was accessible through SSH and immediately it started to record malware activity [25].

Initially it logged dictionary and bruteforce attempts. Thereafter it captured other malware traffic including a Mirai-like bot [26]. When the Mirai malware code was made publically available, it spawned a series of variants [27]. One of these variants became the dominant attacking tool over a 30-day period, until over 100 distinct attacks were recorded on the honeypot. The bash scripts within this bot represented states in the learning environment. Table 1 provides a sample of the bot scripts. It defines a profile of the scripts as being *attacker tools* or honeypot *states*. The attacker tools are relevant for discussion in Task 3.

The Mirai-like ssh bot sample in Table 1 shows that it uses *sudo* and *wget* in its configuration. This allows us to extract from the dataset the rewards given to *wget/action* and *sudo/action*. We concentrated on collecting 100 attacks from this bot. Other SSH malware interacted with the honeypot. But these often used different commands (such as *curl*) or were too infrequent to excessively modify the rewards. The graphs produced from these extractions are presented in Figures 5 and 6.

Figures 5 and 6 reflect the findings from Figure 4 in Task 1. We can clearly see that the adaptive honeypot identifies *allow*, as the best action to take at attack 22 (wget) and attack 9 (sudo). Although the honeypot captured 100 attacks, the graphs do present a move towards converging on the optimum learning policy. As each attack has both a *sudo* and a *wget* command in the attack sequence, we can produce more detailed graphs when compared with those from Heliza. In both graphs, *allow* and *block* are the actions that produce the best reward. Checking the logs from the honeypot dataset, we found blocking the commands sometimes resulted in the command being reissued. This immediately contributes to the transition count but more interestingly it gives insight into the operations of the bot, without resorting to sandboxing for code examination.

Comparison of the number of command transitions on an adaptive honeypot with reduced action set with previous research

We deployed two Cowrie honeypots on the Internet at the same time; one adaptive as detailed in Task 2 and one standard high interaction. To compare performance, we

Table 1. Bot command sample.

Sequence	Bot commands	Profile
1	/gweerwe323f	Attacker tool
2	sudo/bin/sh	State
3	/bin/busybox	State
4	/gweerwe323f	Attacker tool
5	mount	State
6	/gweerwe323f	Attacker tool
7	echo -e '\x47\x72\x6f\x70/' >//.nippon	State
8	cat//.nippon	State
– -	– – – – – – – – – – – – – –	– – – –
38	/gweerwe323f	Attacker tool
39	cat/bin/echo	State
40	cd/	State
41	wget http://Redacted IP/bins/usb_bus.x86 -O – > usb_bus	State
42	chmod 777 usb_bus	State
43	./usb_bus	Attacker tool
44	/gweerwe323f	Attacker tool

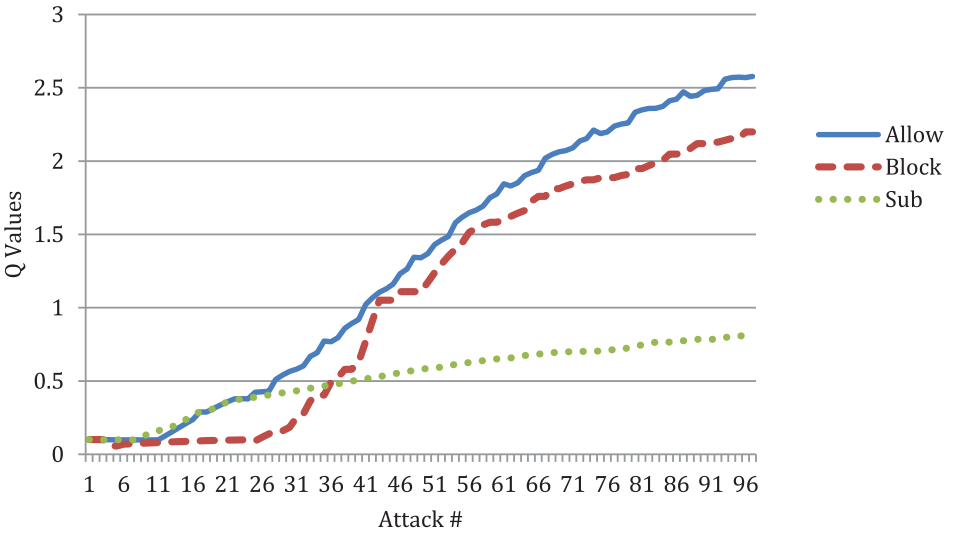

Figure 5. *wget/action* values for reward (5) in live Internet deployment.

