RASSH - Reinforced Adaptive SSH Honeypot

Adrian Pauna[1,*], Ion Bica[1]
[1]*Military Technical Academy, Faculty of Military Electronic and Information Systems, Bucharest, Romania*
[*]Corresponding author (E-mail: adrian.pauna.ro@gmail.com)

***Abstract* — The wide spread of cyber-attacks made the need of gathering as much information as possible about them, a real demand in nowadays global context. The honeypot systems have become a powerful tool on the way to accomplish that. Researchers have already focused on the development of various honeypot systems but the fact that their administration is time consuming made clear the need of self-adaptive honeypot system capable of learning from their interaction with attackers. This paper presents a self-adaptive honeypot system we are developing that tries to overlap some of the disadvantaged that existing systems have. The proposed honeypot is a medium interaction system developed using Python and it emulates a SSH (Secure Shell) server. The system is capable of interacting with the attackers by means of reinforcement learning algorithms.**

***Keywords:* honeypot systems; reinforcement learning; ssh; machine learning.**

I. Introduction

In the context of the evolving and more challenging world of cyber security, one of the things that seems not to change, is the small prevalence of malware for Linux systems. Recent studies show that even though there have been several intentions to create malware that compromises the Linux operating system, still the most common way of penetrating such a system is by brute forcing access via SSH service [1]. Experts show that is difficult for a malware to overpass the restrictions imposed by the lack of root access and by the fast updates available for most of the Linux vulnerabilities [2].

Therefore compromising an SSH service followed by the installation of a root-kit or some other type of backdoor it is still a common attack. The usage of SSH honeypot proved to be a solid way of gathering information about this type of attack. These types of systems are becoming trivial in today's security infrastructure. Honeypot systems such as Kippo, are widely used by Computer Emergency Response Team (CERT) all around the world [3]. In the same time researchers try to improve the honeypot systems and one of the paths they have taken involves the integration of self-adaptive capabilities by means of artificial intelligence techniques. One of the newly developed systems is Heliza, a honeypot system capable of interacting with the attackers by leveraging machine learning techniques [4].

Heliza was developed by G. Wagener, R. State, A. Dulaunoy, T. Engeland and it is implemented using a modified User Mode Linux kernel and a Python implementation of the machine learning algorithm [5]. This implementation has some specific shortcomings such as the complexity of the deployment of UML system corroborated with disadvantages of using a fully operating system as target for the attackers.

In this paper we present a new implementation of an adaptive honeypot system that follows the approach used by Heliza and overcomes several of its shortcomings. Our system uses as reference the code of Kippo honeypot system and functionalities of Pybrain library [6], [7].

The main contributions of this article are threefold. First, we propose a new implementation for self-adaptive honeypots capable of learning from its interaction with an attacker. We have used as reference an existing implementation that uses a well-known reinforcement learning algorithm SARSA and a Markov state model [8]. Secondly, we describe the implementation of the application in Python and we highlight some of the technical improvements we propose. Thirdly we present a succinct comparison of our implementation with the existing one. The comparison was done at an empirical level since the system we propose it is still in a development phase. Our tests concluded that some improved features are notable such as the scalability, portability and low overhead.

Furthermore, in **Sect. 2** we propose a short overview of honeypot systems and their classification. In **Sect. 3** we go more in detail with the description of RASSH reinforced capabilities and the way the system is modeled as an Intelligent Agent. The presentation takes into account similarities with Heliza but emphasis the new features we have implemented. The architecture of RASSH is described in **Sect. 4** followed by technical details of its implementation in **Sect. 5**. The results of our incipient tests issued as a way to compare our system with Heliza are presented in **Sect. 6**. We conclude our work, in **Sect. 7**, with a set of remarks and some future work aspects we think might be useful to be taken into account.

II. Adaptive Honeypot Systems

Honeypot systems are traps set up to attract attackers. From the computer terminology perspective these are computer services or fully operable systems exposed to online access with no production roles for the daily business of the organization. Their main role is to collect information about the attackers they interact with. The assumption is that no –one from inside will interact with them since they have no production role.

