# Self Adaptive High Interaction Honeypots Driven by Game Theory

Gérard Wagener[1 2], Radu State[1], Alexandre Dulaunoy[2], and Thomas Engel[1]

[1] University of Luxembourg
{radu.state,thomas.engel}@uni.lu
[2] SES S.A.
{gerard.wagener,alexandre.dulaunoy}@ses.com

**Abstract.** High-interaction honeypots are relevant to provide rich and useful information obtained from attackers. Honeypots come in different flavors with respect to their interaction potential. A honeypot can be very restrictive, but then only a few interactions can be observed. If a honeypot is very tolerant though, attackers can quickly achieve their goal. Having the best the trade-off between attacker freedom and honeypot restrictions is challenging. In this paper, we address the issue of self adaptive honeypots, that can change their behavior and lure attackers into revealing as much information as possible about themselves. The key idea is to leverage game-theoretic concepts for the configuration and reciprocal actions of high-interaction honeypots.

## 1 Introduction

Simulating failures in order to lure attackers was reported for the first time in the classical paper "An Evening with Berferd" [1], where manual interactions from a human system administrator lured an attacker into revealing many of his tactics and tools. During the operation of an high-interaction honeypot we observed that attackers follow a dedicated goal. We manually interfered with the tools installed and operated by attackers and noticed that some attackers connected back to the honeypot and tried to solve the problems. Some attackers even tried to harden the system aiming to lock out other attackers. Thus, we assume that attackers are rational and follow a specific goal during attacks. We address in this paper a first step towards an automated failure injecting honeypot aiming to disclose as much information as possible about an attacker. According to Lance Spitzner, a honeypot is a resource dedicated to be attacked [2]. Honeypots are frequently used to monitor or lure attackers and serve as baits for attackers. Once honeypots are compromised, attackers can be traced and attacking techniques can be learnt. On the one hand, if a honeypot has to limited capabilities some attack goals can not be reached and not much can be learnt. On the other hand, if a honeypot exposes to many and easily accessible features to an attacker, the attack goal can be easily reached and only a part of the attack can be observed. The goal of an attacker is also often unknown. The challenge addressed in this work is to elaborate an adaptive high-interaction honeypot that

tries to optimize the retrieval of knowledge from an attacker. The level of interaction is a consequence of the capabilities of a honeypot. The more features are implemented in a honeypot, the more interactions are possible between attackers and the honeypot. One way to obtain more interactions, is to partially allow attackers to execute some programs, leading them to explore alternative execution paths and reveal more information about themselves (attack tools) and to disclose other repositories, used for malicious purposes. Similarly, an adaptive honeypot can abnormally terminate the execution of programs by an attacker and lead the attacker to perform other activities, that can provide insightful information to the security community. We address in this paper, the design, implementation and validation of adaptive high-interaction honeypots. The major research challenges that we had to address were:

- to design an adaptive behavior for a honeypot that should be optimal and remain stealthy.
- to implement an effective Linux kernel monitoring solution capable to trace attacker activities on a system.

The remaining paper is organized as follows: Section 2 explains our high-interaction honeypot model. Section 3 formally describes the game between attackers and the honeypot and shows that our honeypot model can be fed with data delivered from a deployed high-interaction honeypot. Our experiments are shown in section 4 and the state of the art activities are summarized in section 5. Finally, the article is concluded in section 6 and future work activities are announced.

## 2 Modeling a High-interaction Honeypot

We consider high-interaction honeypots operating a Linux operating system exposing a SSH server to attackers. The rationale behind this choice is that attackers often use specialized tricks or side effects to detect or evade from low-interaction honeypots and that SSH is a popular attack vector for attackers [3], [4]. We model high-interaction honeypots with a hierarchical probabilistic automaton. This model, as detailed in the following, is needed to frame the honeypot capabilities in the context of game theory.

