J Comput Virol (2011) 7:221–232
DOI 10.1007/s11416-010-0150-4

ORIGINAL PAPER

Heliza: talking dirty to the attackers

Gérard Wagener · Radu State · Alexandre Dulaunoy · Thomas Engel

Received: 18 June 2010 / Accepted: 23 November 2010 / Published online: 29 December 2010

Abstract In this article we describe a new paradigm for adaptive honeypots that are capable of learning from their interaction with attackers. The main objective of such honeypots is to get as much information as possible about the profile of an intruder, while decoying their true nature and goals. We have leveraged machine learning techniques for this task and have developed a honeypot that uses a variant of reinforcement learning in order to learn the best behavior when facing attackers. The honeypot is capable of adopting behavioral strategies that vary from blocking commands, returning erroneous messages right up to insults that aim to irritate the intruder and serve as reverse Turing Test. Our preliminary experimental results show that behavioral strategies are dependent on contextual parameters and can serve as advanced building blocks for intelligent honeypots.

1 Introduction

Over the recent years, honeypots, resources dedicated to be attacked [1], have been widely deployed in operational environments and have been subject of numerous research activities in the scientific community. Although many previous research contributions have addressed the underlying technical and system-level challenges for developing stealthy and secure honeypots, very little has been done with respect to making honeypots intelligent and adaptive. Intelligent honeypots can make a worthwhile difference, since existing honeypots are either too limited and thus provide little valuable information, or are easily identifiable by experienced attackers. We aim to obtain several types of intelligence data from a honeypot, including the software used to escalate privileges as well as the rootkits and custom exploits that an attacker downloads and installs. The tools we are interested in, are tools installed after a successful penetration of the system and used during malicious activities. These tools are usually not collected with traditional malware collectors like Nepenthes [2]. Additional pieces of information that we aim to get are related to the technical background and skill of an attacker, their ethnic and linguistic profile, the repositories used and other targets under attack.

We argue in this article that a new paradigm of adaptive honeypots can provide more intelligence than the established architectures. We describe a prototype called Heliza that can learn how to optimally interact with an attacker. Interactions can be the blocking of a command, the profanity, the return of erroneous results as well as the simple execution of an entered command. Heliza is inspired from the well known Eliza [3] program that lured in the 1970s many computer users into thinking that they were interacting with a therapist or an advanced rule-based system, although Eliza used a very simple and primitive inference algorithm. When designing Heliza, we considered that each interaction has a value in the proper context. For instance, blocking a `wget` command, can lead an attacker to use an alternative repository and this may reveal another attacker controlled location. Insulting an attacker can irritate her and make her reveal her linguistic

G. Wagener (✉) · R. State · T. Engel
University of Luxembourg, 6, rue Richard Coudenhove-Kalergi,
1359 Luxembourg, Luxembourg
e-mail: gerard.wagener.002@student.uni.lu

R. State
e-mail: radu.state@uni.lu

T. Engel
e-mail: thomas.engel@uni.lu

A. Dulaunoy
Computer Incident Response Center
Luxembourg c/o smile - "security made in Letzebuerg",
6, rue de l'Etang, 5326 Contern, Luxembourg
e-mail: alexandre.dulaunoy@circl.lu

background, while the responses of erroneous messages can indicate her technical skills. The major challenge was to define the context and automatically learn the best strategies for each contextual state. Existing honeypots could not implement such strategies. High-interaction honeypots [1] allow any system manipulation by an attacker. Attackers can install and execute arbitrary programs on such a honeypot. The observation of such activities does not reveal their technical skills because every action is permitted and attackers reach their attack goal without any resistance. Furthermore, the linguistic profiles of an attacker cannot be derived from the observation of sequential programs executions. Low- and mid interaction honeypots are easily detected even by a novice attacker, and thus can serve only against autorooters or automated malware.

The main contributions of this article are twofold. First, we describe a new approach for intelligent honeypots that can learn from their interaction with an attacker. We propose a reinforcement learning-based mechanism that leverages the well known reinforcement learning algorithm SARSA [4] and a Markov state model in order to drive an adaptive honeypot. Secondly, we describe the implementation in a Linux based system and highlight some of the technical challenges that we had to face.

The article is organized as follows: The paradigm of adaptive honeypots is discussed in Sect. 2 which describes two different behaviors. One adaptive honeypot is targeted at collecting information from attackers, while another one is intended to keep attacker as long as possible on the system. An adaptive honeypot framework is described in Sect. 3. Experiments are presented in Sect. 4 followed by a summary of the related work in Sect. 5. Conclusions and future work activities are set out in Sect. 6.

2 Adaptive honeypots

Heliza emulates a weak and vulnerable SSH server, which is a popular attack vector [5,6]. Attackers penetrate the system via SSH by performing a brute force attack against accounts on the system. They probe different user accounts with predefined password lists [5]. Shortly after breaking in, attackers come back to the compromised accounts and begin performing malicious activities on the system. Heliza allows information retrieval from attackers to be optimized by leveraging reinforcement learning algorithms.

According to Sutton [4], reinforcement learning is an automated method for goal-directed learning and decision making that works to maximize a numerical reward signal. An agent must discover which actions provide the most reward by trying them out. Rewards can be positive or negative and an agent by default tries to optimize its reward in the long run. A general overview of reinforcement learning is presented in [4] and is shown in Fig. 1. An agent operates in a specific environment and can perform various actions (a_t) at discrete time steps, denoted by the variable t. Each action results in a state change s_t and a reward is given for the selected action (r_t). A classical example is an agent that needs to find the exit of a maze. The agent can move north, south, east and west. Each position in the maze results in a distinct state. When the agent bumps into the wall or wants to make an impossible transition, the agent is punished. When the agent makes a valid movement no reward is given. However, if the agent finds the exit of the maze, it gets a positive reward.

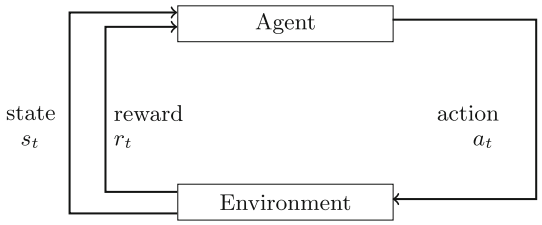

Fig. 1 Overview of reinforcement learning

Heliza is an adaptive honeypot, dedicated to be attacked, and behaves like a learning agent that is continuously under attack. Normally attackers want to execute commands. Heliza has to take decisions as to allow or block these commands. Heliza is also able to forge outputs or insult the attacker. Following a decision, an attacker can enter another command, which results into a state change. Each state change is also linked with a reward.

