12th IFIP/IEEE International Symposium on Integrated Network Management 2011

Adaptive and Self-Configurable Honeypots

Gérard Wagener, Radu State and Thomas Engel
University of Luxembourg
6, rue Coudenhove-Kalergi
L-1359 Luxembourg
{firstname.lastname}@uni.lu
gerard.wagener@ses.com

Alexandre Dulaunoy
Computer Incident Response Center Luxembourg
6, rue de l'Etang
L-5326 Contern, Luxembourg
alexandre.dulaunoy@circl.lu

***Abstract*—Honeypot evangelists propagate the message that honeypots are particularly useful for learning from attackers. However, by looking at current honeypots, most of them are statically configured and managed, which requires a priori knowledge about attackers. In this paper we propose a high-interaction honeypot capable of learning from attackers and capable of dynamically changing its behavior using a variant of reinforcement learning. It can strategically block the execution of programs, lure the attacker by substituting programs and insult attackers with the intent of revealing the attacker's nature and ethnic background. We also investigated the fact that attackers could learn to defeat the honeypot and discovered that attacker and honeypot interests sometimes diverge.**

I. Introduction

Honeypots are valuable resources, designed to be under attack and aiming to gather information about attackers [1]. Many honeypots designs have been proposed and their configuration remains challenging [1]–[3]. On the one hand, if the honeypot is too restrictive, little information can be gathered. On the other hand, if attackers have too much freedom, they can perform a takeover of the honeypot and lock out the honeypot operator. This dilemma leads to a security management problem. Only a few of research activities have focused on making honeypots intelligent and adaptive. On current high-interaction honeypots, which are real systems with real security holes, permissions can be configured to grant or restrict access to given resources. Extensions have been proposed to permit a predefined behavior for a given application [4], [5] based on a policy. However, these paradigms are not suited to design autonomous honeypots because attackers' behavior is barely known in advance and may evolve. In [2] we used game theory as a driving force for making high-interaction honeypots, adaptive targeting information retrieval optimization from attackers. The key challenge was to find the optimal blocking probabilities related to the execution of programs originated by attackers. This was done using traces from the deployment of a high-interaction honeypot. However, in the meantime, attacker behavior could have changed substantively. One possibility for tackling this problem is to use reinforcement learning [6] for the operation of high interaction-honeypots. In this case the honeypot is considered as an agent that tries to optimize a reward signal received from a partially unknown environment as a consequence of having performed various actions. Conceptually, such an approach, includes attackers in the environment and neglects the competition between attackers and honeypot operators as it is the case for a honeypot driven by game theory. In this paper we propose a high-interaction honeypot which optimizes the feedback from attackers including adversarial attacker behavior as reward signal with the aim of being autonomous. Obviously, attackers also use the feedback from the honeypot to find the best strategic behavior. We used fast concurrent learning in stochastic environments in order to create a self-configuring and autonomous honeypot facing attackers. The interaction between attackers and our honeypot is framed in a stochastic game and simultaneous learning and adaptation for the honeypot and attackers are analyzed. An adaptive and autonomous honeypot has been developed on top of User Mode Linux [7] in order to validate the conceptual approach. The paper is organized as follows: Section II explains the evolution of adaptive honeypots. Section III contains the major contribution of this paper. We investigate where the honeypot learns from the attacker and is able to perform self-management, whereas the attacker tries to identify the honeypot's behavior. Section IV describes the architecture and design of an adaptive honeypot based on User Mode Linux. This novel honeypot is evaluated using honeypot traces from a deployed honeypot and the results are presented in section V. Related work is enumerated in section VI, and the paper is concluded in section VII.

II. Motivation

A. Honeypot System Model

In this paper, we investigate high-interaction honeypots operated on a Linux operating system exposing a vulnerable SSH server. SSH servers are popular targets for attackers because they need just to compromise a user-account and then they have full-access to the machine [2], [8], [9]. The system is modeled with an extended hierarchical probabilistic automaton [2]. After having penetrated a SSH server, attackers usually execute sequences of commands in order to achieve their goal. These sequences can be modeled as branches of the automaton. The automaton has two different types of states: macro states and micro states. A macro state represents a program installed on the high-interaction honeypot. The automaton has also micro states, where each command line argument represents a state. For instance, the command line