### 2.1 Honeypot Hierarchical Probabilistic Automaton

Probabilistic automata are often used in the field of pattern recognition [5]. An attacker can connect to a high-interaction honeypot and can execute programs. Downloads can be performed with tools like `wget, curl, ftp`, archives can be extracted with programs like `tar` and `gzip` and so on. We define the states of the automaton as the programs that can be executed on the honeypot. Moreover, we add a state labeled *unknown* in order to describe the fact that new and unseen tools could be installed and used later on. Each program has some program arguments that are passed as array to the `main` function. If no command line arguments are explicitly passed to the program, the first command line argument corresponds to the program name [6]. Moreover, a program can have the same

command line argument than another one but with a different semantic. Thus, we introduce a hierarchy between programs and command line arguments. Each program is formalized as automaton where each state represents a command line argument. The states in an automaton representing a program are called macro states and each macro state contains micro states (i.e. the command line arguments). Some transitions between programs or command line arguments are more likely than others. The program `wget` is often executed previously to the program `tar`. Therefore, each transition can be modeled using a transition probability. The same notation as proposed by Thollard et al. [5] is used. The set $Q_A$ contains the programs installed on the honeypot including an *unknown* state and the set of states for a given program is denoted $Q'_A$. Attackers penetrate the honeypot through the SSH server. Thus, the initial probability for the state */usr/sbin/sshd* is 1 and 0 for all the other states. Moreover the alphabet consists of the commands executed by the attacker. An example of such a hierarchical probabilistic automaton is shown in figure 1[3]. An attacker connects to the honeypot via SSH and stays in the `sshd` state. Next he or she can execute the program `bash` or `uname` with the equal probability of 0.5. After the execution of the program `bash`, the programs `wget`, `rm` and `uname` have the same likelihood to be executed, namely 0.25. The program `ls` is executed with the command line arguments `-la`, `-l` with an equal probability of 0.5.

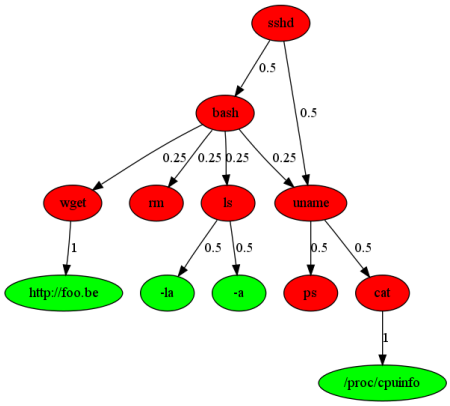


**Fig. 1.** Honeypot hierarchical probabilistic automaton example

### 2.2 Process Vectors

The transitions between programs are described by the conditional probabilities, capturing the likelihood of one program being executed after a

[3] For the purpose of understanding a simplified automaton is presented.

previous one. We use sequences of program executions from a deployed high-interaction honeypot to determine these probabilities. Such a sequence of programs is considered as process vector which is observed from one attack and where each element is a program that is executed during an attack. An attacker who executes the programs `/bin/bash`, `/usr/bin/wget` and `/usr/bin/tar`, generates the process vector $< /bin/bash , /usr/bin/wget , /usr/bin/tar >$.

### 2.3 Attacker Process Trees

In order to obtain the process vectors, we have to dig into the kernel data structure (on the honeypot) holding process tree information. After having compromised the honeypot, an attacker usually executes programs. Such an execution triggers a `clone` or `do_exec_ve` system call which should be monitored. Multiple attackers can be connected to the honeypot at the same time and the operating system itself is using `do_exec_ve` and `clone` system calls. The system calls that are related to a given attack can be identified as follows: In a Linux operating system each process has a process identifier (PID) and a parent process identifier (PPID) [7]. An attack usually starts with a privilege separated process of the SSH server [8], denoted $p_0$. The process $p_0$ then forks, resulting in a `clone` system call or directly executes a program via the `do_exec_ve` system call. We consider that the process $p_0$ executes a program and creates another copy of the process, denoted $p_1$. The parent process of $p_1$ is thus $p_0$ and the result of the execution of a sequence of programs is a process tree of an attack which is a subtree of the Unix process tree on the honeypot. We define a process tree as a tree structure where each node can contain a process id, a timestamp, a program name or a command line argument resulting from a `do_exec_ve` or clone system call. An edge links two process identifiers with each other, which represents the parent child relationship. Furthermore, in a process tree, each parent of a leaf represents a program name and each leaf represents command line argument (at least the program name [6]).
One process tree is shown in figure 2. The privileged separated process of the SSH server has the process identifier 4121 and is the root of the tree. Two `clone` system calls are done; one results in a process with the process identifier 4127 and another one in the process identifier 4129. The process with the identifier 4127 is created after one second and the process with the identifier 4129 is created after 3 seconds. Then the process with the identifier 4127 executes a program called `/bin/bash` after one second and the process with the identifier `4129` starts the `/bin/uname` program after 5 seconds. The program `/bin/uname` is started with the argument `-a` and the command line arguments `bash` and `uname` represent the respective program names.