A. Clasification

The honeypot systems can be classified according to their level of interaction as [9]:

- **Low interaction:** systems that expose only specific services such as FTP, SSH, HTTP and therefore the attackers do not interact with the hosting operating system;
- **Medium interaction:** systems that expose specific services but with a certain level of access to the underlying operating system; for example SSH service that gives access to an emulated file system;

- **High Interaction:** systems that expose a fully operating system which in fact offers the possibility of fully compromising it.

The honeypot systems can be also classified according to their level of adaptability as:

- **Static:** systems that will always keep the same configuration and will expose the same appearance to the attackers;
- **Dynamic:** systems that will change their appearance automatically based on the attacks they are facing or based on their current network configuration that surrounds them [9].

From the perspective of the level of adaptability one of the approaches used by the researchers has been the creation of self-adaptable honeypot systems. These honeypot systems basically leverage specific Artificial Intelligence (AI) techniques such reinforcement learning (RL), game theory or Case Based Reasoning (CBR), so to interact with the attackers [9].

B. *Heliza a self-adaptive honeypot system*

Heliza, a self-adaptive honeypot system, is capable of interacting with the attackers by means of reinforcement learning [4].

Heliza maximizes the amount of information gathered by learning how to interact with attackers. Heliza leverages machine learning techniques and game theory. Heliza permanently improves its behavior and evaluates several interaction strategies. The system was implemented and deployed over a weak configured SSH server. The authors have compared its performance with other high interaction and low interaction honeypots. Heliza achieves fast learning capability with few human operated interventions based on an underlying Markovian decision process [4].

III. Reinforced Honeypot

Reinforcement learning is an area of machine learning that tries to solve the problem faced by an agent when it must learn behavior by interacting with a dynamic environment by means of trial-and-error mechanism [8].

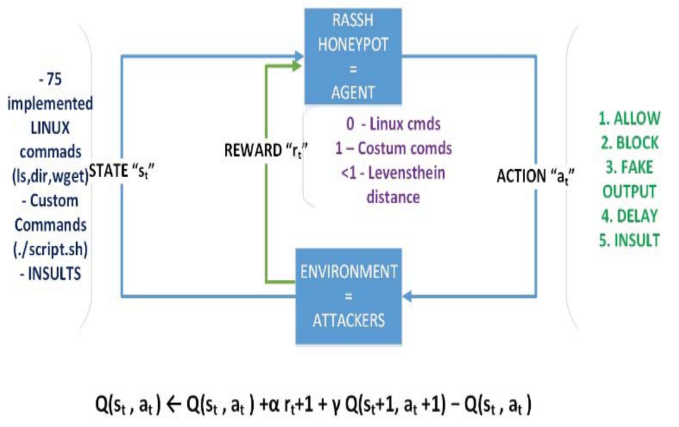
RASSH HONEYPOT = AGENT
- 75 implemented LINUX commads (ls,dir,wget)
- Custom Commands (./script.sh)
- INSULTS
STATE "s_t"
REWARD "r_t"
0 - Linux cmds
1 – Costum comds
<1 - Levensthein distance
ACTION "a_t"
1. ALLOW
2. BLOCK
3. FAKE OUTPUT
4. DELAY
5. INSULT
ENVIRONMENT = ATTACKERS
$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha\, r_t+1 + \gamma\, Q(s_t+1, a_t+1) - Q(s_t, a_t)$

Figure 1. Reinforcement Learning

A standard reinforcement learning model depicted in Fig. 1 assumes that an agent is connected to its environment by means of perception and action.

On each steps the agent:

- Receives some input, denoted i, which gives details about the current state s of its environment;
- Selects an action a, which changes the state of the environment;
- Calculates the value of this state transition by generating a scalar reinforcement signal, denoted r.

The overall goal of this process is to find the policy π, which maps states to actions and maximizes the long-run measure of reinforcement. The environment will be non-deterministic, meaning that the same action in the same state but in different steps might produce different reinforcements.

A. *Environment*

The attackers that manage to compromise the system represent the environment. After compromising the SSH server the attackers issue a sequence of commands that starts with *sshd* and ends with the exit command.