978-1-4244-9221-3/11/$26.00 ©2011 IEEE

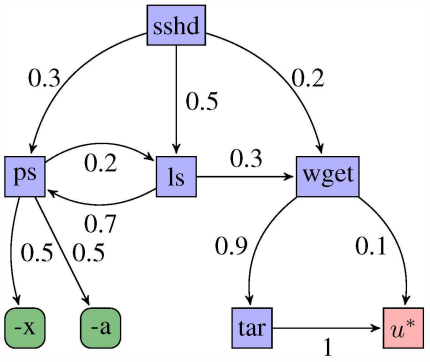

Fig. 1. System model: hierarchical probabilistic automaton

argument `-x` has a different meaning for the program `ps` than for the program `ls`. Hence, we introduced this hierarchy. In addition, the automaton has several special states $u_0, u_1 \dots u_N$ modeling unknown commands. An unknown command corresponds to an input provided by an attacker that does not match a program installed during the initial setup of the honeypot. The state u^* denotes a valid program that was installed by an attacker. If an attacker provides the input `uanme`, then we assume that the attacker made a typographical error instead of executing the program `uname`. Some attackers also insult the honeypot by typing a random string. An illustrative example of such an automaton is shown in figure 1. A simplified automaton is shown. The real automaton has 47 states and cannot be printed. When an attacker connects to the high-interaction honeypot, the program `ps` and the program `wget` are executed with a probability of 0.3 or 0.2 respectively. The program `ls` is executed with a probability of 0.5 and the program `ps` is executed with the command line arguments `-x` and `-a` with equal probability. The tool `wget` is executed with a probability of 0.3 if the previous command is `ls`. After this download, the acquired file is extracted with the program `tar` with a probability of 0.9 or is directly executed with a probability of 0.1.

This model can be extended by allowing adaptive behavior from a honeypot. In each state the honeypot could take a strategic decision based on corresponding actions from an attacker. While simple game theory might provide static views on the efficiency of individual actions [2], the major research challenge addressed in this paper is the design, implementation and evaluation of an adaptive honeypot that can learn and adapt to attackers in nearly real-time. In practice, the transition probabilities are hard to assess. Therefore, it is also assumed that the environment may evolve and that attackers are capable of increasing their skills and defeat the adaptive honeypot. The major conceptual building block in our case is reinforcement learning for stochastic games. In the following we provide a short primer on game theory and Reinforcement Learning (RL) in the context of high-interaction honeypots in order to make the paper self-contained.

B. Game Theory

In the area of applied mathematics, game theory [10] is used to model and to study games between competitors or players. Each player has his or her own interests and can take various actions. Each action results in a payoff. A positive payoff means that a player made a good action and a negative payoff is imposed when a player made a bad action. Players are considered to be rational. Consequently, they try to select their strategy profiles of making actions in such a manner to get the highest reward. In game theory, equilibria concepts were introduced. The most popular is the Nash equilibrium [10]. In a honeypot game this means that neither the honeypot nor the attacker can increase their expected payoff assuming that neither player does not change his strategy during the game. In [2] we defined a game as a 3-tuple $\Gamma = (N_p, (A_i, R_i)_{1 \leq i \leq n})$ between the honeypot and the attacker, where the set of players is $N_p = \{attacker, honeypot\}$. We also have a strategy set for the honeypot A_0 which contains discrete blocking probabilities while the strategy set for the attacker set includes tuples $(Pr(retry), Pr(alternative), Pr(quit))$ where each item describes a probability. When the honeypot has blocked the execution of an attacker's program, he or she could retry, select an alternative program or leave the honeypot according to the respective probabilities [2]. Each player's action results in a reward given by a reward function R_i for each player. An example of a honeypot game is presented in figure 2, which is split into two parts. The lower graph at the left represents a simplified honeypot automaton similar to the automaton in figure 1. The upper graph is a decision tree for the game showing the decisions for each player. Assuming that an attacker has just connected to the honeypot he or she stays in the state `sshd`. The transition towards the state `nmap` yields the highest probability, namely 0.4. This means that the attacker wants to execute the program `nmap`. The Linux kernel intercepts the execution of this system call and a decision must be made whether to allow or block the execution of a program by choosing a blocking probability $Pr(block)$. For each decision, a payoff is distributed to each player. If we consider the case where the honeypot blocks program execution, then the attacker needs to decide if he or she should leave according the probability $(Pr(quit))$, choose an alternative program $(Pr(alternative))$ or simply retry the program execution $(Pr(retry))$. Considering the case where the attacker retries the program execution, the honeypot again needs to decide whether to allow or block program execution. A payoff is distributed to each player. The game is continued until the attacker has reached his or her goal or when he or she leaves. The key question is now which blocking probability leads to the highest expected reward? After having recovered numerical values for the payoffs for each player, we determined Nash equilibria. Nash equilibria are divided into two categories: pure equilibria and mixed equilibria. The simpler case is a pure equilibrium. This means that a player should always select a given strategy, e.g. block the execution of programs with a probability β_0. However, in case of a mixed

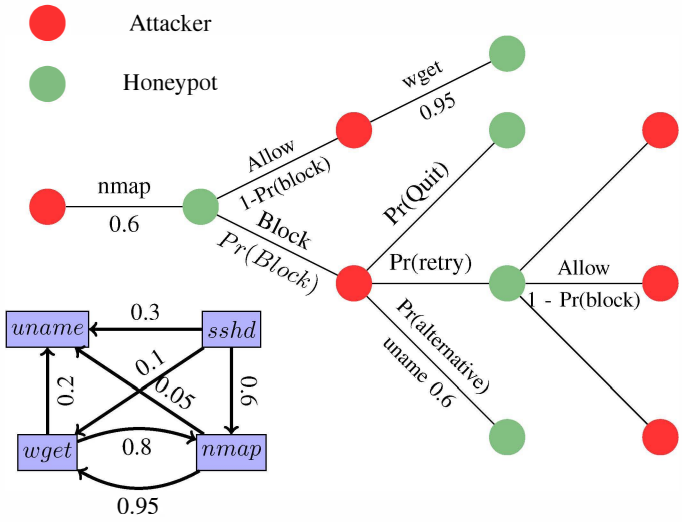

Fig. 2. Honeypot game

equilibrium, the Nash equilibrium points towards a probability of choosing a given strategy. For instance, the honeypot should use a blocking probability of β_0 according a probability of α_0.