### 2.4 Inducing a Honeypot Hierarchical Probabilistic Automaton

We model the honeypot capabilities as a hierarchical probabilistic automaton where each state represents a program. Each state is further-

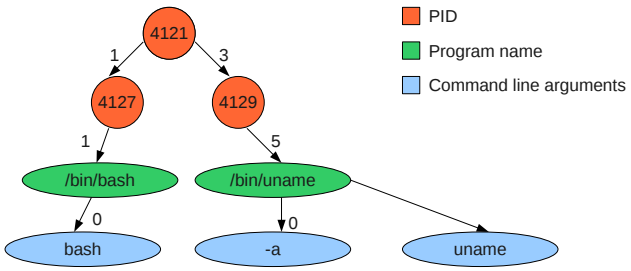


**Fig. 2.** Process tree example

more an automaton on its own, where combinations of command line arguments build the states of the sub-automaton. From a deployed high-interaction honeypot we have extracted the process trees related to an attack. A process tree can be composed of PID nodes, nodes containing the programs that were executed and nodes modeling command line arguments. Due to the fact that the process identifiers change from one attack to another, we are interested to transform these process trees in process vectors describing the sequences of programs that were executed during an attack. The order of program execution is important. A good example is when a tool was downloaded that is then extracted and executed. To recover the order we use the timestamps in the process trees. The time difference of each leaf with the root enables us to determine the position of a program in the process vector. In the example shown in figure 2 the process vector $\overrightarrow{v}$ is $< /bin/bash, /bin/uname >$ because the program `/bin/bash` was executed before the program `/bin/uname`. Each attacker generates a process tree that is converted to a process vector. All these vectors are now inserted in a two dimensional matrix transition matrix. The observed programs are used as labels for the columns and rows respectively. Each cell contains the frequency of how often a couple of programs was observed. The transition probability $P_A$ is computed from the transition matrix. Each cell is divided by the sum of the row. The automaton containing the macro states is created from the transition matrix. In figure 1, each state is represented by a circle and the edges are labeled with the transition probability. For instance, a transition from the macro state `sshd` can be done to the macro state `bash` with a probability of 0.5. Another transition can then be done to the macro state `ls`. The program `ls` can operated in different modes by accepting different command line arguments. In this example the states denoted by "l" and "-la" are micro states and belong to the automaton `ls`. First, the hierarchical probabilistic automaton may be incomplete because it is constructed from honeypot observations. Therefore, we integrate a state in the automaton which is called "unknown". Second,

rare transitions may be unobserved. To counteract this phenomenon we smooth the probabilities that we derived from honeypot observations, where the smoothing factor is denoted $\epsilon$. In this case each probability $> 0$ is multiplied by $(1-\epsilon)$ and from a given state, transitions are created to all other remaining states. If we assume that our automaton has $N$ states and the number of transitions for a given state is $n$, then $N-n$ transitions are created having the probability $\frac{\epsilon}{N-n}$. The automaton has now $N^2$ transitions and is able to capture all possible transitions. In practice, the automaton could be periodically induced in order to adjust the transition probabilities.

## 3 The Honeypot Game

Intuitively, the fact that attackers connecting to the honeypot can be seen as game between the honeypot and the attackers. We assume that attackers try to achieve their goal as fast as possible. They want to minimize the number of interactions with the honeypot. The honeypot aims to maximize the number of interactions or to learn as much as possible from attackers or to distract them as long as possible from real assets. In this article we define two possible actions for the honeypot and three different strategies for an attacker. We determine Nash Equilibriums [9], providing the optimal strategies for both the attacker and the honeypot.

### 3.1 Modeling Attacker and Honeypot Actions

Our current adaptive honeypot can accept or block the execution of a program which is implemented by allowing or blocking the `do_exec_ve` system call in a Linux kernel [7].

**Block a `do_exec_ve` system call.** The honeypot can crash a tool of an attacker which can be derived from blocking the `do_exec_ve` system call. In that case an error code is immediately returned instead of executing the regular code of the `do_exec_ve` system call. Let $Pr(Block)$ be the probability that the adaptive honeypot blocks the system call `do_exec_ve`. This decision is taken at each `do_exec_ve` system call[4].

**Allow a `do_exec_ve` system call.** The honeypot behaves like a normal high-interaction honeypot. The probability to allow a system call is $1 - Pr(Block)$.