2.1 Environment

Attackers compromising the system are modeled as its environment. Attackers penetrate the system via SSH and provide input strings that are usually commands. For instance, they may inspect the system and then try to make it ready for their malicious purpose. A typical attack sequence is for instance `sshd` → `uname` → `wget` → `tar` → `custom`. Attackers may also enter empty commands, typographic errors or insults. Generally, attackers enter sequences of strings denoted $i_0 i_1 i_2 \ldots i_n$ where $i_n \in S^*$. An input i_n is a system command if and only if it belongs to the set $L = \{s_1, s_2, s_3, \ldots, s_n\}$ which contains all bash commands [7] including system programs installed during the setup of the system. Attacker often install customized tools, which we designate the set C, like SSH brute force scanners, rootkits, local root exploits or phishing server software. Hence, $C \subseteq S^*$. This means that all valid programs on the honeypot not previously known are installed by attackers. After having successfully transferred them to the honeypot, they are valid programs on the honeypot and can be executed. Each input that is neither a program nor an ENTER keystroke typed by an attacker is considered to be an insult. Hence, the set of insults is $I = S^* - L - \{empty\} - C$.

Formally, the environment is modeled as a Markov chain with an associated reward process [8]. The underlying finite state space is $S = \{s_1, s_2, \ldots, s_n\}$. Contrary to the model proposed by Xu et al. in [8], each state is a command or a program entered by the attacker or one of three special states. $S = L \bigcup \{insult, custom, empty\}$. A transition to the `empty` state is made when an attacker hits the `ENTER` key on the system. Attackers sometimes do this to test whether the remote connection is still working. When an attacker wants to execute a custom installed program, a transition to the `custom` state is made. We define a relation $z_1 \subseteq C \times \{custom\}$. A string that is entered by an attacker and that is neither a program nor an ENTER keystroke is mapped into the state called `insult`, which is formally defined by the mapping $z_2 \subseteq I \times \{insult\}$.

Like the model proposed by Xu et al., we denote the trajectory generated by the Markov chain $\{x_t | t = s_i; s_i \in S\}$. In our model, a trajectory corresponds to a sequence of inputs provided by an attacker. The dynamics of the Markov chain are described by the transition probability matrix P whose (i,j)-th entry is the transition probability for $x_{t+1} = j$ given that $x_t = i$. The probabilities are due to common input subsequences among sequences.

Environment properties

The `exit` state is an absorbing state. When attackers are in this state they have left the honeypot. Due to the fact, that all attackers penetrate the system through the `sshd` state, all attacks pass through the same node. Hence, it can be deduced that this Markov chain cannot have distinct partitions. An attacker can execute a command multiple times, resulting in a loop in the Markov chain. Due to common transitions among different attackers, the same state can lead towards several other states.

Heliza learns by computing a value action function that gives for each (state, action) pair the long-term utility with respect to the target objectives. Once this function has been learned, Heliza chooses at each state the action with the highest estimated reward.

2.2 Honeypot actions

The added value of honeypots lies in their ability to learn from attackers or to reveal information about them. Heliza is adaptive and capable of taking actions in response to attacker actions. Heliza aims to collect attacker tools and to detect whether the attack is automated or manually performed. Heliza can also be tuned to keep attacker busy. Four actions $a_{1...4} \in A$ are possible for Heliza: It can behave like a standard high-interaction honeypot by allowing (a_1) command executions. When Heliza decides to block (a_2) a command entered by an attacker, it is not executed, but an error code is returned. Heliza can also substitute (a_3) commands. For instance, when the command `w` is executed, aiming to see how many users are logged in, Heliza lies and shows a previously generated content. Another, example is where an attacker executes the tool `wget` with the intention of downloading a customized tool. In this case, Heliza could lie and display a "page not found" error, which may lead an attacker to reveal another malicious repository. An alternative possibility is to swap a few bytes in the downloaded payload. Provos et al. [9] showed that the lifetime of malicious code repositories can be very small (1 h) and if an attacker is connecting to such a site we assume that it is not suspicious if Heliza returns a "page not found" error. Heliza can also insult (a_4) attackers. This action mainly serves as reverse Turing Test [10]. The purpose of such a test is to discover whether an action is being performed by a human being or an automated tool. The insult decision leads to a display of an insult in the terminal of an attacker. An attacker can respond with an insult. By doing so, it is highly likely that the attacker is a human being. Suppose that an attacker has downloaded a customized tool and wants to execute it. Heliza then replies: `Is this all what you want to do?` Some attackers immediately leave when they see a message like this. Obviously, in this case we cannot determine whether this attacker is a human or not. However, some attackers get overwhelmed by emotion and type insults in the terminal. In this case, Heliza can discover that the attacker is a human and can sometimes determine the native language of the attacker. Some attacks are automated and their reaction to insults depends on the capabilities of the script. Some scripts check error codes and the output of the executed command and take appropriate actions. Other scripts have no error handling and just continue the attack.

2.3 Rewards

In the reinforcement learning domain, a learning agent tries to optimize a numerical reward signal. Heliza can use two reward functions depending on the desired behaviors.

Collecting attacker related information

Alata et al. [11] described that attackers often install customized tools on high-interaction honeypots designed for their malicious purpose. Hence, the goal of our reward function is to collect as many attacker tools as possible. Moreover, we are interested in discerning the linguistic features of an attacker. However, the main focus is on customized tools installed by an attacker. The reward function with this purpose is defined in Eq. (1) where i is the input string used by an attacker. Each input i of an attacker sequence of strings $i_0 i_1 \ldots i_n$ is mapped with the states of the Markov chain $s_i \in S$. The normalized Levenshtein distance [12] is denoted l_d and the action taken

Table 1 Sample attacker session

t	i_t	s_t	*action*	*reward*
0	`sshd`	`sshd`	allow	0
1	`ssudo`	`insult`	allow	$\frac{1}{8}$
2	`sudo`	`sudo`	block	0
3	`wget`	`wget`	substitute	0
4	`wget`	`wget`	allow	0
5	`./exploit`	custom	insult	1
6	`I am ...`	`insult`	insult	$\frac{47}{145}$

by Heliza is denoted a_j. The merged set of custom commands and system commands is $Y = C \bigcup L$. If an attacker does a transition to a customized tool, Heliza gets the highest reward (1) and if an attacker executes a system related command the reward 0 is distributed. However, if an insult is entered, the Levenshtein distance between this string and all other programs (x) is computed and the minimal normalized distance is returned as reward.

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

Received insults are particularly useful for gaining information about attackers. For instance, if they swear in their native language, this could indicate their origin and ethnic background. The case where an attacker enters an insult, the minimum normalized Levenshtein distance relative to the customized tools and system commands is returned. If the attacker just made a typographic error, the minimum normalized Levenshtein distance is low. Not much information about the attacker has been revealed except it is highly probable that this attacker is a human being. However, if the distance is close to 1, an attacker has entered a completely unknown input, which may be valuable for Heliza. The reason for normalizing the Levenshtein distance between 0 and 1, is that Heliza should focus on collecting tools than rather collecting attacker insults. The highest reward is still granted when a transition to a customized tool is made.