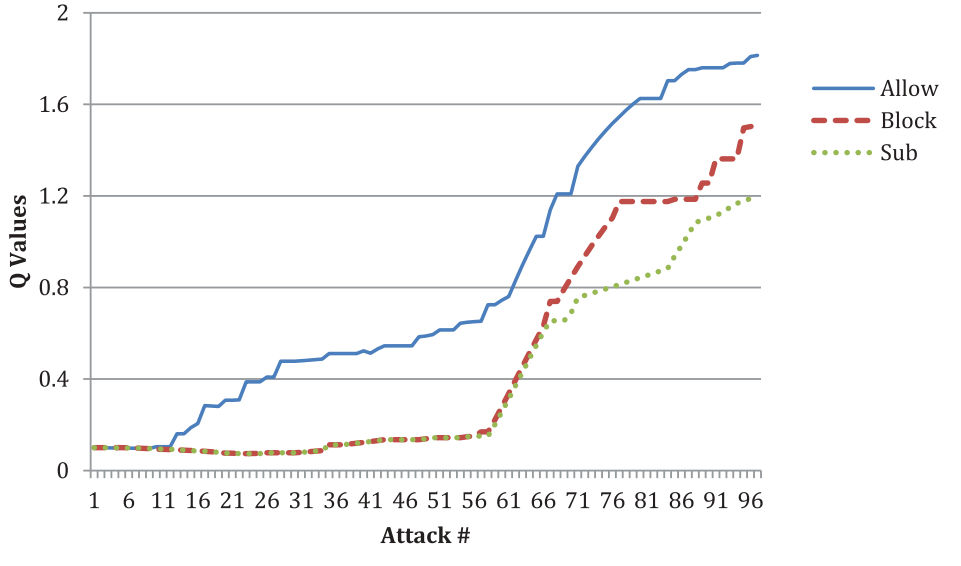

Figure 6. *sudo/action* values for reward (5) in live Internet deployment.

extracted all commands executed on the honeypots. Cowrie stores all interactions in log files as standard. It also stores all downloaded files in a download directory. This makes it possible to accumulate a count of all the commands attempted on the honeypot. These commands all represent interactions post-compromise. Therefore events such as failed attempts, dictionary and bruteforce attacks are excluded as they represent pre-compromise interactions. The dominant interaction came from the Mirai-like SSH bot as seen in Table 1. Other commands such as direct-tcpip and embedded perl scripts also appeared infrequently. These were not recognised states on the honeypot and did not contribute to the reward values. Prior to presenting the results it is important to discuss the format of the bot.

The profile in Table 1 identifies system commands as states and other commands as attacker tools. In reality it has a sequence of 44 commands [25]. It has six commands that can be considered attacker tools. The first three of these exist within the first six commands in the sequence. The remaining three exist in the final seven commands of the sequence.

Comparison with standard high interaction honeypot

The high interaction honeypot only ever experienced the first eight commands in the sequence before the attack terminated. The adaptive honeypot initially only experienced the first eight commands but then the sequence count started to increase as the honeypot learned from its state actions and rewards. Our reward (5) has a goal of maximising transitions. Figure 7 presents a comparison of the cumulative transitions for both honeypots. This includes all command transitions for state and attacker tool commands.

The high interaction honeypot only ever executed 8 commands before the attack ended. This accounts for the linear accumulation of transitions. At attack 16, the adaptive honeypot increased the number of commands in the sequence to 12. This is a very interesting result when compared to the high interaction honeypot's linear evolution. Examining the Cowrie logs, we find that the adaptive honeypot substituted the *cat* command with an *incomplete command* message, for the first time at this point. For the next 30 attacks, the adaptive honeypot executed 12 commands in the attack sequence. The honeypot continued to learn until attack 42 when it allowed all commands in the sequence to be executed. The logs show that the adaptive honeypot allowed *rm* for the first time and continued to do so until the end of all 100 attacks. Overall, we can see that the adaptive honeypot captured 4 times more attack commands than the high interaction honeypot. Investigating the machinations of the bot, through sandboxing and code exploration, could explain this learning evolution. This is however beyond the scope of this article. This task demonstrates that adaptive honeypots significantly increase the size of the dataset that can be captured on a honeypot.