The reason for defining strategy sets with probabilities is arguable. One could imagine defining the strategy set for the honeypot with two elements, namely allow or block the execution of programs. A mixed Nash equilibrium would result into a probability of allowing or blocking the execution of programs. The reason for using probability sets is that there is an uncertainty for attackers and on the observed transition probabilities. For instance, when the honeypot blocks the command `wget`, some attackers give up because they do not see a way to acquire their customized tools; other attackers simply retry the command and clever attackers seek an alternative program for getting their tool. Therefore, we compose a strategy profile as set of tuples containing probabilities of performing various actions. Furthermore, the risk of taking a bad decision using fixed actions is higher than choosing blocking probabilities. In order to have a more precise idea about the impact of erroneous transition probabilities, a quantal equilibrium can be computed as defined in [11]. The idea is that players are assumed to make errors and, in our case the payoffs may be faulty due to uncertain transition probabilities. Quantal response equilibrium analysis describes a function taking as input payoffs and a rationality parameter and outputting the Nash equilibrium for this setting. By iterating this function and by having similar consecutive Nash equilibria, the selected strategy is robust against noisy payoffs (independent of the rationality parameter). However, when using high-interaction honeypots the quantal equilibria shows that the Nash equilibria are not perfectly robust against irrationality and the Nash Equilibrium needs to be periodically recomputed using a new snapshot of the honeypot automaton.

C. Reinforcement Learning

In order to avoid the problem of uncertain transition probabilities RL can be used as an alternative. The advantage of RL is that it embodies model-free approaches [12]. In this case a model does not have to be known in advance. This means that the decisions about attacker actions are independent of the observed transition probabilities. An agent operates in an environment by performing several actions. Each action results in a reward. The objective of an agent is to optimize its received reward. An agent in the context of high-interaction honeypots is the honeypot itself. The agent operates in an environment, defined as a hierarchical probabilistic automaton providing rewards to the honeypot. If an attacker connects to the honeypot and he or she wants to execute a program, then the honeypot needs to decide whether this program execution should be allowed or blocked. After the decision, a reward is distributed to the honeypot. The received reward is then cumulated for each state. When the honeypot reaches a previously seen state, it needs to take a decision. This decision depends on the honeypot's explorer. The choice of the explorer is one parameter to ensure convergence of the learning process. The ϵ-greedy explorer is frequently used. This means that the honeypot almost always takes the action yielding in the highest reward with some random behavior parameter ϵ [6]. Hence, the probability of blocking the program execution does not depend on the transition frequencies in the honeypot automaton. The previously discussed snapshot problem can be avoided because a closed loop is used to estimate the rewards. RL can be more fine-grained than just determining the optimal probability to select a blocking probability. In RL, expected rewards are estimated for each individual state. If an attacker executes the command `wget` and if the objective of the honeypot is to collect programs acquired by attackers, then this program should be allowed, because the allow action yields the highest reward. However, a problem of such an approach is that attackers are considered as part of the environment despite their competitive nature. According to Banerjee et al. [13] considering competitors in a learning scenario as aspects of the environment may mean the environment is no longer stationary and convergence results may be impacted.

III. Fast Concurrent Learning Honeypot

For single learning agents, Markov Decision Processes (MDPs) are popular for modeling the environment with an agent [6], [14]. However, this modeling framework is not suited when multiple players are taken into account. Stochastic games were proposed as extension to traditional MDPs. According to Hu et al. [14], a 2-player stochastic game Γ is a 6 tuple $< S, A^1, A^2, r^1, r^2, p >$, where S is the discrete state space, A^k is the discrete action space of player k, $k = 1, 2$, $r^k : S \times A^1 \times A^2 \rightarrow R$ is the payoff function for player k, $p : S \times A^1 \times A^2 \rightarrow \Delta$ is the transition probability map, where Δ is the set of probability distributions over state space S. Such a model can be derived from our honeypot automaton presented in section II-A. The state space S remains the same, containing the programs installed on the honeypot.