An example of an attacker session is presented in Table 1 where the variable t represents discrete time steps. The column labeled with i_t shows the input that an attacker provided which is mapped to a state s_t in the Markov chain. For a given transition made by an attacker, Heliza can take an *action* and gets a *reward*. In this example, an attacker connected to the honeypot at time 0 and wants to get to the `sshd` state. Heliza allows this transition and gets a reward of 0. The attacker then wants to execute the command `ssudo`, which is classified as an insult. However, the attacker simply made a typographic error. For this simplified example, the Levenshtein distances are computed between the input `ssudo` and all the installed programs {`sshd`, `sudo`, `wget`}. The resulting set of Levenshtein distances is {1, 2, 5}. The minimal Levenshtein distance is 1, which means that only one character needs to be edited to get to the string `sudo`. Hence, the normalized Levenshtein distance becomes $\frac{1}{8}$. In step 2, the attacker notices the typographic error and enters the correct command. This time, Heliza blocks the command. The attacker decides in step 3 to download a local root exploit. Heliza decides to return a forged output stating that the requested page was not found. The attacker then selects another malicious repository and this download is allowed. The attacker wants to execute the local root exploit. A transition to the state `custom` is made because the program was not known during the honeypot setup. The reward 1 is returned because the honeypot has collected a customized tool from an attacker. Heliza decides to print the text `Are you stupid enough to execute this crappy tool...`. The attacker is overwhelmed by emotion and types `I am not stupid dude, it is time for revenge...`. This time, the normalized Levenshtein distance becomes $\frac{47}{145}$ which rewards the honeypot of having revealed an entire sentence from the attacker.

Keeping attackers busy

A straight-forward reward is to take into account the delay between two successive commands expressed in seconds. A higher delay means a longer reaction time of the attacker in handling partial attack failures. Hence, we define a function $\delta : S \times S \times A \times \mathbb{N} \rightarrow \mathbb{R}$. The reward function defined in Eq. (2) returns the temporal difference needed to transit from the previous state to the current state by taking the action a_j at the time i.

$$r_d(s_i, a_j) = \delta(s_{i-1}, s_i, a_j, i) \quad (2)$$

2.4 Learning agent

Heliza is a learning agent, and attackers act as its environment. According to Sutton [4], an agent has the ability to perform a set of actions in various situations (states). Each action is awarded with a positive or negative reward. The purpose of reinforcement learning is to find the optimal policy to select the most promising actions in given states. Formally, a policy π is defined as a stochastic rule used by an agent to select actions [4]. Reinforcement learning is divided into two categories: off-line learning and on-line learning [4]. Monte Carlo methods are frequently used for off-line learning methods and time difference learning methods are used for on-line learning. Either method requires complete knowledge about the environment and both try to optimize received rewards. The purpose of Monte Carlo policy evaluation methods is

to estimate the value of a given state s under a policy π [4]. However, in the context of adaptive honeypots, we are interested in evaluating state action pairs rather than states. An attacker whose rootkit execution has been blocked may chose another path in the hope to achieve the initial attack goal.

The objective of Heliza is to incrementally discover the policy for choosing actions in given states, which is usual for on-policy methods [4]. Attackers connect to Heliza and perform some malicious activity resulting in state transitions. In the state `exit`, they leave the honeypot which means that they have reached an absorbing state. This phenomenon gives the possibility of breaking down the learning method into episodes. The policy is being evaluated at the end of an episode. The State Action Reward State Action (SARSA) method is a straightforward method of on-policy learning method [4]. The general form is presented in Eq. (3). The goal is to estimate the reward Q according to a policy, for a given state s_t and a given action a_t. Due to the fact that an environment is unknown for an agent, an explorer has to decide to explore or to exploit the learned knowledge. This is a fundamental problem in reinforcement learning. We used the ϵ-greedy explorer because convergence to optimal Q values has been proved with such an explorer [13]. The environment is explored according a random component ϵ and the environment is exploited according the learned Q values. An estimation of Q at time t is augmented by the received reward r_t plus a discounted (γ) estimated future reward, taking into account a step size parameter (α). In practice, the rewards are set retroactively and in the adaptive honeypot scenario no discounting ($\gamma = 1$) is done because the beginning and the end of an episode are known. An episode begins when an attacker connects and ends when an attacker leaves at which time the estimated values are computed. A default value of 0.05 is used as step size parameter (α) [14].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t) \right] \quad (3)$$

3 Adaptive honeypot framework

We have developed a high-interaction honeypot generic framework, denoted AHA, where different adaptation mechanisms can be implemented. The framework consists of a modified Linux kernel running in user-space, that outsources the decision-making process to the host operating system. The decisions are then implemented in Python which is easier to perform than implementing a dedicated kernel module. The AHA framework consists of a customized User Mode Linux (UML) [15], a custom developed library with common functions, the AHA daemon and a configuration file. UML is a Linux kernel running in user-space or in other words, it runs

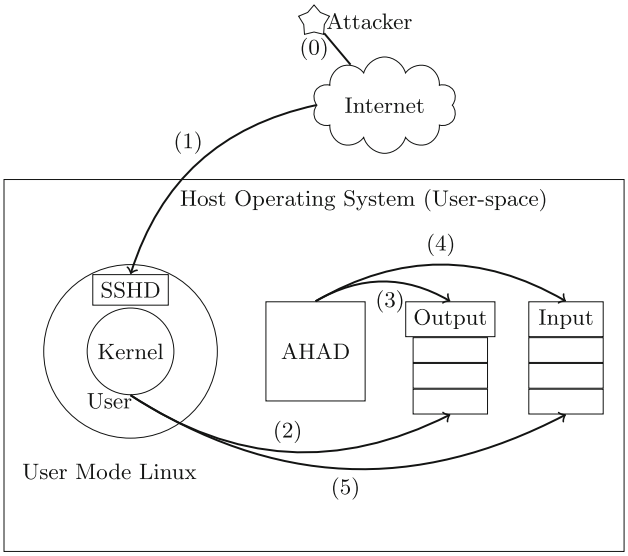

Fig. 2 Overview of the adaptive honeypot framework overview

on a Linux system like any other program and does not need extra privileges. The UML kernel is often called guest kernel and the kernel running the UML is denoted host kernel. UML can be built from an arbitrary recent Linux kernel version by specifying the UM parameter. The reason why UML was chosen, is that from an implementation point of view, it provides functions to interact with the host-operating system that are ready to use. In theory attackers could detect UML [16] and a better alternative is to perform a virtual machine introspection [17]. However, this approach involves more implementation efforts and is even not bullet proof [17]. The scope of this article is to have a proof of concept of adaptive high-interaction honeypots. An overview is shown in Fig. 2. The honeypot is usually operated at a public IP address. Attackers constantly scan the Internet and so discover the honeypot (step 0). They then normally start a brute force attack against the honeypot, by probing different user accounts with associated passwords from large dictionaries. After a while, they manage penetrating the honeypot (step 1). The kernel operating in the virtual machine transmits each argument of a `sys_execve` system call[1] to the host operating system (step 2) where a daemon, denoted AHAD, captures it (step 3) and takes a decision as to whether this system call should be allowed, blocked, substituted or responded with an insult. The daemon puts the decision in the input queue (step 4). The kernel in the guest operating system waits τ milliseconds for the response from the daemon and fetches the decision (step 5). If it receives it within this time frame, it implements the decision. If it does not receive it, the system call returns

[1] Bash internal commands, the fact that an attacker hits the `ENTER` key or typographic errors are not visible in kernel space for the sys_execve function. Hence, it is mandatory to set a hook in bash to propagate these commands to kernel space.

an error, namely ENOMEM, which leads the attacker believe that there is no memory left on the machine. We empirically determined that the maximal time frame τ for suspending a system call is 50 ms. This makes the system slower but still usable.