Comparison with Heliza adaptive honeypot

Figure 7 presents cumulative data for all transitions on an adaptive honeypot. It demonstrated that it performed almost four times better, transitioning to attacker commands, when

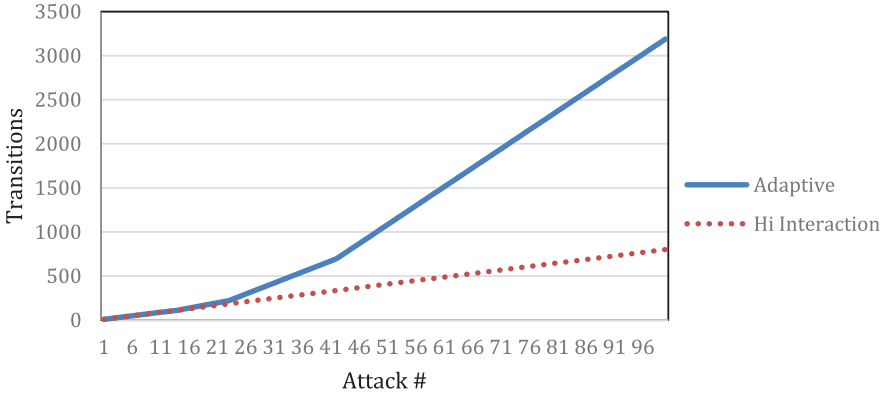

Figure 7. Cumulative transitions for all commands.

compared to a standard high interaction honeypot. To compare it with the performance of Heliza, we examine the logs to specifically look at the transition to attacker commands. This is presented in Figure 8. Again it demonstrates that the adaptive honeypot does perform better than a standard high interaction honeypot. However it does not perform four times better, but only 63% better. This can be explained by revisiting the structure of the bot dominant on the honeypot. Three of the attacker tools reside in the final seven commands of the attack sequence. The adaptive honeypot experienced the complete command sequence from attack 42 onwards. From that point onwards, it interacted with 6 attacker tools until 100 attacks episodes were captured.

It is difficult to compare similarities with previous research when deploying honeypots in real-world experiments. One variable that affects the results is the type of traffic encountered on the honeypot. Malware evolves very quickly and mutate into different versions with alternative attack methodologies. The structure of the bot encountered on our honeypot has a particular methodology and significantly impacts the overall results in Figures 7 and 8. We view Figure 7 as a truer reflection of the effectiveness of an adaptive honeypot. It also reflects our rewards, see Equation (5), which rewards our learner when the attack command is either a system command or an attacker command.

Discussion

Summary

The research presented here uses a novel state action space formalism for automated malware. It builds upon exiting work in this area and improves the RL process implemented on an adaptive honeypot. We presented an improvement in convergence towards the optimum learning policy and a clear indication of a preferred action at an early stage. This policy contributes to a significant increase in command transitions and subsequent dataset volume. The primary purpose of a honeypot is to capture datasets that can be used to analyse the methods of malware developers. New communication methods and end devices for existing technologies such as IoT, industrial IoT, gaming, etc., increase the attack potential. This research highlights the dominant role played by automated traffic for malware propagation. This role is evident on captured honeypot datasets. It is very important to note that all

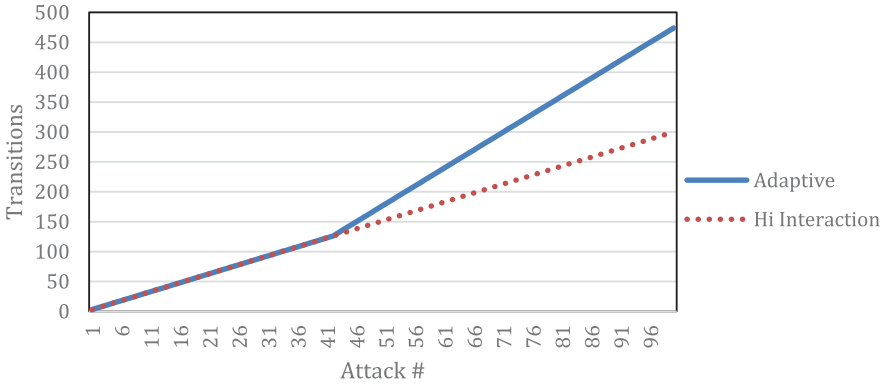

Figure 8. Cumulative transitions for attacker tools.