We denote this as a sequence of strings $i_0 i_1 i_2 \ldots i_n$ where $i_n \in S^*$. The i_n strings will fall in one of the following categories:

- Linux commands implemented in the application: $L = \{s_1 s_2 s_3 \ldots . s_n\}$;
- Customized tools developed or downloaded on the local system : $C \subseteq S^*$;
- Insults - we consider any command that doesn't fall in the above categories nor is an ENTER command:
- $I = S^* - L - \{empty\} - C$.

The environment will be modeled, formally as a Markov chain with an associated reward process with the underlying finite state space [4]. Therefore we have implemented a limit set of 75 commands: $S = \{s_1 s_2 s_3 \ldots s_{75}\}$.

We consider also as part of the environment the insults, the custom commands and the empty ones: $S = L \cup \{insult, custom, empty\}$.

The exit state is an absorbing state because in that state the attackers already left the honeypot [4].

B. *Actions*

Similar to Heliza, RASSH implements a set of actions by which the honeypot interacts with the attackers. This set comprises five actions, four of them being similar from the functionality perspective but different from the implementation perspective.

We detail the actions below:

- **Allow action**: it has identical functionality and implementation with Heliza ; it allows the execution of a command inserted by an attacker;
- **Block action:** it has identical functionality as Heliza (blocks the execution of a command) but the implementation differs in the sense that for each command we can have specific blocking messages;
- **Fake output action:** it has identical functionality as Heliza (fakes the output of a command) but the implementation differs in the sense that for each command we can have a specific faked output;
- **Insult action:** it has identical functionality as Heliza (insults the attackers) but the implementation differs in the sense that each attacker gets geo-localized and the insult message is in the correspondent language;
- **Delay action:** this is a newly implemented that has as target delaying the execution of a command. Its utility resides in the fact that attackers might consider the system being exhausted and will try to download other tools that might have less resources usage.

C. *Rewards*

The creators of Heliza proposed two reward functions corresponding to the two goals they wanted to achieve:

- Decoying attackers to download custom software;
- Keeping the attackers on the honeypot as much as possible.

By adding the fifth action, that delays the execution of a command, the second reward function becomes irrelevant and therefore for our implementation we have used only the reward function that has as goal "collecting attacker related information".

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

where $Y = C \cup L$.

As depicted in the formula:

- The highest reward is received when the attacker executes a custom command (1);
- The smallest reward is received when the attacker executes a command from those 75 implemented in the system (0);
- The minimal normalized value of the Levenshtein distance between the introduced string and all the other programs, when an insult is introduced [10].

D. *Learning agent*

The main purpose of our adaptive honeypot is to evaluate state actions pairs rather than states. Basically what we try to achieve is to discover a policy for choosing actions in given states. This type of learning resides in the category of on-policy methods [8]. Because every attack starts with the sshd command and ends with the exit command, the learning method can be broken down into episodes resulting that the policy is being evaluated at the end of each episode. For the implementation we have also used State Action Reward Action (SARSA) algorithm because it is a straightforward method for on-policy learning [8].

The equation that defines the SARSA algorithm is presented below. As next step, we had to define the values for the parameters:

- $\in -greedy$ explore because of the convergence to optimal **Q** values [11];
- $\gamma = 1$ which means no discounting factor because we know when each episode start and ends up;
- $\alpha = 0,5$ a default value.

$$Q\left(s_t,a_t\right) \leftarrow Q\left(s_t,a_t\right) + \alpha\left[r_{t+1} + \gamma Q\left(s_{t+1},a_{t+1}\right) - Q\left(s_t,a_t\right)\right] \quad (2)$$

As you will notice all the selected parameters are similar with the one used by Heliza and the main reason for this was so to be able to achieve the comparison between the two implementations, we will present in the next chapters.

IV. Architecture of the System

For the implementation of RASSH we have used existing software tools and applications such Kippo and Pybrain toolbox. We have integrated new developments to achieve the desired outcome. We mention that the application still needs further testing, improvements being mandatory.

A. *Kippo*

Kippo is a medium interaction honeypot system, developed in Python that logs SSH brute force attacks and offers an emulated shell that acquires all the commands introduced by the attacker post-compromise.

The system is widely used by many organizations that are active in the area of cyber security such as CERTs, antivirus vendors and universities. The system is able to collect valuable information because of its interesting features such as:

- Fake Debian shell [6];
- Fake file system;
- Commands that offer downloading capacity such as *wget*.