In order to export information from the guest-kernel to the host-kernel we used dedicated functions included in UML. Among these functions are functions to open sockets or to create files. We modified the UML code and used these functions to exchange information between the host operating system and guest kernel. We did some small surgery in the system call wrapper `sys_execve` with the purpose to intercept all programs that are scheduled to be executed.

A unique message identifier i is generated based on the executed CPU cycles of the host operating system. Inside the UML, the program name with its arguments is taken from user-space, and copied into a static c data structure m_i. This unique message identifier i serves as filename on the host operating system, where the static data structure m_i is written in binary form. In practice, the exchange of binary messages is faster than the exchange of textual messages. The `sys_execve` is then suspended for τ milliseconds. On the host operating system the AHA daemon is notified via the INOTIFY [18] interface that a new message arrived in the output queue. It immediately fetches the message, remembers the message identifier i and decodes it with the Python module c types (useful to handle binary data), takes a decision and puts another binary c structure $\bar{m}_i$ with the same identifier i in the input queue. The reason to select different queues is to reduce the number of synchronization steps as these are computationally expensive. After τ milliseconds the suspended system call `sys_execve` resumes, takes the message and parses it by simply accessing the fields of the binary structure and implements the decision. On one hand, when the system call should be allowed, the regular code of the wrapper is executed. On the other hand, when the system call should be blocked, the error code ENOPERM is returned. When the output of a program is forged, the program name of the desired executed program is swapped with a hidden installed program in the UML capable to forge outputs. From an implementation point of view, insulting an attacker is similar to substituting commands, because in that case the program of the desired executed program is substituted with an hidden program labeled insult that randomly selects insults from a predefined list. It is essential that the daemon distinguish between programs executed by the system itself and by programs executed by attackers [6]. Therefore, the process tree data structure is analyzed [6]. Each program that belongs to a sub-tree of a privileged separated process of `sshd` [19] is identified as a program executed by an attacker. Other programs belong to the system itself and by default are allowed. PyBrain [14], a Python based-library, is used for the implementation of the reinforcement learning part functionality. A prototype of AHA is publicly available [20].

4 Experiments

We operated a high- and a low-interaction honeypot to evaluate Heliza. Each honeypot was operated until 349 successful attacks have been observed. From these experiments we recovered honeypot traces and stored them in an SQLite database [21] which is freely available [22]. A honeypot trace is a chain of inputs provided by attackers and is composed of the following elements:

uid is a unique identifier which distinguishes different attacker sessions. An attacker session starts when the SSH server clones a privileged separated process [19] and ends when this process dies, from which it can be deduced that the attacker left the honeypot.

id is a numerical strictly monotonically increasing identifier that identifies the input that an attacker gives. This identifier is essential to establish the order of the inputs that an attacker has provided. Further details about recovering attacker sessions regarding concurrency issues between attackers and the system itself are presented in our previous research activities [6].

input is the input or command an attacker entered. The input can be a command, a misspelled command or an insult from an attacker.

next input specifies the next input an attacker provided in her session. Usually it is the next command but sometimes it can also be a misspelled command or an insult.

action is the action the honeypot took. For instance, allow, block, substitute or insult.

delay records the time difference expressed in seconds between the two inputs. Due to system and network buffering delays we determined a slowdown factor which was taken into account for further processing.

Table 2 (left part) describes general statistics about the honeypot experiments including attacker traces of the honeypot setups. We have observed 349 different successful logins on each SSH server. The shortest attack duration is between 0 and 1 s which is mainly due to automated brute-force tools that were run against the honeypots aiming to establish a list of successfully exploited user accounts. The maximal attack duration was 97 min. In this SSH session an attacker did heavy configuration work on the system and compiled large programs which took some time. For a standard high-interaction honeypot, the average attacker session lasted for approximately 3 min and most attacker session durations were below 3 min (83%). Heliza most frequently performed allow actions closely followed by the number of occasions

Table 2 Dataset description

General statistics		Country code	Proportion (%)
Number of attacker sessions	349	RO	47
Minimal attack duration (s)	0	DE	16
Maximal attack duration (s)	5,849	ES	10
Average attack duration (s)	162	Unknown	4
SD of attack duration (s)	509	LU	4
Proportion of allow actions (%)	31	IT	4
Proportion of block actions (%)	22	MK	4
Proportion of substitute actions (%)	30	LB	3
Proportion of insult actions (%)	17	NL	2
		GB	1
		BE	1
		US	1
		FI	1
		AT	1
		FR	1

Table 3 Attacker insult analysis

Command	Frequency	Language	Proportion (%)
exit	15.77	Undefined	51.8
ls	11.16	Typographic errors	17.1
cd	9.95	Romanian	11.8
uname	5.82	English	9.2
ps	5.82	Smiley	5.3
last	5.09	Slovak	5.3
wget	4.61	Croatian	1.0
id	4.36	Polish	1.0
w	4.36	German	0.2
others	33.06		

Table 4 Final action values

	Used reward: r_d				Used reward: r_t			
	allow	substitute	block	insult	allow	substitute	block	insult
tar	100	203	55	127	5.55	5.15	4.94	1.96
sudo	101	101	146	196	5.37	1.16	3.71	4.17
chmod	199	121	140	71	5.33	5.50	8.85	8.05
uname	184	202	190	159	5.02	4.81	4.58	5.49
kill	65	1	295	220	1.83	2.82	5.77	1.82
insult	189	188	199	190	5.42	5.57	5.29	4.69
custom	194	170	163	189	5.66	5.10	4.95	5.37
ps	194	183	214	140	4.82	5.14	4.71	5.44
wget	175	202	163	146	6.34	5.53	5.24	5.20
bash	202	118	37	172	4.93	2.86	3.56	3.90
last	64	81	202	106	0.99	1.07	4.85	2.50

on which it forged output for the attackers. Heliza explicitly blocked 31% of the inputs and deliberately insulted attackers in 17% of the cases. The country code corresponding to the IP addresses used by attackers was looked-up and the distribution is shown in the right part of Table 2 (right part). Most attackers came from Romanian IP addresses. 16% of the attackers came from German IP addresses and 10% of the attackers had a Spanish IP addresses.