Attackers often find out that the honeypot is not immediately ready for their malicious activities. Thus, they download their tools, install them and execute them. They download tar balls containing a pre-compiled version of their program or the source code is downloaded and compiled on the honeypot. In both cases attackers often configure their programs on the honeypot. All these actions results in interactions with the honeypot. In our hierarchical probabilistic automaton, the interactions with the honeypot result in transitions from one state to another one. We assume that attackers are rational and that they select the next transition

[4] Practically, an explicit error code of the system call could be returned.

on the most probable path in the automaton. In the game between attackers and the honeypot we define three actions for attacker when their transitions are blocked.

**Retry of a command.** Attackers can retry a command from a failure. First a failure might be due to a syntax error. The second reason might be a timeout that emerged during the program execution. For instance, an attacker may to download a file and a network timeout may emerge. In this case another repository might be disclosed. Third, the execution of a program might produce an undesired effect. A wrong command line argument might have been used. The program is executed again with a different command line argument. Let $Pr(Retry)$ denote the probability that attackers execute the same command again.

**Select an alternative solution.** A downloaded program may fail during execution. Some attackers try to debug the problem on the honeypot. They can check the configuration file of the program or run an inspection tool like `strace` on the program. They might try to download another program or to download the source code of the program that will be compiled on the honeypot. No matter which option they select, their behavior can be classified in a category describing the actions of choosing an alternative solution for obtaining their goal. Let $Pr(Alternative)$ denote the probability that attacker select an alternative command to achieve their initial goal.

**Quit.** Some attackers check the capabilities of the honeypot and if they suspect a trap or a worthless system, then they will leave. Let $Pr(Quit)$ describe the probability that attacker quit.

The relation 1 holds for the attacker strategies.

$$Pr(Quit) + Pr(Retry) + Pr(Alternative) = 1 \tag{1}$$

An example of attacker and honeypot strategies is shown in figure 3. We observe that an attacker tries to invoke the command `nmap` (a popular network scanner). The honeypot might allow the execution (with the probability $1 - Pr(Block)$) and in this case the attacker continues and executes the program `wget` (with a probability of 0.95). If the tool `nmap` is not allowed by the honeypot, the attacker can decide to either quit (with a probability of $Pr(Quit)$) or to retry the execution of `nmap` or to execute another command (for instance `uname` - with a probability of 0.6). The execution of `nmap` was blocked and its probability was equally distributed among the transitions to the states `wget` and `uname`. The probabilities used by attackers to choose the next command to be executed can be estimated from an operational high-interaction honeypot. The probabilities used by the honeypot to block the execution of a command is a configuration setting and reflects the strategy played by the honeypot. Similarly, the probabilities used by the attacker to either quit the session, or retry a command (and consequently to choose another command) give the strategy played by the attacker.
The main question is related to what are the optimal settings for both the honeypot ($Pr(Block)$) as well as for the attacker ($Pr(Quit)$, $Pr(Retry)$, $Pr(Alternative)$).

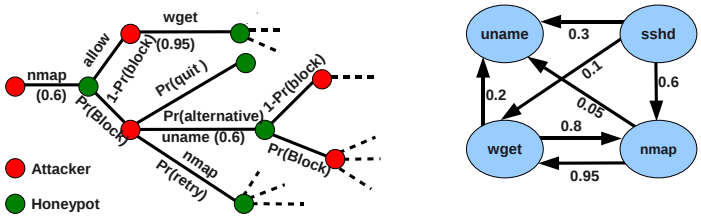


**Fig. 3.** Honeypot game example

### 3.2 Modeling Attacker and Honeypot Games

We reuse the definitions and notations proposed by Amy Greenwald [9] in order to formally describe our games between attackers and the honeypot. The game between the attacker and the honeypot has two players. Thus, $N = \{honeypot, attacker\}$. The honeypot can block `do_exec_ve` system calls with different probabilities. The set $A_h$ corresponds to the set of blocking probabilities the honeypot can choose. An attacker can choose to retry a command, to search for an alternative command or to leave. We define an attacker strategy with a 3-tuple ($Pr(Retry)$, $Pr(Alternative)$, $Pr(Quit)$) and the set $A_a$ contains all these strategies. One purpose of game theory is to compute the optimal strategy profiles for the players which results in the computation of Nash Equilibrium. A Nash equilibrium in the context of honeypot game means that neither the honeypot nor the attacker can increase their expected payoffs assuming that neither player does not change his strategy during the game.

