Self-Adaptive Honeypots Coercing and Assessing Attacker Behaviour

Gérard Wagener

Université du Luxembourg
PhD-FSTC-2011-10
The Faculty of Science, Technology and Communication

École doctorale IAEM Lorraine

DISSERTATION

Defense held on 22/06/2011 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DU LUXEMBOURG
EN INFORMATIQUE

ET

DOCTEUR DE L'INSTITUT NATIONAL POLYTECHNIQUE DE LORRAINE
SPÉCIALITÉ: INFORMATIQUE

by

Gérard WAGENER
Born on 18 March 1982 in Luxembourg (Luxembourg)

Self-Adaptive Honeypots Coercing and Assessing Attacker Behaviour

Dissertation defense committee

Prof. Dr. Ulrich Sorger, Chairman
Professor, Université du Luxembourg

Prof. Dr. Marc Dacier, Deputy chairman and reviewer
Senior Director at Symantec

Prof. Dr. Thomas Engel, Supervisor
Professor, Université du Luxembourg

Dr. Hab. Radu State, Member
Researcher, Université du Luxembourg

Prof. Dr. Eric Filiol, Member and reviewer
Directeur du laboratoire de virologie et de cryptologie opérationnelles ESIEA Ouest

Dr. Hab. Olivier Festor, 2nd supervisor
Research Director at the INRIA Nancy - Grand Est

Abstract

Information security communities are always talking about ”attackers” or “blackhats”, but in reality very little is known about their skills. The idea of studying attacker behaviors was pioneered in the early nineties. In the last decade the number of attacks has increased exponentially and honeypots were introduced in order to gather information about attackers and to develop early-warning systems. Honeypots come in different flavors with respect to their interaction potential. A honeypot can be very restrictive, but this implies only a few interactions. However, if a honeypot is very tolerant, attackers can quickly achieve their goal. Choosing the best trade-off between attacker freedom and honeypot restrictions is challenging. In this dissertation, 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. Rather than being allowed simply to carry out attacks, attackers are challenged by strategic interference from adaptive honeypots. The observation of the attackers' reactions is particularly interesting and, using derived measurable criteria, the attacker's skills and capabilities can be assessed by the honeypot operator. Attackers enter sequences of inputs on a compromised system which is generic enough to characterize most attacker behaviors. Based on these principles, we formally model the interactions of attackers with a compromised system. The key idea is to leverage game-theoretic concepts to define the configuration and reciprocal actions of high-interaction honeypots. We have also leveraged machine learning techniques for this task and have developed a honeypot that uses a variant of reinforcement learning in order to arrive at the best behavior when facing attackers. The honeypot is capable of adopting behavioral strategies that vary from blocking commands or returning erroneous messages, right up to insults that aim to irritate the intruder and serve as a reverse Turing Test distinguishing human attackers from machines. Our experimental results show that behavioral strategies are dependent on contextual parameters and can serve as advanced building blocks for intelligent honeypots. The knowledge obtained can be used either by the adaptive honeypots themselves or to reconfigure low-interaction honeypots.

Acknowledgements

At first and foremost I want to thank the people who enabled setting up this PhD framework. I want to thank Prof. Dr. Thomas Engel and Dr. Hab. Olivier Festor acting as my PhD supervisors. This framework is integrated in an industrial context and would not have been possible without the continuous support of Alexandre Dulaunoy and Thomas Schneider of the satellite operator and Internet Service Provider SES. An essential financial role was played by the FNR and I want to thank the whole FNR team for their help. Besides the strong industrial partner, two universities have been involved in this project namely the university of Luxembourg and INPL Nancy. I also had the opportunity to interact with the MADYNES team headed by Olivier Festor of the Loria laboratory. I want to thank all the organizational staff from each entity for helping me in this complex setup. The results of these research activities are strongly influenced by the fruitful and endlessly long discussions with Radu State and Alexandre Dulaunoy to hear their additional scientific and technical advice. I also want to thank my office colleagues at SES, the university of Luxembourg and the Loria laboratory for their constructive discussions. I want especially thank Dominic Dunlop of the university of Luxembourg for his thorough proof reading of my documents. I also want to thank the reviewers Eric Filiol and Marc Dacier for their useful suggestions. Besides strong professional support, I also benefited from a devoted social support during my professional activities. I want to thank my mother, my two brothers and my sister for their encouragement. I am also in debt to my friends during this period, and especially my girlfriend, for unstinting understanding and patience.

”The present project is supported by the National Research Fund,Luxembourg”

aides à la formation recherche

Fonds National de la Recherche Luxembourg

In honor of my father who died in 2001.

Contents

List of Figures

List of Tables

Chapter 1

Résumé en français

1.1 Introduction

Dans la dernière décade le nombre des rapports de vulnérabilités des logiciels a explosé. Ce phénomène a été décrit par Frei et al. [54] et nous avons pu confirmer ce phénomène dans le chapitre 2. La prévalence d'une nouvelle technologie est suivie par une nouvelle famille de rapport de vulnérabilités. Les vulnérabilités décrites dans ces rapports sont exploitées par des attaquants jusqu'au moment de la stabilisation de la technologie. Ce phénomène est même accéléré par l'interconnexion globale des systèmes. Dans le début des années 2000, Spitzner [154] a découvert qu'une machine à été compromise quelques minutes après avoir été connectée à Internet. Sept années plus tard Provos et al. [130] ont fait une expérience similaire ce qui confirme la persistance du problème. La plupart des internautes connectent leurs systèmes à n'importe quel réseau et exposent tous les services selon le paradigme prêt à tourner. Les internautes ne veulent pas passer du temps à configurer leurs systèmes ayant comme objectif un maximum de sécurité. Ce comportement de masse a menée à la propagation des vers informatiques qui infectaient des milliers d'ordinateurs en très peu de temps. Provos et al. [130] ont constaté que la naissance des programmes malicieux ne cessent pas de croître et que les réactions manuelles sont inefficaces. Les constructeurs de systèmes d'exploitations et les fournisseurs Internet ont fait des efforts énormes pour sécuriser leurs solutions. Des organismes officiels chargés d'assurer des services de prévention des risques et d'assistance aux traitements d'incidents ont fait des efforts pour éduquer les utilisateurs en vue de combattre les attaques contre les systèmes informatiques. Dans les débuts des années quatre-vingt dix, Cheswick [27] a recommandé de diminuer les services exposés à l'extérieur afin de réduire le risque de se faire attaquer. Cette approche pragmatique a été formalisée par Howard et al. [72]. Les auteurs ont formalisé la surface d'attaque qui est dérivée par protocoles de communications et les droits d'accès au sein d'un système d'exploitation. À coté du rôle d'éducation les organismes officiels chargés d'assurer des services de prévention des risques et d'assistance aux traitements d'incidents mesurent les attaques et ils étudient le comportement des attaquants. Dans ces travaux, nous nous concentrons sur les attaques contre des victimes arbitraires qui sont recherchées par des méthodes automatiques. L'objectif d'un attaquant est d'utiliser leurs ressources. Ces attaques sont particulièrement intéressantes car elles peuvent être mesurées de façon automatique. Les attaquants balaient des plages d'adresses IP publiques. Un hôte qui répond à une tentative de connexion est une victime potentielle. Les attaquants peuvent aussi monter un serveur auquel ont accès des utilisateurs légitimes et qui se font compromettre durant leur visite. Par contre, ces types d'attaques ne font pas l'objectif de nos travaux. Après avoir découvert une cible, les attaquants essaient d'y accéder. Ensuite, ils essaient d'avoir accès à la machine. Ils peuvent soit faire une attaque de dictionnaire contre un service légitime, soit utiliser un code d'exploit. Ramsbrock et al. [135] ont étudié le comportement des attaquants qui ont réussi à pénétrer la cible. Ils ont modélisé ce comportement avec une machine à état. Un état représente un type de comportement des attaquants. Un attaquant peut soit changer le mot de passe, soit faire l'inventaire des logiciels installés ou du matériel utilisé soit télécharger des logiciels supplémentaires et les installer. En intégrant les modèles de découverte de cible, le processus d'exploitation de la cible, et les activités par la cible

nous aboutissons à un modèle d'anneau présenté dans la figure 2.2. Chaque anneau correspond à un type d'activité d'un attaquant ou type d'information sur les attaquants. Par exemple, l'anneau, noté R_0 contient toutes les activités de découverte de la cible. L'anneau, noté R_1 décrit tous les processus d'exploitation. Après avoir pénétré un système, les attaquants entrent dans une phase de reconnaissance où ils inspectent le système. Cette phase est représentée par l'anneau R_2. Dans l'anneau R_3 les attaquants modifient le système. Un exemple est de garantir des accès aux futures attaquants ou d'effacer les traces de pénétration. Finalement, les attaques entrent dans l'anneau R^* modélisant les objectifs de l'attaquant. Le modèle des anneaux est hiérarchique et correspond à la profondeur des attaquants sur le système et est relié au niveau de contrôle de l'attaquant. Sur l'anneau R_0 ils n'ont pas de contrôle sur le système par contre sur l'anneau R^* ils contrôlent le système. La technologie des pots de miel est utilisée pour observer et étudier les attaquants. Un pot de miel est un système sans objectif légitime et toute activité sur un tel système est par défaut suspect. Chaque anneau représente un type de connaissance sur les attaquants et différents types de pots de miel ont été proposés pour collecter des informations sur des attaquants. Les pots de miel destinés pour l'anneau R_0 collectent des informations sur la source des attaquants comme leur adresse source et le service demandé. Les exploits utilisés par les attaquants sont identifiés avec des pots de miel opérés dans l'anneau R_1. Les connaissances sur les attaquants contenues dans les anneaux R_0 et R_1 peuvent être collectées avec les pots de miel à faible interaction. Par contre les observations d'attaques reliées aux anneaux R_2, R_3 et R^* sont collectées avec les pots de miel à forte interaction. L'avantage de la technologie des pots de miel c'est qu'il n'y a pas de faux positifs comme c'est le cas pour les systèmes de détection d'intrusion. Par définition, toute observation sur un pot de miel dû à une attaque peut être considéré comme attaque. La technologie de pot de miel est une technologie complémentaire aux systèmes de détection d'intrusion et permet d'étudier les actions des attaquants.

Les pots de miel sont soit utilisés pour mesurer des attaques soit pour apprendre les méthodologies d'attaques implémentées par les attaquants. Dans nos travaux, nous nous concentrons sur le deuxième objectif. Dans le début des années 2000, Spitzner [154] a défendu la thèse selon laquelle les attaquants peuvent apprendre à la communauté des chercheurs en sécurité informatique leurs techniques d'attaques, leurs outils utilisés et leur stratégies employées. En observant la décennie de travaux de recherche sur les pots de miel, nous constatons qu'une partie des travaux ont participé dans la course des pots de miel. Dans une telle course, un nouveau pot de miel est suivi par des techniques de détection ou d'évasion. Cohen [31] a présenté les techniques d'illusion dans le contexte des attaques informatiques. Par contre, les seules techniques d'illusion utilisées dans le contexte des pots de miel est d'émuler des faux services. Ces techniques sont adaptées à collecter les outils utilisés par les attaquants et les prédécesseurs des attaques. Les pots de miel ont été déployés de façon distribué et les tactiques et stratégies des attaquants ont été dérivées grâce aux données collectées par les différents pots de miel. Par contre, les pots de miel eux mêmes ne sont pas capable de dériver les stratégies des attaquants de façon automatique et les pots de miel en eux-mêmes ont un comportement de sondes statiques. Nicomette et al. [112] ont découvert qu'il existe des attaques automatiques et des attaques manuelles ce qui est difficilement distinguable. Les pots de miels sont classifiés comme pot de miel à forte interaction et pot de miel à faible interaction. Cette classification est assez rudimentaire et ne prend pas en compte les capacités des attaquants. L'objectif de ces travaux est de déterminer les capacités des attaquants en construisant des pots de miel qui résistent de façon stratégique aux attaques. Ainsi une réaction d'un attaquant peut être déclenchée. Afin d'aboutir à cet objectif les effets de bord des observations doivent être examinés précisément. Comme détaillé dans le chapitre 3, il y a une course entre opérateurs de pots de miel et les attaquants. Certes, les pots de miel ont des faiblesses qui peuvent être prises en compte en utilisant des techniques d'illusion lors des attaques. Les attaquants peuvent mal interpréter les réponses reçues des pots de miel, et ainsi leurs comportements peuvent mieux être étudiés. Le but est de modéliser ces différents objectifs et d'aboutir à ces objectifs de façon automatique.

Dans nos travaux, nous élaborons des pots de miel adaptatifs qui sont capables de changer de manière continue leur comportement avec le but de découvrir les meilleures actions. Ce manuscrit est divisé en deux parties. La première partie contient un résumé des travaux de recherche antérieurs. La deuxième partie contient une description de nos travaux de modélisation ainsi que les expérimentations des nos pots de miel adaptatifs.

1.2 État de l'art

Cette partie est divisée en deux chapitres. Nous commençons à résumer les activités antérieures dans le domaine des pots de miel et nous identifions leurs limites. Ensuite nous donnons un aperçu sur les théories que nous utiliserons dans nos contributions.

1.2.1 Pots de miel

L'idée d'observer des attaquants sur des système informatiques grâce à des leurres est née dans les années quatre-vingt. Stoll [158] a observé une attaque dans le laboratoire Berkeley. Il a pris la décision de distribuer des faux documents et il a surveillé les accès à celles-ci. Après un certain temps, il a pu identifier l'attaquant. Un peu plus tard dans les années quatre vingt-dix, Cheswick [26] a eu l'idée de déployer des faux services accessibles depuis Internet afin d'observer des attaques. Dans les années 2000, Spitzner [154] a introduit la terminologie des pots de miel qui est de nos jours acceptée dans la majorité des communautés de sécurité informatique. Un pot de miel est une ressource dédiée à être découverte, exploitée et attaquée. Les pots de miel sont fréquemment utilisés pour collecter des données sur les attaquants. La plupart des travaux se concentre à collecter des données techniques des attaquants, comme leur adresse source, leurs logiciels et les répertoires des logiciels malicieux. D'autres travaux se concentrent sur les études de comportement des attaquants qui ont réussi à compromettre un système. Une condition essentielle d'un pot de miel ayant comme but d'étudier le comportement des attaquants est que celui-ci soit attractif pour les attaquants et qu'il imite un système réel. Dans le cas contraire, les attaquants se rendent compte qu'il ne s'agit pas d'un système réel, et en conséquence ne l'attaquent pas ou se déconnectent rapidement. Dans le chapitre 3, nous étudions les tendances sur les pots de miel grâce aux données de Google. Le moteur de recherche Google a accès à des milliards de requêtes des utilisateurs. Google a créé des indices par mot clef. Une synthèse de ces indices est publiquement accessible, de l'année 2004 jusqu'à présent. Un indice plus élevé que son prédécesseur indique que le mot clef est devenu plus populaire. Si l'indice est plus petit que son antécédent le mot clef a perdu de popularité. En 2004, Provos et al. [128] ont publié Honeyd, un pot de miel à faible interaction. Le code source de ce pot de miel a été publié et le pot de miel est facile à utiliser. Ceci avait comme conséquence que de nombreux utilisateurs ont commencé à utiliser cet outil. En 2006, les réseaux de programmes malicieux appelés botnets étaient une menace omniprésente sur Internet et les pots de miel ont été utilisés afin de les étudier et de les combattre, ce qui explique la croissance de l'indice. En comparant les pots de miel récents utilisant la technologie d'introspection des machines virtuelles avec les pots de miel de type Honeyd, nous constatons que la complexité logicielle est beaucoup plus élevée pour ces nouveaux types de pot de miel. Ceci a comme conséquence qu'ils sont opérés par des communautés plus restreintes et plus par le grand public. Pour confirmer cette affirmation nous avons étudié les nombres de publications mensuelles scientifiques sur les pots de miel. Bien que le indices de Google incluent toutes les communautés d'utilisateurs, le moteur de recherche Google Scholar se concentre sur les bases de données scientifiques. L'analyse détaillée est décrite dans le chapitre 3. Dans l'année 2008, l'introspection des machines virtuelles est à la mode pour construire des pots de miel ce qui exige des connaissances systèmes poussées et implique un risque plus élevé que celui relié aux pots de miel à faible interaction.

Différents types de classifications ont été proposés pour les pots de miel. Seifert et al.[147] les classifient selon six critères qui sont énumérés de façon informative. Le premier critère est le niveau d'interaction avec les attaquants. Le niveau d'interaction est dérivé du nombre de fonctionnalités exposées aux attaquants. Le deuxième critère de classification est le niveau de capture de données. Un pot de miel peut collecter des évènements, des attaques et des intrusions. Le troisième critère est le niveau de confinement du pot de miel. Le quatrième critère distingue des pots de miel distribués des pots de miel autonomes. Le cinquième critère est dérivé des moyens de communications avec le pot de miel. Le sixième critère distingue les pots de miel ayant un rôle de serveur avec les pots de miel qui prétendent être un client. Malgré cette fine classification nous utiliserons dans la suite la classification élaborée par Spitzner [154] prenant en compte le niveau d'interaction avec les attaquants car elle nous permet le mieux de situer nos travaux.

Un pot de miel à faible interaction est un système qui émule des faux services. Selon notre modèle des anneaux, des informations reliés aux anneaux R_0, et R_1 peuvent être recueillies comme l'adresse source des attaquants, le service demandé. Un attaquant se connectant à un tel service reçoit un contenu préfabriqué. Les attaquants n'ont ni le droit de stocker ni d'exécuter leur propres programmes. Cette contrainte assez stricte permet de réduire le risque d'opération de pot de miel. Ce bas risque permet d'opérer ces pots de miel à large échelle. Fin des années quatre-vingt-dix, Cohen [31] a proposé un outil d'illusion (Deception Tool Kit (DDK)). L'idée est de mélanger sur un hôte des services réels et des services émulés. Dans un scénario d'attaque d'un tel hôte un attaquant doit distinguer les vrais services des faux ce qui implique une augmentation de charge de travail. Le pot de miel à faible interaction le plus populaire est Honeyd. Ce pot de miel est capable d'émuler des piles réseaux. Ainsi un seul hôte peut émuler plusieurs hôtes émulant différents systèmes d'exploitations. Avec un seul hôte des topologies entières de réseaux peuvent être émulées ce qui est facile à gérer. Honeyd contourne la pile réseau de son hôte réel et répond à toutes les requêtes émulées. Honeyd permet de définir plusieurs personnalités qui correspondent à une pile de réseaux et une collection de programmes qui peuvent être rattachés aux services demandés. Chaque programme est responsable d'émuler un service dédié ce qui peut être vu comme un désavantage de cette approche. Leita et al. [91] ont proposé une approche pour dériver ces faux services à partir des traces réseaux.

Cette approche est assez puissante car elle permet aussi d'émuler de faux programmes malveillants pour lesquels le code source n'est pas disponible. Les pots de miel à faible interaction permettent de gérer les premières interactions avec les attaquants et sont donc suffisants pour quantifier et mesurer les attaques. Par contre, ils ne permettent pas d'étudier le comportement des attaquants qui ont réussi à compromettre un système. Dans notre modèle avec les anneaux, les attaquants essaient d'exploiter le système, ce qui est modélisé avec l'anneau R_1. Ceci est fait grâce à un code d'exploit. Un tel code est souvent encodé dans un message mal-formé. Lors de la phase de décomposition de ce message dans le service attaqué, une partie de celui-ci est exécutée par erreur. Une interruption du service est la conséquence si cette partie contient des données aléatoires. Cependant, les attaquants encodent du code machine dans cette partie qui est exécuté sur l'hôte. L'exécution de ce dernier donne le contrôle d'exécution aux attaquants. Ce code machine est appelé shell code car, pour des raisons historiques, ce code donnait normalement accès à une ligne de commande. Souvent ce code machine télécharge un programme malveillant qui prend le relais de l'attaque. Si un tel message arrive à un pot de miel décrit précédemment, le code d'exploit peut être observé. Baecher et al. [7] ont proposé un pot de miel qui est capable d'émuler des vulnérabilités connues et qui télécharge les programmes malveillants complémentaires. Malgré cette interaction supplémentaire, les auteurs qualifient de faible interaction ce type de pot de miel. D'autres propositions de faux services capable d'émuler des vulnérabilités connues ont été suggérées [6, 60]. Une hypothèse de ces approches est que la vulnérabilité est connue et peut être émulée. Les attaques qui exploitent des vulnérabilités inconnues ne peuvent pas être collectées avec les pots de miel précédemment discutés. De plus, les activités des attaquants après l'exploit (anneaux R_2, R_3, R^*) ne sont pas prises en compte avec les pots de miel à faible interaction.

Par contre, ces activités sont observées avec les pot de miel à forte interaction. Historiquement, ces pots de miel sont les plus anciens. En fait, des ressources réelles sont exposées aux attaquants. Ceux-ci peuvent utiliser toutes les fonctionnalités du système observé. Cheswick [26] a baptisé ces ressources comme des machines jetables ayant des failles de sécurité réelles. Il recommande de capturer tout le trafic réseau dédié à ces machines afin d'observer les activités des attaquants. À cette époque cette approche était parfaitement valide car la majorité des protocoles de communications utilisés par les attaquants étaient non chiffrés. De nos jours, avec l'utilisation des réseaux sans fil publiquement accessibles, les protocoles de communications chiffrées sont devenus très populaires. Si un attaquant utilise de tels protocoles, ces commandes ne peuvent pas être observées, ce qui n'est pas acceptable pour un opérateur de pot de miel. Souvent, celui-ci est légalement responsable de ses ressources et quand il les offre à des attaquants, il peut participer à des attaques vers des tiers [2]. Un contrôle strict est nécessaire durant l'opération de pots de miel pour éviter les dégâts collatéraux. Dès que des attaques sont lancées vers des tiers, l'opérateur doit pouvoir intervenir et déconnecter les attaquants. Spitzner [154] recommande d'avoir plusieurs points de vue d'observation afin de garantir un contrôle des

attaquants. Il a proposé d'observer les attaquants au niveau réseau mais aussi au niveau système. Comme exemple de contrôle du système, il a donné l'exemple d'une ligne de commande modifiée qui exporte toutes les commandes vers un autre hôte. Les attaquants ont changé ce programme et leurs actions passaient inaperçues. La réaction de la communauté des opérateurs de pot de miel est l'observation d'une couche plus basse dans le noyau du système d'exploitation [9]. Balas et al. [9] ont proposé un module noyau dans cet objectif. Un peu plus tard, McCarty [101] a publié une méthode pour détecter et désactiver ces techniques d'observation. La communauté des opérateurs est alors descendu à un niveau plus bas et a ainsi choisi le point d'observations à l'intérieur d'une machine virtuelle.

Parmi les activités de recherche se concentrant sur l'amélioration système des pots de miel, des activités connexes se sont créées comme la gestion des pots de miel, la collaboration des opérateurs de pots de miel et des nouvelles approche d'analyse de données dans les pots de miel. Les pots de miel servent principalement de source de données sur les attaquants. Nous avons synthétisé un modèle à anneaux où chaque anneau représente une couche d'information sur les attaquants. Nous avons constaté que les informations relatives aux anneaux R_0 et R_1 peuvent être collectées avec les pots de miel à faible interaction et les informations relevant de R_{0-3} et R^* peuvent être collectées avec les pots de miel à forte interaction. Les pots de miel à forte interaction couvrent plus d'information sur les attaquants que les pots de miel à faible interaction. Par contre, la gestion des pots de miel à faible interaction est plus facile et le risque d'opération est plus faible. De ce fait, les pots de miel à basse interaction passent mieux à l'échelle. Les opérateurs de pots de miel sont souvent légalement responsables de leurs pots de miel et ils doivent éviter que les attaquants prennent le contrôle absolu du pot de miel en attaquant des tiers. Cohen [31] a discuté dans le contexte des attaquants des techniques d'illusion. Les pots de miel à faible interaction émulent des faux services et utilisent une technique d'illusion. De plus, les pots de miel à forte interaction essaient de ressembler le plus à des hôtes normaux (de systèmes de production). Les deux types de pots de miel utilisent des techniques de déception statique et très peu de travaux ont été effectués pour rendre les pots de miel eux-mêmes plus intelligents et adaptatifs. Dans le contexte des pots de miel à forte interaction, un attaquant peut compromettre le système, le modifier, installer et lancer ses propres outils. Si le but de l'attaquant était d'exécuter ses outils, il aboutit à son objectif d' attaque sans aucune résistance venant du pot de miel. L'attaquant n'a pas été défié car toutes les actions ont été permises. Dans le cas où le pot de miel aurait interféré de manière stratégique avec les actions de l'attaquant, des comportements supplémentaires auraient pu être observés. Ceci permettrait de dériver les capacités des attaquants confrontés à la résolution d'un problème. Les pots de miel pourraient utiliser des techniques de déception dynamiques avec l'objectif d'apprendre les capacités des attaquants ainsi que leur nature et leur bagage linguistique. Afin de modéliser des pots de miel intelligents et adaptatifs, nous donnons un aperçu des fondations théoriques utilisées dans nos contributions.

1.2.2 Apprentissage dans les jeux

Dans le chapitre 4, nous nous concentrons sur la théorie des jeux et sur l'apprentissage par renforcement dans le contexte théorique des processus de décision Markovien ainsi que des processus stochastiques. De plus, nous mettons en évidence des liens entre ces différentes théories. La théorie des jeux permet de modéliser des jeux entre joueurs où chacun d'entre eux a ses propres intérêts et dépend des autres. À chaque joueur est associé un ensemble d'actions et chacune d'elle est associée avec un gain ou une perte. Donc, chaque jouer a une fonction de gain. Dans le cas où le gain d'un joueur correspond à une perte de l'autre, le jeu est dit à *somme nulle*. Dans le cas où la magnitude de gain d'un joueur est différente de celle d'un autre joueur le jeu est dit à *somme générale*. Les jeux peuvent être représentés soit sous forme normale soit sous forme extensive. La forme extensive est plus appropriée pour modéliser des jeux répétitifs. La forme normale permet de calculer des équilibres de Nash qui définissent les meilleures stratégies pour chaque joueur dans le contexte des jeux simples. Pour les jeux sous forme extensive, les jeux sous forme normale doivent être dérivés afin de calculer un équilibre de Nash. Les stratégies calculées à partir d'un équilibre de Nash ne sont pas forcément Pareto-optimal. Une stratégie d'un joueur peut détériorer le gain d'un autre joueur. Dans le contexte d'un équilibre de Nash dans un jeu simple, il est supposé que tous les joueurs doivent être rationnels, c'est à dire visant à maximiser

leurs gains. Un non-respect de cette condition résulte en des gains erronés. Le problème des gains erronés peut être analysé avec une analyse d'équilibre quantal en vue de déterminer les impacts sur les équilibres de Nash. La théorie des jeux a déjà été employée dans le cadre des pots de miel. Par contre, la majorité des contributions se concentrent sur des jeux joués au niveau infrastructure qui est composée de machines de production et de pots de miel. Les pots de miel fréquemment utilisés sont des pots de miel à faible interaction. Garg et al. [108] a modélisé un jeu entre deux joueurs. Le premier est le défenseur et le deuxième un attaquant omniprésent. Le défenseur possède une infrastructure qui est composée de machines réelles et de pots de miel. Un attaquant peut attaquer soit une machine réelle soit un pot de miel mais il préfère attaquer le premier type. Il reçoit donc un gain positif s'il attaque une machine réelle et un gain négatif s'il attaque un pot de miel. Pour le défenseur, les gains sont distribués de façon inverse. Les auteurs ont défini les gains de façon manuelle. Lye et al. [98] ont défini un jeu similaire. Par contre ils ont dérivé les gains à partir d'un questionnaire qui a été rempli par les administrateurs de leur université. Ces deux modélisations exigent que les comportements et les gains soient connus à l'avance et donc soient statiques. En vue de résoudre ces problèmes, nous avons envisagé l'utilisation d'apprentissage par renforcement. Dans un tel cas, un agent est capable de faire des actions dans un environnement. Chaque action est soit punie, soit rémunérée. L'objectif d'un agent est de maximiser son gain. Les bases théoriques dans l'apprentissage par renforcement sont les processus de décisions Markovien. Un tel processus est composé d'un ensemble d'états qui peuvent être visités par un agent. Dans chaque état l'agent peut choisir une action parmi un ensemble d'actions. Une fonction définit les gains qui sont distribués après chaque action. Un agent recherche une politique définie par une relation entre les états et les actions. L'objectif de l'agent est de trouver la politique en maximisant le gain à court, moyen, ou long terme. Si un tel problème est modélisé avec un processus de décision Markovien et si tous les paramètres sont connus alors cette politique optimale peut être déterminée grâce aux équations de Bellmann comme décrit dans le chapitre 4. En pratique, quelques paramètres demeurent inconnus ce qui a pour conséquence de rendre impossible le calcul des équations de Bellmann. Habituellement, la fonction de transition dans le processus de décision Markovien ainsi que la fonction de gain sont inconnues. De plus la complexité de calcul des équations de Bellmann est très élevée. Ces deux problèmes nous ont menés vers l'apprentissage par renforcement ("model-free methods"), qui nous permet d'approximer les politiques optimales. Le désavantage majeur d'une telle approche est le compromis entre exploration et exploitation. Un agent n'est pas censé connaître son environnement.