In the early stages of our honeypot development, it was questionable whether attackers would react to insults. Such a reaction would be an immediate disclosure of personal information regarding attacker. Particularly interestingly, we observed 1,011 insults from attackers. From a purely ethical point of view, we cannot print these insults in this article. However, we can give some information (Table 3) about the used language by attackers to insult Heliza. For most insults we were not able to discover the language attacker used. Some insults consisted of only one character or some random keystrokes. 17% of the insults were due to misspelled command, like the command `uanme` where we believed that the attacker wanted to type `uname`. From these attacker inputs it is highly probable that a human being was connected to Heliza, rather than an automated script assuming that most attackers test their malicious automated attacks before running them. Heliza always used the English language to insult attackers and, surprisingly, fewer than 10% of the returned insults were in English and 12% were in Romanian. Some attackers (5%) showed a sense of humor and replied with a smiley. The right part of table contains the top 10 commands entered by attackers after an insult of the Heliza. Figure 3 shows the inputs and commands attackers have provided after they were insulted by Heliza. On the x-axis is presented the number of inputs an attacker provided and on the y-axis shows the amount of attackers. Heliza insulted 86 distinct attackers and 15% of them immediately left the honeypot. However, most of the attackers entered at least one command or an insult. After a manual investigation of the attacker input sequences, we noticed that some attackers believed that the insults are due to other attackers and not from the system itself. Even some attackers replied with the command `wall` which is used to display messages in all the terminals of the users that are connected. Some attackers just pressed enter to clean the terminal and repeated the command which explains that those attackers preferred to continue the attack. Normally the attacker response time for the first insult is larger than the response time regarding another insult. After a while some

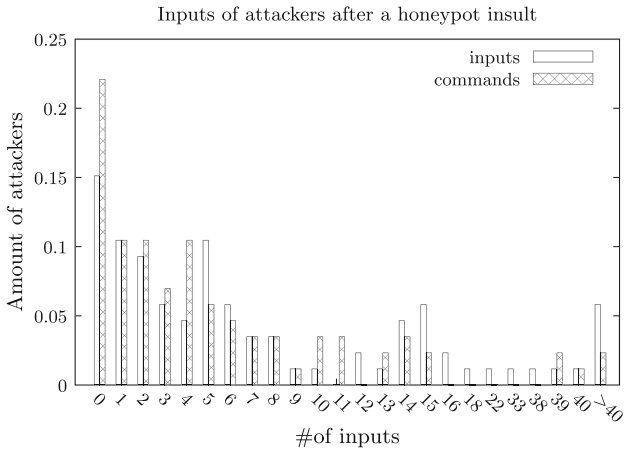

Fig. 3 Inputs entered by attackers after an insult

attackers get annoyed and started to enter successive insults. For such a sequence of insults the delay between such successive insults is <2 s. Other attackers became curious and started to challenge Heliza in order to understand what is going on.

It is worth to mention, that insults can give indication about other compromised machines. For instance, we observed some Romanian insults from German, French and Spanish IP addresses. In this case we assume that Romanian attackers have compromised these machines and used them as rebounds for attacking Heliza aiming connection laundering. The reactions of attackers regarding strategical blocks are also interesting. On average an attacker retries a command one time and there was an attacker who retried a command 116 times. After having done manual analysis of this attacker's traces we assume that this attacker tried to challenge Heliza in order to determine how the decisions are taken. The reaction of attackers namely if attackers continue their attack or if they get annoyed, their persistence facing resistance permit to draw a profile of attackers which may serve as attacker classification criterion (Fig. 3).

4.1 Learning analysis

Heliza was configured with two reward functions defined Eqs. (1) and (2). The honeypot environment Markov chain has 46 states. For space reasons, not all states can be discussed in detail in this article. Thus, we present only the most relevant and some general results. For the purpose of comparison and simplification each attacker connection to Heliza corresponds to an iteration k. Heliza incrementally ($k = k + 1$) computes an action value table describing the various states with the actions that provided the best rewards in the long term [estimated Q values defined in Eq. (3)]. After the final attack ($k = 349$) a stripped action value table is shown in Table 4.

Generally the highest reward for a given state determines the action which should be taken for this state in the long run. This table is quite valuable for an honeypot operator who does not want to setup Heliza but rather simply wants to install static fake services[2]. Despite following the two behaviors, some strategies for selecting an action for a given state are the same. For instance, when an attacker connects to Heliza it is always a good idea to allow the command. An attacker who does not get a command prompt can hardly stay or install custom tools on Heliza. Some commands are frequently used by attackers to explore the compromised system. Heliza has decided to block the command `last` such that the attackers cannot detect other attackers on the system. The program `sudo` is a convenient way to get more user privileges and is often used for attacker maintenance work on Heliza. When Heliza insults an attacker, the attacker needs to investigate the situation, so spending more time on the system. The attacker needs to determine whether the system itself initiated the insult (i.e. provocative error messages configured by system administrators) or if the insult is due from other attackers concurrently connected to the system. However, if Heliza wants to collect tools, this command should be allowed, because it is often used for installing software on the system. If the purpose of Heliza is to collect attacker information, the command `wget` should be allowed[3]. However, if Heliza aims to waste an attacker's time, a forged output should be returned. Attackers usually download their tools as tarballs. Obviously, when transitions favoring custom-installed tools are desired, this transition should be allowed. If the purpose is to detain an attacker, the command *tar* should lie, such that the attacker needs to understand why the required tool is not working. From an implementation point of view the filename passed to the command tar is substituted with another filename.

It has been proved formally that the SARSA always converges if each state is visited an infinity of times and if a greedy learning policy is used [13]. In practice, this means that we need to assess how many attackers are needed to compromise the system in order to have meaningful action value functions. We studied some relevant bash commands, `wget`, `sudo`. The results are presented in Figs. 4, 5, 6 and 7. The graphs 4 and 5 show estimated rewards for `wget`, and the graphs 6 and 7 for `sudo`. In the graphs 5 and 6, Heliza is configured to collect information; in the graphs 4 and 7, to waste attacker's time. Examining Fig. 5, we see that, by iteration 340, Heliza has learned that allowing `wget` is the best strategy for collecting information; in contrast to keep the attacker on time as long as possible, the graph 4 shows that substitution is identified as best by the 50th iteration.

[2] Side effects are unknown.

[3] Assuming that Heliza's outgoing connections are strictly controlled by an Network Intrusion Detection System aiming to avoid collateral damage.