**Computing Payoffs** Respective to attacker and honeypot strategies we propose two honeypot games. The games are different with respect to the payoff computation. We propose the following payoff computations[5]:

**Number of transitions.** We assume that attackers are rational and that they want to achieve their goal as fast as possible. Thus, an attacker tries to minimize the number of transitions in the hierarchical probabilistic automaton. The honeypot tries to learn as much as possible from the attacker. Potential useful information for a honeypot is to collect tools owned by attackers or to discover the sources where they are downloaded from. Hence, the honeypot tries to maximize the number of transitions performed by an attacker. The payoff for the honeypot $R_h^t$ returns the number of transitions performed by an attacker. The more transitions an attacker does, the better it is for the honeypot. Attackers try to minimize their transitions and their payoff function returns $-1$ multiplied by the number of transitions. The less transitions attackers do, the less they are punished in terms

[5] which can be setup on two different honeypots

of payoff. This game seems at first glance unfair regarding attackers. If we assume that attackers want to be undiscovered while they are doing their attack, they have already lost because they connected to a monitored honeypot instead to real assets. The only chance they have is to divulge less information as possible and thus try to minimize the number of transitions.

**Path probability payoff.** The payoff computations purely based on the state transitions ignores the fact whether attackers reached their goal or not. Moreover, the payoff should take into account how likely a path is regarding observations from a deployed honeypot. We are looking for a payoff computation that rewards the honeypot for blocking and that penalizes the attacker when being blocked.

The payoff for the honeypot is shown in definition 2 and the payoff for the attacker is presented in definition 3. The probability $Pr(path)$ denotes the probability of the path the attacker has chosen and $Pr^*(path)$ denotes the probability of the most probable path from the source to the selected destination by the attacker.

$$R_a^p = \frac{Pr(path)}{Pr^*(path)} \tag{2}$$

$$R_h^p = 1 - \frac{Pr(path)}{Pr^*(path)} \tag{3}$$

The more the path probability gets close to the most probable path probability, the payoff for the attacker converges to 1. In this case, the payoff of the honeypot gets close to 0 which is the minimum payoff for the honeypot. If the path of the attacker is diverted due to blocked programs, the path probability chosen by the attacker diverges from the most probable path probability and gets lower than the most probable path probability. Hence, the payoff gets minimized for the attacker and maximized for the honeypot.

### 3.3 Computing Payoffs with Simulations

In order to compute the payoff values for all possible combination of strategies, we use a Monte Carlo simulation. We have built a simulator that uses bootstrap data obtained from an operational honeypot deployed over a period of 3 months. Due to computation and deployment constraints we are forced to do simulations since, doing real world experiments for all possible behaviors would require 2684 different honeypots setups.

**Honeypot Simulator** Attackers and the honeypot select their strategies according to a given discrete probability. These probabilities are fixed and an attack is simulated. The simulation provides the number of transitions an attacker did, the optimal path probability, the path probability for the attacker, the fact that the attacker left and the fact that the maximum number of transitions was reached. The variable *src* specifies the initial state and the variable *dst* stands for the destination state in the

hierarchical probabilistic automaton. Hence, the final state probability, required by the automaton, is 1 for the state *dst* and 0 for all the other states. During a simulation, the transitions performed by an attacker are recorded. The states, that an attacker passed through are kept in a list. When the simulation starts, the attacker enters the initial state. We assume that an attacker chooses the next transitions on the most probable path. If the attacker is not blocked, the attacker follows the same path. An attacker has a fixed goal. If this goal is reached then the simulation ends. ($src = dst$). Moreover the number of transitions during a simulation is recorded and if this number exceeds a defined threshold the simulation ends because we want to avoid endless transitions. The attacker can retry a command or compute the next state and the step is recorded. The honeypot decides to block or allow this step according to the probability $Pr(Block)$. The attacker now decides whether to quit or continue the game according the probability $Pr(Quit)$. If the attacker quits, the simulation ends. If the attacker decides to choose an alternative command, the hierarchical probabilistic automaton is modified due to implementation issues. The probability for the blocked transition is set to 0 and the probability for this transition is equally distributed for all outgoing transitions. An attacker always computes the most probable path and the same path could not be selected due to the 0 probability transition. Of course this effect is undone for the next simulation round. If the attacker decides to retry a command the state *alternative* is set to false, the loop ends and the next round starts.