Dans un tel cas, il doit explorer des actions dans des états donnés et en observer les effets. Ceci est fait dans une phase d'exploration. Dans la phase d'exploitation un agent exploite les connaissances apprises auparavant. Les décisions sont prises par une règle d'apprentissage. Une exigence est que l'environnement doit être stationnaire ce qui entraîne que les probabilités de transition entre états et les gains ne changent pas au cours du temps. Dans un contexte avec plusieurs agents concurrents, ce prérequis n'est pas toujours respecté. Les processus Markoviens de décision ont été étendus en prenant en compte ces contraintes, ce qui a mené à la définition de jeux de Markov stochastiques. Fink [50] a formellement prouvé l'existence des points d'équilibre dans les jeux stochastiques. Ce résultat permet de combiner des approches d'apprentissage avec la théorie des jeux.

1.3 Contributions

1.3.1 Modélisation du comportement des attaquants

Dans les années quatre-vingt dix, Cheswick a étudié le comportement des attaquants en interagissant manuellement avec eux. Grâce à une expérience similaire, nous avons constaté qu'en interférant avec un attaquant nous pouvons étendre nos connaissances à propos de lui. L'objectif de nos travaux est de modéliser, implémenter et d'évaluer des pots de miel intelligents qui s'adaptent aux attaquants en vue de les défier et d'apprendre plus de connaissances sur eux. Dans le chapitre 5, un modèle générique de comportement des attaquants a été présenté et les possibilités d'adaptation. Nous avons focalisé notre travail sur les attaquants qui pénètrent les serveurs SSH, car ce type d'attaque est très populaire. En effet, derrière ces serveurs se cachent souvent

des systèmes puissants et attractifs pour les attaquants. Le modèle de comportement doit être assez générique pour qu'il puisse également être adapté à d'autres types d'attaques. Une fois que les attaquants ont pénétré un système, deux types de comportement sont envisageables. Le premier est appelé comportement d'avancement et le deuxième se nomme comportement de réponse ou réaction des attaquants. Dès que les attaquants ont pénétré un système, ils exécutent une succession de commandes en vue de réaliser une attaque. Le fait d'entrer successivement des commandes définit le comportement d'avancement qui correspond à des transitions dans un automate hiérarchique probabiliste. Cet automate est composé de macro-états représentant des programmes installés sur le système. Chaque macro-état se définit aussi comme un automate probabiliste où chaque état est équivalent à un argument transmis au programme. La hiérarchie est nécessaire car la sémantique des arguments est différente selon les programmes. L'alphabet de l'automate est associé aux chaînes de caractères entrées par les attaquants. Parmi les commandes ou les arguments des programmes, les attaquants ont la possibilité d'entrer des informations non valides comme des commandes pour exécuter leurs propres programmes, des erreurs typographiques, des insultes ou des tests de disponibilité de la ligne de commande. Pour cette raison nous avons introduit trois états spécifiques nommés *insult, custom, empty*. Chaque chaîne de caractère entrée par un attaquant est associée à une des quatre catégories suivantes:

1. Commande système
2. Commande relative à un programme installé par un attaquant
3. Commande vide
4. Insulte ou erreur typographique

Un attaquant peut entrer une commande système, ce qui équivaut à l'exécution d'un programme installé par défaut sur le système. Cependant, si la commande n'exécute pas un programme installé par l'opérateur du pot de miel, une transition dans l'état *custom* est déclenchée. L'observation d'une entrée vide nous indique que l'attaquant a testé la réponse de la ligne de commande et, dans un tel cas, une transition vers l'état *empty* est effectuée. Si une entrée ne correspond à aucun cas précédemment décrit, l'attaquant est considéré comme ayant tapé une insulte dans son terminal. Cette définition d'insulte comprend les erreurs typographiques des attaquants. Dans le chapitre 6, nous utiliserons la distance de Levenshtein afin de classer les différentes entrées qui sont responsables d'une transition dans l'état *insult*. D'un premier point de vue, cette modélisation semble assez abstraite mais elle permet d'être implémentée de façon systématique. En vue d'observer les commandes entrées par un attaquant, nous observons les appels systèmes dans le noyau. Pour chaque appel associé à l'exécution d'un programme, l'identifiant du processus, l'identifiant du processus père ainsi que le nom du programme avec ses arguments sont mémorisés. Ces informations permettent la reconstruction de la structure arborescente des processus qui sont actifs dans le système. Le serveur SSH crée un processus responsable de gérer la connexion d'un attaquant. Ce processus est la racine d'un sous-arbre de l'arbre global des processus du système. Tous les nœuds de ce sous-arbre sont donc corrélés aux exécutions de programmes d'un attaquant. L'analyse des arbres de processus permet de gérer plusieurs attaquants simultanés tout en différenciant les processus invoqués par le système lui-même. Chaque lien entre nœuds dans un sous arbre est étiqueté de la différence de temps entre le père et le fils. Ainsi nous pouvons reconstruire l'ordre des commandes exécutées et nous pouvons transformer les sous-arbres en des suites de commandes entrées par un attaquant. Cette suite de processus est qualifié de vecteur de processus. De plus, nous proposons une méthode pour recueillir les probabilités de transition entre commandes.

Quand un attaquant a pénétré un système, il exécute des commandes pour atteindre son objectif. Un pot de miel défie l'attaquant en interférant avec l'attaque. Nous définissons quatre actions pour un pot de miel adaptatif dans le but de déclencher une réaction chez l'attaquant:

permettre l'exécution d'une commande. Si un pot de miel adaptatif permet l'exécution d'une commande, il se comporte comme un pot de miel à forte interaction traditionnelle. Cette commande est nécessaire pour permettre l'avancement dans la réalisation de l'attaque.

bloquer l'exécution d'une commande. Lors de cette action, le pot de miel adaptatif retourne volontairement une erreur à l'attaquant indiquant que l'exécution du programme désiré a échoué. Un attaquant est bloqué durant son avancement et doit réagir s'il veut aboutir à son objectif.

substituer l'exécution d'une commande. Un pot de miel adaptatif exécute un autre programme à la place de celui qui a été désiré par l'attaquant. Cela est une technique d'illusion qui déclenche également une réaction chez l'attaquant. Son défi consiste alors à comprendre le comportement du système.

insulter l'attaquant. Dans un tel cas le pot de miel écrit une insulte dans le terminal de l'attaquant. Cette approche est une technique d'illusion qui déclenche aussi une réaction de l'attaquant. Elle doit permettre de distinguer si l'attaquant est un être humain ou un robot. Dans le contexte d'un humain cette technique permet d'augmenter le niveau de stress pour le pousser à commettre des erreurs et pour déterminer son origine linguistique.

Grâce à ces interactions supplémentaires avec l'attaquant, la réaction des attaquants est particulièrement intéressante et se caractérise par une des cinq catégories suivantes :

répéter la commande échouée. D'un côté, l'échec d'une commande peut être dû à un mauvais argument ce qui résulte en une erreur syntaxique. L'attaquant peut donc choisir des autres arguments ou réutiliser les mêmes. D'un autre côté, l'échec peut être causé par un temps de débouclement émergé durant l'exécution du programme.

chercher une commande alternative. Après avoir observé l'échec d'une commande, un attaquant peut tenter d'en déterminer la cause. Il a la possibilité par exemple de lancer un outil de traçage comme `strace` en parallèle du programme qu'il désire exécuter. Une autre option est de vérifier ou de modifier la configuration du programme. En pratique, nous avons observé des attaquants ayant téléchargé le code source du programme en vue de le transformer en programme exécutable sur le pot de miel lui-même. Grâce à cette intervention, nous avons pu récupérer le code source habituellement non disponible. Ce type de comportement est significatif d'un attaquant cherchant une solution alternative pour aboutir à son but.

insulter le pot de miel. Toute commande ne correspondant ni à un programme, ni à une commandes vide est considérée comme une insulte. L'observation d'erreurs typographique ou d'insultes est fortement synonyme de la nature humaine, et donc différence celle d'un robot de l'attaquant. Durant nos premières interactions avec les attaquants, nous avons remarqué des attaquants tapant des insultes dans leur terminal. Ceci était souvent le cas quand ils ont reçu une insulte du pot de miel. Une des hypothèses amène à penser que les attaquants supposent que d'autres avaient déjà compromis la machine et sont la source des insultes. Une supposition alternative est le fait que les administrateurs ont configuré les messages d'erreurs de leur système en se servant d'un langage vulgaire. Les insultes sont particulièrement intéressantes car on peut en déduire des aspects sociaux et culturels des attaquants et dans certain cas, par recherche dans des dictionnaires, nous pouvons déterminer le langage utilisé. Durant nos expériences, nous avons pu constater que le pays de l'adresse IP ne correspond pas au langage utilisé. Par exemple, beaucoup d'insultes roumaines provenaient d'adresses IP allemandes. Il est alors facilement imaginable que les hôtes affectés à ces adresses ont été compromis et servent de rebond pour les attaquants.

quitter le pot de miel. Suite à une intervention d'un pot de miel les attaquants peuvent penser qu'ils sont sur un pot de miel à basse interaction et se déconnectent alors du système. D'autres ne trouvent pas de solution au défi imposé et ils quittent le système.

La réaction des attaquants informe sur leurs capacités de résolution de problèmes. Afin de mesurer la réaction d'un attaquant, nous utilisons le temps de réaction entre deux commandes successives ainsi que les chaînes de caractères entrées par l'attaquant suite à cette intervention. La distance de Levenshtein permet quant à elle de

déterminer des erreurs typographique ou des insultes. Une faible distance indique une erreur typographique et une grande distance une insulte.

Ce modèle de comportement des attaquants et des interventions d'un pot de miel nous permet de construire des pots de miel adaptatif. Nous proposons un pot de miel adaptatif qui apprend le meilleur comportement avec des traces d'un pot-de miel à forte interaction. Ensuite nous proposons deux pots de miel apprenant le meilleur comportement de façon autonome.

1.3.2 Apprentissage dans les jeux de pot de miel

Dans le chapitre 6 nous décrivons ces trois pots de miel à forte interaction. L'interaction entre l'attaquant et le pot de miel est modélisé par un jeu en se basant sur la théorie des jeux. Dans un premier temps, nous considérons qu'un pot de miel adaptatif peut soit permettre une commande soit la bloquer. Comme réaction, un attaquant peut réessayer la commande, chercher une commande alternative ou quitter le pot de miel. Dans ce jeu nous définissons deux joueurs: un attaquant et un pot de miel. L'attaquant est considéré unique car notre approche ne s'intéresse pas aux attaquants individuels mais aux informations que le pot de miel adaptatif peut apprendre. Chaque joueur à une fonction de gain dérivée de l'automate probabiliste hiérarchique. Le but d'un attaquant est d'aboutir à son objectif avec le moins d'effort possible et le but du pot de miel est de garder l'attaquant connecté le plus longtemps possible. Ainsi deux types de jeux sont définis. Chaque jeu à sa propre fonction de gain qui prend en entrée les transitions effectuées par un attaquant dans l'automate. Ensuite, nous proposons un simulateur capable d'utiliser des traces de pots de miel à forte interaction en vue de calculer les gains pour chaque joueur. De cette manière, nous pouvons calculer les équilibres de Nash afin de déterminer le meilleur comportement pour chaque joueur selon la théorie des jeux en se concentrant sur les jeux simples. Le désavantage d'une telle approche est qu'elle exige un jeu de données équilibré afin d'estimer les probabilités de transition entre les états. Ces probabilités peuvent à leur tour impacter les gains de chaque joueur. Dans un second temps, nous avons étendu les capacités d'interventions du pot de miel pour augmenter son degré d'adaptabilité. Il peut alors permettre l'exécution d'une commande, la bloquer, la substituer ou insulter l'attaquant. Ce pot de miel est modélisé par l'intermédiaire d'un processus de décision Markovien. Le pot de miel est représenté comme un agent qui opère dans un environnement composé de plusieurs états. Dans notre cas, cet environnement correspond à l'automate hiérarchique probabiliste présenté dans le chapitre 5. Dans chaque état, l'agent a la possibilité d'effectuer des actions. Chaque action dans un état est liée avec un gain défini par la fonction de gain. Nous avons créé deux fonctions de gain. La première a comme objectif de collecter le plus d'informations possibles sur les attaquants comme leurs outils et leurs insultes. La deuxième fonction de gain vise à les garder connectés le plus long possible. Grâce aux processus Markovien de décision, un agent pourrait calculer les actions optimales pour chaque état et ne pas utiliser les probabilités de blocage pour tous les états. Nous avons donc un degré d'adaptabilité plus fin. Avec ces valeurs optimales l'agent pourrait dériver une politique d'opération optimale. Les actions des attaquants influencent l'environnement en effectuant des transitions dans l'automate. D'un côté nous pourrions utiliser une approche par simulation à l'instar de celle utilisée dans le premier jeu. D'un autre côté nous pouvons faire appel à la famille des algorithmes d'apprentissage par renforcement. Plus particulièrement nous nous concentrons sur les algorithmes ne nécessitant pas de modèle exacts de l'environnement. L'avantage de tels algorithmes est qu'ils sont robustes à de légères variations des probabilités de transitions et de la fonction de gain. Ces algorithmes se décomposent en deux parties: l'explorateur et la règle d'apprentissage. Étant donné que l'environnement n'est pas forcément connu par un agent, il doit tout d'abord le découvrir. Cette tâche est accomplie par le composant d'explorateur. Nous avons choisi l'explorateur ϵ - greedy car il a été prouvé que la règle d'apprentissage aboutit à des valeurs optimales pour chaque état visité.

La modélisation d'un pot de miel comme agent et de considérer les attaquants dans l'environnement de l'agent néglige la nature compétitive entre attaquants et opérateur de pot de miel ce qui peut affecter la convergence vers les valeurs optimales. En vue de résoudre ce problème nous utilisons un jeu stochastique. Les actions sont semblables à celles définies dans le processus de décision Markovien et les objectifs des fonctions de gain sont les mêmes. Dans un tel jeu chaque joueur apprend des réactions à chaque état en prenant compte

les actions de son adversaire.

1.3.3 Opération des pots de miel

Dans le chapitre 7 nous décrivons nos contributions complémentaires aux bonnes pratiques de l'opération des pots de miel. Dans le cadre d'un pot de miel à forte interaction, il est conseillé d'utiliser des points d'observation redondants. Un autre aspect important est le contrôle de la source d'observation. Une source d'observation accessible par un attaquant n'est pas nécessairement fiable. Un attaquant peut par exemple effacer ou modifier ses traces. Une source robuste est la capture du trafic réseau. Il est souvent supposé que l'attaquant ne puisse pas prendre le contrôle des équipements réseaux qui inter-connectent le pot de miel. L'équipement qui connecte le pot de miel avec les réseaux publiques est configuré pour dupliquer l'intégralité du trafic permettant ainsi à l'opérateur du pot de miel de l'analyser. Par ailleurs, d'autres sources d'observations, comme le noyau du système ou la machine virtuelle qui opère le pot de miel sont à considérer si l'attaquant utilise des canaux de communications chiffrés. L'opérateur est légalement responsable de son pot de miel. Dans tous les cas et à chaque instant l'opérateur du pot de miel doit être à même de voir toutes les activités sur son pot de miel et, dès qu'il observe que son pot de miel participe à des attaques vers des tiers, il se doit de prendre les mesures nécessaires afin de minimiser les dégâts collatéraux. En cas de constatation d'activités préliminaires qui peuvent mener à des attaques envers de tiers, nous devons arrêter les expériences. Parmi, les bonnes pratiques d'opération de pot de miel à forte interaction, on trouve usuellement:

- Limitation de la bande passante
- Émulation d'internet pour les attaquants
- Déploiement des pare-feux
- Déploiement des systèmes de détection d'intrusions
- Le fait de modifier du trafic vers des tiers afin de le rendre non-nuisible (connection scrubbing)

Nous proposons des contributions additionnelles en nous focalisant sur les méthodes de surveillance des activités du pot de miel, ce qui inclut observations au niveau du réseau mais également au niveau du système d'exploitation. En fonction de la disponibilité de ces sources d'information, nous proposons une implémentation générique dédiée pour construire des pots de miel adaptatifs. Cela se traduit par deux propositions de visualisation de trafic réseau pour ensuite nous concentrer sur l'évaluation de la qualité des outils recommandés dans le cadre des bonnes pratiques d'opération de pots de miel pour extraire les informations. Ceux-ci proposent de capturer et d'analyser l'intégralité du trafic dédié au pot de miel. Cependant, un pot de miel sous une ou plusieurs attaques massives génère rapidement de grands volumes de données qui exigent un temps de traitement important. Durant l'analyse, le pot miel risque d'endommager des ressources des tiers. Afin de diminuer le temps de réaction de l'opérateur de pot de miel, nous préconisons l'utilisation d'une technique de visualisation. Parmi les travaux existants spécifiques à la visualisation des flux réseaux individuels, nous proposons de visualiser des flux agrégés pour palier au problème des données volumineuses et pour avoir un aperçu global sur les activités. Ensuite, notre méthode de visualisation basée sur les coordonnées polaires est capable de:

- détecter des balayage de ports
- détecter des intrusions réussies
- détecter des attaques par recherche exhaustive

Dans le contexte d'analyse des traces réseaux, un problème fondamental relève du niveau de définition de l'identifiant d'un flux réseau. La majorité des outils d'analyse de traces supposent qu'un flux réseau est identifié avec un cinq-uplet composé de l'adresse IP source, du port source, de l'adresse IP destination, du port

destination et du protocole utilisé. Lors des attaques de recherche exhaustive sur un pot de miel un attaquant génère beaucoup de connexion vers le pot de miel. L'adresse du pot de miel ainsi que son port de service est fixe. Le port source est variable. Due au nombre restreint de possibilités de sélection de port source le même port source est réutilisé. Ceci entraîne un fonctionnement incorrect des outils d'analyse du à un mélange de flux. Après une investigation de ce problème nous pouvons déduit un exploit de cette faille. Un attaquant peut créer une bombe PCAP ayant comme but de faire un déni de service sur la machine de l'analyse. Elle consiste en quelques paquets TCP/IP. Cette bombe explose lors de l'analyse de ces paquets et a comme conséquence un effondrement des ressources de la machine effectuant cette analyse. De plus, nous avons décrit une attaque pour cacher un flux dans un autre. Physiquement les paquets correspondant ont été enregistrés mais ignorés par les outils d'analyse que nous avons évalués. Ainsi un attaquant est en mesure cacher le téléchargement d'un outil. Afin d'assurer la collecte des outils téléchargés, nous proposons une méthode dynamique in-vivo d'analyse de programmes malveillants. En combinant cette approche avec les méthodes traditionnelles il est possible de prendre en compte les quelques cas pour lesquels les conditions initiales garantissant le bon fonctionnent du programme ne sont pas garanties durant une analyse in-vitro . Dans le cas des machines virtuelles utilisées dans le contexte de pots de miel à forte interaction, une solution facilement modifiable est souhaitée afin de proposer une preuve de concept pour construire des pots de miel adaptatifs à forte interaction. Notre choix s'est porté sur User-Mode Linux (UML) qui est un système Linux exécuté dans l'espace utilisateur d'un système Linux traditionnel. Parmi ses défauts reportés dans des travaux antérieurs, nous avons mis en exergue une faille et une preuve de concept de déni de service d'un tel système. Face à tous ces problèmes d'opération de pot de miel, nous proposons des techniques de limitation des risques afin de pouvoir faire des expériences du pot de miel adaptatif. Finalement, nous proposons une implantation générique de ce type de pot de miel adaptatif. L'idée principale est que chaque appel système relatif à l'exécution d'un programme doit être confirmé par un démon. L'opérateur de ne doit alors pas rechanger le noyau du pot de miel mais juste ajouter les modules nécessaires d'apprentissage au démon.

1.3.4 Validation expérimentale

Après avoir modélisé le comportement des différents intervenants ainsi que les méthodes d'opération, nous avons réalisé des expériences avec des pot de miel qui sont décrites dans le chapitre 8. Dans une première étape, deux pots de miel ont été déployés dans le but de récupérer des traces d'attaquants servant de base pour les évaluations. Nous avons fait une expérience avec un pot de miel à basse interaction et un pot de miel à forte interaction. L'automate probabilistes hiérarchique est inféré à partir du pot de miel à forte interaction. Cette instance nous sert de référence pour calculer les gains de chaque joueur et dériver les équilibres de Nash pour chaque type de jeux. Ensuite, nous avons instancié et déployé un pot de miel adaptatif avec les stratégies résultantes des équilibre de Nash. Le résultat de cette analyse informe de la probabilité de bloquer les exécutions des programmes relatifs aux attaquants. Il a ainsi été constaté que les attaquants effectuent trois fois plus de commandes comparé à un pot de miel à forte interaction standard en se basant sur la longueur des vecteurs de processus. Cependant, cette approche a trois désavantages. En premier lieu, la stratégie optimale dérivée des équilibres Nash peut dépendre des traces que nous avons utilisées pour le calculer, ce qui implique que les données d'entrée doivent bien être équilibrées. En deuxième lieu, cela suppose des attaquants rationnels alors que ce n'est pas toujours le cas. Finalement, le degré d'adaptabilité du pot de miel est faible. Le pot de miel peut accepter ou bloquer une exécution d'un programme selon une probabilité de blocage sans prendre en compte le contexte de l'exécution. Un attaquant peut ainsi exécuter la même commande plusieurs fois et se rendre compte de la probabilité de blocage. Pour ces raisons et en vue d'augmenter le degré d'adaptabilité nous avons implémenté un pot de miel adaptatif qui est contrôlé avec l'apprentissage par renforcement. Il peut ainsi permettre, bloquer, substituer une commande ou insulter l'attaquant. Avec une telle approche, nous pouvons estimer les meilleurs actions pour chaque état d'après le point de vue du pot de miel. Cette information a été recueillie sous forme d'un tableau où chaque ligne correspond à un état et chaque colonne à une action. Les cellules avec le nombre le plus élevé est le choix préférentiel du pot de miel. Ainsi nous pouvons déterminer les programmes qui doivent être accessibles sur un pot de miel et les programmes qui peuvent être émulés. De

cette façon nous pouvons alors améliorer les pot de miel à basse interaction. Une comparaison en termes de nombre de transitions vers des programmes installés par les attaquants souligne un gain de 3 en faveur du pot de miel adaptatif. Grâce à la première approche d'apprentissage par renforcement, le pot de miel opère dans un environnement incluant les attaquants. Cette approche néglige la nature compétitive établie entre attaquants et pot de miel. Nous avons observé que cette simplification impacte la convergence des valeurs apprises. Afin de palier à ce problème, nous avons mis en œuvre un jeu stochastique dans une autre d'implémentation de pot de miel adaptatif.

1.4 Conclusion et travaux futures

Dès les années quatre-vingt-dix, Cheswick [26] a a montré qu'il est possible d'étudier le comportement des attaquants en interagissant manuellement avec eux. Nous avons fait une expérience similaire sur un pot de miel à forte interaction au cours nous de laquelle un attaquant a installée un outil de balayage de machine qui a monopolisé les ressources du pot de miel. Ainsi aucun autre attaquant ne pouvait se pouvait plus connecter au pot de miel. Nous avons injecté manuellement du code dans cet outil dans le but de provoquer une panne de l'outil. Après un certain temps, l'attaquant est revenu pour inspecter les résultats de son outil, ce qui lui a permis de se rendre compte de la panne. Il a alors relancé l'outil encore une fois pour qui tombera de nouveau en panne. Après cette deuxième panne, l'attaquant a essayé en examiner la cause. Il n'a pas réussi et il a décidé d'installer un autre outil fonctionnant correctement. Grâce à cette intervention stratégique, le pot de miel a pu récupérer deux outils de l'attaquant. Dans ces travaux nous avons proposé un nouveau paradigme de pot de miel adaptatif. Dans le contexte des pots de miel à forte interaction un attaquant peut exécuter des commandes qualifié d'être arbitraires ce qui lui permet d'effectuer les différentes étapes de son attaque. Nous définissions ce type de comportement comme l'avancement de l'attaquant. Un pot de miel adaptatif intervient de façon stratégique pendant les attaques. Dans un tel cas l'attaquant est dévié de son chemin de manière de déclencher une réaction chez l'attaquant pour réaliser son attaque. Nous appelons cette réaction comportement de réponse. Une attaquant peut soit réessayer la commande, soit chercher une commande alternative, soit se déconnecter du pot de miel. En utilisant des critères mesurables comme le temps de réaction ou les entrées de l'attaquant, le pot de miel peut dériver les capacités de l'attaquant. Des attaques automatiques peuvent être différenciés des attaques humaines. De l'un côté si le pot de miel intervient lors d'une attaque automatique alors ces attaques fréquemment échouent. De l'autre côté si l'attaquant répond avec une insulte il est fortement probable que cet attaquant est un attaquant humain. De plus les capacités de résolution de problème peuvent être estimées avec les pot de miel adaptatifs. On peut voir la persistance de l'attaquant. Par exemple si elle réessaie une commande plusieurs fois ou si elle abandonne facilement. De plus on peut voir si elle est assez intelligente de choisir une chemin alternatif pour aboutir à son objectif. Les pot de miel adaptatifs permettent aussi de déterminer la quantité de résistance qu' un attaquant peu subir avant d'abandonner. Toutes ses connaissances sur les attaquants ne peuvent guère être déterminé avec les pots de miel à basse interaction ou à forte interaction. Dans les pots de miel à basse interaction il n'y a pas assez d'interaction avec les attaquants et avec les pots de miel à forte interaction la majorité des actions sont permises et il n'y a pas de résistance aux actions effectuées par les attaquants. Les pots de miel adaptatif peuvent servir des nouvelles sources d'information sur les attaquants permettant d'étudier leurs réactions. L'objectif de mes travaux est de construire des pots de miel adaptatifs qui exploitent de façon autonome ces caractéristiques pour faire face à la large panoplie des attaques présentes sur Internet. La théorie des jeux et l'apprentissage par renforcement permettent au pot de miel de se rapprocher de comportements optimaux paramétrés par l'intermédiaire de fonctions de gains. Deux objectifs sont contrôlés par les fonctions de gains:

- Maximiser les informations sur les attaquants comme leurs entrées ou leurs outils.
- Maintenir les attaquants actifs le plus long temps possible en vue d'étudier leur résistance face aux interventions du pot de miel.