A. Actions

Once an attacker has compromised a machine, he or she usually wants to enter shell commands resulting in program execution. These commands are included in the set of actions for an attacker A^1. When the honeypot blocks the execution of a program, an attacker could retry the same command, select an alternative or leave. In the case where the attacker wants to retry a command, the next state is the previous one. When an attacker wants to leave, the next command is the state `exit` and in case the attacker wants to select an alternative command, the desired state is a different one. The attacker can also insult the honeypot by passing to the states u_n. The honeypot can allow or block a command. Additionally, we added two more actions for the honeypot. For instance, the program of an attacker could be substituted. Attackers often download tools from already compromised machines or shared malicious repositories that are often subject to changes. For instance, an attacker wants to download an IRC bouncing tool and downloads the file `xzf.tar.gz`. After the download, he or she unpacks it and start the binary denoted `./x`. If the honeypot substitutes the tool `./x` with an SSH brute-force tool, the attacker may believe that he or she downloaded the wrong tool and may acquire another tool. By causing this action, the honeypot may have revealed another malicious repository and an additional tool. We also decided to insult attackers by sending an insult to his or her terminal. During the initial implementation experiments we believed that attackers would not react to insults. Doing so would reveal additional information about the attacker, for instance, the language of the attacker's riposte. Sometimes we also observed that the attacker just simply continued the attack in a rapid manner. By looking at the timestamps of successive commands, we believed that this was an automated attack. Hence, insults can serve as a Reverse Turing test to determine whether attackers are humans or machines [15]. Nevertheless, we observed insults from attackers, where some attackers believed that other attackers insulted them and others believed that an administrator insulted them. Thus, we think that insults are particularly powerful deception technique.

B. Rewards

At any point the operation of a stochastic game, the game is in a given state. In the context of our adaptive honeypot, this is the last program that was executed. Each player takes an action. The attacker wants to execute a next program by entering a command and the honeypot decides whether this command should be allowed, blocked, substituted or lead to an insult. Although the reward function r^k takes all these inputs and is capable of returning a reward, the reward function is often unknown to the players. The transition function is also often not known [14]. In this context this means that the next command entered by an attacker is unknown. In such a situation, agents need to explore the environment and their objective is to find an optimal policy that maximizes their expected rewards. According, to Banerjee et al. [13] the distributed reward may depend on the behaviors of the opponents, which makes reward computation challenging, because each opponent has his or her own self-interests.

1) Attacker reward: Assume that an attacker has a dedicated goal, denoted s^*, while penetrating the system. We also suppose that attackers use the easiest and quickest method for achieving this. In the honeypot automaton there is a shortest path, denoted P^* leading from the `sshd` to this goal (state s^*). The attacker knows a priori that he or she can reach s^* passing on average though $|P^*|$ states. On a standard high-interaction honeypot, the attacker can execute arbitrary commands. Hence, the attacker stays on the shortest path by simply executing the needed commands. However, if the honeypot is interfering with the attacker by strategically blocking commands, substituting his or her commands or writing insult to his or her terminal, the attacker gets distracted. He or she could give up, or accept the challenge by choosing alternative commands or retrying the failed command. When the attacker is disturbed, conceptually he or she is detoured from the shortest path targeting s^* and followed another path, denoted P'. Moreover, we assume that attackers are not always enthusiasts and naively try to achieve their goal with an infinitely large effort. When the attacker believes of getting closer to his or her goal, a higher reward is distributed. If the attacker has not reached the goal within $|P^*|$ transitions, he or she tries a little bit more. If he or she is disturbed too much, we believe that the attacker's interest decreases until he or she gives up. A formal definition of an attacker's reward is presented in eq. 1.

$$r^1 = \frac{|P^*|}{|P'|} \tag{1}$$

2) Honeypot rewards: The honeypot's interests are different of those of the attackers. The honeypot wants to reveal useful information from attackers, including his or her customized tools and his or her skills. Obviously, the honeypot does not know the attacker's target in advance. However, it can measure the reaction time of the attacker or if the attacker entered something new. Assume an attacker provides an input c, intending to execute a command or to insult the honeypot. Let δ be the delay between two successive commands s_i and s_{i-1}, measuring the attacker's response time. The reward for the honeypot is defined in eq. 2 where l is the minimal Levenshtein distance [16] between the provided input c and all the known states (see eq 3). If an attacker makes a typographical error, e.g. `uanme` instead of `uname` (l=1), is a good indication that this attacker is a human being rather than an automated script [9]. On the one hand, if the response time is 0 seconds it is probable that a defective attack script has been launched against the honeypot and the honeypot gets a reward of 0. On the other hand, if the delay was larger than 0 seconds it is highly likely that a human intruder is attacking, which is more interesting than automated attack script. When an attacker enters a regular command, the minimal Levenshtein distance becomes 0, which is incremented by 1, in order to avoid an overall reward of zero. Even in this case, for delays larger than 0, the honeypot manages to keep the attacker busy, which is

positively rewarded. The Levenshtein distance increases when the attacker enters a new input which is probably a new tool or an insult from the attacker yielding a high reward. In this case, the honeypot has learned something new from the attacker.