## 4 Experimental Evaluation

We set up a high-interaction honeypot capable to record `do_exec_ve` and `clone` system calls. We directly patched the Linux kernel in order to avoid a detection by address arithmetic which is an attack described by McCarty [10]. We transmit the collected data in kernel space directly to the hardware level in order to avoid that collected information passes through the hands of the attacker. The honeypot is operated with the Qemu a x86 emulator [11]. The kernel inside the Qemu was modified such that process ids are logged. On the host machine this data is put in a database. The honeypot has also an additional network interface where system logs are transmitted to a syslog-ng server like it is the case for current production systems. The default running service is a SSH server which serves as entry point for attackers. We could configure the SSH server that the PAM module `pam_permit` should be used. In this case no password is asked, which may be very suspicious for attackers. Thus, we preferred to modify the `pam_unix` module, which is responsible for password authentication in a Linux operating system. With our patch, the system asks for a password but then neglects all non-privileged user passwords. This implementation choice is also resistant against password changes performed by attacker, because the password is not checked anymore. In theory an attacker could also change the PAM modules but we did not observe this phenomenon during the operation of our honeypot. Moreover, we observed that some attackers installed their own

shell in order to be sure that they do not use a shell with additional monitoring features. Furthermore some attackers replaced the SSH server on the honeypot. An alternative solution is to perform a MITM attack in order to filter the command executed by attackers. However, from an engineering perspective this solution requires additional efforts to become stealthy. From this honeypot we recovered the process trees related to attackers which are sub trees of the Unix process tree on the honeypot. Then we transformed these process trees in process vectors. Each vector corresponds to an attack. From the observed process vectors we created a hierarchical probabilistic automaton to drive the simulation. Our data sets and developed software are publicly available[6].

### 4.1 Data Sets

The honeypot was operated on one public IPv4 address and consisted of a Ubuntu Linux 7.10 operating system. The Linux operating system was executed in a virtual machine operated by Qemu, version 0.9.1. We patched the `pam_unix` module, version 0.99.7.1 in order to facilitate access to the attackers and to mitigate the effects of an attacker that changed the password of a compromised account. We extended the Linux kernel, version 2.6.28-rc6 with the `do_exec_ve` and `clone` monitoring features. The honeypot was operated from 2009-01-21 until 2009-03-09. In this period we observed 637 successful ssh logins and 12140 ssh failures. Despite the patched `pam_unix` module, a high number of ssh failures was discovered. Our `pam_unix` module patch lets the pam_unix module ignore passwords for non privileged user accounts on the honeypot. For 61% of the failed ssh attempts the root account was targeted which was explicitly blocked by our `pam_unix` module patch. Besides the 13 system accounts, we created 12 additional user accounts. Thus, we have 25 non privileged user accounts. Attackers tested 1763 non existing accounts with different passwords which is another explanation for the high number of SSH failures. For the successful logins we observed 183 different IP addresses. Some attackers modified the kernel but the virtual machine was configured in such a way that a reboot was translated into a power off. The kernel changes are noticed because the file system of the honeypot was periodically mounted (loop back) and checksums were computed to detect changes. If the kernel was changed we replaced the modified kernel with the original.

**Process Trees** We recovered 637 process trees. The root of each process tree was the privileged separated process by sshd. The smallest trees have only one node and the tree with the maximum nodes had 1954 nodes. The small trees can be explained due to the fact that a brute force attacks against the SSH server was performed by some attackers with automated tools. The automated tool managed to break into the honeypot and immediately left. The maximum length of a process tree is due to bots that were installed on our honeypot. The bot master had

[6] `http://quuxlabs.com/~gerard/jogy-experiment`

long sessions with the bot in order to operate it. Due to data processing capabilities we stopped to reassemble the tree if the length is longer than 100 nodes. The average number of nodes per process tree is 105 with a standard deviation of 231.

**Process Vectors** Each process tree was converted in a process vector aiming to extract the program sequences done by an attacker. The longest process vector is composed of 85 programs and the smallest one contains only 1 program. The average process vector length is 6.16 with a standard deviation of 2.81.