L'avancement des attaquants est modélisé avec un automate hiérarchique probabiliste qui contient deux niveaux d'états. Les attaquants communiquent des entrées qui sont des chaînes de caractères. Ces entrées sont associées à des états de l'automate. Les états du premier niveau sont appelés macro-états et représentent des programmes installés sur le pot de miel. Les programmes installés par les attaquants sont représentés par un état spécial appelé *custom*. Les entrées des attaquants qui sont vides sont transposées dans un état appelé *empty*. Les entrées ne pouvant être affectées à un état selon les règles précédentes sont regroupées dans un état spécifique nommé *insult*. Chaque macro état est un automate qui a des paramètre du programme comme états. Dans cet automate hiérarchique, les attaquants peuvent faire des transitions où chacune d'elle corresponde à une étape d'une attaque. L'avantage de cet automate est qu'(i) il est possible de l'intégrer dans les pots de miel à forte interaction et (ii) qu'aucune aucune intervention humaine n'est nécessaire. Nous avons proposé une méthode automatique qui prend des données brutes comme entrée obtenues par le noyau du système d'exploitation. Grâce aux arbres de processus, les commandes exécutées par le système lui-même peuvent être différenciées de celles des différentes attaquants. Cette distinction est nécessaire afin de ne pas rendre instable le système d'exploitation. En effet, la perturbation des processus vitaux pourrait induire une panne du système d'exploitation. Les instances d'automates hiérarchiques nous ont permis de construire des pots de miel adaptatifs ayant des buts spécifiques. Un objectif d'un pot de miel est contrôlé avec la fonction de gain qui est construite à partir des composants mesurables comme les entrées données par les attaquants et le temps écoulé entre deux entrées successives. Parmi les objectifs nous avons définis (i) la collecte des outils des attaquants ou le déclenchement de nouvelles entrées sur le pot de miel et (ii) le fait de garder les attaquants le plus actif possible.

Les outils des attaquants ainsi que leurs insultes sont particulièrement intéressants pour les opérateurs de pot de miel. Les outils malveillants des attaquants permettent d'améliorer les logiciels anti-virus. De plus, si un opérateur observe un tel outil sur une machine légitime il est fortement probable que cette machine ait été compromise. En collectant des insultes, nos modèles peuvent prédire avec une certaine probabilité qu'un attaquant est un être humain réactif aux pannes. Dans certains cas nous arrivons même à identifier le langage utilisé qui peut être comparer avec le pays correspondant à l'adresse source de l'attaquant. L'objectif de garder les attaquants le plus longtemps actif possible permet d'étudier leurs réactions et de se concentrer sur les attaquants intéressants. Les attaques automatiques ont de faibles capacités de gestion des erreurs et échouent directement face aux interventions stratégiques du pot de miel. Les attaquants novices abandonnent rapidement et les plus curieux restent. Les pots de miel adaptatifs permettent d'exploiter une nouvelle source d'information sur les attaquants qui consiste en la réaction de ces derniers.

Les objectifs de nos travaux étaient d'élaborer des pots de miel adaptatifs qui déclenchent des réactions chez les attaquants sans s'engager dans la course entre développeurs de pots de miel et attaquants. Nous sommes convaincus que la création des pots de miel adaptatifs est indépendant de la technologie utilisée et nous avons donné une preuve du concept où nous avons identifié trois types d'attaques sur nos pots de miel adaptatifs. Deux classes d' attaques viennent des pots de miel à forte interaction et une classe d'attaque est dédiée aux pots de miel adaptatifs.

- Attaques du système
- Attaques du noyau
- Attaques comportementales

Nos pots de miel adaptatifs interviennent au niveau des commandes exécutées par les attaquants. Ceci nécessite une distinction claire de celles exécutées par le pot de miel et celles exécutés par le système lui-même. Ce problème a été résolu en analysant les sous-arbres de processus affilié au processus d'entrée qui est dans notre cas le serveur SSH. Une fois entré dans le système, un attaquant peut exécuter des commandes arbitraires. Il peut par exemple installer un programme qui continue la suite des opérations de façon automatique. L'attaquant peut aussi modifier ensuite le système de telle façon que le système lui-même exécute le programme injecté. Le pot de miel adaptatif permet par défaut toutes les actions du système et donc aussi

l'exécution de ce programme. Un attaquant pourrait aussi installer un serveur de gestion de commandes local sur le pot de miel. L'attaquant se connecterait ensuite à un tel serveur à travers un tunnel SSH. Dans un tel cas le sous-arbre de processus relié à cet attaquant a une hauteur de 1 et le pot de miel adaptatif n'interfère pas avec les commandes lancées par le serveur local. Une piste pour contrer ce problème est l'analyse par marquage [81]. Chaque donnée d'origine d'un attaquant est marquée et est surveillée tout au long de son existence sur le système. Des efforts supplémentaires doivent être faits dans le contexte d'observation du système de fichiers du pot de miel. En effet, en modifiant le système, un attaquant pourrait modifier les états de celui-ci. Si un attaquant se rend compte que quelques programmes sont toujours permis dû au mécanisme d'apprentissage, il pourrait remplacer ces programmes par d'autres habituellement bloqués ou substitués. Une contre-mesure à une telle attaque consiste à calculer des sommes de contrôle des programmes inclus lors de la création du système. Ainsi, un remplacement d'un programme aurait pour conséquence une modification de la somme de contrôle. Le pot de miel peut soit prévenir de telles attaques en interdisant le remplacement des programmes connus, soit arrêter de fonctionner si un tel changement est observé.

Les pots de miel adaptatifs que nous avons proposés dans nos travaux interviennent au niveau noyau du système. De ce fait, une menace potentielle serait d'attaquer le noyau du système. En vue d'accéder à celui-ci l'attaquant dispose de plusieurs options comme par exemple en installant un module noyau. Ceci peut être évité en configurant un noyau monolithique lors de sa configuration. L'attaquant pourrait aussi simplement remplacer le noyau. Par contre, l'attaquant devrait redémarrer le système. Si le pot de miel est implanté dans une machine virtuelle, celle-ci peut être instrumentée de manière à substituer l'action de redémarrer le système une mise à l'arrêt de la machine. Si un opérateur d'un pot de miel observe un système à l'arrêt, il peut alors vérifier son noyau et constater qu'il a été modifié grâce à des sommes de contrôle pré-établies. L'opérateur du pot de miel a ainsi la possibilité de remplacer le noyau modifié par le noyau initial. Un attaquant pourrait aussi directement modifier la mémoire du noyau initiale. L'opérateur du pot de miel doit alors prévenir ces accès. Enfin, l'attaquant a toujours la possibilité d' utiliser un code d'exploit dédié pour le noyau. Dans ce cas, l'opérateur du pot de miel devra donc veiller à mettre à jour le noyau régulièrement. Nous avons modifié directement le code source du noyau afin d'éviter qu'un attaquant désactive nos fonctionnalités d'observation et d'adaptation. Ceci a comme conséquence que les mises à jours du code source doivent être réalisées avec notre noyau modifié. Actuellement, les pots de miel tendent à faire les observations à l'intérieure d'une machine virtuelle. Bien que cette technique soit plus difficile à tromper, il existe des attaques contre cette nouvelle technologie que nous avons discutées dans le chapitre 3 au cours duquel nous avons mis en évidence le fait que les pots de miel à forte interaction ne peuvent exclure tout risque résiduel, ce qui doit être accepté par l' opérateur du pot de miel.

Dans le chapitre 6, nous avons présenté un jeu entre un pot de miel et un attaquant omniprésent. Selon la théorie des jeux, dans les jeux simples, les joueurs doivent être rationnels ce qui n'est pas garanti pour les attaquants. Les script kiddies sont utilisés par des novices qui essaient les attaques décrites par des experts. De même, la spécialisation des attaquants peut varier. Il y a des attaquants présentant des connaissances solides pour un système d'exploitation Windows mais souffrant de lacunes sur le système d'exploitation Linux [197]. De plus la discipline des attaquants est variable. Quelques attaquants sont plus rigoureux que d'autres. Afin de contrer ces problèmes, la théorie des jeux propose des mécanismes pour mesurer ces erreurs. Un pot de miel adaptatif opéré avec l'apprentissage par renforcement apprend son comportement lors de son opération. Par contre, avec cette approche, la nature compétitive entre les attaquants et les pot de miel adaptatifs est ignorée ce qui induit des problèmes de stabilisation de l'apprentissage. En modélisant un jeu stochastique prenant en compte les actions de l'adversaire,la stabilisation des valeurs apprise est améliorée mais le problème de rationalité réapparaît. Afin de mesurer la magnitude de cette incertitude, nous devrons suivre l'approche décrite dans [74]. Il faut alors faire des simulations des attaquants qui ne suivent pas l'équilibre de Nash et calculer l'impact sur les valeurs apprises.

En plus de ces problèmes de stabilisation, un attaquant est également en mesure d'attaquer la méthode d'apprentissage. Comme déjà discuté précédemment, un attaquant pourrait changer les programmes du système et ainsi influencer les transitions entre états afin que le pot de miel apprenne des valeurs incorrectes. Dans un

tel cas, un attaquant visé à apprendre les détails du pot de miel en sacrifiant ses objectifs initiaux. Une autre condition dans les jeux simples stipule qu'un joueur ne change pas de stratégie. Cependant, rien n'empêche un attaquant de le faire en réalité afin d'obtenir des avantages supplémentaires. Par exemple, un attaquant peut abuser des commandes pour aboutir à son objectif. S'il veut lister les fichier dans un répertoire, il peut soit utiliser la commande légitime soit utiliser un effet de bord d'une autre commande. Nous pourrions intégrer dans notre modèle des super - états relatifs à la sémantique des commandes. Cette solution nécessiterait une intervention manuelle de l'opérateur du pot de miel. Normalement, les attaques comportementales précédemment décrites exigent que l'attaquant ait conscience de la présence du pot de miel adaptatif. À chaque exécution d'un programme déclenché par un attaquant, nos pots de miel adaptatifs décident d'intervenir où non. Sur des systèmes Linux, il y a souvent des groupes de commandes qui sont exécutées ensemble. Un attaquant peut trouver suspect si une partie d'entre elles échoue et les autres réussissent. Afin de résoudre ce problème, la sémantique des commandes devrait être inclus dans le modèle du pot de miel adaptatif.

Dans nos travaux nous avons modélisé et évalué des pots de miels adaptatifs. Nous avons exploré la théorie des jeux et l'apprentissage par renforcement dans le contexte des pots de miel à forte interaction. Nous avons étudié les pots de miel adaptatifs dans le contexte de pots de miel SSH. Étendre le paradigme de pot de miel adaptatif sur d'autres types de pots de miel est envisageable comme ceux dédiés à la voix sur IP ou la messagerie. De plus, nous avons considéré uniquement des pots de miel de type serveur. Nous pourrions aussi créer des pots de miels adaptatifs de type client qui émuleraient des utilisateurs se faisant attaquer. Ce champ d'utilisation donnerait lieu à l'utilisation des algorithmes de renforcement plus complexes à l'instar d'agents collaboratifs partageant les informations apprises. En plus des jeux simples considérés dans cette thèse, une approche alternative pour prendre en compte l'incertitude des joueurs est d'intégrer la théorie bayesienne en modélisant des jeux dits bayesiens. Nous songeons également à explorer différents types de jeux comme les jeux de Stackleberg où de multiples joueurs collaborent et où chaque joueur a un rôle spécifique. Il faut aussi considérer la possibilité de modéliser des jeux entre pot de miels collaboratifs hiérarchiques en exploitant les théories d'apprentissage par renforcement qui sont en cours de développement.

Chapter 2

Introduction

2.1 Context

Many organizations have extended their business capabilities and rely on external providers to accomplish important missions and business functions [114]. Often, they offer services to external partners or they rely on external services. These services are outside their organizational boundaries and are not under the control of the organization. Besides these complex business interactions, the complexity of software is continuously increasing, resulting in an increase of implementation flaws. According to Stefan Frei et al. [54] the number and diversity of attacks has exploded: In the time period between 1996 and 2006, the number of software vulnerability reports[1] steadily increased. After examining the publicly available dataset of vulnerabilities reports from 1999 to 2010, we can confirm this trend[2] (see figure 2.1). The appearance of a new technology is usually followed by a new class of vulnerabilities. These vulnerabilities are then exploited by attackers in an exploitation process, until the technology matures. This cyclic phenomenon is accelerated by the fact that more systems are globally interconnected and accessible from any country in the world[24]. In this century the Honeynet community headed by Spitzner [154] made the observation that an unprotected computer would be compromised within a few minutes. Provos et al. [130] made a similar observation seven years later which shows the persistence of the problem. They also found 19 different malicious programs on a vulnerable machine that was directly connected to the Internet for 24 hours. Owners of information systems simply follow the plug-and-play paradigm, where they just connect their systems to the Internet and expect that everything works: they do not bother to identify the services they really need and shut them down or mitigate access to risky services. This lazy behavior has led to self-propagating malicious software, known as worms, which infested the Internet following the turn of the century. Provos et al. [130] reported that the birthrate of malicious software known as malware is steadily increasing and human intervention is too slow [130]. Operating system vendors and Internet service providers have undertaken tremendous efforts to secure their solutions in order to mitigate these outbreaks. Computer Security Incident Response Teams (CSIRTs) and security researchers have run large-scale compaigns to educate users and teach them basic security practices [114].

Cheswick et al. [27] advise that an exposed system should run as few programs as possible. A system with a minimal number of exposed programs is easier to monitor and to manage. This pragmatic approach was formalized and extended by Howard et al. [72]. In order to perform a risk analysis, they developed an attack surface model with three abstract dimensions to Microsoft operating systems. They considered enablers, communication protocols and access rights. If fewer services are exposed, fewer enablers are exposed, resulting in a smaller attack surface. The larger a system's attack surface, the more likely it is to be exploited. Thus, software vendors and system administrators can reduce the attack surface by exposing fewer services. Un-

[1] The number of vulnerability reports is measurable. CVE assigns a unique number to each report. However, a software vulnerability is not in a one to one relation with a vulnerability report. Some software vulnerabilities triggered multiple reports, and a report can addresses multiple systems. Hence it may describe multiple instances of the vulnerability.

[2] A detailed description of these measurements is presented in the appendix section A.

fortunately, some systems can offer little space for reconfiguration and other systems are not under control of administrators. Proxy servers and firewalls have been proposed to control outgoing and incoming connections [44].

By the year 2000, the web presence of organizations had became a crucial business factor. Unfortunately, the developers of these solutions assumed a clean and well-formed inputs from external users. This presumption led to a pandemic of cross-site scripting exploits [78] and SQL-injection attacks. In addition to controlling or hijacking visitors' hosts, attackers can also control the databases that feed the attached web applications. The countermeasure developed by CSIRTs is the definition of best practices for web development with the intent of education of web developers.

Besides their education mission, CSIRTs also study the behaviors and origins of attackers. Attackers actively look for vulnerabilities in new technologies and try to exploit them. A successful exploit usually leads to a system being penetrated through an exposed service, which is a program that is remotely accessible [47]. Usually, the attackers' the first step is to gather information about the target. Attacks are divided into two generic categories. In the first, when attackers have selected a particular victim, for instance a specific industry or a given organization, the term *targeted attack* is used [28]. Targeted attacks are often related to industrial espionage or sabotage. In the second category, when attackers do not care which victim they have selected, the term of random attack is used [28]. In this case, an attacker is just interested in gain access to the resources of the selected victim. A hijacked resource is called a zombie, and is used to attack other targets [201]. An attacker who we call in thesis Eve, could simply probe an address range. If a host replies to the probes, she has discovered the host. This behavior is usually defined as *scanning* [154]. Alternatively she could set up a rogue publicly-available server or compromise a frequently visited server such that users visiting this server are infected. In this case, the term *drive-by malicious software* is used [131]. Such attacks are out of the scope of this thesis.

Attackers often distribute attacking tasks. For instance, groups of attackers specialized in scanning activities share or sell their results to other attackers [112]. Assuming that Eve has discovered a host, she has to exploit it to get access. Sometimes, she could apply a ready-to use-proof-of-concept code [53]. If this is not available she can develop her own exploit code. If she decides to publish it without previously notifying the owners of the system the term *zero-day exploit* is used [125]. If the exploit is successful, she obtains access to the host and the host is compromised. Li et al. [93] used attack observations and created attacker profiles based on the characteristics of the selected target, the characteristics of observed events, and the consequences of the events. Ramsbrock et al. [135] observed attacker behaviors after the successful take-over of a machine and modeled attackers' behaviors as a state machine having seven states. Once Eve had entered a machine, she could check the hardware or software configuration, she could change passwords or change the overall configuration of the compromised system. Following, these activities, Eve usually downloads and installs customized tools and executes them for her malicious purpose.

A combination and simplification of the scanning activities, exploit process and attacker behaviors following the penetration results in the ring hierarchy is presented in figure 2.2. Each ring corresponds to a generic activity by an attacker and also shows observation levels of attacks. We defined five rings of attacker behavior: discovery, exploit, reconnaissance, customization, and post-attack. The deeper (the higher the index of) the ring, the nearer to success is the attack. Having discovered the target, modeled with ring R_0, Eve tries to penetrate it via exploits grouped in the ring R_1. After penetrating the target, she enters a reconnaissance phase, represented by ring R_2. She tries to discover system details, such as the version of the operating system and the installed software. Using this knowledge, she starts to prepare the system. In the ring R_3 she often takes steps to assure further logins and wipe traces of her break-in, then starts to abuse the system. An attacker has a dedicated attack goal presented with ring R^*. For instance, she may want to steal confidential information or hijack the system in order to performing additional attacks.

Each ring in figure 2.2 also represents a type of knowledge about attackers. An attacker's source IP address and source port can be observed in ring R_0. A lookup of the Internet service provider together with the corresponding country of an attacker can be performed. The exploits used are contained in the knowledge of

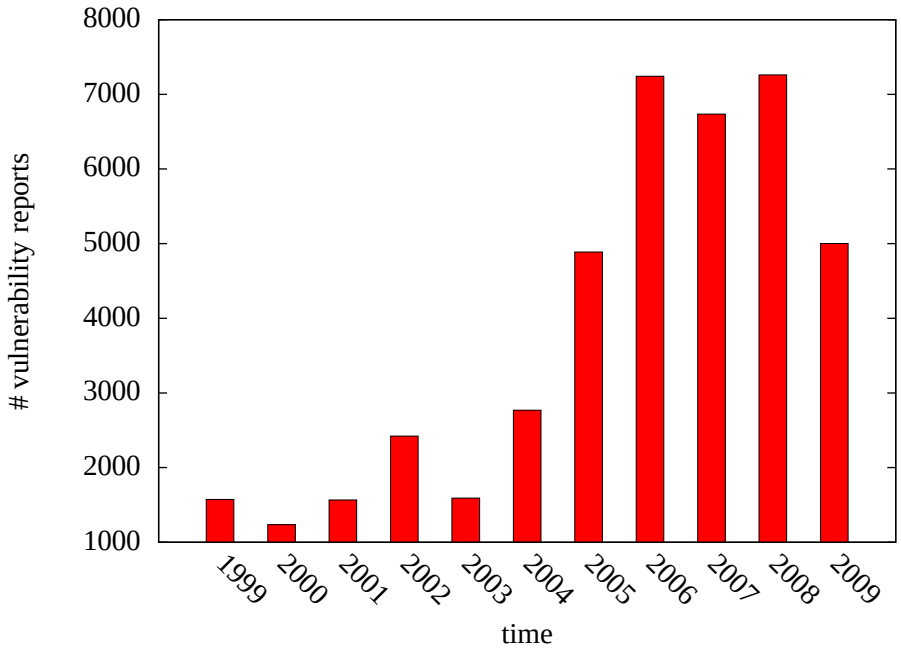


Figure 2.1: Vulnerability Reports Published by MITRE

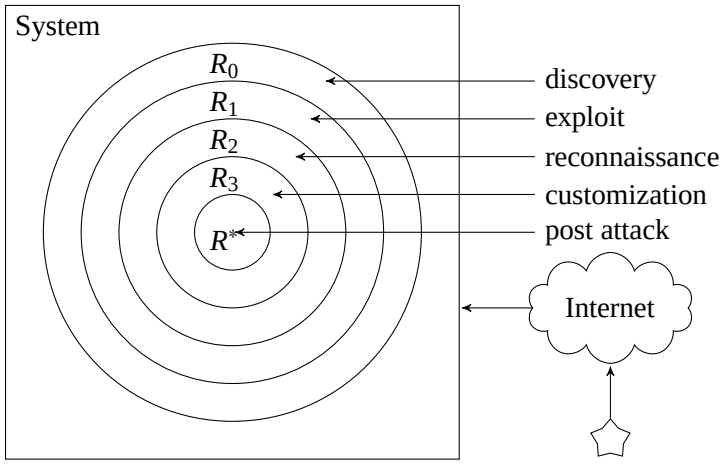


Figure 2.2: System Attack Hierarchy

ring R_1. An attacker's strategy for system investigation can be observed in ring R_2. Some attackers do a careful investigation while others attackers show little interest and simply try to proceed. If a system is not ready for the intended malicious activities, attackers download customized tools. These downloads reveal malicious code repositories and the tools themselves. This knowledge is presented in ring R_3. Malicious code repositories can be taken down and file signatures can be generated for anti-virus solutions. Rings R_2 and R_3 include knowledge of an attacker's activities following break-in, reconnaissance and customization. The presence of attackers is detected by intrusion detection systems which raise an alert during an attack. However, indented knowledge about attackers is currently assessed by deceptive systems known as honeypots. This knowledge is injected in intrusion detection systems. A honeypot is a system to be probed and attacked but which has no legitimate purpose [154]. Different kinds of honeypots have been implemented and are specialized to populate one or more rings of knowledge. One one hand, low-interaction honeypots perform well in gathering information belonging to rings R_0 or R_1. An attacker's source IP address, source ports and the first interaction with a service are recorded. Some low-interaction honeypots are even capable of handling exploits (ring R_1) [7], [60]. For known exploits, they can even download the injected code. High-interaction honeypots on the other hand are used to get more information related to the rings R_2 to R^*. With high-interaction honeypots, it is also possible to collect information concerned to the rings R_0 to R_1. High-interaction honeypots are particularly useful for observing attackers' behavior after a successful penetration. Assets of interest are the tools used and the strategies followed by attackers. Compared to classical intrusion detection systems, honeypots do not have to distinguish between legitimate and malicious events because there are no legitimate events by design. Hence, all activities on a honeypot are suspicious by default and therefore not legitimate. False positives can also be present in the context of honeypots. Such events are triggered by backscatter and misconfiguration [105]. Intrusion detection systems must have a signature or an heuristic to detect an attack. Honeypots simply expose a vulnerable service and so they respond to unknown attacks [4]. Hence, honeypots are a powerful technique for gathering malicious software [7], [130], [139] or information about to attackers [138], [154]. The major advantage of honeypots is that the threat collection process can be partially automated, instead of collecting the malicious software from compromised production machines [130]. Although, honeypots do not prevent attacks, they are complementary tools for detecting and studying attackers. Furthermore, with the information gathered, they provide a means for improving intrusion detection systems and anti-virus solutions.

2.2 Problem Statement

Honeypots are praised as solution for learning from attackers: the National Institute of Standards and Technology even recommends honeypots as a security control mechanism [114]. Early in this century, Spitzner [154], claimed that most security professionals designing security products are ignorant attackers' tools, motivation and tactics. This fact is a strategic advantage for the attacker [154]. Spitzner [154] on page 2 propagates the message that "Have the enemy teach us its own tools, tactics and motivations". Looking back at the last ten years (discussed in chapter 3) of honeypot development and operation, this vision could be updated to "Have the enemy teach us its own tools, ~~tactics~~ and ~~motivations~~".

Honeypots are resources designed to be attacked. They should be completely passive without any legitimate production purpose. Obviously, due to the legal constraints of honeypot operators, appropriate mitigation techniques must be employed when facing real attackers. Various level of operation at risks are used, ranging from the emulation of services to the exposure of real services. As discussed in chapter 3, honeypots have been proposed for the emulation of dedicated services. Low-interaction honeypots allow to study the first interactions with attackers and high-interaction honeypots allow to observe attacker activities after they have compromised a system.

A variety of information levels are recorded according to our ring model. Thus, honeypots focus on the observation of attackers dependent of their interaction level. The interaction level is defined by the features exposed to attackers, instead of actively interacting with attackers in a strategic way. In the context of high-interaction honeypots most research activities have participated in the arms race between attackers and honey-

pot operators. In this self-sustaining phenomenon, honeypot operators propose a new emulation technique or monitoring method and attackers try to defeat it.

Cohen [31] discusses the utility of deception techniques in the context of attackers. Low-interaction honeypots emulate fake services and thus lure attackers. However, the deception technique employed by state-of-the-art honeypots is simplistic. A honeypot has quite a limited arsenal to respond to an attacker. Firstly, honeypot can only offer the emulated service to an attacker. Secondly, the implemented deception technique is applied in a deterministic way due to the fact that a honeypot can choose only this behavior. Hence, we deduce that current honeypots are static observation devices instead of intelligent deception systems.

Despite, these limitations as has been previously discussed, current honeypots perform quite well in collecting programs related to attackers. Also, pre-attack patterns like scanning activities can be assessed quite efficiently. Security researchers have performed distributed honeypot operations [39] and the jointly observed results can be analyzed. From these, researchers have tried to derive the tactics of attackers, for instance, the existence of common attack patterns or behaviors. This assessment is either done manually or partially automated based on the observation results. However, state-of-the-art honeypots are not capable of revealing attack patterns themselves, due to their fixed capabilities. Hence, they lack additional capabilities which they could select as a strategic response to attacks. Due to this lack of choice among different responses provided to attackers only the default behavior of attackers such as the pushing of shellcode or the deploying of malicious software can be observed.

Nicomette et al. [112] report that some attacks are completely automated. Instead of observing a human attacker, the honeypot operator observes an attack script or a malicious program that interacts with the honeypot. Distinguishing between human and automated attackers is challenging, because only the effects of operations can be observed. In addition, it is difficult to draw meaningful conclusions about the skills of an attacker when observing the effects of program executions on a passive honeypot which has strictly deterministic behavior. If additional knowledge should be learned about an attacker, the new type of information has to be measurable in order to draw meaningful conclusions. However, honeypots can observe only the effects of attack operations. Consequently, side-effects must be carefully inspected in order to derive this additional information. As discussed in detail in chapter 3, there is an arms race between attackers and honeypot operators. Attackers try to avoid honeypots because they are monitored devices with no legitimate purposes. From an economic point of view such systems are unattractive for attackers. These systems are by definition designed to fool an attacker and so generally prevent her from reaching her attack goal. From the arms race we can deduce that emulation of fake services are an essential weakness of honeypots, but attackers too have their weaknesses. For instance, they may misinterpret the effects of their actions. Therefore, honeypots having additional features to challenge attackers should act more strategically and defend their interests to so as reveal more information from attackers.

2.3 Contributions

In this thesis we address autonomous and adaptive honeypots that offer graduated challenges to attackers with the purpose of revealing their true nature, skills and linguistic background. The thesis is divided into two parts. The first part describes the previous work and the second part contains our contributions.

Chapter 3 presents the related work on honeypots. It starts with the pioneering activities of the late 1980s, on the observation of attackers in information systems, followed by deception techniques in information systems. At the start of the current century a commonly accepted terminology on honeypots was introduced, providing an enabler for the novel honeypot research communities. Honeypot taxonomies are presented using the commonly accepted honeypot categories, namely low-interaction and high-interaction honeypots. A low-interaction honeypot provides fake services that try to mimic real ones whereas a high-interaction honeypot is an actual vulnerable system, exposed to attackers. Distinct communities work on these different categories of honeypots following a risk management approach. The highest threat is that

loosing the control over a deployed honeypot, so getting involved in attacks against third parties. On one hand, the risk in operating low-interaction honeypots is quite low and they are useful for detecting the presence of attackers. On the other, the risk in operating high-interaction honeypots is quite high because an actual vulnerable system is exposed to attackers, who could hijack it. Hybrid solutions which allow the operational risk to be mitigated are discussed. Security researchers [39] use honeypots to measure attack activities in a distributed large-scale manner. Other security researchers focus on the analysis of the data that is collected via honeypots. In practice, honeypots have been used to collect malicious software. From an attacker perspective, malicious software is often a precious asset. It serves them as tool for malicious activity and therefore they want to stop it falling into the hands of security researchers. Should this happen, anti-virus industry can develop appropriate countermeasures resulting in a disruption of the malicious business activities in which attackers are often involved. Hence, the attackers point of view is considered and the technical limits of state-of-the-art honeypots are summarized. Besides, these technical limitations, neither systems are adaptive and act in a strategic manner such that these weaknesses are compensated.