traffic encountered on our adaptive and standard high interaction honeypots, deployed on the Internet, was totally automated. There was no human interaction in the log files, verified by the commands' timestamps. The methodology in this article has demonstrated improved learning and overall data collection for adaptive honeypots using RL. Malware developers continuously seek to compromise new technology for malicious reasons or for financial gain. Honeypots provide security groups with valuable insight into attacker behaviour. It is but one tool used in the fight against malware propagation. By increasing the quantity of attack interactions, we believe that this methodology can contribute to the fight.

Future work

AWS makes it very easy to locate an Internet facing honeypot at a particular geographic location. We deployed our standalone honeypot on AWS EC2 service. This honeypot incorporated PyBrain and data from the learning process was stored in a backend MySQL database. AWS also provides other services such as Virtual Private Cloud (VPC) and Relational Database Services (RDS). Using these services we can connect the series of honeypots (honeynet) through VPC and have them updating a central database located on one regional EC2 instance. As each honeypot in the series experiences an attack episode it updates the central database. Subsequent attacks at all other honeypots continue this process of updating the central reward values. Our work now focuses on improving the time it takes for a honeynet to converge upon the optimum policy.

Disclosure statement

No potential conflict of interest was reported by the authors.

Notes on contributors

Seamus Dowling is a lecturer with Galway Mayo Institute of Technology (www.gmit.ie). He specialises in networking, data communications and virtualisation. He jointly initiated and runs GMIT's Cisco Academy at the Mayo Campus. He holds a BSc degree in computing from Waterford Institute of Technology and an MSc degree in computing from Sligo Institute of Technology. He is currently pursuing PhD with NUI Galway and OSNA. Previous research includes implementing and analysing DDoS attacks. He has worked in both the private and public sectors in IT support, administration and management roles since 1993.

Michael Schukat is a lecturer and researcher in the Discipline of Information Technology at the National University of Ireland, Galway. He is principal investigator of both the OSNA (Open Sensor Network Authentication) cyber security research group (www.osna-solutions.com) and the Performance Engineering Laboratory (http://pel.it.nuigalway.ie/index_files/home.php) @ NUI Galway. His main research interests include security/privacy problems of connected real-time/time-aware embedded systems (i.e. industrial control, IoT and cyber-physical systems) and their communication/time synchronisation protocols. He is actively involved in various security working groups on a European (e.g. COST Action Cryptacus) and International level (e.g. US-NIST CPS Public WG). Originally from Germany, Dr Schukat studied Computer Science and Medical Informatics at the University of Hildesheim, where he graduated with MSc (Dipl. Inf.) in 1994 and PhD (Dr rer. nat.) in 2000. Between 1994 and 2002, he worked in various industry positions where he specialised in deeply embedded real-time systems across diverse domains, such as industrial control, medical devices, automotive and network storage.

Enda Barrett is a lecturer and researcher at the National University of Ireland Galway. In 2013, Enda received his PhD in computer science from NUI Galway. His PhD research investigated the application of a subset of machine-learning techniques known as reinforcement learning to automate resource allocations and scale applications in infrastructure as a service cloud computing environments. Upon completion of his PhD, Enda joined Schneider Electric as a research engineer on a globally distributed innovation team. His main research interests include machine learning, distributed computing and cyber security.