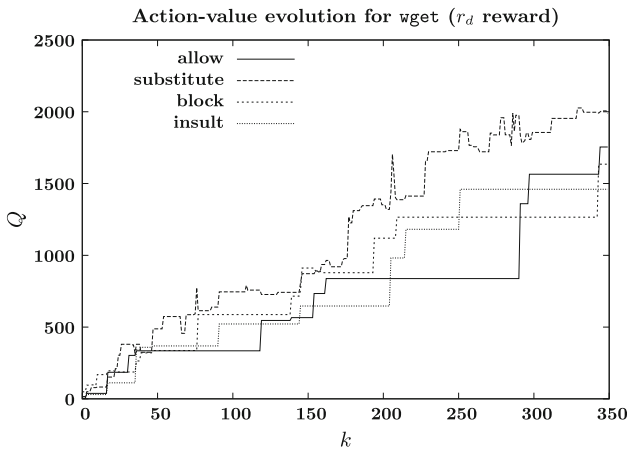

Fig. 4 Action-value evolution for `wget` (r_d reward)

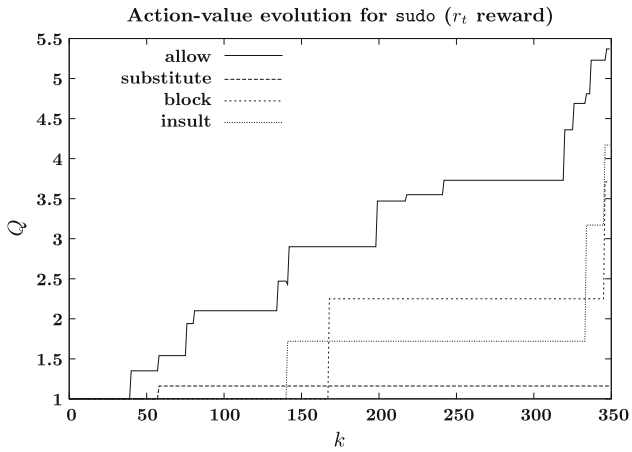

Fig. 6 Action-value evolution for `sudo` (r_t reward)

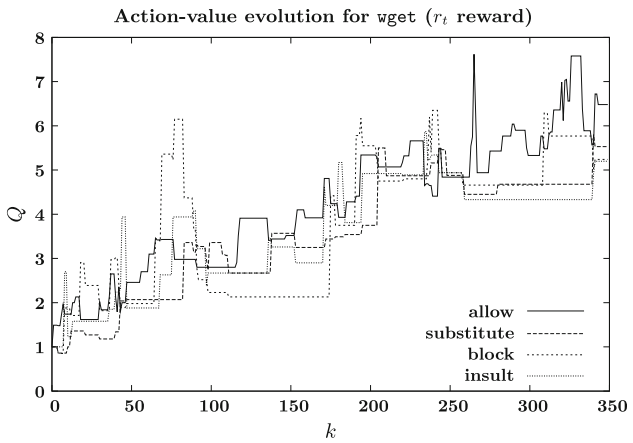

Fig. 5 Action-value evolution for `wget` (r_t reward)

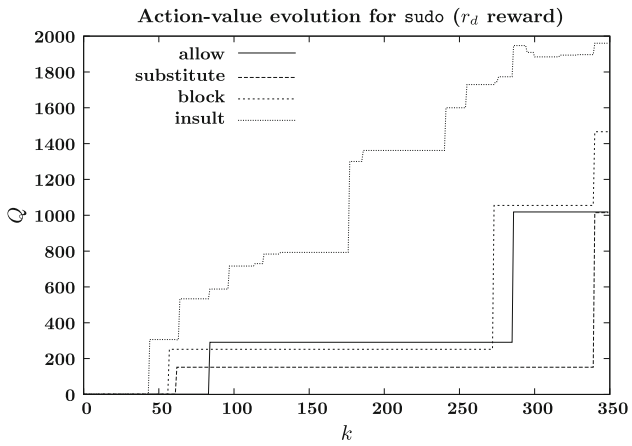

Fig. 7 Action-value evolution for `sudo` (r_d reward)

Similarly, Heliza learned after the 40th iteration that the execution of the `sudo` should be allowed when the purpose is to collect attacker related tools (Fig. 6). In Fig. 7, Heliza learned at iteration 44 that attackers should be insulted for keeping them busy.

4.2 Honeypot comparison

In this section we evaluate the performance of Heliza by comparing it with a standard high-interaction honeypot and a low-interaction honeypot called Hali which emulates behaviors for all the commands executed by attackers. Hali is a fake shell and was developed by ourselves. When an attacker connects to this shell she receives a login and a shell implemented in Python. The output of each command is forged. By default, the standard high-interaction honeypot allows all executions of programs; program executions are neither blocked nor substituted nor are attackers insulted. The purpose of this comparison is to determine whether Heliza reveals more information from attackers than typical low- or high-interaction honeypots. Figure 8 shows that attackers make more transitions to custom installed programs on Heliza than on a regular high-interaction honeypot when Heliza is configured to collect attacker-related information. The x-axis shows the iteration number k, on the y-axis is the cumulative number of transitions to custom installed commands by attackers is shown.

The order of attacks may differ among the different honeypots. Alata et al. [23] reported that attacker launch automated attacks against a honeypot in a first step and come later back to perform the real attack. During the operation of a honeypot these two kinds of phenomena might be mixed. Hence, the cumulative number of transitions is considered in order to make the comparison more robust taking into account an equal set of attacks for each honeypot. For approximately the first 25 attackers Heliza has the same performance than a standard high-interaction honeypot in terms of transitions to attacker related programs. However after 347 successful attacks, Heliza provides an increase of three times in transitions to attacker related commands. This increase is partially due to blocked attacker-related programs. However, an analysis of the actions taken by Heliza for transitions