Chapter 4 presents theories dedicated to the conceptual design of autonomous and adaptive honeypots capable of defending their interests to obtain a given type of knowledge from attackers. We start by providing a short evolution of game theory which is essential to formally frame opponents in a competitive environment. In order to introduce equilibrium concepts, a short primer on formal games is presented. Rational players always try to achieve such an equilibrium and equilibria can be discovered with a wide variety of algorithms. Previous work on the application of game theory to information security is described. Game theory has already been addressed in the context of honeypots. The majority of the contributions focus on games at the infrastructure level, where honeypots are used as passive information security sensors which cannot take individual strategic decisions. In games, players can make errors which may impact their behaviors. Countermeasures and mitigation techniques are discussed. In practice, not all the parameters of a game are known a priori, which motivates further investigations in the domain of goal-oriented learning. First, we introduce Markov Decision Processes to provide a theoretical foundation. They formally describe an agent which operates in an environment with the purpose of optimizing a reward signal. Each decision taken by an agent is rewarded or punished. However, either not all the parameters are known, or some of the assumptions of the learning model are not true in practice. Therefore, learning methods are enumerated such that optimal rewards can be learned instead of formally computed. The proposed learning techniques are then extended with stochastic games, in which two competitive opponents operate in a shared environment where each agent has its own interests. This approach permits the modeling of games where each player learns from its interaction with its opponent.

Chapter 5 presents a transversal model of attacker and honeypot behaviors, which is used in the adaptation mechanisms described in chapter 6. The generic behavior of attackers is consolidated from previous literature. Attackers input sequences of strings to a compromised system. These strings are sequences of characters, and correspond to commands, invalid commands, typographic errors or insults. These strings are then mapped to states in a hierarchical probabilistic automaton. Each program installed on the compromised system corresponds to a macro state. We introduce a state hierarchy due to the semantic difference of command line arguments. A program is a distinct automaton and also has states which model the command line arguments. A major requirement of this model is that it can be implemented in automated adaptive honeypots without human intervention. Therefore, we formally describe a process where direct system observations are transformed such that they can be mapped onto the hierarchical probabilistic automaton. An attacker executing commands on the honeypot causes transitions in the hierarchical probabilistic automaton. The purpose of this thesis is to make honeypots autonomous and adaptive instead of deterministically emulating a behavior as it is the case for current honeypots. The interventions of adaptive honeypots trigger responses from attackers. We defined three actions for an adaptive honeypot. Firstly, an adaptive honeypot can allow a program execution. In this case it behaves

exactly like a traditional honeypot. Secondly, it can prevent the execution of a program for strategic reasons. Third, it can substitute a program execution in order to assess the skills of an attacker. Finally, it can insult an attacker, aiming to irritate her and to provoke a response that reveals her social and linguistic background. In response to these interventions by the honeypot, the attacker can retry a command, select an alternative command, insult the honeypot or leave it.

Chapter 6 presents an adaptation mechanism for honeypots, such that the level of adaptation is continuously increased and improved. A honeypot usually follows the objective of acquiring information from attackers. As a first step, games are modeled between a honeypot and an omnipresent attacker rather than individual attackers. Two games are modeled with appropriate payoff functions for each player. These payoffs are computed with Monte Carlo simulations, which use an instantiated hierarchical probabilistic automaton as input. From these payoffs the optimal strategy profiles are computed for each player by following Nash equilibria. For the honeypot the best blocking probability is the most promising result from these computations. However, this adaptation mechanism is quite coarse-grained and neglects the contextual state of an attack. Moreover, the optimal blocking probabilities are derived from simulations which assume perfectly equilibrated bootstrap data. In order to address these issues and to increase the level of interactions of an adaptive honeypot, we apply a goal-oriented learning approach, namely reinforcement learning. An adaptive honeypot, Heliza, is modeled as an agent which optimizes reward signals in an environment. In this context, the omnipresent attacker is a part of the environment, which is modeled by the previously developed hierarchical probabilistic automaton. An attacker attempts to make a transition to a given state. Heliza has to decide whether this transition should be allowed, blocked or detoured. Each decision yields a reward which is optimized. By following such an approach, the competitive nature of the relationship between attackers and Heliza is neglected, and this may impact the learning processes. Therefore, we modeled the interaction between an attacker and the honeypot as stochastic game and used a learning algorithm to approximate optimal behaviors.

Chapter 7 The operation of honeypots is a risky operation. Three major threats have been identified. Firstly, attackers could take over the honeypot such that the operator losses control over it. Secondly, an attacker could perform stealthy operations with effects that are not directly visible to the honeypot operator. Thirdly, attackers could abuse the honeypot to attack third parties. Hence, accurate monitoring techniques must be be used to monitor operation. The greatest threat is that attackers could attack third parties resulting in legal consequences for the honeypot operator. Therefore, the honeypot operator needs to be aware of all the ongoing activities in which the honeypot is involved. Hence, all the network traffic related to the honeypot is recorded and firewalls and intrusion detection systems are deployed as described in the best practices for honeypot operation [154]. However, the volume of recorded traffic may quickly increase when the honeypot is under heavy attack. While still assuming a quick reaction by the operator, we propose two novel network visualization techniques. The first is an approach which uses aggregated networks as inputs, aiming to highlight activities at subnet layer. The second in contrast aims to detect the nature of outgoing attack. Attackers could push volatile programs to the honeypot such that no traces are left on the honeypot's file system. Therefore, we evaluated network forensic tools in order to ensure that we can recover such programs from the network traces. However, we determined fundamental design flaws in the network forensic tools. This led us to examine the system call layer, which can be considered as an additional monitoring layer. The monitoring of the malicious programs installed and operated by attackers has the advantage that attackers must first ensure that all the prerequisites of the tools are fulfilled, a requirement which is often not met for dynamic in-vitro analysis systems [190]. We finally propose a generic adaptive honeypot framework where different learning algorithms can be quickly implemented and evaluated and which is freely available [174].

Chapter 8 describes the experiments performed during our research activities. A regular high-interaction honeypot was set up to recover the traces of attackers. A low-interaction honeypot was also operated to recover traces which were later used for comparing different adaptive honeypots. From the high-interaction

honeypot we recovered the process trees, which are transformed into process vectors. These process vectors were then used as input data for the generation of our hierarchical probabilistic automaton. We implemented a simulator that used such an automaton as input and which could compute the payoffs for each player. These payoffs were then used to compute the optimal strategy profiles for each player. We implemented an adaptive high-interaction honeypot which was configured with the results of the simulations and noticed an increase in interactions with attackers. We increased the level of interaction by operating an adaptive high-interaction honeypot driven by reinforcement learning. We noticed even more interaction with attackers by comparing the traces of the adaptive honeypot with the traces from the low-interaction and the regular high-interaction honeypot. However, with the learning algorithm used we noticed a slow convergence to optimal values due to the fact that the competitive nature of attackers and honeypots were being neglected. Therefore we operated another adaptive honeypot using a learning algorithm derived from stochastic games to take this phenomenon into account.

Chapter 9 formally summarizes our research activities and enumerates potential future work. We suggest a new paradigm of adaptive honeypots to reveal more information about attackers. These novel honeypots adapt their behavior to attackers. The adaptation is controlled with game theory and reinforcement learning aiming at automated operation. We illustrated the example of adaptive high-interaction honeypots exposing a vulnerable SSH service. However, we believe that the adaptation paradigm to attackers is not related to a given technology. We could also develop additional games between attackers and a honeypot at different levels than those at the operating system kernel. We leveraged adaptive honeypot games between an omnipresent attacker and a honeypot. We could explore also more complex game scenarios considering multiple collaborative players.

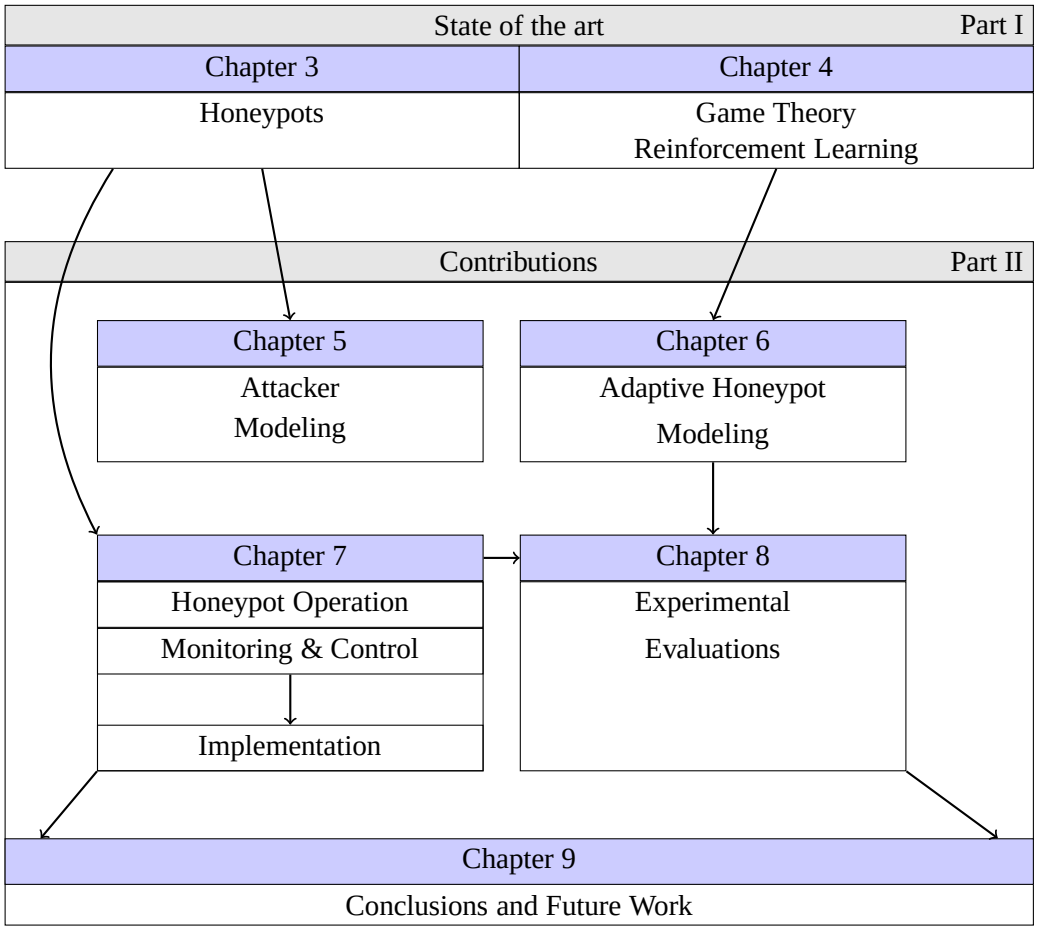


Figure 2.3: Chapter Organization

The dependencies of the chapters are shown in figure 2.3. The previous work is summarized in the first part and our contributions are shown in the second part. The previous work allowed the modeling of attacker behaviors which enabled the modeling of adaptive honeypots that take into account game theory and reinforcement learning. The foundations of these two theories are presented in the first part. We identified challenges in the previous activities on honeypots and developed honeypot monitoring and control mechanisms influencing the design choice of our generic adaptive honeypot framework. We have evaluated our various models of adaptive honeypots using the generic adaptive honeypot framework. Finally, we conclude this dissertation and summarize the limitations of our adaptive honeypot framework and the various interaction models including experimental aspects.

Part I

State of the art

Chapter 3

Honeypots

3.1 Honeypot Evolution

In the late eighties, Stoll [158] reported an attack on the Lawrence Berkeley Laboratory. Instead of keeping the intruder out, he took a novel approach. He decided to provide counterfeit military documents as bait and monitored access to them. These documents were accessed by the intruder, who was then tracked. A little later, in the early nineties, Cheswick [26] became interested in the attacks on his Internet gateway. He had the idea of adding some pseudo-services with no production purpose and recording all activities related to those services. After a while an attacker used these services which provided actual content. The author noticed that the attack came from a stolen account and that the services needed to be improved. The decision was taken to manually interact with the attacker by providing him with forged content.

At this time only a few attackers were active and with the help of system administrators, the official authorities took the necessary steps to physically discover attackers and call them to account. Today the scale of attacks has changed [54]. In the late nineties, Cohen [31] discussed several deception techniques that could be used to lure attackers. A tool, the deception tool kit (DTK), was developed allowing pseudo services to be defined in advance. Popular examples are TCP services returning a previously generated content to attackers. The major contribution of Spitzner [154] is a commonly accepted honeypot terminology for the throw-away machines [26] and deception devices [31] proposed earlier by others. The commonly-accepted definition is shown in definition 1.

Definition 1. *"A honeypot is security resource whose value lies in being probed, attacked, or compromised." [154] page 23.*

A honeypot is particularly useful for gathering information about attackers, such as their technological and ethnological background [139]. Most work in the honeypot communities is centered on the gathering of technical information such as source addresses, malicious software and the locations of malicious code repositories. Honeypots should closely mimic real production systems in order to be attractive to attackers [154]. Spitzner [154] proposes adding user accounts to a system, creating email accounts, forging documents and synthesizing a command history.

The honeypot movement started in the beginning of this century to grow. More researchers worked on honeypots. Google offers a publicly available service to determine trends on given subjects. This service is mainly offered to journalists, economists and investors because these communities often analyze trends on given subjects. In the past, these communities have had to base their data on monthly governmental reports that are frequently amended and which lack hard data [76]. Figure 3.1 shows Google's search trend [76] for honeypots in the field of information security. The higher the trend, the more popular the keyword is. The detailed experiment is described in the appendix in chapter B. On the x-axis is represented the time from 2004 to 2010 expressed in months. The data was only available from 2004, which explains why the period from 2000 to 2004 is not covered. On the y-axis is shown the search index. The first peak was mid-2004. At this

time Provos et al. [128] implemented a honeypot called *Honeyd*. The paper was published an had a high impact.Moreover, the authors published the source code. *Honeyd* is reliable and easy to use. Furthermore, the operation at risk of such an honeypot is almost zero. Hence, many people were interested in it, explaining the peak in interest. The following years the interests slightly decreased. However, at the end of 2006, botnets became an emergent threat on the Internet and became popular in the news. Zou et al. [201], proposed a method for detecting and tracking botnets with honeypots. Portokalidis et al. [125] modified the popular virtual machine Qemu [18] with additional monitoring features targeted at malicious software analysis and honeypot operation. The authors made the source code publicly available and many other researchers started to use it [130], [153]. One year later, Xuxian et al. [194] extended Qemu with further monitoring features such as the availability to observe executed processes on the guest system. To the best of our knowledge, this implementation is not freely available. The drawback of modified Qemu is its performance, because every instruction is emulated on the host CPU. In the same area, researchers [89], [125], proposed automated generation of intrusion detection signatures from honeypots. This conceptual approach led to new avenues for research. Instead of generating signatures, Leita et al. [91] generated scripts for *Honeyd*. At this point in time *Honeyd* was still the least risky honeypot to operate [91].

The evolution of the number of scientific publications in the honeypot area is presented in figure 3.2. A detailed description of the methodology, is presented in the appendix B. The x-axis shows the time while the y-axis gives the number of scientific publications. In 2000 Spitzner [154] introduced honeypot terminology, leading to a small number of publications. *Honeyd* was a major milestone in the field of low-interaction honeypots. *Honeyd* does not require expert knowledge to be operated and the operational risk is low, because counterfeit services are emulated in user space. Due to the fact that the source code of *Honeyd* was publicly available, numerous researchers [40], [65], [89], [91], conducted experiments with *Honeyd* and proposed extensions resulting in the significant growth of papers from 2004 to 2005. *Honeyd* was also particularly useful in detecting Internet worms [40], [120].

Although the CPU manufacturer Intel had already introduced a new virtualization technology in 2006, the major break-through for the honeypot community came in 2007. Compromised honeypots can more easily be switched off and reinstalled when virtualization is used. Conceptually, the external monitoring of an emulated system was not new, as it had been demonstrated previously by Dunlap et al. [45]. However, Intel's virtualization technology (VT) revived the idea of external monitoring. In the case of VT, ring zero from Intel architectures, is not the lowest anymore. Intel introduced an hypervisor ring, where additional honeypot monitoring capabilities can be implemented [43]. Many other researchers made new propositions based on virtual machine monitoring in the context of honeypots [68], [155]. The publication peak in 2008, mainly resulting from the vitual machine introspection revival and observed in figure 3.2, is corresponding only to a minor peak in Google's search index in figure 3.1. The search index trend can be seen to be decaying. It reached a peak in 2004, due to *Honeyd*. The NIST organization enumerates the honeypot technology in their control mechanisms. The honeypot control mechanism is defined as: "The information system includes components specifically designed to be the target of malicious attacks for the purpose of detecting, deflecting, and analyzing such attacks." p.119 [114]. A typical implementation of this control mechanism is the deployment of low-interaction honeypots detecting the presence of attackers.

The operation of high-interaction honeypots is still risky, which discourages people from using it. High-interaction honeypots are particularly useful to study the behavior of attackers after they penetrated a system. We believe that most people interested in information security are willing to study break-in attempts and less people are interested in studying the behaviors of attackers after a successful penetration. Hence, we deduce that only a small community is interested in this information. Only a few organizations such as CSIRTs or groups of information security researchers are interested in studying the behavior of attackers after they have penetrated a system. In addition, the lack of publicly available source code prevents people to use it and to extend it. Consequently a smaller number of Google queries are observed resulting in a decrease in the presented Google trend.

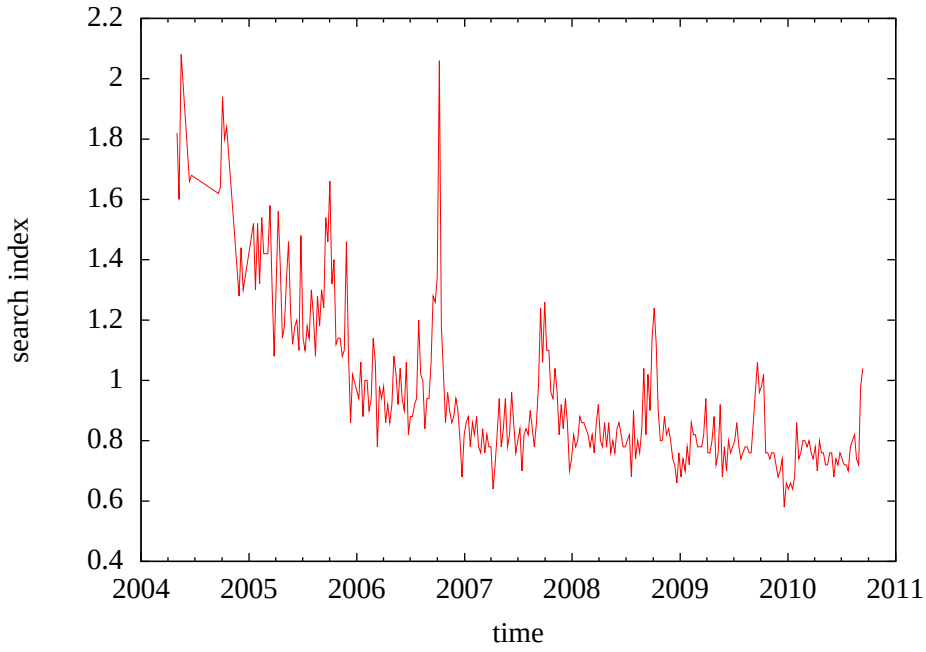


Figure 3.1: Search Index Trend

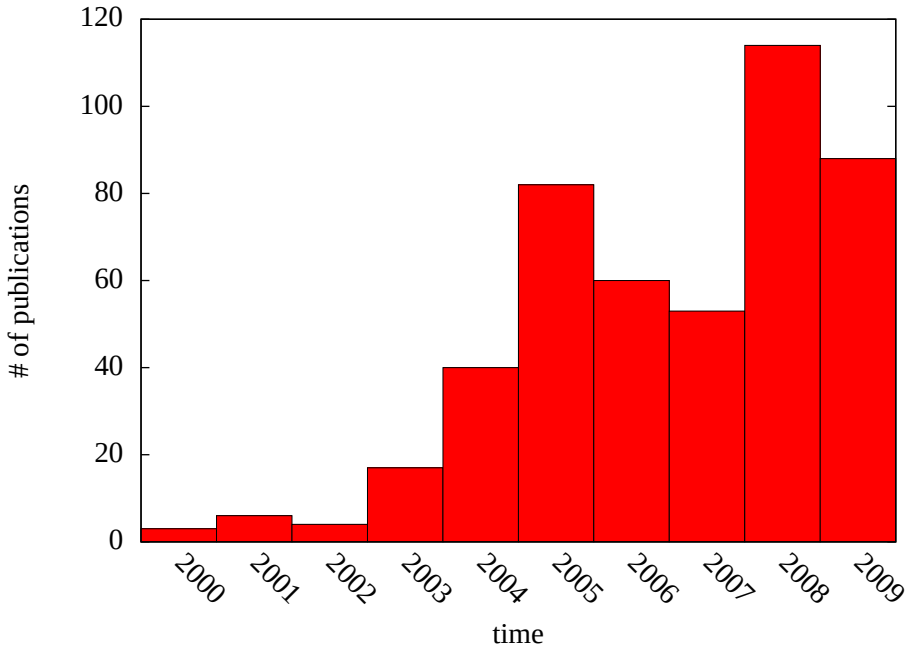


Figure 3.2: Scientific Publications about Honeypots

3.2 Honeypot Classifications

Data control and data capture are essential paradigms for a safe honeypot operation [154]. Honeypot operators should know at any time what is happening to the honeypot and the monitoring system should have multiple layers in order to mitigate the failure of a layer [154]. McCarty [101] reported a case where an attacker switched off the honeypot's monitoring features. This is a worst-case scenario for honeypot operators. If attackers were able to abuse the honeypot for performing further attacks, honeypot operators would be legally responsible for their systems and would actively participate in attacks. Therefore, it is essential to be able to classify honeypots. When a honeypot operator encounters a new honeypot implementation and if the honeypot category is known, then a decision on whether to operate the honeypot or not is easier to take. Seifert et al. [147] classified honeypots according to six classes: interaction level, data capture, containment, distribution appearance, communication interface, and role in a multi-tier architecture.

Interaction-level Honeypots usually expose some functionality to an attacker. In order to limit the attacker's control, the exposed functionality is limited in some manner. From the point of view of interactions, honeypots are divided into low- and high-interaction honeypots [130], [147], [154]. The term mid-interaction honeypot is also sometimes used [130] to specify honeypots emulating services having more features than low-interaction honeypots. *Multipot* is a medium interaction honeypot designed for Windows platforms capable to emulate known vulnerabilities. Having received the shell code[1] Multipot [77] tries to emulate it with the purpose of downloading the additional payload. The interaction between attackers with a compromised system is often modeled with attack-trees [145] where a node corresponds to a stage of an attack and the edges represent the logical connections between nodes.

Data capture Spitzner [154] and Cheswick et al. [27] advise the use of multiple monitoring and data-capture layers. Seifert et al. [147] propose the use of a honeypot's data capture capabilities, which describe the type of data it is able to capture, as honeypot classification criteria. They define four values for this category: events, attacks, intrusion, and none. When a honeypot collects data such as scanning activities, it collects events; when it collects data on a brute-force attack on an account, it collects attacks and when it collects data about an attacker who has penetrated the system, it collects information about intrusions.

Containment In most countries, honeypot operators are legally responsible for their systems [2]. Best practice shows that a firewall should be put in front of the honeypot in order to protect other organizations and to prevent it being actively involved in attacks on third parties [154]. Nicomette et al. [112] configured a firewall in front a honeypot such that attackers could not reach third parties through the honeypot. Such firewall configurations with additional intrusion detection systems are called honeywalls [25]. Having penetrated a system, attackers often want to download tools from the Internet [112], [154]. If the firewall blocks all connections from the honeypot to the Internet, the honeypot is unattractive to attackers and attacks can only be partially observed [112], [154]. Spitzner [154] proposes limiting the number of Internet connections from the honeypot. This allows attackers to download their tools but not to perform destructive attacks [154]. Alata et al. [2] propose simulating external hosts for the honeypot by using dynamic connection redirection mechanisms. Seifert et al. [147] defined these mitigation techniques as *containment*. They identified four containment techniques. Firstly, a honeypot can block actions of attackers. Secondly, a honeypot could defuse attacker actions. In this case, the attacker can connect to the target, but the content of the connection is tampered with so as to remove dangerous payloads. Thirdly, a honeypot can also slowdown an attacker. An example is to artificially slow down connections related to attacks. Finally, a honeypot can block all actions by attackers.

Distribution appearance Seifert et al. [147] introduced the distribution appearance class as a classification criterion. Honeypots can be stand-alone or distributed systems. A stand-alone honeypot only interacts

[1] A detailed explaination of shell code is presented at page 35.

with the attacker and his environment, while a distributed honeypot interacts both with the attacker and with additional entities. Examples are automated log file analysis or attacker tracking programs.

Communication interface Seifert et al. [147] also uses the communication interface to classify honeypots. The communication interface defines the means by which the information about attackers can be collected. For instance, network traffic can be collected or system logs can be recovered, via an Application Programming Interface (API).

Role in multi-tier architecture Seifert et al. [147] describe server-side honeypots. Such a honeypot passively waits for connections from attackers and waits to be exploited. He also identified client honeypots, which try to mimic a user whose machine gets compromised when he or she visits a malicious web server. This thesis mainly focuses on server-side honeypots.

The voluntary interaction with attackers is the riskiest part of honeypot operation [192]. When a new honeypot design is published, it is usually classified according its interaction level. Low-interaction honeypots have often a lower software complexity than high-interaction honeypots and hence are easier to manage. Honeypots are divided into research and production honeypots: production honeypots are systems for detecting the presence of attackers, whereas research honeypots are used to study attacks [154].

Low-interaction Honeypots

The *Honeyd* community leveraged the low-interaction honeypot definition shown in 1. Low-interaction honeypots try to mimic services that are frequently the subject to exploitation. Attackers connect to such a honeypot and provide some input such as their source IP address, source port and the requested service. However, attackers are usually not allowed to store information and execute their own programs on the honeypot. Hence, the operation of low-interaction honeypots is less risky than allowing arbitrary code execution. Low-interaction honeypots are typically production honeypots intended to assess the presence of attackers. The deployment of low-interaction honeypots helps to understand the evolution of automated threats such as the propagation of malicious software.

Definition 1. *Low-interaction honeypots simulate only services that cannot be exploited to get complete access to the honeypot [71].*

Cohen [31] describes a variety of low-interaction honeypots. He proposed the deception tool kit DTK to emulate pseudo services. The main purpose is to lure automated attack tools and waste attacker's time. When an attacker scans a host that deploys DTK, she sees many vulnerabilities and needs to determine which vulnerabilities are real and which are fake. Obviously, the attacker should not be able to automate this task, and thus must spend a lot of time to test them sequentially. Hence, the workload of the attacker is increased and the defender gains time to track the attacker.