ORCID

Seamus Dowling http://orcid.org/0000-0001-8722-2009

References

1. Bellovin SM. Packets found on an internet. SIGCOMM Comput Commun Rev. 1993;23(3):26–31.
2. Pa YMP, Suzuki S, Yoshioka K, et al. IoTPOT: analysing the rise of IoT compromises. EMU. 2015;9:1.
3. Wagener G, Dulaunoy A, Engel T. Heliza: talking dirty to the attackers. J Comput Virol. 2011;7(3):221–232.
4. Hayatle O, Otrok H, Youssef A. A Markov decision process model for high interaction honeypots. Information Security Journal: A Global Perspective. 2013;22(4):159–170.
5. Pauna A, Bica I, 2014, May. RASSH-Reinforced adaptive SSH honeypot. *Communications (COMM), 2014 10th International Conference on* 1–6); IEEE.
6. Provos N, A virtual honeypot framework. *USENIX Security Symposium*; 2004, Aug. (Vol.173, p. 1–14).
7. Baecher P, Koetter M, Holz T, et al. The nepenthes platform: an efficient approach to collect malware. In: International Workshop on Recent Advances in Intrusion Detection. Berlin: Springer; 2006. p. 165–184.
8. Portokalidis G, Slowinska A, Bos H. Argos: an emulator for fingerprinting zero-day attacks. ACM SIGOPS Operating Systems Review. 2006 Oct;40(4):15–27.
9. Pouget F, Dacier M, Pham VH, 2005. On the advantages of deploying a large scale distributed honeypot platform. In Proceedings of the E-Crime and Computer Evidence Conference
10. Watson D, Riden J. The Honeynet project: data collection tools, infrastructure, archives and analysis. In Wombat Workshop on Information Security Threats Data Collection and Sharing, WISTDCS '08; IEEE; 2008. p. 24–30.
11. Ramsbrock D, 2007, "Profiling attacker behavior following SSH compromises", 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07), Edinburgh.
12. Holz T, Raynal F, 2005, June. Detecting honeypots and other suspicious environments. In *Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC* IEEE p. 29–36.
13. Dowling S, Schukat M, Melvin H, A ZigBee honeypot to assess IoT cyberattack behaviour. *Signals and Systems Conference (ISSC), 2017 28th Irish*; 2017, Jun. p.1–6. IEEE.
14. Rubin BS, Cheung D. Computer security education and research: handle with care. IEEE Secur Priv. 2006;4(6):56–59.
15. Ghourabi A, Abbes T, Bouhoula A. Characterization of attacks collected from the deployment of Web service honeypot. Security and Communication Networks. 2014; 7(2):338–351.
16. Goseva-Popstojanova K, Anastasovski G, Pantev R. Using multiclass machine learning methods to classify malicious behaviors aimed at web systems. In *Software Reliability Engineering (ISSRE), 2012 IEEE 23rd International Symposium on*; p. 81–90. 2012, Nov. IEEE.
17. Owezarski P, Unsupervised classification and characterization of honeypot attacks. *In Network and Service Management (CNSM), 2014 10th International Conference on*; 2014, Nov.p. 10–18. IEEE.

18. Wagener G, Radu State DA, Engel T, Self-adaptive high interaction honeypots driven by game theory. *SSS*; 2009, Nov. p. 741–755.
19. Valli C, Rabadia P, Woodward A, 2013, "Patterns and patter - an investigation into ssh activity using Kippo honeypots", Australian Digital Forensics Conference.
20. Schaul T, Bayer J, Wierstra D, et al. PyBrain. J Machine Learn Res. 2010;11:743–746. Published 2/10.
21. Dagon D, Gu G, Lee CP, et al.. A taxonomy of botnet structures. In *Computer Security Applications Conference, 2007. ACSAC 2007. Twenty-Third Annual*; 2007, Dec p. 325–339). IEEE.
22. Navarro G. A guided tour to approximate string matching. ACM Computing Surveys (CSUR). 2001;33(1):31–88.
23. "Cowrie Honeypot",[cited Date: 2017 Sep 06] Available from: http://www.micheloosterhof.com/cowrie
24. "An adaptive honeypot using reinforcement learning implementation",[cited Date 2017 Dec 19) Available from: https://github.com/sosdow/RLHPot
25. Chen YZ, Huang ZG, Xu S, et al. Spatiotemporal patterns and predictability of cyberattacks. PloS One. 2015;10(5):e0124472.
26. "SSH Mirai-like bot",[cited Date 2017 Nov 28]. Available from https://pastebin.com/NdUbbL8H
27. Kolias C, Kambourakis G, Stavrou A, et al. DDoS in the IoT: mirai and other botnets. Computer (Long Beach Calif). 2017;50(7):80–84.