$$r^2 = \delta \times (l+1) \tag{2}$$

$$\forall s \in S \; l = \min Levenshtein(c, s) \tag{3}$$

C. Learning Honeypots and Attackers

Having defined the two agents, the environment and the rewards, the learning method for each agent is presented in this section. A straightforward method is to integrate Nash equilibrium computations into the expected reward computations of a traditional RL. Hence, Hu et al. proposed Nash-Q [14]. However, there are some open issues concerning such a solution. According to Junling Hu et al. the algorithmic complexity of finding an equilibrium in matrix games is unknown. Bikramjit et al. prove that the minmax-Q and Nash-Q are equivalent in the purely competitive domains [13]. However, the advantage of the minmax-Q algorithm that it is more resource-efficient. The purpose of an adaptive high-interaction honeypot is to incrementally learn the optimal policy for choosing actions in given states. In the context of adaptive honeypots we use the minmax-learning proposed by Banerjee et al. [13] and define a stochastic game between an attacker and a honeypot. As we have two agents, lets denote a player k and $\bar{k}$ the competitor. The parameter γ comes from traditional RL and represents the discounting rate while α represents the learning rate [6]. In a given state at a given time t, s_t, each player performs an action a^k. For each player k at time t a reward r_t^k is distributed and the estimated values are updated according to eq. 4. The estimated rewards and the greedy explorer serve for the self-configuration of the honeypot. If a player is greedy and if positive rewards have been observed for a given state and action, the player decides to take this action.

$$Q_k^{t+1}(s_t, a_t^k, a_t^{\bar{k}}) = \begin{array}{l}(1-\alpha_t) Q_k^t(s_t, a_t^k, a_t^{\bar{k}}) \\ +\alpha_t[r_t^k + \gamma Q_k^t(s_t, a_{t+1}^k, a_{t+1}^{\bar{k}})]\end{array} \tag{4}$$

IV. Implementing an adaptive high-interaction honeypot

A recent trend in the development of high-interaction honeypots is to perform the monitoring of attacker activities from an external point of view. Virtual machines are a popular vehicle for this purpose [3]. However, for the purpose of quick prototyping, User-Mode-Linux (UML) is better suited. UML is a variant of a default Linux kernel that runs in user space on top of a Linux operating system, denoted the host operating system. It differs from a virtual machine because the hardware is not emulated. UML provides functions to communicate with the host operating system and, from a monitoring perspective, the tedious task to implement hardware to kernel space mappings can be avoided. Hence, we transformed UML into a framework where additional adaptation mechanisms, like game theoretical decision making or RL, can be quickly implemented. An overview of this framework, which we call AHA, is shown in figure 3. Attackers are constantly scanning networks for new victims (step 0) [2], [8], [9]. Once they have found the honeypot, they start to penetrate the system (step 1). The Linux authentication module (PAM) was modified to let attackers easily in [2]. The SSH server establishes a command-line shell for the attacker by creating a new process (p_a). When a process is executed, our modified UML puts a message in an output queue (step 2), describing this process. This message is fetched by a daemon called AHAD (step 3). Each time the SSH server establishes a command-line shell for an attacker, the process identifier is memorized and a cloned process tree [17] is maintained in the daemon. When a process identifier belongs to a subtree related to an SSH session (having p_a as root), the daemon knows that this process belongs to an attacker and passes the decision to a customized class implementing a game theoretical approach or RL. By default all processes belonging to the system itself are allowed. Processes related to attackers can be allowed, blocked, substituted, or an insult can be sent to the attacker's terminal. The AHA daemon puts the decision in the input queue (step 4). The modified UML then polls the decision (step 5) and implements it. If a process execution should be blocked, the kernel returns an error message; if the process is allowed, the regular code of UML is executed. When the attacker is insulted or if the program is substituted, the process' arguments are swapped in kernel space. Execution performance is a critical aspect. We empirically determined that a process execution cannot be suspended longer than 50ms. If a process invocation is suspended for too long, the system becomes unusable and unstable. Therefore, the AHA daemon just focuses on taking decisions. When the CPU is idle, the program Aha_worker merges the exchanged messages in the queues into a log file and cleans the queues. The program Aha_eye uses this log file and is capable of determining the sequences of commands an attacker entered during an SSH session.

V. Experimental Results

In a first step an adaptive high-interaction honeypot was set up for 48 days with the purpose of recovering traces from attackers. In a second step the adaptive honeypot is compared with a plain high-interaction honeypot. The recovered traces are the ground truth for learning evaluation and serve for setting up the honeypot automaton. The honeypot randomly changed its behavior regarding the execution of programs by attackers. The choices of selecting actions followed a uniform distribution and it is assumed that attackers respect the semantics of commands in order to have meaningful transition probabilities among states. The developed software and the experimental datasets are freely available in our git repository [18]. The traces contain 47 states including programs installed during the setup of the honeypot. Programs installed by attackers are mapped to the state u^* and insults to the states $u_{1..n}$ which are mapped to computed minimal Levenshtein distances with respect to installed programs. We observed 4360 different