Liston [95] used a similar approach to slowdown self-propagating malicious software. He developed a program called Labrea, which uses unused IP address space aiming to decelerate worms. When a worm connects, Labrea artificially slows down the connection attempt. It listens for network packets using the *pcap* library [79]. When there is a connection attempt, it periodically announces a TCP window size of 0, simulating network congestion on the targeted host. Most TCP/IP implementations on attacking hosts then wait for a time slot t and retry. Furthermore, the interval t grows exponentially if the congestion persists increasing the slowdown factor. Honeyd is a program that runs in user space emulating a complete TCP/IP stack. This means that the program does not need to run in kernel space, which reduces the operational risk. It can be configured to respond to requests for an entire sub-network, meaning that it responds to each request to any host address h_a in this network. When an attacker probes the address h_a, *Honeyd* catches the corresponding IP packet via the *pcap* library and crafts a response packet that is sent via a raw socket [186]. The reason for using raw sockets with *pcap* is that arbitrary packets can be intercepted and crafted. A regular TCP/IP stack is not designed to

respond to arbitrary IP addresses. A honeypot operator can configure forged outputs for any given service. For instance, a Perl script could suply the code for a minimal web server that uses the standard input and standard output file descriptors. This script is then attached to port 80 which is the default port for web servers communicating via HTTP. When an attacker probes this port, the customized program is executed and the content is returned to the attacker. Each TCP/IP stack has its own characteristics and can be fingerprinted [57]. Thus, for low-interaction honeypots it is important to mimic these behaviors in order to be similar to a real TCP/IP stack and so to lure network scanners [128]. *Honeyd* uses the signatures database of the network scanner Nmap and it can be configured to make a given IP address emulate the TCP/IP stack of a particular operating system. Provos et al. [128] call such a behavior a personality. The configuration file of *Honeyd* allows arbitrary routing topologies to be emulated using manipulated TTL (Time to Live [156]) fields .

Honeyd is a powerful tool for emulating known services. The back doors of malicious programs have been reverse-engineered and scripts have been developed to emulate those back doors. An example is the Kuang2 back door[2] [65]. However, some malicious programs use their own communication protocols on random ports and have not yet been reverse engineered. Hence, Tillman proposes Honeytrap [164]. This program listens for TCP packets. When a SYN packet [156] hits Honeytrap, it dynamically assigns a listener to that port and can respond according to four modes. Firstly falsified content can be returned to the attacker. In this case Honeytrap behaves like *Honeyd*. Secondly, Honeytrap can mirror the connection attempting to connect to the same port on the attacker. An attacker is often a compromised machine and is looking for a vulnerable service s_0. Due to the fact that the machine is already compromised and is running the service s_0 it is likely to interpret the requests. Honeytrap mirrors the received request to the attacker and records the resulting conversation. Thirdly, Honeytrap can forward the requests to another host such as a high-interaction honeypot. Finally, Honeytrap has an ignore mode. In this mode, the packets are just dropped. Bakos [8] used a different implementation approach in order to attract connections to arbitrary ports. If a service is not running on a regular host, the corresponding TCP/IP stack replies with a TCP [156] packet having the RST flag [156] set to tell the requester that the service is not running [157]. In contrast of emulating a TCP/IP stack [128], [95], Bakos [8] use *iptables* to redirect the traffic to a dedicated service capable of replying. Antonatos et al. [5] exploited the DHCP protocol in order to get more IP addresses for a host aiming for greater network visibility. Their honeypot is called Honey@home and requests additional IP addresses by sending DHCP requests to a local DHCP server. The required addresses are then used for setting up tunnels to high-interaction honeypots.

Now that we have discussed the monitoring techniques for multiple IP addresses and arbitrary ports, we enumerate techniques for determining the content to be returned to an attacker. For instance, efforts have been undertaken to simulate real production services with *Honeyd* [65]. This is usually achieved with scripts written in bash, perl, python, Windows Command-Line Shell Language, Visual Basic or JavaScript. However, *Honeyd* is not particularly bound to a particular scripting language. It provides input, output and error channels and passes information through environment variables to use the script, a process that is executed by the operating system. When an attacker sends a request, it can be read from the standard input file descriptor, while replies are communicated through the standard output file descriptor. *Honeyd* sets environment variables for the source IP address, the source port, the destination address, the destination port and the protocol used (UDP, TCP, ICMP) [156]. These variables can be accessed and further processed by the script. *Honeyd* ships with some useful default scripts [65], for instance, the banner of an SSH server. Obviously, an attacker can only connect to this pseudo-service and can neither perform a complete key-exchange nor get a login to the machine. There are also scripts that emulate a telnet login on common CISCO routers. These always respond to the attacker that credentials are incorrect and emulate connection timeouts. [65] also presents a script that lists a directory for an IIS server (Microsoft's proprietary web server). Email server scripts have also been developed, as have pseudo-FTP servers returning static content.

Leita et al. [91] propose a more advanced alternative to manually implementing scripts: automated generation of scripts from recorded honeypot traces. Their solution, Scriptgen, is composed of four functional

[2]Many malicious programs establish back doors. These present themselves as new services that, for example, allow the attacker continuing access to the compromised system.

modules. The message sequence factory module uses a *pcap* file containing captured network traffic as input, and produces messages. A message is defined as the longest consecutive set of bytes going in the same direction. Two possible directions are handled: the first is from client to server and the second is from server to client. The authors use these messages with their state machine builder module in order to generate raw state machines, which are composed of edges and states. An edge is annotated with an exchanged message recovered from traces and the observation frequency. The authors then simplify these raw machines with a module called the state machine simplifier. To do this, they first do macro clustering based on a sequence alignment approach, then perform a region analysis, where they group homogeneous sequences of bytes. Both clustering methods help to reduce the number of states and edges. The regrouped states become abstract states. Finally, they use the simplified automaton for feeding a script generator in order to generate scripts for *Honeyd*. The authors acknowledged that their generated scripts are only approximations to real services but they claim that they are sufficient to fool automated attacks.

In the previous static configuration examples[3] for *Honeyd* [65], only the initial interactions are recorded. On one hand, using such an approach reduces code complexity and thus the risk that attacker can take over the honeypot is reduced. On the other, not much can be learned from attackers because only the first interactions can be observed. The counterfeit SSH server, proposed in [65], can only present a banner to the attacker. In this case only the source IP address can be logged; the credentials of attackers cannot be observed because this level of complexity is not implemented in the script. Coret [34] implemented a pseudo SSH server, Kojoney, simulating a complete SSH login and some basic bash commands. All his code is implemented in Python, and an attacker cannot execute or install her own programs. Hence, this Kojoney is classified as a low-interaction honeypot.

Allowing the execution of arbitrary code by attackers is a dangerous decision. The attacker could switch of the monitoring capabilities [101] or abuse the honeypot to attack third parties. In this case, the honeypot actively participates in malicious attacks and in most countries the honeypot operator is legally responsible. Attackers often exploit vulnerabilities. They send a crafted request, known as an exploit, to a vulnerable service. The vulnerable service wrongly interprets the request in such away that the crafted request is partially executed. The hostile code that is executed in a defective service is called shellcode. Historically, this code spawns a shell enabling an attacker full access to the machine [122]. For self-propagating worms, the infected code usually downloads another malicious program instead of spawning a shell [122]. These shellcodes are particularly dangerous, because when they hit their target, they allow attackers to take over a service without any human intervention. In the network intrusion community much efforts has been undertaken to detect shellcode [122]. Historically an attacker simply added their shellcode in a legitimate requests without any attempt to hide it. For a traditional buffer-overflow exploit, the return address is overwritten. However, finding this address depends on the process running the service and stack offsets differ slighly. Attackers use a NOP sled, which is often considered as landing-zone. This means that the beginning of the shellcode starts with No OPeration (NOP) instructions in order to make the code more robust against slightly different offsets [116]. When the offset is only one byte ahead, the opcode alignment might be incorrect leading to invalid instructions and results in a failure of the shellcode. For the instruction detection community such an attack is relatively easy to detect. They simply need to search for a 0x90 byte sequence defining the NOP sled [122]. Erickson et al. [47] noticed that instructions using one word are need to constructed a landing zone. The richness of Intel's x86 instruction set provides many other alternatives. For instance, the instruction to increment the register ECX by 1 has an opcode of 0x41. When the intrusion detection system interprets this opcode, it sees the ASCII code `A` and some intrusion detection systems consider that ASCII codes are benign. The advantage of this approach is that a landing zone populated with such irrelevant instructions is still robust for attackers because the value of the ECX register is discarded anyhow. Most exploits want to download another program or to spawn a shell. Due to Intel's i386 instruction set shellcodes share common features [47]. Intrusion detection systems use these common parts to detect shellcode. As a countermeasure, attackers encrypt their shellcode resulting in a small routine that decrypts the remaining shellcode. After the exploit, the shellcode is decrypted in the defective

[3] Scriptgen is excluded.

service and the decrypted shellcode is executed [122]. Even in this case an intrusion detection system can still create a signature for the encrypted hostile shellcode. Hence, attackers have created polymorphic shellcodes which slightly changes a few instructions such that the shellcode looks different for each target. Intrusion detection systems have two choices: they can try to decrypt the shellcode or they can emulate it.

The danger of self-propagating malicious software comes from the fact that the exploit process is fully-automated. An infected machine scans for other vulnerable machines. After having found one, the exploit is automatically performed and the new infected machine starts to infect other vulnerable machines. Malicious programs often share the same spreading code, but download a different malicious program. The previously described honeypots can capture the connection attempts of malicious machines and the exploit code. However, they are unable to download the malicious program. Hence, Baecher et al. [7] developed a tool called *Nepenthes*. The purpose of *Nepenthes* is to collect malicious programs. It exposes services on a host denoted collector which emulates fake vulnerable services. When a compromised machine connects to the collector it can upload the shellcode. The corresponding vulnerability module on the collector decodes the shellcode and downloads the malicious program as if it were a normal infected host. However, the program is not executed but stored for later analysis.

Baecher et al. [7] define that Nepenthes is a low-interaction honeypot because it simply emulates known vulnerabilities. When new vulnerabilities are discovered, a vulnerability module must be developed for *Nepenthes*. Nepenthes is implemented using the C++ programming language. A worst-case scenario is a software vulnerability in *Nepenthes*' implementation. Provos et al. [130] illustrated that malware collecting tools can be detected. After such a detection an attacker could craft a special request to take over the collector. Hence, Göbel [60] developed a similar tool called Amun. The major difference is that Amun is developed in Python which is less vulnerable to buffer overflow attacks. In the worst case, attackers could crash the collector by triggering an exception instead of taking it over. The latest example of this type of honeypot is Dionaea [6] able to handle IPv4, IPv6 and voice over IP services.

One one hand, low-interaction honeypots are partially useful for detecting attackers. The source address and the requested service can be recorded. In the best case, known exploits can be emulated and malicious programs can be downloaded. The software complexity of low-interaction honeypots is usually low and attackers cannot execute arbitrary code on them. Hence, their operational risk is low, which is an advantage for those who wish to run them in large numbers. On the other hand, they cannot collect malicious programs that use unknown vulnerabilities. Furthermore, an attacker's activity after having exploited a system cannot be observed with a low-interaction honeypot. However, all the rings of our ring hierarchy, presented in section 2, can be addressed with high-interaction honeypots.

High-Interaction Honeypots

Cheswick [26] mentions throwaway machines having real security holes with the sole purpose of being attacked a long time before the honeypot movement. These services can be compromised by attackers and their behavior can be studied. He also discusses jails created with the `chroot` command. This command permits a different system root to be set for a process. The new root may be a subdirectory of the overall file system and a program running in this jail can only access the files in this subdirectory and its descendants. He concludes that such a jail is not perfectly secure and not entirely invisible. He advised using throwaway machines with real security holes that are externally monitored by a second machine. He proposes capturing all the network traffic and thus observes an attacker's activity. This was a valid solution for its time, because attackers connected to the service using clear-text protocols for remote access. Hence, their activities could be monitored. However by the mid-nineties, encryption was gaining popularity. Most telnet services [126] were replaced with SSH [196] and a significant fraction of communications became encrypted. In 2000, Spitzner [154] called these throwaway machines high-interaction honeypots. In order to overcome the problem of monitoring encrypted network traffic, data capture should be performed on the honeypot itself. Spitzner [154] proposes the use of a modified command line shell to monitor an attacker's activity [154]. He also defended the idea that no information about attackers should be locally stored on a honeypot: if information is locally stored, an attacker could tamper

with or delete the recorded information. Attackers often installed their own command line interpreters on honeypots [9]. This prevents an attacker from being monitored by the honeypot's shell. To counteract this trend, the monitoring features moved into kernel space. Balas et al. [9] implemented Sebek a Linux kernel module for monitoring an attacker's keystrokes and related file accesses. Sebek uses the rootkit technology initially developed by attackers who wished to hide their presence on compromised machines. It detours the read system call, and so is able to capture the content of opened files and the standard input stream. Hence, even encrypted communications can be captured because the monitoring happens after the decryption. Moreover, Sebek transmits the acquired data over a network to a server that is unlikely to be under the control of an attacker.

A little bit later McCarty [101] published an article in which he describes techniques to detect these additional monitoring features. Attackers inspect the addresses of the system calls. When a function is detoured its address appears in an unexpected address range. McCarty [101] also describes techniques that attackers could use to switch off the monitoring features. Dunlap et al. [45] argue that the kernel monitoring approaches depend on the integrity of the operating system and assume that the operating system is trustworthy. As proved by McCarty, this is not always the case. Hence, Dunlap et al. [45] suggest performing the monitoring at virtual machine level. This approach implies that the honeypot is operated in a virtual machine. A virtual machine is a program that emulates a complete operating system and is run under another operating system. A popular open-source virtual machine is a Linux or Windows operating system that uses the emulator Qemu [18]. Portokalidis et al. [124] extended Qemu with additional monitoring features in order to detect zero-day exploits. These features make it suitable for the operation of honeypots. The authors developed a generic method to detect stack smashing, heap corruption and format string attacks. The key idea is to detect them at the CPU level. Qemu is a virtual machine allowing each executed instruction to be monitored and controlled. Something that is hardly possible with a kernel level approach. The honeypot of Portokalidis et al. [124] generates signatures when an attack is encountered. A honeypot operator can see the exploit when it happens but is still not able to track all the regular commands that an attacker has entered. Hence Xuxian et al. [194] extended Qemu such that the executed programs be recovered. With such an approach, a honeypot operator is able to determine the actions of an attacker following a successful exploit. Qemu translates each instruction in user space resulting a high performance overhead comparing to the hardware-assisted virtual machines.

As previously discussed, with low-interaction honeypots attackers can only perform a limited set of interactions with the honeypot. This limitation helps the operator to control the honeypot. Consequently, this model scales better than the model of high-interaction honeypots. Hence, low-interaction honeypots are better suited for assessing attacks and for collecting a significant amount of data [112], [167]. However, low-interaction honeypots are not suitable if the intend is to study attacker's behavior within a system due to the limited amount of exposed features. While had Cheswick [26] discussed throwaway machines with real security holes in the early 90s and Bellovin had implemented pseudo services it was not until 2000 that evolve quickly. Figure 3.3 shows the evolution of the numbers of scientific publications about low- and high-interaction honeypots. A detailed description about the experiment is presented in appendix B. Publications in 2000 and 2001 dealt only with high-interaction honeypots, describing the experiments made by the authors did with high-interaction honeypots, explaining what the attacker did. The deception toolkit had been available since 1998. In 2003 first implementations of *Honeyd* appeared. Gupta [113] discussed the effectiveness of both tools. He pinpointed their limits and proposed an iterative approach to the operation of honeypots because the attackers' tactics and strategies are unknown. In 2006 virtual machines gained popularity for the operation of honeypots. The advantage of virtual machines is that they can be set up easily and can be reset to a clean state without reinstalling a machine from scratch. This process can even be partially automated.

The breakthrough for high-interaction honeypot can be observed in the year 2008, when the VT technology is used for building honeypots. Instead of making the observations in kernel space they are performed at virtual hardware level. The major disadvantage of virtual machine introspection is that assumptions must be made on the interpretation of the collected data. A deeper monitoring level implies a larger interpretation effort. Inside a virtual machine, memory chunks and CPU registers could be inspected. A system call number is put in a

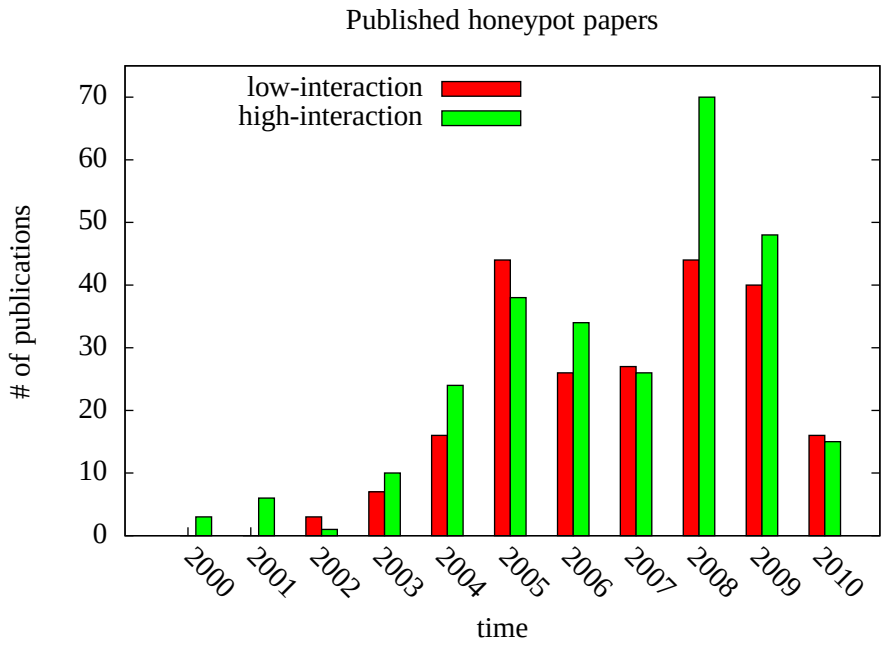


Figure 3.3: Scientific Publications about Low- and High-interaction Honeypots

register. A honeypot operator must know it and to which system call its value corresponds. An attacker could thus change the system call numbering scheme by modifying the local kernel. In this scenario, a honeypot operator could believe that an attacker read a file but in reality she wrote a file. The honeypot operator must also be capable of determining the results of a system call. The CPU registers EAX or RAX contain the return value of a system call. If an attacker redesigns these return values in the local kernel, the honeypot operator might believe that the system calls failed but they were successful in reality. Hence, we believe that the operation of a high-interaction honeypot always includes a residual risk that has to be accepted by the honeypot operator. However, high-interaction honeypots are particularly interesting to study attacker behaviors following a successful penetration.

3.3 Honeypot Research Activities

This powerful idea of honeypots, leads to numerous research activities. The major trends of research activities are grouped in this section.

3.3.1 Attacker Observation and Information Gathering

The basic idea of honeypots is to provide a counterfeit infrastructure in a dangerous environment. This infrastructure is closely monitored. When, the infrastructure is attacked, intruders' behavior is studied and information is gathered from the attackers. Low-interaction honeypots only provide a limited set of features for attackers, but they are quite useful for detecting attacks in networks. A connection to *Honeyd* discloses the source IP address of an attacker. The requested service is also revealed. An attacker can also connect to a high-interaction honeypot. Portokalidis et. al. [124] proposed Argos a customized virtual machine to detect zero day exploits. The high-interaction honeypot community is also interested in post-exploit activities. Most research efforts have focused on improving the monitoring capabilities of high-interaction honeypots. Alata et al. [3] modified the TTY driver installed on the honeypot which enabled them to mirror the attacker's terminal

and gather key-strokes dynamics. Moreover, they track system calls to the kernel for the case attackers bypass a terminal. Attackers often switch of the terminal's echo and blindly type commands. Xuxian et al. record only processes that are executed [194]. Alata et al. [3] introduced an additional system call that is used within their exposed program to differentiate system calls related to attackers and the system itself. Xuxian et al. [194] simply trap all the system calls. Zuge et al. [199] use high-interaction honeypots to collect self-propagating malicious software. The authors compared the amount of data they collected with their high-interaction honeypot with the programs they collected from the low-interaction honeypot *Nepenthes* running the two experiments the same time. They collected more unique malware samples with their high-interaction honeypot because Nepenthes only emulates a subset of vulnerabilities. The authors used the TCP stream reassembly engine *snort* from the open source intrusion detection system and looked for Portable Executable (PE) files in the network streams tapped from the honeypot. This approach is used to collect malicious software targeted at Microsoft Windows operating systems. In addition, they monitored the file system of the virtual machine that is operating the honeypot. If a new malicious program is installed, it is automatically copied. The file system is periodically remotely imaged and all the files are traversed. Each file is compared with a list of initial files installed on the honeypot. If a new file appears, a new malicious program has been detected. The advantage of this approach compared to those used by Nepenthes, is that samples that use unknown malicious programs can be collected. Hence, low-interaction and high-interaction honeypots enable to collect programs related to attackers. However, the current honeypots do not reveal any information about the usefulness of these programs from the attacker's points of view. For instance, a tool to unpack commonly used archives has a lower value to an attacker, than a database program including stolen information.

3.3.2 Honeypot Management

The operation of a high-interaction honeypot often involves costs. The major challenge is to handle arbitrary code execution by attackers. Honeypots may crash and need to be restarted, or additional firewall rules must be configured. The reason for this partially retroactive method is that the attackers' actions and objectives are often unknown. The operation of a honeypot is an iterative process [113]. Initially assumptions are made about attackers' behaviors. After a while, attackers violate these assumptions. The honeypot operator has to adjust the honeypot according to new assumptions. Further important considerations are the scalability and network visibility. A honeypot that is operated on only one public IP address has a lower visibility than a collection of honeypots. Honeypot operation is subject to legal restrictions. If the honeypot gets involved in further attacks that involve third parties, the honeypot operator is legally responsible [2]. Hence, Spitzner [154] and Balas et al. [10] propose letting attackers in, but using intrusion detection systems and customized firewalls to prevent them from attacking third parties. Chamales [25] calls these firewalls *honeywall*s. Portokalidis et al. [125] and Jianwei et al. [199], claim that *honeywall*s are perfectly suited to high-interaction honeypots that aim only to collect the first program that is acquired by attackers. In case an attack exploits the services and pushes the shellcode, the honeypot operator has already achieved his goal. However, if the objective is to study attacker behaviors after the break-in, providing a *honeywall* is not really effective. Attackers usually try to connect to the Internet after they have compromised a system [2], [3], [181]. When they fail, they usually give up and leave the honeypot. Alata et al. [2] proposed a customized Linux firewall that can dynamically redirects connections or simulate or drop them. They give an example in which an attacker scanned an entire network. In this case, the customized kernel forged connections to make an attacker believe that Internet connectivity was available. In the scanning example, only SYN/ACK [156] or RST TCP packets are forged. However, if an attacker connects to a pseudo-service, a content has to be returned. In essence, this is the same problem faced by low-interaction honeypots.

The risk of not attacking third parties can be reduced by using a number of different strategies. Xuxian et al. [193] proposed a hybrid honeypot solution. The low-interaction honeypot *Honeyd* is installed at different locations on different IP addresses forwarding the traffic to a centralized cluster of high-interaction honeypots. This centralized approach helps to handle failures. On the client side, code complexity is reduced, resulting in a low failure rate. However, the operational risk of the centralized high-interaction honeypot remains, and a trust

relationship has to be established among the entities operating the low-interaction honeypots and the centralized high-interaction honeypots. Vrable et al. [173] address the scalability issues of high-interaction honeypots. They claim that most high-interaction honeypots waste CPU cycles during the long periods when they are waiting for attackers. The authors exploit these cycles using a distributed architecture of high-interaction honeypots to emulate on-demand a large fleet of high-interaction honeypots while using minimal physical resources. Most of the proposed concepts to mitigate the damage resulting from honeypots are static solutions based on fixed assumptions. However, honeypots themselves do not address self-management aspects such that they assess the benefits and losses during operation.

3.3.3 Distributed Honeypot Operation

Setting up of a honeypot on a public IPv4 address permits an observation of only $\frac{1}{2^{32}}$ of the address space[4]. A Linux kernel can be configured to have more than one IP address per network interface [127]. However, it is recommended that no more than 16 addresses should be used in order to avoid instability [94]. If a larger address space is to be monitored, more physical interfaces are needed. This limitation can be overcome by using *Honeyd*. In this case the kernel TCP/IP stack is not used, but is emulated in user space. *Honeyd* enables entire subnetworks to be emulated. However, no organization can emulate all free IP addresses. Firstly, an organization can use only IP addresses that they own. Several organizations such as [141] manage the address allocation. Secondly, the networks need to be properly routed in order to guarantee the stability of the Internet. In this case, only owned networks can be monitored, and no meaningful assumptions about the neighboring networks can be made. Hence, Dacier [38] described the setup of a distributed network of honeypots composed of a set of low-interaction honeypots. He created a project, Leurré.com to which people could contribute by setting up a low-interaction honeypot. Information is centrally logged and each partner can access the common collected data. In 2008, the project comprised 50 different platforms in 30 countries. Initially (Leurré.com V.1.0), *Honeyd* was deployed aiming to reduce operational risk. The EURECOM institute offered a Compact Disk with the necessary software to facilitate deployment. They also offered a centralized log collection facility and granted access to partners. In Leurré V.1.0., The authors focused mainly on the measurement of scanning activities, but subsequently the authors focused on collecting malicious programs with their low-interaction honeypots.

The Leurré.com project evolved into an EU project called WOMBAT [39]. The authors used SGNET instead of honeyd. The authors collected malicious programs using feedback from dynamic malware analysis. Any pointers to other malicious programs found by the analysis were acquired. The authors of this project have three objectives: Data acquisition, data enrichment and threat analysis. The NoHa, a framework 6 EU project [88] is a similar European project to set up honeypots. In the intrusion detection community, the SurfIDS [160] is a distributed intrusion detection system project capable of identifying malicious traffic by means of shared information retrieved from other sensors. The purpose of these projects is to collaboratively share threat information and collect malicious data on a large scale, aiming to provide a better visibility and understanding of global malicious activities. An exhaustive list of distributed malicious data collection projects can be found in [99]. The advantage of distributed honeypots is that a higher visibility of malicious activities is reached than with a single honeypot. The higher visibility frequently results in a larger amount of data that is collected with such solutions. Hence, data aggregation techniques and an evaluation of data processing tools is mandatory for a meaningful interpretation of the data.

3.3.4 Honeypot Data Analysis

Threat collection and measurement is of particular interest is particularly for the anti-virus industry. Their portfolio of commercial security products must keep track of the attackers. In addition to the submission of malicious software by their customers, they also use honeypots. Once they have a malicious program, they

[4]The number 2^{32} is only a theoretical value. Some addresses should not be assigned and some addresses should not be routed [157] resulting in a smaller address space.

can generate a signature for it then update and distribute their malicious software database. A customer hit by the new threat can then detect the infection and in some cases undo its effects. In order to undo the effects of malicious software, these effects must be known. Because the source code is seldom available, the software must be reverse-engineered. Two approaches are used: A static analysis of the malicious program can be performed [49]. In this case the program is not executed, but disassembled. The machine code is transformed into assembly code which is semi-automatically analyzed. Malicious software developers are aware of this technique and use tricks to confound the analysis. In order to counter such evasion techniques, a malicious program can be dynamically inspected [49]. To do this, the program is executed in isolation and environmental changes are recorded. A fundamental problem of a dynamic analysis is that it is not known if all code branches are executed. This problem can be reduced to the Halting problem [49]. In addition, attackers try to detect such environments and provide erroneous analysis results on encountering them. Malicious software researchers try to make the environment robust against such attacks. Ironically, this has led to a an arms race situation similar to that which the intrusion detection and honeypot community has encountered. In the malicious software analysis community a similar downward spiral like that previously discussed, can be observed: Function hooks have been moved from user space [190] to kernel space and have ended up in virtual hardware space [43].