All the logs can be store in a Mysql database so that you can review them later and even replay them in a lively manner with tools such as *playlog.py* [6].

B. *Pybrain*

As describe by T. Schaul et al., Pybrain is a "versatile machine learning library for Python" [7]. It provides flexible but yet very powerful algorithms for machine learning. The library has as strict dependency only the SciPy library and implements learning algorithms and architectures ranging from supervised learning and reinforcement learning to evolutionary methods [12]. One of its interesting features is the ability to connect various types of architectures and algorithms.

V. Implementation

As stated previously, RASSH is an evolution of Kippo achieved by implementing in Python the necessary modules that leverage reinforcement learning. The architecture of the system, presented in Fig. 2, comprises several modules with specific roles: Honeypot module, Actions module, RL module.

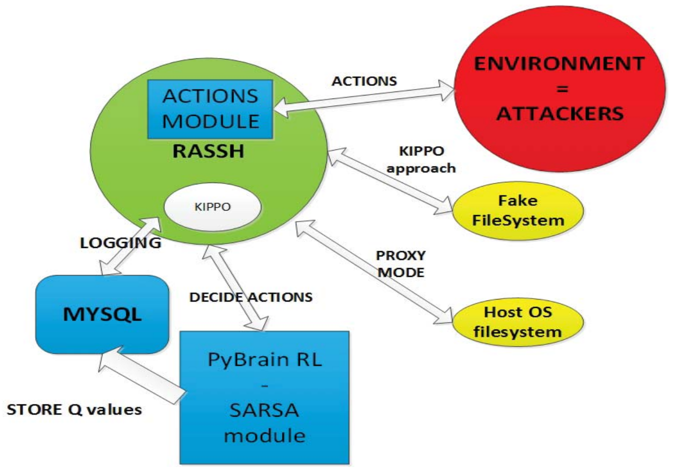

Figure 2. RASSH architecture

A. *The Honeypot module*

This module implementation relies on the existing code of Kippo but improves its capabilities through the implementation of several new commands (*useradd*, *userdel*, etc.). Beside the emulated commands that existed (*ifconfig*, *wget*, *ping*, etc.), we have implemented new ones that use a different approach that we call proxy mode. In proxy mode, the honeypot module acts as a proxy between the Linux operating system shell and the emulated one accessed by attackers. This way, before executing a command at the operating system level, the module waits for the actions decided by the RL module.

B. *The Actions Module*

This module acts as an intermediate between the honeypot module and the RL module. Its purpose is to implement the actions in the shell offered to the attackers by the Honeypot module. The decision of which action should be triggered and when, comes from the RL module.

This module is independent of the implementation method of the commands (emulated or proxy). The following approach was used for each of the actions:

- **Allow action:** it is straightforward meaning that the module permits the execution of the command;
- **Block action:** if RL module decides to trigger this action, for each of the commands there is an error message stored in the database that is printed on the shell and the command does not get executed;
- **Fake output action:** if RL module decides to trigger this action, for each of the commands there is a fake output message stored in the database that will be printed on the shell and the command does not get executed. The stored output is a modified copy of a normal one and for different types of commands it is listed line by line (for ex. Ping command output);
- **Delay action:** it is also a straightforward implementation meaning that if this action is triggered, a sleep line code with a specific interval will be in the loop and after the expiration of the interval, the command is executed;
- **Insult action:** if RL module decides to trigger this action, the module geo-localizes the IP address and an insult message stored in the database in the native language will be printed on the shell. The command will not get executed.

C. *The RL module*

This module uses the existing implementation of the SARSA algorithm of Pybrain and models the Reinforcement Learning problem that we presented in the previous chapter [13]. In Fig. 3 we detail the code of the RL module. As you can see PyBrain facilitates a very simple implementation by providing the necessary components of the reinforcement learning algorithms in its library. The module saves all the Q-values in the Mysql database.

```
class RL:
   def __init__(self):
       self.av_table = ActionValueTable(4, 5)
       self.av_table.initialize(0.1)

       learner = SARSA()
       learner._setExplorer(EpsilonGreedyExplorer(0.0))
       self.agent = LearningAgent(self.av_table, learner)

       env = RASSHEnv()
       task = RASSHTask(env)
       self.experiment = Experiment(task, self.agent)

   def go(self):
     self.experiment.doInteractions(1)
     self.agent.learn()
     def rl_start_thread():
   t = threading.Thread(target=rl_run)
….
class HASSHEnv(Environment):
      indim = 5
     outdim = 4

   def getSensors(self):
      val = getCurrentCommand()
      return [val,]

   def performAction(self, action):
      doAction(action)
```