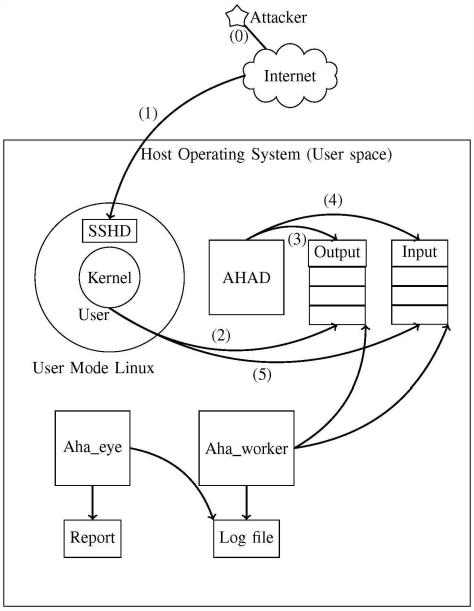

Fig. 3. Adaptive Honeypot Alternative - AHA

transitions between programs. In addition, the delays between two successive programs were recorded, serving as reward for the honeypot. Attackers did also typographic errors or typed 1183 insults. A manual analysis of the insults shows that 17% are typographic errors. For 28.5% we identified the used language. Attackers used English, Romanian, Croatian, Polish and German. Some attackers showed a sense of humor and replied with a smiley (5%). We also observed Romanian insults coming from French, German and Spanish IP addresses. Hence, it is highly probable that these hosts have been compromised serving as bounce for further attacks. Attackers installed 61 different programs on the honeypot and made 695 transitions to their own program. It was also observed that different attackers often installed the same programs on the honeypot. In order to recover the payoffs for the attackers, the last program execution was recorded where the honeypot allowed all the transitions. This means that the honeypot had not interfered with the attacker and we assume that the attacker reached their goal. The observed delays and minimal Levenshtein distances permit us to compute the rewards.

In the following we addressed how fast the learned Q values stabilize. During the experiment no discounting factor was used ($\gamma = 1$) with the purpose of arriving at a worst case scenario for stabilization. By selecting $\gamma \ll 1$, the values should stabilize faster [6]. Due to space reasons only a few relevant states are represented in figures 4, 5, 6 and 7. The discrete time t is shown on each x-axis, while the y-axis gives the learned Q value. The state `ls` is a typical system command for listing files and sometimes mandatory for performing an attack, but in general this command is not dangerous assuming that there are no confidential files on the honeypot. Hence, the attackers and the honeypot have similar interests when this command is allowed. In figures 4 and 5 the Q value evolution for both players is represented. When an attacker wants to continue the attack, this command should be allowed. Sometimes, the honeypot also insulted the attacker or substituted the command `ls` due to the ϵ-greedy explorer, but the respective learners noticed that these actions are not suited for this state. However, sometimes the interests of an attacker and the honeypot diverge. For instance, this is the case for the tool `wget`. This tool is frequently used to download arbitrary files and is often one of the last steps of an attack. Thus, when an intruder wants to continue the attack, the best choice from the honeypot is to allow this command (figure 6). If the honeypot substitutes the execution of this program, an attacker usually finds alternative commands for installing the desired customized tool. From the honeypot's perspective it is better to substitute the execution of the program `wget` in order to keep the attacker longer active (figure 7).

For each command entered, the honeypot needs to decide whether to allow this command to block the program execution, to substitute the output of the program or to insult the attacker. Q values are then recorded for each player. Figures 8 and 9 show the average Q values, represented on the y-axis, for the honeypot in different states, shown on the x-axis. Rewards are distributed retroactively. This means that an attacker is at state s_i and wants to move to state s_{i+1}. The honeypot reacts by choosing the action a_j. The state transition is made or not made according to the action a_j. After this move the rewards are distributed for each player for the state s_i. The learning spawns over multiple SSH sessions. In figure 8 a subset of states {`id`, `sh`, `bash`, `unset`} can be identified, where the highest averaged Q value occurs when the attacker retries a command. The commands in the set are usually common system commands and when the attacker arrives at these states, the attack is usually continued by retrying successive commands. For instance, the attacker has a shell meaning that he or she is at the `bash` state and he or she wants to execute `wget`. Even when this transition is blocked, the attacker can still continue the attack by retrying the command. The situation is different for the `insult` state. Attackers sometimes reach the insult state u_n when they make a typographical error or when they enter an unknown command. When the honeypot returns an error like command not found, the attacker assumes that the program is not installed on the system and tries to use a different program. Hence, the highest average Q value results in the selection of an an alternative path. Figure 9 presents the average Q values when the honeypot substitutes the command. The programs `w`, `uname` and `uptime` are first commands entered by attackers in order to identify the system that has been compromised. In our traces, the command `w` is often the first command entered. When the honeypot allows the transition to this state and then starts substituting successive commands like `id`, `uptime` the forged output is often not consistent with the output of the program `w`. We assumed that some attackers then realized that they were connected to a fake shell, identified the honeypot and started to type insults in the terminal, inducing a large Levenshtein distance leading to a high reward.