Besides installing programs on a honeypot attackers often interact with files on honeypot. An operator should always be aware of exactly what has happened on a honeypot. In his PhD thesis, Fairbanks [48] described new forensic techniques focusing on EXT3 and EXT4 file system structures. File system changes are recovered from the file system journal. His Timekeeper tool permits the recovery of sequences of file system events such as file modification, file creation and file removal. Nicomette et al. [112], operated a high-interaction honeypot for 419 days. They exposed a vulnerable SSH server hosted on a Linux operating system to the Internet. The SSH server was constantly under attack and brute force attacks were launched against it. The authors of the paper focused on differentiating between human and automated attacks. In addition, they analyzed the user name and password pairs that the attackers used. They clustered these pairs and compared the lists used by attackers with popular password crackers. Their purpose was to identify attacker communities which share password lists and compromised accounts. The authors concluded that attackers use customized dictionaries and discovered that there are specialized attacker communities for focusing on brute-force attacks. According to the authors, only a few dictionaries were shared among attackers. Due to the long-term nature of the experiment, the authors observed that some dictionaries were stable over time. Furthermore, the authors noticed that high- and low-interaction honeypots running at the same time were targeted by different communities of attackers: none of the IP addresses observed on their high-interaction honeypot appeared in the records of the low-interaction honeypot deployed in the Leurré project. The authors used the presence of typographic errors and the mode of data transmission with the honeypot as criteria to distinguish human attackers from automated attacks. Automated attacks transmit the data faster, and more often group it into blocks, than human attackers. An examination of the TTY buffer [33] enabled the authors to identify copy and paste actions on the honeypot and thus they were able to distinguish script kiddies from more experienced attackers. They concluded that they observed more script kiddies than experienced attackers. They also found out that attacks are composed of various attack phases and that the attackers use different source addresses to perform attacks. Besides the collection of malicious tools and the assessment of the nature of attackers, additional information could be assessed such as their reactions to failures aiming to reveal their skills and social background.

3.4 Detecting Honeypots

Honeypots have become popular and their deployment has increased. Attackers are aware of this trend, and try to avoid. A honeypot tries to mimic a legitimate server. This leads to artifacts that are searched for attackers. Provos et al. [128] note that this happens as early as the in the discovery phase. When an attacker scans a network he may analyze the arrival of network packets. A Windows kernel behaves differently from a Linux kernel; each host's TCP/IP stack has its own fingerprint [57]. These differences allow an attacker to expect the services that are available. In [130] Provos et al. considers the case where a host's TCP/IP stack behaves like that

of a Linux host yet announces typically Windows services. Obviously, an attacker can determine this mismatch, assume that the host is a honeypot or a gateway and stay out. In [128], Provos et al. discuss techniques for fooling commonly-used fingerprinting techniques. Looking for artifacts on a host is also a valid approach. The resulting Linux kernel can be complied as regular program and emulate an entire operating system. This kernel is called User-Mode-Linux (UML) and is popular for setting up high-interaction honeypots. Holz et al. [70] described UML detection techniques. For instance, the CPU information from the /proc file system already reveals that UML is in use. A listing of running processes shows the processes emulating the kernel, which are not present on a regular system. The kernel command line also shows that a UML is running. Linux kernel messages also give some configuration hints concerning UML. The set of running processes is also a different from that on a regular Linux system. Wagener et al. [180] went a step further. After successfully identifying the User Mode Linux, they could generate a system call with specially crafted parameters to take down the entire emulated operating system. The system call causes a kernel panic consequently file system content is not synchronized with the hard drive and information that might help the honeypot operator to investigate the attack is lost.

Virtual honeypots are often emulated with a VMware virtual machine [130]. Unlike UML, VMware allows to emulate arbitrary Operating Systems. A simple test an attacker can make allows to query the serial numbers of the hardware used. This information is the same for all virtual machines. Early VMware versions also had implemented a back door, which could be detected by attackers [70]. To do this, an attacker needed to implement a small assembler routine.

Honeypots are sometimes operated in conjunction with a debugger in order to trace all function calls executed by the emulated operating system. Holz et al. [70] describe techniques to detect debuggers. The previously discussed detection techniques for detecting User Mode Linux and VMware virtual machines prove only that a virtual machine is running. The detected virtual machine might also have a legitimate production purpose. Balas [9] proposes Sebek, a kernel monitoring technique. He detours commonly used system calls resulting in address-range changes in the system call table. Bill McCarty [101] reports attacks which can detect these changes. The author also describes a methodology to recover the original address and a technique to dynamically update the system call table. If this is done, the monitoring techniques can be disabled by attackers. The out-of-the box monitoring techniques proposed by Dunlap et al. [45] and Xuxian et al. [194], make a major assumptions about memory structure and system call order [68]. If an attacker interferes with these assumptions, the external analysis results may be corrupted.

3.5 Summary

This chapter summarizes the evolution of honeypots. The breakthrough on attacker monitoring came with the honeypot terminology proposed by Spitzner [154]. A honeypot is a resource designed to be attacked in contrast to classical intrusion detection systems no distinction has to be made between legitimate and malicious activities. All activity on a honeypot is by definition suspicious because the honeypot has no production purpose and the activity is certainly not legitimate.

Spitzner [154] also proposed a classification for honeypots, which has been adopted by the honeypot community dividing honeypots into low-, mid- and high-interaction honeypots. However, in academic communities mid-interaction honeypots are uncommon. A low-interaction honeypot emulates pseudo services without supporting infrastructure. The most widely-used low-interaction honeypot is *Honeyd* [128]. It allows conclusions to be drawn about attacker presence and attack trends. A high-interaction honeypot is a real resource having a full-blown operating system that is exposed to attackers. Obviously, these operating systems must have additional covert monitoring features in order to monitor attackers. There is an arms race between attackers and honeypot operators. Once honeypot operators have developed a new monitoring techniques, attackers try to evade it. This has resulted in a trend to make observations at progressively lower levels of a system. The host intrusion detection system has faced a similar downward spiral [59]. However, the honeypot community does not aim at simply detecting intrusions but also tracking attacker activities after a successful exploit.

Name	Reference	Type	Capabilities	Source Code			Maintained	Operational costs	Limits
				Technology	Available	Functional			
DTK	[31]	low	R_0	Unix	Yes	Yes	No	low	L_1
Honeyd	[128]	low	R_0	PI	Yes	Yes	Yes	low	L_1
Labrea	[95]	low	R_0	PI	Yes	Yes	Yes	low	L_2
Honey@home	[5]	mixed	R_{0-3}	Linux	No	No	No	low	L_4
Kojoney	[34]	low	R_{0-2}	PI	Yes	Yes	Yes	low	L_3
Nepenthes	[7]	low	R_{0-1}	Linux	Yes	Yes	Yes	low	L_1
Amun	[60]	low	R_{0-1}	Linux	Yes	Yes	Yes	low	L_1
ReVirt	[45]	high	R_{0-3}	Linux	Yes	Yes	No	high	$L_5 + L_7$
Sebek	[9]	high	R_{0-3}	Linux$^+$	Yes	Yes	No	high	L_5
Argos	[125]	high	R_{0-3}	Linux	Yes	Yes	No	high	L_5
VMScope	[195]	high	R_{0-3}	VT	No	No	No	high	L_6
Dionaea	[6]	low	R_{0-1}	Linux	Yes	Yes	Yes	low	L_1

Table 3.1: Honeypot Evaluation Grid

Spitzner [154] divides honeypots into production and research honeypots. A production honeypot is a counterfeit resource deployed in operational infrastructure with simply to detect the presence of attackers. For this application, low-interaction honeypots are particularly interesting due to their low operational risk. Research honeypots are used to study attackers' behaviors. Research honeypots have been used to detect zero-day exploits [125], which are known only to the attacker community, and to generate signatures for intrusion detection systems [89]. They are also used to study post-attack behaviors. Most high-interaction honeypot publications summarize specific observations, [3], [135] or propose novel monitoring techniques [9], [45].

Table 3.1 summarizes the proposed honeypots that have been previously discussed and that are frequently used by security researchers. The first column gives the name of the honeypot. On the second column is denoted if it is high-interaction or low-interaction honeypot. Some authors have proposed mitigation and management techniques for honeypots [5], [193]. The key concept is to combine low-interaction honeypots with high-interaction honeypots. In this case, the term *mixed* is used. The capabilities column indicates the information that can be gathered with the given honeypot by reference to the ring hierarchy set out in chapter 2. The fourth column shows the technology used. If it is a program that uses generic programming interfaces, the term PI (Platform Independence) is used. If hardware virtualization technology is used the term VT appears. Otherwise, the operating system's family is used. Columns five to seven describe aspects of the source code. Available source code is extremely useful for making an accurate evaluation. It permits researchers to reproduce experiments, to analyze implementation details and to extend the approach without reinventing the wheel. We show whether the source code is available and functional from an external point of view. If the source code is not available we assume that it is not functional. We define a functional program as a program that does exactly do what is described in the related publication. The last column includes a codification about the limits of a given honeypot. These are explained in table 3.2. In the context of distributed architectures, a honeypot is distributed in two parts. A front-end with a minimalistic design aiming to expose a service that is then relayed to the back-end. Sometimes the software of the front-end is available but the the back-end unreachable. In such a case it is likely that the community abandoned their infrastructure.

Operating systems constantly evolve and source code needs to be maintained. For instance, function names change or disappear in updates of shared libraries. Well-maintained code, means that people still are working on the project and that the program can be used on modern systems. Two criteria have been used for determining if a project is maintained. A honeypot such as *Honeyd*, Labrea and Nepenthes are included in standard Linux distributions which are maintained. Other honeypots, like Kojoney Argos or Dionaea are maintained by the

Code	Description
L_1	Considerable development efforts required for each service or vulnerability
L_2	Attacker is only slowed down
L_3	Root privileges are required to run the honeypot
L_4	Back-end service is not available
L_5	Honeypot can be taken over by attackers
L_6	Honeypot is not available. Hence the experiments cannot be reproduced
L_7	Source code is obsolete and not maintained
$^+$	A Microsoft Windows version is available
PI	Platform independent

Table 3.2: Codifications

main developers. The eighth column describes operational costs. The term low is used when a honeypot can be operated without much human monitoring. This is frequently the case for low-interaction honeypots, which usually run with limited user privileges on protected machines. Even when an attacker manages to take over the honeypot, she has still only limited access.

In contrast, the operation of a high-interaction honeypot usually needs a lot of human effort. Attackers have full access to the machine. Despite mitigation techniques attackers could find unconventional methods of bypassing them. If they are able to cause damage to other parties, the honeypot operator is legally reliable. Furthermore, data is collected at a low level of the honeypot and additional efforts is needed to reconstruct the overall activity of the honeypot. For instance, when all network traffic is recorded from the honeypot, it must be reassembled and the communication protocol need to be analyzed. Using the knowledge ring model introduced in chapter 2 and following a generic attack scheme, attackers usually attempt to discover potential victims (ring R_0). When they have found a victim they exploit it (ring R_1). After a successful exploit, they usually start to investigate the system (ring R_2) and start abusing it (ring $R_3 - R^*$). This knowledge is assessed with the previously described honeypots.

3.6 Limitations

The initial purpose of research honeypots is to study attackers and determine their tactics and tools [154]. As summarized in table 3.1, low-interaction honeypots are particularly well-suited for collecting data at the information rings R_0 to R_2. By setting up pseudo-services, attackers can be identified and exploits can be collected and inspected. Their advantage is their low management. A high-interaction honeypot exposes an entire vulnerable system to attackers. An attacker can execute arbitrary commands on a high-interaction honeypot because all commands are permitted. On high-interaction honeypots attackers can customize the system by using their own tools. Usually, attackers manage to penetrate the system and install malicious programs. According to Spitzner [154] on page 2 *"Have the enemy teach us its own tools, tactics, and motivations"*. Cheswick [26] and Bellovin [20] pioneered the idea of interacting with attackers and described their encounters. Spitzner [154] and Raynal et al. [139] explicitly set up a high-interaction honeypot and described their observations. Ramsbrock et al. [135] went a step further and modeled attacker behavior on a high-interaction honeypot as a state machine derived from observations made during the earlier operation of high-interaction honeypots. Their model describes typical attacker actions following the compromise of a system, and tells a honeypot operator what he should expect while operating a high-interaction honeypot. However, this model assumes that all commands are allowed for an attacker, and the honeypot does nothing to resist an attack. They follow the paradigm of exposing an infrastructure to an enemy without any protection or resistance. The enemy assaults the infrastructure and her actions are observed. Following such an approach only a limited set of strategies and tactics of an attacker is likely to be observed because attackers face no resistance during the attack, and may react differently in such a case.

Most of the research related to high-interaction honeypots has focused on the arms race between attackers and honeypot operators. The publication of a new attacker monitoring technique is followed by another one describing how to detect or evade this novel approach. To a certain extent such research activities are necessary in order to ensure that attackers continue to be lured and to avoid any suspicion of counterfeit infrastructures. When an attacker detects an obviously fake infrastructure, she may not attack it, or if she has attacked it may backtrack quickly. Cohen [31] discusses deception techniques in the context of confronting attackers. His idea is mix real services with fake services such that an attacker has to discover the services that are useful to her. This approach has been formally described using game theory [98], [108] such that an attacker has to decide how to attack a real or a fake system with each decision being associated with a price. However, little effort have been made to make honeypots themselves more intelligent and adaptive with in order to automated and augment information retrieval from attackers. Current high-interaction honeypots have static behavior and can be only operated with a predefined configuration. Such an approach allows malicious programs to be collected. The skills of attackers or their tactics and strategies when facing resistance are not assessed. Attackers are not challenged on the path to reaching their goal because all operations are permitted, with the result that the honeypot operator gains little idea of their skills. If they have encountered resistance or additional constraints, they might reveal more information about themselves. Some attackers may give up, and leave the honeypot. Other attackers may look for alternative paths to their goal. The choice of an attacker is particularly interesting for a honeypot operator because it would allow him to classify attackers according their skills. Furthermore, after system identification, some attackers expect a particularly behavior from the compromised system. In order to optimize information retrieval from attackers, a honeypot's strategies should not be predictable by an attacker. If an attacker is able to correctly predict the behavior of a honeypot, this must mean that it is behaving statically like a classical high-interaction honeypot. Adaptive honeypots use deception techniques in order apply increasing resistance against attackers. Adaptive honeypots may reveal more information about attackers, such as their strategies regarding their final goal or their ethnological background. From a high-level perspective, attackers and honeypot operators are opponents in a competitive environment. Attackers want to reach their goal without being discovered. Honeypot operators want to reveal information about attackers. Therefore, we explore game and learning theories in the context of high-interaction honeypots in order to model interactions with attackers and to optimize information retrieval from attackers.

Chapter 4

Learning in Games

4.1 Game Theory

In 1928, von Neumann [171] introduced Game Theory in his article “Zur Theory der Gesellschaftsspiele”. He formalized the case of n players who play a game. Each player’s result is influenced by his own and his opponents’ actions. Each player wants to achieve a good result at the end of the game. In 1944, von Neumann with the economist Morgenstern formalized this concept in economics [172]. The breakthrough in game theory came in 1950, when Nash published his thesis ”Non-Cooperative Games” [109]. He introduced and proved the Nash Equilibrium (NE) and received the Nobel prize for this achievement. Informally, every time people have to deal with one another, a game is played [21]. Each player is able to perform a variety of actions, and each action results in a payoff. A Nash equilibrium is a set of strategies among several players in a game if and only if each strategy is the best reply to the other [21]. The prisoner’s dilemma is a famous problem analyzed in game theory [169]. In this thought experiment, the police arrest two people. Both are suspected of having jointly committed a crime. The police lack sufficient evidence to convict them. The suspects are put in separate cells such they cannot communicate with each other. Each suspect is offered the following deal: If a suspect defects against the other, and if the other does not defect, he will be released (0 years of prison), while the other will go to prison for 12 years. If neither suspect testifies, both have to go to prison for 1 year due to the lack of evidence. If both testify, both will go to prison for 10 years and this situation is called Nash Equilibrium. In this situation the players act selfishly and want to be better off independent of the other player’s decision. Game theory is particularly useful for modeling the interactions among several players having different interests. It allows reasoning about rational players’ strategic interactions. The theory is popular in economics and politics [115].

A game may be formalized as either a strategic form or an extensive form game [55]. The strategic form is less complete than the extensive form, but it is better suited for identifying dominant strategies and to computing Nash equilibria. A strategic form game [55] is composed of a finite set of players $N = \{0, 1, 2, \ldots, n\}$. Player 0 represents ”Nature” and is exogenous to the game [169]. Each player i has either a finite set of pure strategies S_i or a mixed strategy set. A pure strategy set is a discrete set of actions or strategies a player can perform, and a mixed strategy set for a player i is defined in eq. 4.1 and represents the probability distribution, denoted $\Delta(S_i)$ over a finite strategy set S_i [64]. For a mixed strategy set, a strategy profile is $\prod_i Q_i$ and is abbreviated (q_i, q_{-i}), where $-i$ denotes the other players. A mixed strategy profile is more generic than a pure strategy profile. For a pure strategy profile, the probability of selecting a given strategy is always 1. Therefore, the mixed strategy profile is considered in the following.

$$Q_i = \Delta(S_i) = \left\{ q_i : S_i \to [0, 1] \mid \sum_{s_i \in S_i} q_i(s_i) = 1 \right\} \tag{4.1}$$

Each player i also has a payoff or utility function $r_i : S \to \mathbb{R}$, where $S = S_1 \times \cdots \times S_n$. The expected

1\2	D	C
D	-10,-10	0,-12
C	-12,0	-1,-1

Table 4.1: Prisoner's Dilemma - Payoffs

payoff for a player i, who is playing strategy s according the probability q is defined in eq. 4.2, where $q(s) = \prod_{j=1}^{N} q_j(s_j)$.

$$\mathbb{E}_{s\sim q}[r_i(s)] = \sum_{s_\epsilon S} q(s) r_i(s) \tag{4.2}$$

The prisoner's dilemma can be formalized as a strategic game [169]. The set of players $N = \{1, 2\}$ is composed of the two players. Each player has two strategies: to defect D or to confess C. Hence, both players have the same set of strategies $S_1 = S_2 = \{D, C\}$. In some cases the suspects go to prison. The police offers a special deal, which is presented in the payoff matrix 4.1 [169]. If both players defect they both go to prison for 10 years. Going to prison is a negative payoff (-10) for the players. If one player defects and the other does not, the second goes to prison (payoff -12) and the first is freed. If neither player testies, both go to prison for 1 year (payoff = -1).

In strategic form games, the choices of the players are made simultaneously in one round. In an extensive form game [169] each player can make more moves in a game. Hence, interaction among players can be modeled. Strategic form games are represented with a decision tree K. Each node in this tree is a decision node and represents a player.

A game represented by the extensive form includes the following components [169]:

Set of players. This set contains n players and is denoted $N = \{0, 1, 2, \ldots, n\}$. External events result in actions by player 0 and are usually caused by nature. The extensive form game shown in figure 4.1 has two players $N = \{1, 2\}$.

Available actions. In particular situations a player $i \in N$ can perform k actions or strategies s_k^i. An example decision tree presented in figure 4.1 and is read from left to right. The first node is first player's decision. Two actions are available: s_1^1 and s_2^1. If player 1 chooses the strategy s_1^1, player 2 has to make a decision, choosing between actions s_1^2 or s_2^2.

Order of moves. The extensive game form permits successive strategies or actions to be modeled. One player can trigger an event before another one does. Each node in the decision tree represents a player and each edge corresponds to an action by that player. The order of the moves is represented by a branch in the decision tree. If player 1 makes the decision s_1^1 and player 2 then decides to perform the action s_2^2, the order of moves is s_1^1, s_2^2.

Information sets. An information set is a partition of the history of moves for a given player. It is used to model incomplete information for a player.

Payoffs. Each action results in a payoff. The payoff is usually distributed at terminal nodes and is represented by a real number. In the case where a player's gain is exactly equal to the other player's loss, the term, *zero-sum game* is used; otherwise the term *general sum game* is used [169].

Formalization permits the modeling of a game among n players and subsequently analysis. Looking at the prisoner's dilemma presented in table 4.1, the strategy to defect D is a dominant strategy [169]. In any situation, it gives a better payoff than the alternative. For instance, if one player defects and the other does not it gives 0 payoff to the player that defects. In the prisoner's dilemma, the players act selfishly: they want to get the

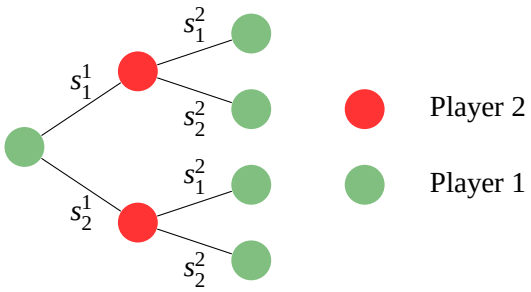


Figure 4.1: Extensive Form Game

best payoff for themselves. If both defect both get a negative payoff. If the players had had the opportunity to agree to not defect they would both have been better off. According to Fudenberg et al. [55] one player's selection strategy can hurt other players. For instance, in a zero-sum game one player's profit is the other's loss [55]. The strategy of defecting is also a Nash Equilibrium. It is valid for players to try to maximize their own payoff while ignoring the other players' goals. Therefore, the Nash equilibrium is not *Pareto optimal*. A formal definition is presented in [52]. *Pareto efficiency* comes from the *Pareto improvement*. If a parameter in a model can be improved without deteriorating another one, a *Pareto improvement* has occurred.

A Nash equilibrium shows the strategies that maximize a player's payoff. If such a strategy exists, the notion of *pure Nash equilibrium* is used. Sometimes a pure Nash equilibrium does not exist. However, Nash [109] proved in his theorem 1 that *every finite game has an equilibrium point*. A Nash equilibrium is defined as a strategy profile such that all players of the game select strategies that are the best responses to the strategies of the other players [64]. The best strategy q_i^* for a player facing other players q_{-i} is defined in eq. 4.3[1].

$$\forall q_i \in Q_i, \mathbb{E}[r_i(q_i^*, q_{-i})] \geq \mathbb{E}[r_i(q_i, q_{-i})] \tag{4.3}$$

There are pure or mixed equilibria. A mixed equilibrium is a probability distribution over strategies. A pure equilibrium is a special case of a mixed equilibrium, where the probability of selecting a particular strategy is 1. A strategic game permits straightforwardly computation of the Nash Equilibria because there is only a single interaction among the players. A decision tree of such a game has only a depth of one. Games formalized as an extensive form game permit to model more interactions among players. Repeated games [55] are games having more than one interaction. In order to compute a Nash equilibrium, all the selected strategies with their payoffs have to be known. A repeated game is divided into sub-games, where Nash equilibria can be computed in each sub-game [55]. The computation of Nash equilibria addresses the question of which action pair for each player maximize the payoff for each player. Because, the exact algorithmic complexity of computing Nash equilibria is unknown [123], this computation is a trade-off between execution speed and completeness. Although some algorithms do not identify all the Nash equilibria meaning that these algorithms are not complete, they are fast [123]. Other algorithms, are more exhaustive and require unrealistic amount of processing power. A good starting point is to clearly identify the type of game. On one hand, generic algorithms for strategic games have been proposed and are ready to use. On the other, when dealing with an extensive game, its particular sub-games needs to be identified because every extensive game is different. For each sub-game, Nash equilibria can be computed. If the game is a strategic game, then ready-to-use algorithms can be applied. The next step is to establish whether the game is zero-sum or general sum. A large family of algorithms have been proposed to search for Nash equilibria. These can be organized into two categories: Those in the first category can solve strategic games and those in the second can solve games in the extensive form. The category of strategic games can be further diveded into two classes: algorithms addressing the first class are solve zero-sum games, while those in the second solve general-sum games.

[1] $\mathbb{E}$ is the expected value.

Due to the fact that the exact algorithmic complexity of finding Nash equilibria is unknown, a trade-off has to be made between completeness, meaning to find all possible Nash equilibria, and execution time. The oldest but still frequently-used, algorithm was set out by Lemke [92]. An algebraic proof was proposed for the existence of equilibrium points for a two-person, non zero-sum games. The authors also propose an approach for searching at least one equilibrium point, which is identified via linear complementarity. The algorithm uses combinatorial search to find a Nash equilibrium, resulting in a exponential runtime. Shapley [149] showed that this algorithm does not necessarily find all equilibrium points. Mangasarian [100] showed the enumeration of all Nash equilibria is a polytope problem extending Lemke's approach to find all the Nash equilibria. Porter et al. [123] described an algorithm that is capable of solving both zero-sum and general-sum games. The authors try to increase the execution speed by minimizing the time taken to find the first Nash Equilibrium point. In practice players sometimes do not know their exact payoff. These errors may impact the equilibria and when this impact is analyzed, is called *Quantal response equilibrium* [61]. Errors are incrementally introduced in the payoffs and the Nash equilibria are recomputed in order to see how stable the equilibria are.

Game theory permits to formalize the interaction between players having different interests. Players can perform various actions, each of which is related to a cost or a reward. Recently, this theory has been applied in the context of information security. Often, two parties are modeled: an defender and an omnipresent attacker. For instance, Schmidt et al. [144] formalized a game between an attacker and a defender in the context of intrusion detection systems. A defender can deploy intrusion detection systems in a communication network, but each deployment involves a cost. The authors describe four games differing in network characteristics, and the number of accuracy of intrusion detection systems. They concluded that the accuracy of detecting attackers is significantly reduced when attackers can find a way to exploit defects in intrusion detection systems even when the defender uses an optimal placement strategy.

O'Donnell [115] formally explains the reason that most malicious programs are targeted for Microsoft Windows platforms. He states that Macintosh users believe their platform is more secure than those of by Microsoft, but argues that the reason for their being fewer malicious programs on Macintosh than on Microsoft platforms is that of economic motivation. He created a formal game between attackers and users. Attackers can attack a Windows system or a Macintosh system. In his reward model he included the market share of the system, the value of the system and the probability of successfully defending a system. He discovered that, for some systems the market share factor makes a strategy dominant. This means that an attacker will always get a higher payoff when attacking a more widespread system. Even the worst-case reward for attacking such a system is better than attacking a less popular system.

As discussed in chapter 3, Cohen described the idea of mixing real systems with fakes ones. In shadow honeynets, production machines and honeypots are intermixed [4]. With such an approach attacks can be partially observed. When attackers discover the infrastructure, they leave traces on honeypots as well as on production systems.

Garg et et al. [108] formalize an equivalent game involving two players, an attacker and a defender. The authors create a formal framework for modeling deception in honeynets. Attackers prefer to assault regular hosts rather than honeypots, while defenders want to lead intruders to attack honeypots. Hence, an attacker can probe a regular host or a honeypot. If she hits a regular host, a positive reward is given to the attacker and a negative reward to the defender. If the attacker probes a honeypot, the rewards reversed. Hence, each player has different interests. The authors' game is in the extensive form. This means that each player has information sets modeling imperfect information. The game between attackers and defenders is played sequentially, taking into account the opponent's moves. The game has four stages. The defender has to decide whether host at a particular address is to be a regular host or a honeypot. When an attacker probes a host he does not necessarily know a priori it is a honeypot or a legitimate machine. The defender also uses deception techniques. A host having the role of a honeypot could respond to an attacker as it were a legitimate host, and a legitimate host could respond in a way that suggests it is a honeypot. The defender could also utilize disclosure strategies meaning that neither a regular host nor a honeypot would lie about its role. Garg et et al. [108] focus on a game theoretical framework for honeynets using deception techniques. They give an example of manually defined

payoffs and information sets for each player, then show how to compute Nash equilibria by partitioning the extensive-form game into sub-games having the strategic game form. However, the authors do not explain how deception techniques are used in this context.