Figure 3. RL module implementation

We have tried to use as much as possible the features of Pybrain so to be able to test other reinforcement learning algorithms implemented into the library.

VI. EXPERIMENTS

The starting point of our implementation was the idea of creating a new honeypot system that overcomes the flaws that exist in other solutions. For our survey, the only existing implementation similar to RASSH we have encountered was Heliza and therefore our experiments focused on its improvement.

During our research we haven't issued a fully comparative test between the two implementations. Empirical tests have shown several improvements such as:

- Improvement of the scalability of the system: Heliza depends on the UML implementation which is dependent on the linux system;
- Improvement of the Reverse Turing Test capability: Heliza issues insult only in English and its creators observed that attackers tend to leave the system when their nationality differs;
- Improvement of the learning capability: Heliza use only the SARSA algorithm implementation.

A. Improvement of the scalability of the system

Because of the use of Kippo as reference, a system that proved to be very scalable being used by many experts and organizations involved in the cyber security area, we consider RASSH a scalable self-adaptive honeypot.

On the other hand, Heliza, as stated also by its creators is "a proof of concept of adaptive high interaction honeypots".

Its scalability derives also from its implementation:

- while Heliza uses UML kernel and this approach involves more implementation efforts and is even not bullet proof [14];
- RASSH uses Python libraries that are portable and require less implementation efforts.

Table I
RASSH compared to Heliza

Feature	RASSH	Heliza
Fake File system	X	X
Adaptive Actions (the implemented actions adapt to the origin of the attacker and the type of command)	X	
Low Overhead (the application CPU usage is low)	X	
Portability (the application may run on different operating systems)	X	
Tested RL algorithms (a benchmark of the implemented reinforcement learning algorithms)		X
Comparative Tests (a comparison with common honeypot systems)	X	X
Proxy mode commands (offer direct access to the hosting operating system)	X	

B. Improvement of the Reverse Turing Test capability

The vast majority of the attacks a SSH honeypot collects are generated by automated tools. A simple test, such as activating the logging feature of the SSH service shows that on the Internet there are "bots" that scan entire ranges of IP addresses with the only purpose of brute forcing their user accounts. Some of these bots are configured so that once they compromise an account; they issue a predefined set of commands that collect details about the system. In the same manner a human attacker first tries to discover details about the system. One way to differentiate bots to humans, as proposed by the creators of Heliza, is to use the concept of Reverse Turing test [15]. Our tests show that by issuing insult messages in the native language corresponding to the source IP address country, improves the capacity of determining the ethnicity of the attacker. Compared to Heliza, in which case 15% of the attackers left the system after receiving an insult in our case less than 10% had the same behavior.

C. Improvement of the learning capability

Studies show that there is a real interest in using artificial intelligence features to develop adaptive honeypot systems [16].

RASSH is one of the first that makes use of well-known machine learning libraries, offering to experts involved in the creation of algorithms means to easily integrate them with Honeypot systems capabilities. The purpose of our work was not to test the feasibility of these algorithms or their optimization.

Nevertheless we have set up an experiment with our SARSA implementation by using the same dataset the authors of Heliza have used [17].