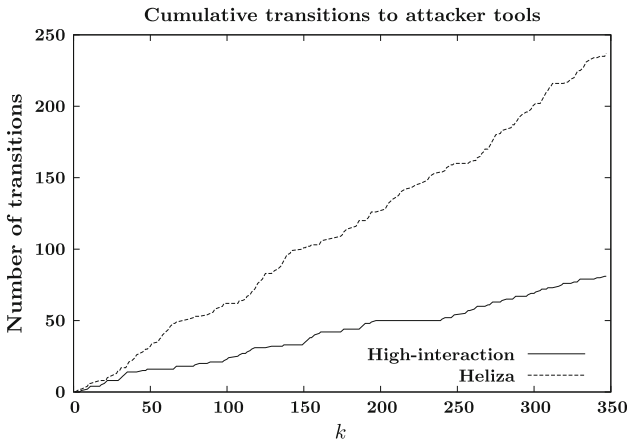

Fig. 8 Attacker related-information collection comparison

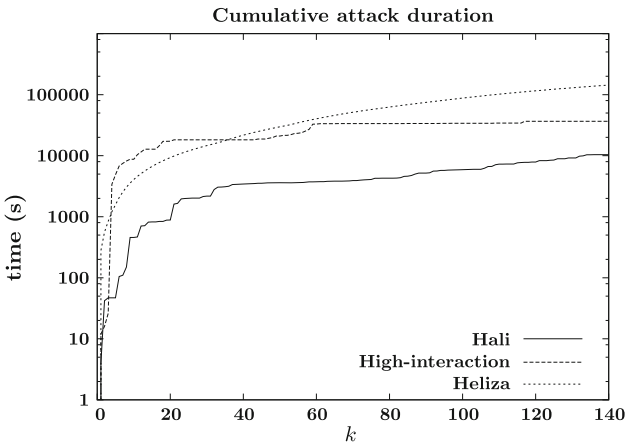

Fig. 9 Maximizing attack duration comparison

to the execution of custom programs shows that 20% of the attacker-related programs have been blocked. These programs have been substituted in 26% of the cases. Heliza allowed transitions to these programs 21%. Finally, Heliza decided in 32% of the cases to insult an attacker when such a transition is made. The reason for not showing the comparison with Hali is that no installations of custom tools has been observed for the first 350 attackers. Obviously, if every command is forged, the installation process for attacker-tools fails.

Figure 9 shows the comparison results of Heliza with Hali and a standard high-interaction honeypot are shown in terms of attack duration (Eq. 2). On Hali, attackers cannot install tools. Usually they try for a while and then give up. At first glance ($k < 30$), the standard high-interaction honeypot performs better than Heliza in terms of keeping the attacker busy. These short sequences of commands are often entered by automated scripts. If Heliza interferes with them they often fail. At the beginning of operation Heliza may take wrong decisions inducing attackers to leave. However, after approximately the 30th iteration Heliza keeps attackers longer than the standard high-interaction honeypot.

5 Related work

Sutton's introduction [4] and the reinforcement learning survey by Kaelbling [24] give a broad overview of the reinforcement learning area. They also show numerous application examples, mainly in the area of robotics. A classic example is a robot that needs to find the exit of a maze. Different reward models exist to parameterize the behavior of the robot. Examples include punishment induced by energy constraints; others do not punish and simply give a positive reward when the robot finds the exit. In other examples, the agent is even punished for bumping into walls. Reinforcement learning often considers an agent that operates in an environment and receives positive or negative rewards in response to taken actions. Among the more spectacular autonomous agents are helicopters which are able to perform aerobatic flight manoeuvres controlled by reinforcement learning [25]. Reinforcement learning has also been explored and extended for collaborating multi-agents [26]. Hierarchical learning among agents has been studied by Barto et al. [27]. Gambardella et al. tackle the well-known traveling salesman problem using an experimental reinforcement-based approach [28]. Learning agents do not always collaborate. Littman et al. explore the possibility of learning agents that are opponents by introducing game theoretical concepts. The example of two opponents playing soccer is given and the game is simulated.

In this article we use reinforcement learning for honeypots that wish to optimize information retrieval from attackers. In 1991 Cheswick [29] described where he manually retrieved information from an attacker by providing forged content to the attacker and by simulating system failures. In 1992 Bellovin [30] discussed trap programs, running dummy services, aiming to collect attacker related information. Cohen [31] discussed various deception techniques useful for information systems. In his Deception Tool Kit (DDK) known vulnerabilities are emulated. However, Heliza uses deception techniques in conjunction with machine learning techniques. In 2002 Lance Spitzner defined honeypots as resources dedicated to be attacked [1]. McCarty stated that there is a race between attackers and honeypot operators [32]. Once honeypot operators have found a means to observe and mitigate attacker actions, clever attackers find ways to circumvent these techniques. Recently, development efforts have been undertaken to perform the monitoring at a virtual hardware layer. However, these approaches have some limitations and could theoretically be circumvented [33]. We deliberately do not want to participate in this race. Obviously, attackers could theoretically detect and sidestep from

the adaptive honeypot framework, but we focus on information retrieval optimization from attackers, trying to lead them to reveal as much useful information about themselves as possible. An essential issue in operating a honeypot is to select the emulated services. Chowdhary et al. [34] propose to emulate services based on observed traces. In this article we focus only on SSH attacks because SSH is popular attack vector and is widely used [5,11].

In 2006, Alata operated a high-interaction honeypot and described some behaviors of attackers. From the architectural point of view, they added a system call into the Linux kernel that is used by a modified SSH server aiming to identify attacker commands. During our experiments we noticed that some attackers replaced the SSH server. Hence, we logged all the executed programs and filtered them taking into account process trees [6]. Similarly to Alata et al. we recorded from legitimate production machines user-accounts that are frequently probed and we created these accounts on the Heliza. However, we preferred to patch the Linux authentication module than just providing week passwords in order to avoid that attackers can lock out other attackers by changing the user-account password. Cheswick [29] and Alata et al. [11] considered typographical errors of attackers for detecting their nature. Monrose et al. [35] went even further and identified users by analyzing keystrokes dynamics. However, applying such an approach to attackers is challenging because the attacker profile is hardly known in advance. Moreover, if the emotions of an attacker has been addressed with a insults, the keystrokes dynamic may change. Alata et al. [11] used the backspace criterion for determining if the attack is originated by a human being. However, we compute the Levensthein distance between the attacker's input and all the valid programs. This takes into account all different keystrokes and can differentiate swear words from similar commands. The first attempt to build an adaptive high-interaction honeypots was undertaken by Wagener et al. [6]. They explore game theory in the context of high-interaction honeypots. In this scenario, a high-interaction honeypot could allow or block the execution of a program. The Nash Equilibrium is computed, resulting in a mixed equilibrium identifying the optimal blocking probabilities for the honeypot. Besides blocking or allowing command execution, Heliza has further capabilities such as doing a Reverse Turing Test or forging the output of command execution. Strictly following the state-of-the-art honeypot classification into low-mid and high-interaction honeypots [1,34], Heliza cannot be classified, because it sometimes behaves like a low-interaction honeypot by returning a forged content. It can also accept only malicious programs like it is the case for mid-interaction honeypots. It also can simply accept arbitrary commands as it is the case for high-interaction honeypots. Pouget et al. [36] propose a methodology to compare honeypots in terms of attacker interactions and attack patterns. In this article we focus on optimizing interaction and information retrieval, namely downloaded customized tools and insults learned from attackers or keeping them busy.