Carrol et al. [46] formalize a similar game between defenders and attackers. The authors model a signaling game in a network composed of honeypots and regular machines. The defender has to decide which machines should be honeypots and which should be regular machines. In addition, a defender can use camouflage techniques to make a honeypot look like a regular host and a regular host like a honeypot. An attacker can assault either a honeypot or a regular machine. Attacking a honeypot yields a loss for the attacker. A honeypot is monitored, and when an attacker assaults such a machine, she reveals her attacking techniques to the defender. The authors derived four strategies for the defender according to her capabilities, and nine strategies for the attacker. The defender strategies are based on the decision as to whether a host should reveal its true nature or not. Based on received signals from the defender, the attacker can attack without determining the system type, or after first determining the system type. Alternatively, she could retain from attacking the system. She could also disbelieve the system of its true nature. The authors define a payoff structure for each action, giving the gains and losses for each player. The authors then established equations that must hold for computing the expected payoffs in the context of *Bayesian equilibria*. A formal definition of a Bayesian equilibrium is shown in [56]. Using these equations, the authors discover ten equilibria. The authors made two case studies in order to determine which equilibria is the most appropriate. In the first, they discuss a network where 10% of the systems are honeypots. In this scenario the authors concluded that the defender should disguise all hosts as honeypots. In the second study, scenario the authors considered automated attacks such as botnets, and determined that the defender should always either reveal the true nature of a system, or always claim the opposite one.

Lye et al. [98] formalized a game between administrators and attackers. Their input was data by a network manager employed at their university. The authors identified three attack scenarios: firstly, an attacker could deface the main web site on their public web server. Secondly, an attacker could launch internally a denial of service attack. Finally, an attacker could steal confidential data. The authors are not interested in an individual attacker, and therefore assume an omnipresent attacker. The authors consider a general-sum game between attacker and administrator, because an attacker's gains does not necessarily have the same magnitude as the administrator's costs. Their infrastructure is modeled as graph representing a network. A node in this graph is a device like a workstation or a router. The external world is modeled as a single node. Each edge in the graph corresponds to a physical or virtual communication channel. On this graph, the authors introduce super states which represent the state of the network and are composed of the individual states of each physical node together with a traffic state. Each node state corresponds to a dedicated software configuration. The authors define eleven attacker actions derived from their survey, an administrator can perform eight actions. The authors give a high-level description of these actions. For instance, an attacker can install a virus or a sniffer program and an administrator can remove compromised accounts or remove sniffers. Both players can also do nothing. For the super states, the authors assign probabilities for transitions among their super states. Formally, they model the game as a stochastic game[2] game with a set of super states, a set of actions for each player, a set of transition probabilities, and a set of rewards for each player. They also introduce a discount factor and model time using discrete steps. The authors examine several attack scenarios and show how Nash equilibria can be computed in this stochastic game. They used MATLAB to compute the equilibria and report that the execution time was between 30 and 45 minutes. Hence, this implementation cannot be applied on real-time systems.

Grossklags et al. [80] formally analyzed the situations involving attackers and defenders on a shared infrastructure. They developed five security games: They investigated the case of total protection, best shot game, weakest link game, the weakest target with mitigation techniques, and the weakest target without mitigation techniques. Each player in each game has its own parameterized utility function. The authors used parameterized utility functions. The most relevant parameters are the protection level and the self-insurance level. Due to their generic approach with parameterized utility functions, the authors were able to compute Nash equilibria

[2] A definition of a stochastic is presented at page 58.

Step 1	Environment	You are in state 23. You have 3 possible actions.
Step 2	Agent	I go for action 3.
Step 3	Environment	You received a reward of 4. You are in state 12. You have 2 possible actions.
Step 4	Agent	I go for action 1.
Step 5	Environment	You received a reward of -5. You are in state 34. You have 5 possible actions.
Step 6	Agent	I go for action 2.

Table 4.2: Reinforcement Learning - Operation Example

and then discuss the parameters, searching for best and worst cases in each type of game. The upper and lower bounds also permit the authors to investigate the impacts of having more players in each type of game.

Kantzavelou et al. [83] focused on a special type of attacker, namely the insider. An insider may have different objectives than those of the organization that she is working for. Insiders are divided into two categories: traitors and masqueraders. Traitors have been granted extra privileges and exploit them to work against their own organization. Masqueraders pretend to be someone else in order to gain privileges. The activities of insider are different from those of an external attacker. The authors define four actions for an insider. Firstly, an insider can work normally by performing permitted tasks. Secondly, an insider can do unintentional damage as result of making mistakes. Thirdly, an insider can embark on a pre-attack phase. Finally, an insider make her attack. The authors define the payoffs via preferences. A further preference defines which strategies are preferred by a player. For instance, an insider prefers a commendation from her manager to a warning. The authors used a repeated game model. The authors manually define payoffs for each scenario, and solved the game by computing Nash equilibria. The authors also carried out a quantal response equilibrium analysis [61]. In such an analysis, the rationality of players and erroneous payoffs are inspected.

In contrast to the situations presented so far, there are cases where the payoffs are not known in advance at all. Hence, a contribution of this thesis explores reinforcement learning where a player can learn her payoff during her interactions. A primer of reinforcement learning is presented in the next section followed with a combination of reinforcement learning in stochastic games where learning approaches are combined with game theoretical paradigms.

4.2 Reinforcement Learning

Kaelbling et al. [82] give a broad overview of reinforcement learning. A typical reinforcement learning problem is shown in figure 4.2 and a reinforcement learning scenario [82] is shown in table 4.2. An agent interacts with its environment and has a set of sensors to perceive its environment through which it receives a reward signal r_t [161]. Each action performed by the agent is rewarded or punished. The agent tries throughout its life to optimize this reward signal. The environment is defined as discrete set of states S, and the agent has a discrete set of actions A and a set of rewards R. The agent interacts with its environment only at series of discrete time steps[3].

At each instant t, the agent identifies its state s_t in its environment. The agent then decides to perform an action a_t, and as consequently makes a state change to the state s_{t+1}. The agent's action is rewarded or punished with a reward r_t. In order to maximize its long-term rewards, the agent must learn a policy π , mapping states to actions.

4.2.1 Markov Decision Process

This section presents a short primer on Markov decision processes (MDP), which provides a theoretical basis for reinforcement learning [161]. An agent acting in an environment takes decisions, which result in state

[3]Kenji [86] discusses reinforcement learning in continuous time and action space, but this topic is not relevant for the further understanding of our contributions.

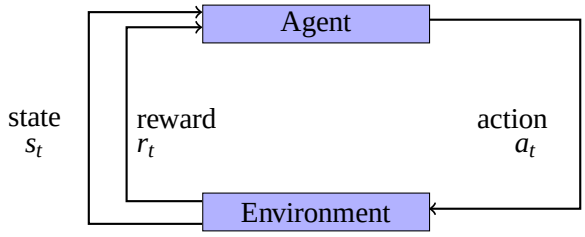


Figure 4.2: Reinforcement Learning Problem

changes. In the most generic way the probability of reaching state $s' \in S$ and receiving reward $r \in R$ depends on the agent's previous actions and its previous state changes, defined in eq. 4.4 [161]. However, if the next state s' depends only on the previous state s_t and the agent's action $a_t \in A$, the environment conforms to the Markov property, which is defined in eq. 4.5 [161].

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t, r_t, s_{t-1}, a_{t-1}, r_{t-1}, \ldots, r_1, s_0, a_0\} \tag{4.4}$$

$$Pr\{s_{t+1} = s', r_{t+1} = r \mid s_t, a_t\} \tag{4.5}$$

Each environment conforming to the Markov property can be formally modeled as a Markov decision process [161]. Here we consider the case of finite Markov processes, because we are discussing the formal roots of reinforcement learning. A finite Markov decision process, simply denoted as a Markov decision process[4], is composed of

- A finite set of states S.
- A finite set of an agent's actions A.
- A transition function $T : S \times A \to PD(S)$, where $PD(S)$ is the probability distribution over the set S.
- The reward function $R : S \times A \to R$ defining the distributed rewards.

According to Stutton et al. [161], a Markov decision process defines the one-step dynamics of the environment as shown in eq. 4.6.

$$\mathcal{P}^a_{ss'} = Pr\{s_{t+1} = s' \mid s_t = s, a_t = a\} \tag{4.6}$$

The expected rewards are defined in eq. 4.7 and depend on the current state, the previous state and the action performed [161]. The expected value is denoted E.

$$\mathcal{R}^a_{ss'} = E\{r_{t+1} \mid s_t = s, a_t = a, s_{t+1} = s'\} \tag{4.7}$$

An agent must find a policy π that maximizes its received reward in the long run. Two kinds of policies have been defined:

Stationary policy An agent learns a mapping from the state space to the action space $\pi : S \to A$.

Non-stationary policy An agent learns a probability to be in state s and taking action a_t, $\pi(s_t, a_t)$.

[4]Infinite Markov decision processes are described in [148] and are not mandatory to conceive our contributions.

A fundamental problem of an agent in an environment is that it has to learn the value of being in a state s, and how profitable it is to perform an action a in a given state s. The quality of the states and the state-action pairs are defined with respect to the expected future rewards [161]. An agent is follows a policy π defining the actions it should take. During this evaluation, an agent estimates the value of a state, denoted V^π, in order to see if it is worth visiting in this state. In addition, it evaluates a the value of a state-action pair under a policy π, denoted $Q^\pi(s, a)$. An agent can maintain V^π and Q^π as parameterized functions. If these functions are recursive, they are called Bellman equations [161]. The Bellman equation defining the value of a state s under the policy π is shown in eq. 4.8. A detailed derivation can be found at [161] page 91. The optimal value of V^* can then be calculated according eq. 4.9 [161]. Similarly, an agent can optimize a state-action pair Q^* by applying eq. 4.10.

$$\begin{aligned} V^\pi(s) &= E_\pi \left\{ \sum_{k=0}^{\infty} \gamma^k r_{t+k+1} \mid s_t = s \right\} \\ &= \sum_a \pi(s,a) \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^\pi(s') \right] \end{aligned} \tag{4.8}$$

$$\begin{aligned} V^*(s) &= \max_a E\left\{ r_{t+1} + \gamma V^*(s_{t+1}) \mid s_t = s, a_t = a \right\} \\ &= \max_a \sum_{s'} \mathcal{P}^a_{ss'} \left[\mathcal{R}^a_{ss'} + \gamma V^*(s') \right] \end{aligned} \tag{4.9}$$

$$Q^*(s,a) = E\left\{ r_{t+1} + \gamma \max_{a'} Q^*(s_{t+1}, a') \mid s_t = s, a_t = a \right\} \tag{4.10}$$

The value $V^*(s)$ is the unique solution of the Bellman equation, and defines the optimal value of a state. The variable γ is a discounting parameter used for handling indefinitely interactions. If an agent's interaction with the environment can be broken down into episodes, meaning that the start time and the end time are known, γ is set to 1 such that no discounting is done. However, if either the start time or end time is unknown, future rewards are usually decayed by setting $\gamma < 1$ so as to reduce future rewards. The act of breaking down an agent's interaction into episodes are is sometimes defined as the agent's horizon [98].

finite horizon A finite horizon is used when the lifetime of an agent is known in advance. This means that an agent has to optimize its behavior within a finite number of steps. In this case, the discounting factor γ can be set to 1.

infinite horizon If an agent's lifetime is unknown in advance, the term infinite horizon is used. In this case, a discounted reward accumulation or an average reward model can be used.

- **discounted rewards** Rewards are geometrically discounted according a discount factor [82]. This discount factor defines if the agent optimizes its reward signal over the long or the short run. The discount factor is usually a number between 0 and 1 and is multiplied by the received rewards. On one hand, a low discount factor reduces the received reward and targets long-term operation. On the other, if the highest discount factor (1) is used, the agent focuses a short-term operation.
- **averaged rewards** The average rewards in a delimited operation window are used for reward accumulation. The operation window does not necessarily correspond to an episode but the concept is quite similar. It consists of a successive lists of rewards established in a continuous process.

As alternative to the algebraic computation, the optimal policies they can be determined by dynamic programming approaches [161]. Agents' decisions are often represented with backup diagrams like that shown in figure 4.3. Looking at the left tree in figure 4.3, an agent is optimizing the value of a state V^* according to equation 4.9. The agent is in a state s and performs an action a. The agent gets a reward r and a transition to the state s' is made. Each such interaction corresponds to a branch of the backup tree, and the number of potential transitions resulting from all potential actions increases is quickly. Hence, the term *full-backup* is used, because all possibilities are taken into account. The right tree in figure 4.3 shows the case where an agents optimizes a

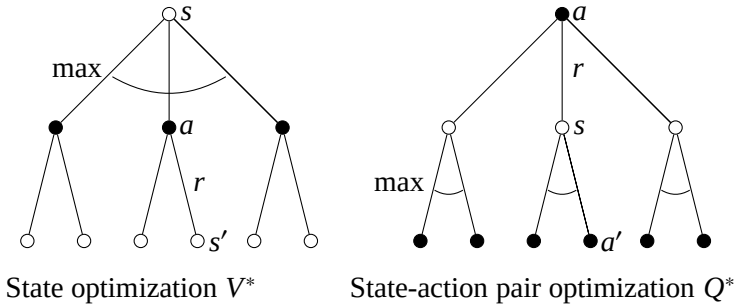


Figure 4.3: Full Backup Representation

state-action pair Q^* defined in eq. 4.10. The term of full-backups is also used for eq. 4.10. Each policy has an associated value function that either optimizes the value of a state $V(s)$ or the value of a state-action pair $Q(s,a)$. When the optimal value-functions are known, the optimal policy π^* leading to this optimal value-function can be discovered. This is due to the fact that the optimal value functions for a state s, or for a state-action pair, can be expressed as $V^*(s) = max_\pi V^\pi(s)$ and $Q^*(s,a) = max_\pi Q^\pi$ respectively. In order to find an optimal policy, policy iterations have been proposed with an algorithmic complexity is $O(|A||S|^2+|S|^3)$ per iteration [82], taking into account that there are at most $|A|^{|S|}$ distinct policies [82]. Furthermore, the discounting factor γ can increase the algorithmic complexity [82]. Despite the fact that optimal policies can be mathematically formulated and algebraically solved, it is expensive in terms of memory usage and computational power to determine them [161].

4.2.2 Learning Agents

Although Markov decision processes allow the formal modeling of agents operating in an environment and the algebraic computation of optimal policies, some parameters of Markov decision processes are unknown in practice. Frequently unknown parameters are the reward function or the transition probabilities. An agent must interact with the environment to discover the optimal values for a state or a state-action pair through learning algorithms. When not all parameters are known, the term *model-free learning* is used [82]. However, when following a model-free approach, the fundamental exploration-exploitation trade-off emerges. An agent has to discover its environment. Having explored the environment, the agent could exploit its knowledge or continue its exploration. There might be some hidden states yielding higher rewards. An agent may take an action yielding a high reward but which may result in low rewards in the future. Hence, the far-reaching effects of actions are unknown. An agent usually has two components: an explorer, a learning rule.

Explorers

When discounted rewards are used, an allocation index can be used. An agent remembers the number of times an action has been chosen and the number of times that positive rewards were received for this action. An agent has thus a history of rewards for a given action in a given state. This technique is known as *Gitten's allocation indices*. In addition, Bayesian reasoning approaches have been proposed to tackle the exploration-exploitation problem [82]. Alternative explorer families include randomized strategies. In this case an action is chosen according a probability $1-\epsilon$, and a random action is selected according the probability ϵ. The ϵ-greedy explorer follows this scheme [161]. The Bolzmann explorer [82] has a temperature parameter. If this parameter is high, an agent is more willing to explore and if this parameter is low, the agent exploits its acquired knowledge. The temperature parameter is decayed over time, with the result that an agent stops exploring in the long run. In addition Kaelbling et al. [82] record the number of successes and the number of trials in order to improve the exploration-exploitation trade-off. When the explorer is tightly coupled with the exploitation the

term *on-policy* is used; otherwise the term *off-policy* is used.

on-policy An agent learns a policy π while following it. The value of a policy is estimated while it is being used for control [161].

off-policy An agent learns a policy while following another.

Learning rules

Monte Carlo methods provide one way of tackling a partially unknown environment in order to compute the optimal values. These methods require sequences of states, actions and rewards [161]. These sequences are recovered either from on-line methods or from simulations. Monte Carlo methods exist for episodic tasks, as demonstrated in eq 4.11, where R_t is the actual return at time t [161]. The optimal policies are discovered via general policy iteration methods [161]. An advantage of Monte Carlo methods is that the estimate of a state s does not depend on estimates of the other states [161]. This means that Monte Carlo methods are even suited to non-Markovian tasks [161]. A major drawback of Monte Carlo techniques is that some states may never be visited.

$$V(s_t) \leftarrow V(s_t) + \alpha \left[R_t - V(s_t)\right] \tag{4.11}$$

In many application fields [161], an agent's interactions with an environment cannot be divided into episodes, because the start or end times are not known. Hence, temporal difference (TD) learning methods which are also model-free methods have been proposed. TD learning leverages iterations over the estimated value of a state using immediate rewards as input. Estimated values are updated at each time step instead at the end of an episode. A basic TD learning method is shown in eq. 4.12, where α is a constant step size parameter [161]. As shown in eq. 4.12, the estimated value of a state depends on estimates of other states. Consequently, estimation errors may be propagated.

$$V(s_t) \leftarrow V(s_t) + \alpha \left[r_{t+1} + \gamma V(s_{t+1}) - V(s_t)\right] \tag{4.12}$$

In order to mitigate these error propagations, a new component, called an *adaptive heuristic critic* (AHC) is introduced [161]. An example, taken from [161], is shown in figure 4.4. The AHC is a separate memory structure for an agent, which represents its policy structure. This component criticizes the actions of an agent. A formal definition of an AHC is given in eq. 4.13. The TD error is computed from the new state, the previous state and the received reward. This error can then be included in an agent's policy selection method. An example is the Gibbs softmax method is presented in [161] page 189. AHC methods are only applicable for on-policy algorithms [161]. Another approach to reducing these propagated errors is to use $TD(\lambda)$, which is a combination of Monte Carlo methods and TD methods. The λ parameter defines eligible traces and consists of a temporary record of the frequencies of an event, such as a visit to a state or the performing of an action [161]. The case where λ is set to 0 results into $TD(0)$. It has been shown that $TD(\lambda)$ methods converge faster to optimal values than the simple $TD(0)$ method [161].

$$\delta_t = R_{t+1} + \gamma V(s_{t+1}) - V(s_t) \tag{4.13}$$

Q-learning [188] is easier to implement than AHC because only one component has to be considered [82]. The basic Q-learning form is presented in eq. 4.14.

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_t + \gamma \max_a Q(s_{t+1}, a) - Q(s_t, a_t)\right] \tag{4.14}$$

According to Kaelbling et al. [82], a major advantage of Q-learning is that it is exploration-insensitive. The Q values converge to the optimal values independent of the agent's explorer while the data is being collected. The agent has still to explore, but the details of how the exploration is done do not impact convergence [161].

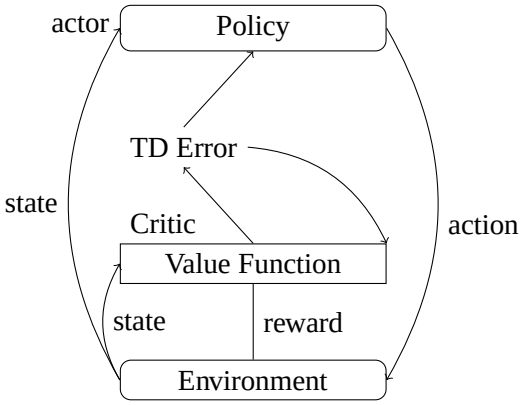


Figure 4.4: Adaptive Heuristic Critic

Q-learning is designed for discounted infinite-horizon Markov decision processes [82]. Undiscounted rewards can also be handled with such an approach if a reward-free state is reached and if this state is regularly reset [82]. Q-learning is a model-free approach [188], and allows an agent's environment to be partially unknown. A disadvantage of Q-learning is that each state has to be visited an infinity of times in order for the Q-values to converge to the optimal Q-values. The Q(λ)-learning approach [119] combines the approach of using a adaptive heuristic critic and reinforcement learning.

Q-learning is an off-policy algorithm and SARSA, an alternative model-free approach, is an on-policy algorithm. It takes the next state into account, the next action and the next reward [161], explaining the name of the algorithm: *S*tate - *A*ction - *R*eward - *S*tate-*A*ction. The general form of SARSA learning is presented in eq. 4.15. SARSA has also be extended with eligible traces SARSA(λ) [161].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha \left[r_{t+1} + \gamma Q(s_{t+1}, a_t) - Q(s_t, a_t) \right] \tag{4.15}$$

Sutton et al. [161] show numerous application examples, mainly in the area of robotics. A classic example is a robot that needs to find the exit of a maze. Different reward models exist to parameterize the behavior of the robot. Examples include punishment induced by energy constraints; others do not punish, and simply give a positive reward when the robot finds the exit. In other examples, the agent can even be punished for bumping into walls. Reinforcement learning often considers an agent operating in an environment and receiving positive or negative rewards in response to actions taken. Among the more spectacular autonomous agents are helicopters which are able to perform aerobatic flight manoeuvres controlled by reinforcement learning [1]. Reinforcement learning has been explored and extended for collaborating agents [163]. Hierarchical learning among agents has been studied by Barto et al. [13]. Gambardella et al. [58] tackle the well-known traveling salesman problem using an experimental reinforcement-based approach. To the best of our knowledge, there is no literature concerning the use of reinforcement learning in the context of high-interaction honeypots.

4.3 Multi-Agent Learning Founded on Game Theory

Markov decision processes (MDP) are frequently used to model an agent in a dynamic environment. An agent must learn a policy that maps states to actions by optimizing its reward signal. Often some parameters of the MDP, especially the probabilistic transition function, are unknown and a learning approach must be used. In some cases, the successive states should also be taken into account by an agent instead of just maximizing a reward signal. However, the previously discussed reinforcement learning techniques only consider an agent in an environment make the assumption that the environment is stationary. When there are more than two agents, a simplistic approach is to consider opponents as the environment [90]. However, an environment containing additional agents is constantly changing [74]. The environment may not be stationary anymore

being is influenced by the other agents rather than generated by a stochastic process [74]. In this section we consider reinforcement learning where more than one agent is present, and the agents are opponents. The learning approaches should be feasible in close to real time. Stochastic games are introduced that handle the uncertainty of agents dealing with other agents better than traditional Markov decision processes.

Stochastic Games

Shapley [150] introduced stochastic games. A stochastic game is composed of a finite number of players who may occupy a finite number of positions. A position corresponds to a state in a traditional Markov decision process. At each state a player chooses an action from a finite action set. Each action results in a reward. The transitions from state to state are described by transition probabilities, which are jointly controlled by the players. The interesting aspect is that these two players could have different objectives. Zachrisson [198] extends such games to Markov games. The theoretical foundations providing a bridge between game theory and stochastic games are provided by Fink [50] who investigated the existence of equilibrium points. Littman [90] described a reinforcement learning approach in order to solve a two player zero-sum game modeled as Markov game. He defines a Markov game with a set of states, denoted S, and a collection of action sets $A_1 \dots A_k$ where k denotes the number of players. The state transitions are described by the transition function T of the current state and the actions of each player. $T : S \times A_1 \cdots \times A_k \rightarrow PD(S)$, where $PD(S)$ is a probability distribution over the state space. In addition a reward function R is associated with each player. $R_{i \in \{1,\dots,k\}} : S \times A_1 \times \cdots \times A_k \rightarrow \mathbb{R}$. In this game the author uses a single reward function, where one agent tries to maximize its reward signal and another player tries to minimize it.

The major difference between a Markov decision process and a Markov game is the definition of the policy that an agent has to discover. In a traditional Markov decision process, a policy is a mapping from the state space to the action space [90]. However, in Markov games, a policy is a mapping from the state space to a probability distribution of an action space [90]. The author gives the example of the rock-paper-scissors game, where a deterministic policy by an agent can be easily defeated. Such games are better modeled by a Markov game. The advantage of a Markov game is that the uncertainty of the opponent's moves can be included in the probabilistic choice of a player's actions. In a traditional Markov decision process, the optimal deterministic policy $\pi : S \rightarrow A$ can be used to determine the quality of a state action-pair, denoted $Q(s, a)$. In addition, the reward of each player depends on the actions of each player, the current state and the state transitions controlled by the Markov property [74]. Littman [90] modified this statement by introducing an opponent's action, denoted o, and estimates the quality of the state-action-action triple, denoted $Q(s, a, o)$. Instead of formally computing the optimal policy, Littman [90] uses the previously described Q-learning algorithm. A formal convergence proof was later published in [96].

Hu et al. [74] also model a stochastic game with more than one agent. The authors propose a combination of Q-learning and Nash equilibria and create a bridge between reinforcement learning and game theory. They adapt Q-learning in general-sum games. The authors work on the stationary property of the environment. An environment containing more agents is constantly changing, and the formal guarantees of Markov decision processes do not always hold. In addition each agent might have different interests, as is usually the case for noncooperative zero-sum and general-sum games. General-sum games have the advantage that arbitrary reward models can be established [74]. Therefore, the authors propose Nash-Q, which is a variant of the Q-learning algorithm [188] that takes Nash equilibria into account. Their algorithm is applicable in a stochastic game, and its goal is to optimize a reward signal. Each agent estimates the Q-values for each state and for each other agent. Obviously, the exact Q-values of opponents are unknown to an agent, so each agent has to learn them and thus reveal each opponent's strategy. After having revealed these strategies, a Nash equilibrium is computed. A Nash equilibrium is computed for each stage of the game. This means that when the game is in a given state, each player performs an action and a state transition is made. At each such moment each agent seeks a Nash equilibrium. The authors noticed that the algorithmic complexity of their algorithm is dominated by the computation of Nash Equilibrium, for which they use the Lemke-Howardson method [35]. A part from, the Nash equilibrium computation, the space complexity of their algorithm is linear in the number

of states, polynomial in the number of actions and exponential in the number of agents. The authors use an infinite sampling technique and decay their learning rate. Under these conditions, they prove convergence towards optimal Q-values. However, three additional assumptions must hold: Firstly, each state has to be visited infinitely often; Secondly, each agent has to update its Q-values according to the current state and the actions of each player; and thirdly, each stage game has an optimal point or a saddle point. Hu et al. also performed some experiments in the context of multiple agents acting in a shared environment. Firstly, they study learning convergence. The authors achieved nearly optimal Q values when each state-action tuple is visited just 95 times on average, despite their formal proof indicating that each state has to be visited infinitely often. The authors then studied the phenomenon of malfunctioning agents. When some agents behaved randomly, instead of following the computed Nash equilibrium, the impact of optimal Q-values for the other agents was found to be low. Another phenomenon that the authors studied is the presence of multiple Nash equilibria. As evaluation criterion that use agents that reach optimal Q-values by always selecting a given Nash equilibrium. This method is possible due to the usage of the Lemke-Howson algorithm which always returns Nash equilibria in a fixed order.

4.4 Summary

In this chapter we have presented a short primer on game theory and reinforcement learning based on Markov decision processes. These theories have been linked in order to model multiple agents having different interests in a shared environment. The normal-form representation of a game consists of a set of players with a defined set of actions and a payoff structure. The extended form can be used if a game is repeated multiple times. Nash equilibria of a normal form game can be computed in order to find the optimal strategies for each player. These optimal strategies are not necessarily Paretto optimal. In game theory, the major assumption is that players are rational, and aim to optimize their payoff. In practice, this is not always the case. Therefore, the quantal response equilibrium is introduced in order to study how stable a Nash equilibrium is. The irrationality of a player is reduced to erroneous payoffs.

Game theory has already been applied in the context of honeypots. However, the majority of contributions define games at the level of infrastructure composed of production machines and honeypots. Moreover, the payoff structures have been manually defined [108] or derived from a survey [98]. Such approaches are only suited to static models. However, an attacker's interests cannot always be known in advance, and may change. Therefore this thesis considers reinforcement learning. Reinforcement learning is founded on Markov Decision Processes (MDPs). In an MDP, an agent interacts with its environment by performing actions. Each action results in a reward or a punishment. An agent must find a policy that maps states to actions. If all the parameters of an MDP are known, the optimal values for a state or a state-action pair can be algebraically computed according Bellman's equations. The optimal value can then be used to determine the optimal policy that leads to these optimal values. However, some parameters such as the reward distributions or the transition probabilities are often unknown. These cases can use a model-free learning approach, where an agent tries to achieve the optimal values by following a trial-an-error approach. A model-free learning agent is composed of an explorer and a learning rule. The explorer handles the trade-off between exploration and exploitation, and the learning rules define how the optimal values are updated. The learning rules described require a stationary environment where the probabilities of receiving a reward or making state transitions do not change over time. When an environment is shared among multiple agents having different interests, this assumption is often violated. MDPs have been extended to stochastic games (Markov games) where multiple agents are formally modeled. Fink [50] formally showed the existence of equilibrium points in such games. This statement shows the existence of Nash equilibria in stochastic games and the combination learning approaches with game theory. Game theory and reinforcement learning enable to formally model the interactions with attackers including incomplete information. The models are used to improve interactions with attackers in order to reveal a maximum information about them in an automated fashion.