### 4.2 Simulation Results

The hierarchical probabilistic automaton was set up using the process trees. We obtained 91 different programs (states). Each program is on its own an automaton based on the command line arguments. To simplify the automaton, we removed the first command line argument which corresponds to the program name in a Linux operating system. On average, programs have 9.72 command line arguments. The program with the most observed command line arguments has 181 arguments and some programs have one program argument. The standard deviation of the program arguments per program is 23.5. A large number of command line arguments can be explained by substitutions done by the program `bash` [12]. For instance the argument `*` is substituted by the program `bash` with a file list in the current directory. Moreover the hierarchical probabilistic automaton contains 581 different transitions. To model unknown or unseen transitions we smoothed the transition probabilities. Due to the fact that in our simulator the attacker selects the path with the highest probability, the smoothing factor is selected in such a way that the path probabilities are not affected. We evaluated the smoothing factor from $4.48 \cdot 10^{-15}$ to $4.48 \cdot 10^{-3}$ which are multiples of 10 of the lowest path probability. For each smoothing factor, we computed the average number of transitions from the initial states (always /usr/sbin/sshd) until the final states (last programs executed by attackers). In the range of $4.48 \cdot 10^{-15}$ to $4.48 \cdot 10^{-6}$ the average number of transitions remains constant and for values larger that $4.48 \cdot 10^{-6}$ the average number of transitions linearly decreases due to the fact that an attacker can select artificial shortcuts. We used a smoothing factor of $4.48 \cdot 10^{-08}$, which does not change the number of average transitions and is large enough to avoid rounding errors. The number of transitions increased to 8281 which is the square of the number of states which can be explained that we have a fully interconnected automaton.
The hierarchical probabilistic automaton was used to simulate attacks in order to compute the average payoff. We simulated the honeypot strategies ($Pr(Block)$) and attacker strategies ($Pr(Quit)$, $Pr(Retry)$, $Pr(Alternative)$) in a range of 0 and 1 in a step of 0.10 respecting the relation 1. In a first step we evaluate the impact of blocking system calls of an attacker. We noticed that the average of transitions performed by an attacker increases with the blocking probability.

In a second step, we computed Nash Equilibriums using the game theory simulator Gambit [13]. Only mixed equilibriums have been found. If we consider the first game (upper half of the table 1) then one mixed Nash equilibrium exists: for instance, the honeypot can decide to use either a blocking probability of 0.10 or of 0.90. It should use 0.10 in 54% of the cases and 0.9 in 46% of the cases. The attacker should use $Pr(Quit)$ equal to 0.3 or 0.4 with associated probabilities 0.73 and 0.27 respectively. Similarly, value choices according to the table can be set for $Pr(Retry)$ and $Pr(Alternative)$. The second game, (lower half of the table 1), has also a mixed equilibrium: the honeypot should use three different blocking probabilities (0.4, 0.7, 1) with corresponding probabilities 0.3, 0.51 and respectively 0.19. This is interesting, since blocking all transitions ( $Pr(Block) = 1$ ) should be done in 19% of the cases. The attacker can also set his optimal strategies with respect to this table.

**Table 1.** Gambit simulation results

| $R_h^p$ | | $R_a^p$ | | | |
|---|---|---|---|---|---|
| q | $Pr(Block)$ | q | $Pr(Quit)$ | $Pr(Retry)$ | $Pr(Alternative)$ |
| 0.54 | 0.1 | 0.73 | 0.3 | 0.4 | 0.3 |
| 0.46 | 0.9 | 0.27 | 0.4 | 0.2 | 0.4 |
| $R_h^t$ | | $R_a^t$ | | | |
| 0.3 | 0.4 | 0.14 | 0.6 | 0.2 | 0.2 |
| 0.51 | 0.7 | 0.26 | 0.8 | 0 | 0.2 |
| 0.19 | 1 | 0.6 | 0.8 | 0.1 | 0.1 |

## 5 Related Work

The article of McCarty [10] describes an arm race between honeypots and attackers: attackers improve their techniques as soon as new monitoring techniques are deployed, which furthermore leads to defenders improving their previous approach. Some researchers modified shells aiming to observe the commands that an attacker used [2]. The major assumption of such a strategy is that the attacker does not change the shell on the honeypot. Other papers considered kernel patching [10] to mitigate this attack. Recently, virtualization based solutions [14] allowed to monitor from an external point of view high-interaction honeypots. Our operational deployment results are in line with the observations made by Eric Alata et al. [4] and Daniel Ramsbrock et al [3]. However, we preferred to patch the Linux authentication module PAM in order to avoid the case where attackers can lock out other attackers by changing the password of a compromised account. Next, we preferred to extract the process trees related to a SSH server instead of patching SSH as it is proposed by Eric Alata et al, because we have observed several times