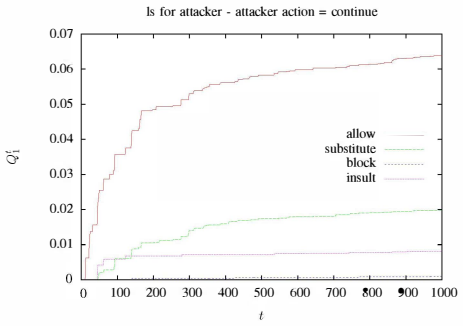

Fig. 4. Q value evolution for the state ls from the attacker's perspective

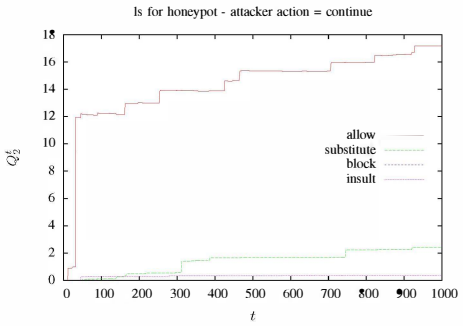

Fig. 5. Q value evolution for the state ls from the honeypot's perspective

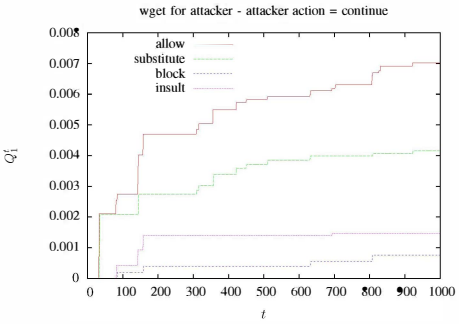

Fig. 6. Q value evolution for the state wget from the attacker's perspective

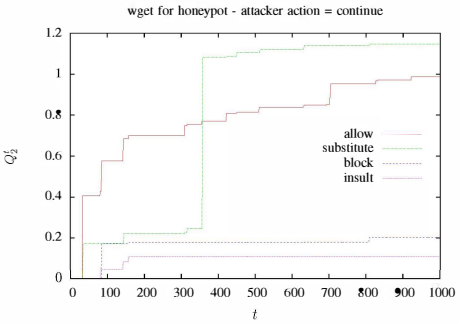

Fig. 7. Q value evolution for the state wget from the attacker's perspective

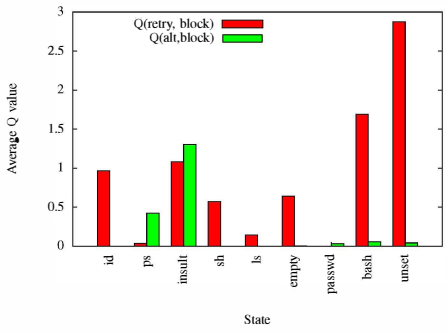

Fig. 8. Impact when the honeypot blocks transitions

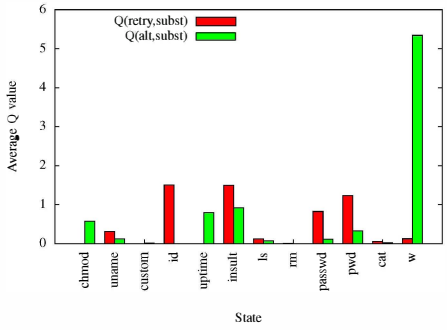

Fig. 9. Impact when the honeypot substitutes program executions

We implemented a fast concurrent learning module for our AHA framework, which is publicly available [18]. The resulting honeypot was operated for 8 hours in a controlled environment and 15 attacks have been observed. The average attack duration is 260 seconds on the honeypot, which confirms the average attack duration of 238 observed during the experiments driven by traces. In order to do a more fine grained analysis, a larger dataset is needed for doing a meaning full comparison with a high-interaction honeypot, but these preliminary results already confirm that our approach can be implemented and operated.

VI. Related Work

In the late eighties Stoll introduced the concept of providing a fake infrastructure serving as trap for attackers [19]. A little later Bellovin discussed the concept of throw-away machines with real security holes for monitoring and tracking attackers [20]. Spitzner defined honeypots as resources designed to be attacked [1]. From an interaction point of view, honeypots are divided into three categories, namely low-interaction, mid-interaction and high-interaction honeypots. Machines with real security holes are considered as high-interaction honeypots and attackers can execute arbitrary commands on them. From an operational aspect these honeypots are the most difficult to manage, due to the large universe of actions an attacker can perform. When mid-interaction honeypots are used, this universe is restricted by emulating a set of known vulnerabilities. Nepenthes [21] is a popular tool for collecting self-

spreading malicious software. The added value of this tool is to collect malware and not to study the vulnerabilities. The safest honeypot is a low-interaction honeypot. In this case connections from attackers are simply logged. Attackers can only connect to this service but they cannot push shell code or execute arbitrary commands. However, few efforts have been undertaken to make honeypots intelligent or adaptive. Tillmann developed a tool called honeytrap, which passively listens for connections and, if it sees a connection attempt, dynamically associates a service and mirrors back already-received data. [22]. Alata et al. [9] deployed a high-interaction and reported some observations about attacker behaviors. They identified different attacking phases. Attackers assault a system account and then come back after a while. If they are still able to get into the system they start their malicious activities like SSH brute-force scanning or IRC bouncing. In both cases the attacker's purpose is to misuse the honeypot in order to hide connections. They also noticed that some attackers made some typographical errors and they assumed that these errors are due to human attackers. However, we also take into count the delay between two successive commands aiming to differentiate between human attackers and faulty automated scripts. Most of the honeypot activities focused on detection and evasion techniques related to honeypots [3], [23]. Lye et al. considered game theory. They defined a game between attackers and network administrators spawning over multiple machines [24]. However, our game is played only one machine and at operating system level.