6 Conclusion and future work

In this article we have described Heliza a honeypot prototype that leverages machine learning techniques in order adapt its behavior to attackers. We define behavior in terms of several actions that can be taken: blocking, executing the command, returning errors, or insulting. The applicability of each action is dependent on the context, that on the command to be executed command as well as the history of commands. We have leveraged an on-line reinforcement algorithm to map the context of actions to the action to be taken. This work can be used to develop a new generation of honeypots that exhibit learning capabilities and adaptability which reduces the risk of honeypot operation. Our current work consists in extending the underlying system with higher-order Markov chain state models and addressing more application-specific deployment targets. Heliza could be used as information source for studies of social backgrounds of attackers. For instance, Heliza could swear in different languages. The keystrokes dynamics taking into account attacker emotions could also be explored. Heliza has the limit that clever attackers could misuse commands for achieving their goals. An attacker could for instance use the state `perl` to list programs instead of using the command `ls`. Further research needs to be done about attackers that know how Heliza is working and who try to poison the learning process or who try to evade or takeover Heliza.

References

1. Spitzner, L.: Honeypots: Tracking Hackers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA (2002)
2. Baecher, P., Koetter, M., Dornseif, M., Freiling, F.: The nepenthes platform: an efficient approach to collect malware. In: Proceedings of the 9th International Symposium on Recent Advances in Intrusion Detection RAID, Springer, pp. 165–184 (2006)
3. Weizenbaum, J.: Eliza—a computer program for the study of natural language communication between man and machine. Commun. ACM **9**(1), 36–45 (1966)
4. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). The MIT Press, Cambridge, MA (1998)
5. Ramsbrock, D., Berthier, R., Cukier, M.: Profiling attacker behavior following SSH compromises. In: DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks, Washington, DC, USA, IEEE Computer Society, 119–124 (2007)
6. Wagener, G., State, R., Dulaunoy, A., Engel, T.: Self adaptive high interaction honeypots driven by game theory. In: SSS '09: Proceedings of the 11th International Symposium on Stabilization, Safety,

and Security of Distributed Systems, Berlin, Heidelberg, Springer-Verlag, 741–755 (2009)
7. Newham, C., Vossen, J., Albing, C., Vossen, J.: Bash Cookbook: Solutions and Examples for Bash Users. O'Reilly Media, Inc., Sebastopol (2007)
8. Xu, X., Xie, T.: A reinforcement learning approach for host-based intrusion detection using sequences of system calls. In: ICIC (1). 995–1003 (2005)
9. Provos, N., Mcnamee, D., Mavrommatis, P., Wang, K., Modadugu, N.G.: The ghost in the browser analysis of web-based malware. In: Proceedings of the 1st conference on First Workshop on Hot Topics in Understanding Botnets, USENIX Association, Cambridge (2007)
10. Coates, A.L., Baird, H.S., Fateman, R.J.: Pessimal print: a Reverse Turing Test. In: Proceedings of the International Conference on Document Analysis and Recognition (ICDAR), 1154–1158 (2001)
11. Alata, E., Nicomette, V., Kaaniche, M., Dacier, M., Herrb, M.: Lessons learned from the deployment of a high-interaction honeypot. In: EDCC '06: Proceedings of the Sixth European Dependable Computing Conference, Washington, DC, USA, IEEE Computer Society, 39–46 (2006)
12. Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. **33**(1), 31–88 (2001)
13. Singh, S.P., Jaakkola, T., Littman, M.L., Szepesvári, C.: Convergence results for single-step on-policy reinforcement-learning algorithms. Mach. Learn. **38**(3), 287–308 (2000)
14. Schaul, T., Bayer, J., Wierstra, D., Sun, Y., Felder, M., Sehnke, F., Rückstieß, T., Schmidhuber, J.: PyBrain. J. Mach. Learn. Res. **11**, 743–746 (2010)
15. Wright, C., Cowan, C., Morris, J.: Linux security modules: General security support for the linux kernel. In: Proceedings of the 11th USENIX Security Symposium. 17–31 (2002)
16. Holz, T., Raynal, F.: Detecting honeypots and other suspicious environments. In: 6th IEEE Information Assurance Workshop, United States Military Academy, West Point (2005)
17. Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings Network and Distributed Systems Security Symposium, 191–206 (2003)
18. Love, R.: Kernel korner: intro to inotify. Linux J. **2005**(139), 8–12 (2005)
19. Provos, N., Friedl, M., Honeyman, P.: Preventing privilege escalation. In: SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium, Berkeley, CA, USA, USENIX Association, 231–242 (2003)
20. Wagener, G.: Aha source code repository. http://git.quuxlabs.com
21. Owens, M.: Embedding an SQL database with SQLite. Linux J. **2003**(110), 2–5 (2003)
22. Wagener, G.: Aha dataset. http://quuxlabs.com/~gerard/datasets
23. Alata, E., Nicomette, V., Kaaniche, M., Dacier, M., Herrb, M.: Lessons learned from the deployment of a high-interaction honeypot. In: Dependable Computing Conference, 2006. EDCC'06. Sixth European, 39–46 (2006)
24. Kaelbling, L., Littman, M., Moore, A.: Reinforcement learning: A survey. J. Artif. Intell. Res. **4**, 237–285 (1996)
25. Abbeel, P., Coates, A., Quigley, M., Ng, A.Y.: An application of reinforcement learning to aerobatic helicopter flight. In: Advances in Neural Information Processing Systems 19, MIT Press (2007)
26. Tan, M.: Multi-agent reinforcement learning: independent vs. cooperative agents. In: Proceedings of the Tenth International Conference on Machine Learning, Morgan Kaufmann, 330–337 (1993)
27. Barto, A.G., Mahadevan, S.: Recent advances in hierarchical reinforcement learning. Discrete Event Dyn. Syst. **13**(1–2), 41–77 (2003)
28. Gambardella, L.M., Dorigo, M.: Ant-Q: a reinforcement learning approach to the traveling salesman problem. In: Proceedings of the ML-95, 12th International Conference on Machine Learning, Morgan Kaufmann, 252–260 (1995)
29. Cheswick, B.: An evening with Berferd in which a cracker is lured, endured, and studied. In: Proceedings of Winter USENIX Conference, 163–174 (1992)
30. Bellovin, S.M.: There be dragons. In: Proceedings of the Third Usenix Unix Security Symposium, 1–16, September 1992
31. Cohen, F.: A note on the role of deception in information protection. Computers & Security **17**(6), 483–506 (1998)
32. McCarty, B.: The honeynet arms race. IEEE Secur Priv **1**(6), 79–82 (2003)
33. Xuxian, J., Xinyuan, W.: "out-of-the-box" Monitoring of VM-Based High-Interaction honeypots. In: RAID, 198–218 (2007)
34. Chowdhary, V., Tongaonkar, A., Chiueh, T.: Towards automatic learning of valid services for honeypots. In: ICDCIT, 469–470 (2004)
35. Monrose, F., Rubin, A.: Authentication via keystroke dynamics. In: CCS '97: Proceedings of the 4th ACM Conference on Computer and Communications Security, New York, NY, USA, ACM, 48–56 (1997)
36. Pouget, F., Pouget, F., Holz, T., Holz, T.: A pointillist approach for comparing honeypots. In: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA 2005), LNCS 3448, Springer-Verlag, 51–68 (2005)