that attackers changed the SSH server. Although our implementation is based on a modified kernel running in Qemu, the conceptual approach using game theory with high-interaction honeypots can also be used in the context of virtualization based solutions. Garg et al. [15] also used game theory, where they established a different game: an attacker is rewarded if he or she probes a real machine and punished when he or she probes a honeypots. Bistarelli et al. [16] propose high level attack trees and associated attacks with countermeasures, where each action is linked to a cost or payoff. Game theory was also used in the general context of dependability and network security [17] for predicting future attacks.
Our honeypot model is based on a hierarchical probabilistic automaton bootstrapped with operational data from a high-interaction honeypot. Attacker actions are frequently grouped in high-level attack categories which describe the automaton [3]. Our approach is different from the work described by Kong-wei Lye et. al [18], because we recover the states and the transitions probabilities from a deployed honeypot compared to a manual definition. In order to compute the payoffs for the formal game, we used a simulation strategy that is similar to the approach used by Shishir et al. [19].

## 6 Conclusions and Future Work

This paper proposes a new paradigm for adaptive high-interaction honeypots that rely on game theoretical concepts as main driving force. We modeled the interaction between the honeypot and an attacker as a game, where appropriate payoff functions model the behavior goals observed in the real world. We derive the best strategies from the well known Nash equilibrium and use operational honeypots in order to parametrize the game model. We make the strong assumption that hackers are always rational - this might be not the case with all attackers. The obtained results permit practical solutions for designing adaptive high-interaction honeypots. The adaptability is given by blocking one system call according to the optimal blocking probabilities. We leveraged data obtained from a deployed high-interaction honeypot in order to parametrize our model. We plan to investigate other game theoretical models, where repetitive and iterative learning from the past is possible. From an implementation point of view, we are not focusing on indirect attacks: for instance, we do not fully model attacks, where a script is added and gets executed later by the system itself. Moreover, our automaton may be biased by the specifics of the deployed honeypot and attackers that are aware of the game could poison the transition probabilities. This motivates further research on simplifying and comparing hierarchical probabilistic automata from different honeypots. We also planned to compare adaptive and non adaptive honeypots.

## References

[1] Cheswick, B.: An evening with Berferd in which a cracker is lured, endured, and studied. In: In Proc. Winter USENIX Conference.

(1992) 163–174
[2] Spitzner, L.: Honeypots: Tracking Hackers. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA (2002)
[3] 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 (2007) 119–124
[4] 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. (2006) 39 – 46
[5] Vidal, E., Thollard, F., de la Higuera, C., Casacuberta, F., Carrasco, R.: Probabilistic finite-state machines-part I. IEEE Trans. Pattern Anal. Mach. Intell. **27**(7) (2005) 1013–1025
[6] Mitchell, M., Samuel, A.: Advanced Linux Programming. New Riders Publishing, Thousand Oaks, CA, USA (2001)
[7] Love, R.: Linux Kernel Development (2nd Edition). Novell Press (2005)
[8] 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 (2003) 16–16
[9] Greenwald, A.: Matrix games and nash equilibrium (2007) Lecture.
[10] McCarty, B.: The honeynet arms race. IEEE Security and Privacy **1**(6) (2003) 79–82
[11] Bellard, F.: Qemu, a fast and portable dynamic translator. In: ATEC '05: Proceedings of the annual conference on USENIX Annual Technical Conference, Berkeley, CA, USA, USENIX Association (2005) 41–41
[12] Newham, C., Vossen, J., Albing, C., Vossen, J.: Bash Cookbook: Solutions and Examples for Bash Users. O'Reilly Media, Inc. (2007)
[13] Turocy, T.: Gambit (2007) http://gambit.sourceforge.net/.
[14] Dinaburg, A., Royal, P., Sharif, M., Lee, W.: Ether: malware analysis via hardware virtualization extensions. In: CCS '08: Proceedings of the 15th ACM conference on Computer and communications security, New York, NY, USA, ACM (2008) 51–62
[15] Garg, N., Grosu, D.: Deception in honeynets: A game-theoretic analysis. In: Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC. (2007) 107–113
[16] Bistarelli, S., Dall'Aglio, M., Peretti, P.: Strategic games on defense trees. In: Formal Aspects in Security and Trust. (2006) 1–15
[17] Sallhammar, K., Helvik, B.E., Knapskog, S.J.: A framework for predicting security and dependability measures in real-time. International Journal of Computer Science and Network Security **7**(3) (2007)
[18] Lye, K.W., Wing, J.M.: Game strategies in network security. International Journal of Information Security **4**(1) (February 2005) 71–86
[19] Nagaraja, S., Anderson, R.: The topology of covert conflict. Technical report, University of Cambridge (2005)