VII. Conclusion and Future Work

In this paper we have presented a self-management paradigm for high-interaction honeypots. We have leveraged fast concurrent learning in stochastic games as a conceptual building block for self-configuring honeypots. We consider self-configuration equivalent to learning strategic decisions facing attackers. We have framed the self-configuration within a stochastic game, where the honeypot learns state-specific reactions. A reaction is a choice from a set of possible actions which can be allowing, blocking or substituting a command or insulting the attacker. We have equally assessed learning what an attacker can do in order to accommodate a broader model in which an attacker can also be adaptive. We have implemented and deployed a prototype in order to establish a ground truth and experimented using a minmax Q algorithm in order to configure and validate our contribution. As far as we know, our paper is the first that describe an adaptive honeypot and its attackers. We have focused in this paper only on the defensive aspect but we are convinced that automated penetration and assessment tools can leverage our formalism in order to improve attack efficiency. Further research is needed, to improve the state representation. Experienced attackers could replace states on the honeypot and poison the learning process. Our honeypot model could also be extended by respecting system command semantics. More complex competitive learning algorithms could be explored, taking into account multiple attackers that may collude.

References

[1] L. Spitzner, *Honeypots: Tracking Hackers*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2002.

[2] G. Wagener, R. State, A. Dulaunoy, and T. Engel, "Self adaptive high interaction honeypots driven by game theory." in *SSS*, ser. Lecture Notes in Computer Science, vol. 5873. Springer, 2009, pp. 741–755.

[3] X. Jiang and X. Wang, ""Out-of-the-Box" monitoring of VM-based high-interaction honeypots," in *RAID'07: Proceedings of the 10th international conference on Recent Advances in Intrusion Detection*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 198–218.

[4] S. Smalley, C. Vance, and W. Salamon, "Implementing SELinux as a Linux Security Module," NAI Labs, NAI Labs Report #01-043, Dec 2001. [Online]. Available: http://www.nsa.gov/selinux/doc/module.pdf

[5] M. Bauer, "Paranoid penguin: an introduction to Novell AppArmor," *Linux Journal*, vol. 2006, pp. 13–17, 2006.

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

[7] J. Dike, *User Mode Linux*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2006.

[8] D. Ramsbrock, R. Berthier, and M. Cukier, "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, 2007, pp. 119–124.

[9] E. Alata, V. Nicomette, M. Kaaniche, M. Dacier, and M. Herrb, "Lessons learned from the deployment of a high-interaction honeypot," in *Dependable Computing Conference, 2006. EDCC'06. Sixth European*, 2006, pp. 39–46.

[10] A. Greenwald, "Matrix Games and Nash Equilibrium," 2007, lecture.

[11] J. Goeree, C. Holt, and T. Palfrey, "Regular Quantal Response Equilibrium," *Experimental Economics*, vol. 8, no. 4, pp. 347–367, December 2005.

[12] L. P. Kaelbling, M. Littman, and A. Moore, "Reinforcement Learning: A Survey," *Journal of Artificial Intelligence Research*, vol. 4, pp. 237–285, 1996.

[13] B. Banerjee, S. Sen, I. Sen, and J. Peng, "Fast Concurrent Reinforcement Learners," in *In Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, 2001, pp. 825–830.

[14] J. Hu and M. P. Wellman, "Multiagent Reinforcement Learning: Theoretical Framework and an Algorithm," in *In Proceedings of the Fifteenth International Conference on Machine Learning*. Morgan Kaufmann, 1998, pp. 242–250.

[15] A. L. Coates, "Pessimal Print: A Reverse Turing Test," in *ICDAR '01: Proceedings of the Sixth International Conference on Document Analysis and Recognition*. Washington, DC, USA: IEEE Computer Society, 2001, pp. 1154–1159.

[16] G. Navarro, "A guided tour to approximate string matching," *ACM Comput. Surv.*, vol. 33, no. 1, pp. 31–88, 2001.

[17] R. Love, *Linux Kernel Development (2nd Edition)*. Novell Press, 2005.

[18] G. Wagener, "AHA Source Code." [Online]. Available: http://git.quuxlabs.com/?p=aha-linux-2.6/.git;a=summary

[19] C. Stoll, "Stalking the wily hacker," *Commun. ACM*, vol. 31, no. 5, pp. 484–497, 1988.

[20] S. M. Bellovin, "There Be Dragons," in *Proceedings of the Third Usenix Unix Security Symposium*, September 1992, pp. 1–16. [Online]. Available: http://www.cs.columbia.edu/~smb/papers/dragon.pdf

[21] P. Baecher, M. Koetter, M. Dornseif, and F. Freiling, "The Nepenthes platform: An efficient approach to collect malware," in *In Proceedings of the 9 th International Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer, 2006, pp. 165–184.

[22] W. Tillmann, "Honeytrap."

[23] B. McCarty, "The Honeynet Arms Race," *IEEE Security and Privacy*, vol. 1, no. 6, pp. 79–82, 2003.

[24] K.-w. Lye and J. M. Wing, "Game Strategies in Network Security," in *Proceedings of Foundations of Computer Security Workshop 2002*, 2002.