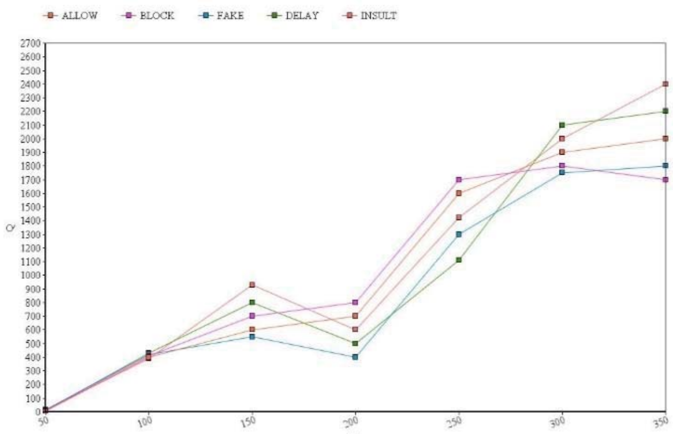

Figure 4. RASSH test action-value evolution for wget (r_d reward)

The results are promising and, as you can see from the graph presented in Fig. 4, our system behavior was similar.

We consider that by using a common machine learning library developed in Python, future work on the improvement of the system becomes less tedious.

VII. CONCLUSIONS AND FUTURE WORK

In this paper we have presented a SSH self-adaptive honeypot system, which we call RASSH, capable of learning behavior by interacting with its attackers. The concept is not novel, but our implementation differs from the existing ones in many ways, bringing more scalability and creating the prospects for a framework that can be used for further developments. Our experiments show that the system is lighter than previous ones creating the possibility of being used also by non-experienced technical persons. Also the experiments show that by using well known machine learning libraries, experts from the area of artificial intelligence might become more easily involved in creating algorithms feasible for the specificity of honeypots system. At the moment, RASSH is not a fully operational system, so as future work we foresee improvements regarding the coding of the application and integration with other Reinforcement Leaning algorithms such as the collaborative ones [18].

REFERENCES

[1] S. A. Mokhov, M. Laverdière, D. Benredjem “Taxonomy of Linux Kernel Vulnerability Solutions”, Innovative Techniques in Instruction Technology, E-learning, E-assessment, and Education 2008, pp 485-493.

[2] T. Kojm "The ways of viruses in Linux HOW SAFE?", http://www.linux-magazine.com/w3/issue/62/Viruses_in_Linux.pdf , Retrieved 2014-02-07.

[3] ENISA report: "Proactive Detection of Security Incidents" https://www.enisa.europa.eu/activities/cert/support/proactive-detection/proactive-detection-of-security-incidents-II-honeypots/at_download/fullReport , Retrieved 2014-02-07.

[4] Gérard Wagener, Radu State, Alexandre Dulaunoy, Thomas Engel "Heliza: talking dirty to the attackers",Journal in Computer Virology August 2011, Volume 7, Issue 3, pp 221-232.

[5] 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).

[6] Kippo: A ssh honeypot. http://code.google.com/p/kippo/, Retrieved 2014-02-07.

[7] T. Schaul, J. Bayer, D. Wierstra, M. Felder, F. Sehnke, Y. Sun, T. Ruckstieß, J. Schmidhuber: "PyBrain" Journal of Machine Learning Research 11 (2010) 743-746 ,Published 2/10.

[8] Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction (Adaptive Computation and Machine Learning). The MIT Press, Cambridge, MA (1998).

[9] Wira Zanoramy Ansiry Zakaria, Miss Laiha Mat Kiah "A review of dynamic and intelligent honeypots." ScienceAsia 39S (2013): 1–5.

[10] Navarro, G.: A guided tour to approximate string matching. ACM Comput. Surv. 33(1), 31–88 (2001)

[11] 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)

[12] SciPy a Python-based ecosystem of open-source software for mathematics, science, and engineering. In particular, these are some of the core packages, http://www.scipy.org/ ,Retrieved 2014-02-07.

[13] A. Pauna , RASSH : https://code.google.com/p/rassh/

[14] Garfinkel, T., Rosenblum, M.: A virtual machine introspection based architecture for intrusion detection. In: Proceedings Network and Distributed Systems Security Symposium, 191–206 (2003)

[15] 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)

[16] Zakaria WZA, Mat Kiah ML (2012) A review on artificial intelligence techniques for developing intelligent honeypot. In: Proceedings of the 3rd International Conference on Next Generation Information Technology, Seoul, pp 696–701.

[17] Wagener, G.: AHA dataset. http://quuxlabs.com/~gerard/datasets

[18] P. Stone, Learning and multiagent reasoning for autonomous agents. In The 20th International Joint Conference on Artificial Intelligence, pages 13–30, January 2007.