Part II

Contributions

Chapter 5

Modeling Adaptive Honeypots

Simulating failures in order to lure attackers was reported for the first time in the classical paper ”An Evening with Berferd” [26], where manual interactions from a human system administrator lured an attacker into revealing many of his tactics and tools. During the operation of a high-interaction honeypot, we had a similar experience. We observed an attacker who installed a bandwidth-greedy attack tool. The attacker launched this tool, which had the effect of allowing no further attackers could attack the honeypot, so that nothing interesting could be observed. We manually injected some code into the attacker's tool such that it failed. After a while, we saw that the attacker reconnected to our honeypot and investigated the records of her tool noticing that the tool crashed. The attacker retried her command, restarted the program which failed again. After a few minutes the attacker tried to debug the program and still was not able to determine why it failed. After another couple of minutes, the attacker acquired an additional tool having similar features to the first. We observed that attackers try to achieve their own goal. We manually interfered with the tools installed and operated by attackers and noticed that some attackers reconnected to the honeypot and tried to solve the issues we created. Some attackers even tried to harden the system aiming to lock out other attackers. Thus, we assume that attackers are rational and each attack has a purpose. We address in this chapter an automated failure injecting honeypot aiming to reveal as much information about an attacker as possible. The challenge addressed in this work is to elaborate an adaptive high-interaction honeypot that attempts 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 between intruders and the honeypot are possible. 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 such as tools, skills and repositories used in attacks. Similarly, an adaptive honeypot can abnormally prevent the execution of programs initiated by an attacker and lead the attacker to perform other activities, which can provide insightful information for the security community. The case of high-interaction honeypots operating a Linux operating system exposing a SSH server to attackers is considered. SSH is usually available on servers having large amounts of processing power and bandwidth. These servers are usually highly available and have only a few numbers of downtimes and once the systems are accessed a full command line is available such that nearly any arbitrary command can be executed. Hence, SSH is a popular attack vector for attackers [3], [112], [135], [181].

Table 5.1 shows a typical attack scenario on a SSH server, compatible with the observations of Nicomette et al. [112]. An attacker discovered the SSH server and managed to hijack an account. She connects to the server with the credentials (step 0) and obtains a shell. Due to the fact that the system is unknown to the attacker, a system identification is done. The attacker first needs to know which privileges she has. For instance, this is done with the command `id` (step 2). Another useful piece of information is the kernel version which can be queried with the command `uname` (step 4). Furthermore, the attacker might be interested to see which programs are currently running on the system (step 6). The attacker, then decides to download a malicious tool (step 8). The attacker then uses the tool on the compromised system (step 10). On traditional high-interaction honeypots, an attacker has reached her goal and the honeypot operator has collected one program and its origin.

The attack sequence without any riposte from the honeypot is `sshd` → `id` → `uname` → `ps`→ `wget` → `custom`. Attackers may also enter empty commands, typographic errors or insults. Attackers provide inputs as sequences of strings. An input is a system command if and only if it is a bash command [111] or if it is related to a program installed during the setup of the system. Attacker frequently install their own tools, like SSH brute force scanners, rootkits, local root exploits or phishing server software. This means that all valid programs on the honeypot not previously known are installed by attackers. After having successfully transferred them to the honeypot, they are valid programs on the honeypot and can be executed. Each input that is neither a program nor an ENTER keystroke typed by an attacker is considered to be an insult. An adaptive honeypot interferes with the commands entered by attackers with the purpose to learn more about them. The attacker is challenged an some resistance faced her attack and she has to react. For instance, she could simply retry the program execution. Another option is to determine why the program execution failed by debugging the system. Attackers could also decide to simply download another tool or desperate attackers give simply up. In the example presented in table 5.1 the honeypot decided to return the error `Permission denied` (step 11). The attacker decided to get more permissions with a local root exploit which she retrieves from another location.

Due to this strategical interference of the honeypot facing the attacker, the honeypot operator retrieved more information about her. First, the honeypot operator learned that this particularly attacker did neither give up, nor retried the command. However, this attacker looked for an alternative way to achieve her attack goal. Second, the honeypot operator collected an additional program related to an attacker which may be interesting for anti-virus industry. Third, the honeypot operator discovered another repository under the control of an attacker. Fourth, the honeypot operator could measure the attacker's reaction time. If it is very small it is likely that the honeypot faced an automated attack. A larger delay among successive commands means that an attacker needed more time to choose the next step. Within this time frame, an attacker could simply looked up the error code in a public search engine or she could have interacted with other attackers to choose the next steps.

A naive idea would be to block all the programs installed by an attacker. However, following such an approach, an attacker would immediately find out that she cannot execute her own programs and the observation of a honeypot operator stops when a tool is downloaded. Therefore, the challenge that we addressed was to frame this kind of interactions among a honeypot operator and an attacker in theoretical frameworks which is exploited the honeypot to take the right decision aiming to maximize information retrieval from attackers. An overview of the model of adaptive honeypots is shown in figure 5.1. In the context of high-interaction honeypots the behaviors of attackers and the behaviors of the honeypots are defined in the next two sections. An attacker has usually an attack goal and enters successive commands to reach this goal. This kind of behavior we define as advances of an attacker. The purpose of this work is to elaborate adaptive honeypots that interfere with an attacker advances and as consequence an attacker has to respond to the strategical actions of the honeypot. An attacker can leave the honeypot, retry the executed command, select an alternative command, insult the honeypot or leave the honeypot. As behaviors for the honeypot we defined four actions. The honeypot can allow advances of an attacker, it can block the advance, substitute the command of an attacker or insult her.

5.1 Modeling Attacker Behavior

On high-interaction honeypots nearly arbitrary programs can be executed. Ramsbrock et al. [135] modeled an attacker behavior on a high- interaction as state machine with seven generic states:

CheckHW Attackers check the hardware configuration of the compromised system in order to determine whether it is worth to continue the attack or to stop it.

CheckSW Attackers check the current software on the honeypot in order to prepare it for further attacks.

Password Attackers often change the password of a stolen account aiming to lock out system administrators and other attackers.

ChangeConf Attackers change the configuration of the system.

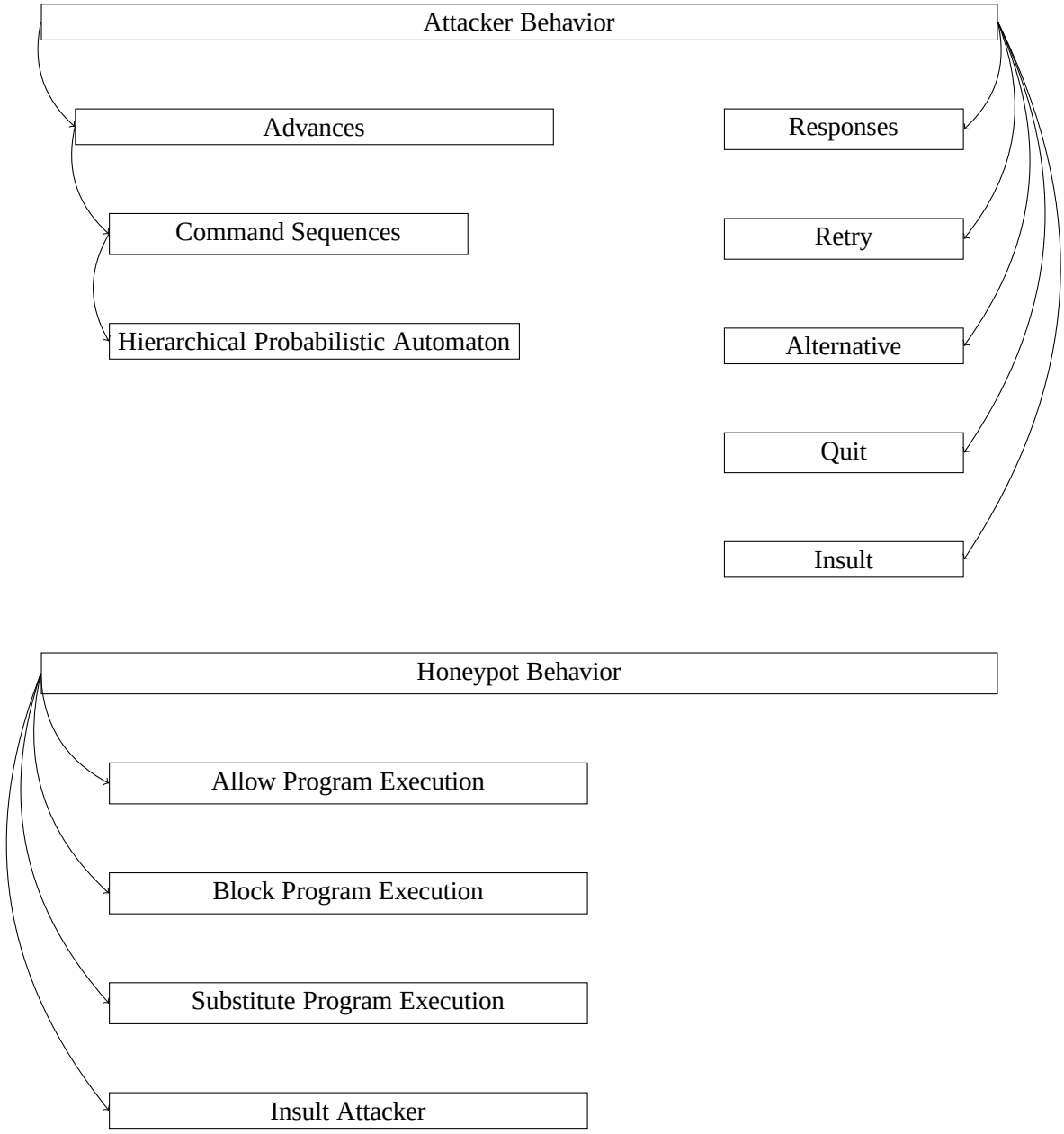


Figure 5.1: Overview of the Model Structure of Adaptive Honeypots

Step	Attacker	Honypot	Comment
0	SSH connect		Attacker penetration
1		Returns shell	Full access
2	id		System identification
3		Execute id	
4	uname		System identification
5		Execute uname	
6	ps aux		System identification
7		Execute ps	
8	wget URL_0		Acquire tool
9		Execute wget	
10	$./ssh - brute$		Misuse the system
11		Return error	Strategical block
12	wget URL_1		Additional tool
13		Execute wget	
14	./local-exploit		
15		Execute local-exploit	Make attacker happy

Table 5.1: Attack Scenario on a High-Interaction Honeypot

Download Attackers often acquire additional software in order to continue their attack.

Install Attackers often install malicious software on compromised machines.

Run Attackers run and operated their malicious programs on the stolen account.

This behavior gives a high-level overview of attacker behavior and is coarse grained. However, it enables to abstract from individual attackers which is already a good starting point for modeling an adaptive honeypot which objective is to be capable of interacting with a variety of attackers rather than with a particular class of attackers. Therefore, an omnipresent attacker is modeled able to perform generic actions. However, in the previously discussed attacker behaviors, the details on how attacker reach their goals are abstracted too. In addition, it is difficult to automatically map observed system commands to these states. The semantic richness of a full operating systems permits a nearly infinity of alternatives to reach a given state. Therefore, we aim to elaborate a more generic model, where attackers enter sequences of strings as inputs on an adaptive honeypot. The states for a hierarchical probabilistic automaton are derived from these inputs. Hence, attackers entering inputs perform transition in this automaton. An attacker can advance in her attack by performing the next transitions. However, if she is detoured from her attack she can respond. Therefore, we define two abstract behaviors for an attacker: the advance operation and the response operation. We define that attackers do their advance operations in a hierarchical probabilistic automaton and that they can select among four choices as response when they have derived from their attack sequence.

5.1.1 Hierarchical Probabilistic Automaton

Once attackers have compromised a system, they start to enter sequences of commands. Usually, they start to investigate the system, prepare the system and abuse the system [135]. In essence, they enter commands on the system until they reach their goal. If they are not disturbed we define this behavior as advance operation. We extended the model of Ramsbrock et al. [135] with an hierarchical probabilistic automaton in order that it can be induced with observed program execution from attackers. Probabilistic automata are often used in the field of pattern recognition and computational linguistics [170]. They are particularly interesting to add probabilities to a given structure [170]. An attacker who is attacking a honeypot is such a structure. He or she

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` etc. These sequences of program executions are then considered as transitions in an automaton. We define the states of the automaton as the programs that can be executed on the honeypot. Moreover, three additional states are added. First, the state *custom* describes valid programs on the honeypot that are installed by attackers. Second, the state *empty* describes the behavior of the command line shell when an attacker entered an empty command and hit the ENTER key on her keyboard. Third, the state *insult* is added because attackers sometimes enter invalid commands that are either typographic errors or insults. Each program has some program arguments which are passed as an array to the `main` function upon execution. If no command line arguments are explicitly passed to the program, the first command line argument corresponds to the program name [104]. Moreover, different programs may have the same command line arguments, differing only in semantic. Thus, a hierarchy between programs and command line arguments is introduced. 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. For instance, 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. [170] is used. Let the tuple $\mathcal{A} = (Q_A, \Sigma, \delta_A, I_A, F_A, P_A)$ be the automaton where:

- Q_A is a set of states
- Σ is the alphabet
- I_A: $Q_A \rightarrow [0, 1]$ (initial state probabilities)
- $\delta_A \subseteq Q_A \times \Sigma \times Q_A$ (set of transitions)
- $P_A : \delta_A \rightarrow [0, 1]$ (transition probabilities)
- $F_A : Q_A \rightarrow [0, 1]$ (final state probabilities)

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[1] 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 5.2. An attacker connects to the honeypot via SSH and stays in the `sshd` state. Next he or she can execute the program `ps` with the probability of 0.3, the program `ls` with the probability of 0.5 or the program `wget` with a probability 0.2. After the execution of the program `ps`, the programs `ls` can be executed with a probability of 0.8 and the program `tar` with a likelihood of 0.2. From the state `ls` and `tar` the attacker can reach other states with the respective probabilities. For the sake of readability, the first command line argument for each program and the command line arguments for the programs `ls`, `wget` and `tar` are omitted.

During the operation of a high-interaction honeypot executed programs can be observed and two questions have to be answered

- Which program executions are related to an attacker and which ones to the system itself?
- What are the relationships among the program executions?

[1]The probability that an attacker owing credentials of the system is performing a login on the honeypot.

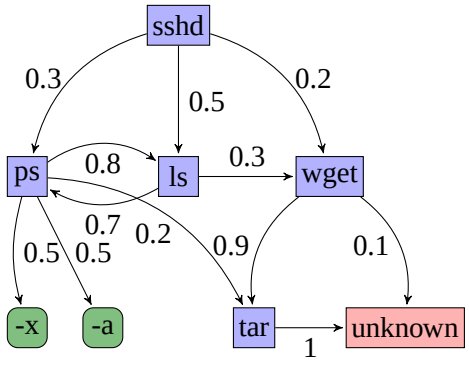


Figure 5.2: Honeypot Hierarchical Probabilistic Automaton

$\vec{A_1} =< sshd,\ bash,\ uname >$

$\vec{A_2} =< sshd,\ bash,\ ps >$

$\vec{A_3} =< sshd,\ bash,\ uname,\ wget >$

Transition matrix

	sshd	bash	uname	ps	wget
sshd		3			
bash			2	1	
uname					1
ps					
wget					

Figure 5.3: Recovering Transition Frequencies

Process Vectors

Transitions between programs are described by conditional probabilities, capturing the likelihood of one program being executed after a previous one. Sequences of program executions from a deployed high-interaction honeypot are used to determine these probabilities. Such a sequence of programs is considered as a 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 $< /bin/bash , /usr/bin/wget , /usr/bin/tar >$. An example of transition probabilities is shown in figure 5.3. Three process vectors are shown $\vec{A_{1..3}}$. In the transition matrix each column and each row corresponds to a state. The content of the cell contains the transition probability for the state written in the column to the state representing a row. Each process vector starts with the state `sshd`. The first observed program that is executed is `bash`. Hence, the transition `sshd` → `bash` has been observed three times. The third executed program of the process vectors $\vec{A_1}$ and $\vec{A_3}$ is the state `uname` and has been observed 2 times. The transitions `bash` → `ps` and the transition `uname` → `wget` appeared only once. These numbers of transitions are then normalized such that probabilities emerge. Unknown transitions are in a first step tackled with a smoothing approach which described on page 71 resulting in a fully interconnected automaton. The created unobserved transitions have low probabilities in order to not influence the later payoff computations. The fact of including only the last command respects the Markov property defined in chapter 4 which is a requirement for the next interaction model.

Attacker Process Trees

In order to obtain the process vectors, the Linux kernel data structure, which resides on the honeypot and holds the process tree information has to be inspected. Attackers usually execute programs, as soon as they manage to compromise a honeypot. That in turn, triggers a `clone`[2]or `sys_execve` system call which has to be monitored. As multiple attackers can be concurrently connected to the honeypot, and also, as the system itself is using `sys_execve` and `clone` system calls, a distinction needs to be introduced. 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) [97]. An attack usually starts with a privilege separated process of the SSH server [129], denoted p_0. The process p_0 then forks, resulting in a `clone` system call or directly executes a program via the `sys_execve` system call. It is considered 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. A process tree is defined as a tree structure where each node can contain a process id, a timestamp[3], a program name or a command line argument resulting from a `sys_execve` or clone system call. An edge links two process identifiers with each other. This 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). Let T_i^c be a tree induced by `clone` and `sys_execve` system calls. Thus, T_i^c is an ordered pair (V, E) such that V is the set of nodes and E is the set of edges. V contains the tuples (p_i, t_i, m_i, c_i). In fact a node consists of a process identifier (p_i), a timestamp (t_i), a program name (m_i) or a command line argument resulting from the execution of a `sys_execve` or `clone` system call. An edge $e \in E$, denoted as $x_i x_j$, links two process identifiers with each other, representing the parent child relationship in a Unix process tree context. Each `clone` or `sys_execve` message contains the parent process identifier which enables to recover each edge. Program names and command line arguments are extracted from the `sys_execve` system call. Furthermore the map $\delta : E \to \mathbb{N}$ links edges with time differences between successive nodes as it is defined in eq. 5.1.

$$\begin{aligned} &n_i n_j \mapsto d \\ &\delta(\underbrace{(p_i, t_i, m_i, c_i)}_{=n_i} \ \underbrace{(p_j, t_j, m_j, c_j)}_{=n_j}) = t_j - t_i \end{aligned} \tag{5.1}$$

One process tree is shown in figure 5.4. 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 made; 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 ($\delta(4121\ 4127) = 1$) 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 program `/bin/uname` after 5 seconds. The program `/bin/uname` is started with the argument `-au` and the command line arguments `bash` and `uname` represent the respective program names.

Assembly of a Honeypot Hierarchical Probabilistic Automaton

We model the honeypot capabilities as a hierarchical probabilistic automaton where each state represents a program. Each state is furthermore an automaton on its own, where combinations of command line arguments build the states of the sub-automaton.

Process trees related to attacks are extracted from a live high-interaction honeypot. 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

[2]The traditional term for creating a new process is called fork [104]. However, we use the term *clone* as it is used in the Linux kernel's source code.

[3]A timestamp includes the seconds elapsed since the Unix epoch (First January 1970) and the milliseconds concatenated with the number of CPU instructions queried by the instruction RDTSC.

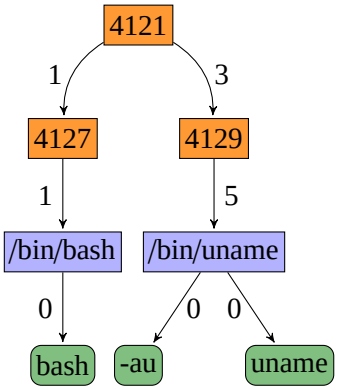


Figure 5.4: Process Tree

in transforming these process trees to process vectors which describe the sequences of programs that were executed during an attack. H is the set of paths from the root x_0 to a leaf x_k and such path is is denoted $x_0x_1x_1x_2 \ldots x_{k-1}x_k$. The map $\lambda : H \rightarrow \mathbb{N}$ describes the sum of the edge weights of the paths as shown in eq. 5.2. In the example sketched in figure 5.4, λ of the path 4121 4127 $/bin/bash$ becomes 2 and λ of the path 4121 4129 $/bin/uname$ becomes 8.

$$\lambda(x_0 \ldots x_i x_{i+1} \ldots x_k) = \sum_{(x_i x_{i+1}, d) \in \delta} d \tag{5.2}$$

The order of program execution is important. A good example would be a tool that is first downloaded from a remote location and then extracted from an archive and finally executed. The program names represent the macro states of the hierarchical probabilistic automaton and are included in the set Q_A. To recover the order we use the timestamps in the process trees. This can be described by a map $\overline{\lambda} : H \rightarrow Q_A \times \mathbb{N}$ shown in eq. 5.3 which associates programs with their execution order.

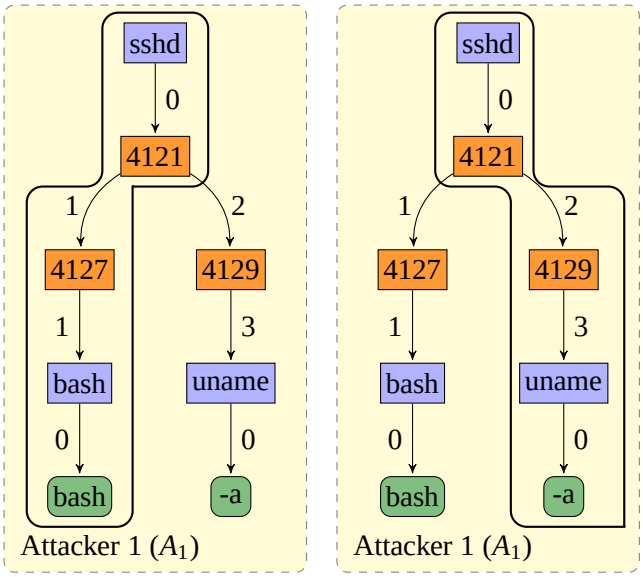
$$(\underbrace{x_0x_1 \ldots x_{k-1}x_k}_{=z}) \mapsto (x_{k-1}, \lambda(z)) \tag{5.3}$$

A process vector, denoted as $\overrightarrow{v}$, represents the program sequence executed by an attacker and is defined in eq. 5.4 (d denotes the position in the vector). A process vector has a minimal length of 1 and always starts with the program `/usr/sbin/sshd`. In order to avoid program position conflicts in a process vector, the unique number of executed CPU cycles needs to be included in the time stamp. In the example shown in figure 5.4 the process vector $\overrightarrow{v}$ is $< /bin/bash, /bin/uname >$ because the program `/bin/bash` was executed before the program `/bin/uname`.

$$\forall (x, d) \in \overline{\lambda}(H), \overrightarrow{v}_d = x \tag{5.4}$$

Figure 5.5 illustrate an example how process vectors are recovered from process trees. At the top of the figure is shown a duplicated process tree. On the left side, the left branch is selected. The branch starts with the state `sshd` and ends with the state `bash`. The sum of the edges (eq. 5.2) for this branch is 2. According to eq. 5.3 bash is at position two of the generated process vector. The sum of the edges of the right branch is 5. This means that the program `uname` is at position 5 of the process vector. The resulting process vector is < sshd, bash, uname >.

Each attacker generates a process tree that is converted to a process vector. All these vectors are then inserted in a two dimensional matrix transition called transition matrix shown in figure 5.3. The observed programs are used as labels for the columns and rows respectively. Each cell contains the frequency of how



$\sum ts$	Program name
0	*sshd*
2	*bash*
5	*uname*

Figure 5.5: Process Vectors Recovery

often a pair 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 5.2, each state is represented by a rectangle 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 `ls` with a probability of 0.5. Another transition can then be done to the macro state `ps`. The program `ps` can operated in different modes by accepting different command line arguments. In this example the states denoted by ”-x” and ”-a” are micro states, presented with rounded rectangles, and belong to the automaton `ps`.

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 ϵ. 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. The recovered transition frequencies shown in figure 5.3 are smoothed in table 5.2. Attackers executed three times bash immediately after the SSH login. This program execution is done automatically when an attacker logs in the system instead of directly executing remote commands. Hence, the transitions probability from the state `sshd` to the state `bash` is one. The SSH service also permits to directly execute commands on the remote system. For instance, an administrator can execute one her local machine the command `ssh user@host tar -cO /home > local_backup.tar`. In this case no shell is opened on the remote host. The program `tar` is remotely executed and the standard output, retrieved on the local machine and is redirected to a file denoted `local_backup.tar`. This technique has the advantage that no temporary archive needs to be created on the remote server which is not always possible to disk constraints. Obviously, arbitrary commands like this can be remotely executed. The smoothing of the probabilities takes these scenarios into account. A predefined constant ϵ is removed from the transition `sshd` to `bash` and equally distributed for the other transitions. Each transition from `sshd` to the other states than `bash` gets then $\frac{\epsilon}{5-1}$.

	sshd	bash	uname	ps	wget
sshd	$\frac{\epsilon}{4}$	$1\ (1-\epsilon)$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$
bash	$\frac{\epsilon}{3}$	$\frac{\epsilon}{3}$	$\frac{2(1-\epsilon)}{3}$	$\frac{(1-\epsilon)}{3}$	$\frac{\epsilon}{3}$
uname	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$	$1\ (1-\epsilon)$
ps	$\frac{\epsilon}{4}$	$1(1-\epsilon)$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$
wget	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$	$1(1-\epsilon)$	$\frac{\epsilon}{4}$	$\frac{\epsilon}{4}$

Table 5.2: Smoothed Transition Probabilities

Automaton Properties

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

5.1.2 Attacker Responses

Four actions are defined for an attacker when they do not reach their estimated macro state:

Retry of a command If executing a command fails, attackers may try executing it again. A failure might occur due to a syntax error or a timeout during the program execution. For instance, an attacker may download a file and a network timeout may occur. An attacker may choose another URL and another repository might be disclosed. The execution of a program can also produce an undesired effect if a wrong command line argument has been used. The program will then be executed again, with a different command line argument resulting in a different micro state but remaining in the same macro state.

Select an alternative solution The execution of a program may fail. 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 also might try to download another program or to download the source code of the program in order to compile it on the honeypot. No matter which path they choose, their behavior can be classified in a category describing the actions of choosing an alternative solution so that they can reach their goal.

Insult An attacker could enter an unknown command in the terminal that is either an insult or a typographic error. As response to an insult in the attacker's terminal she could insult the honeypot or the other attackers.

Quit Some attackers check the capabilities of the honeypot and if they suspect a trap or a worthless system, then they will leave.

5.2 Honeypot Behaviors

An adaptive honeypot is capable of adapting to attackers such that it can change its behavior such that it interferes with an attackers advances. The adaptation is done at the level of executing programs on the honeypot in kernel space. For each program execution of an attacker, an adaptive honeypot can take four different actions:

allow If an adaptive honeypot allows the execution of a program, it behaves like a regular high-interaction honeypot, because it does not interfere with the execution flow.

block An adaptive honeypot can also strategically block the execution of a program, encouraging the attacker to retry the command, to debug the process or to choose a different command. For instance, an attacker installed a buggy ssh brute force scanner. If an attacker launches the tool with a given command line switch s_1 and if the adaptive honeypot decides to simulate a segmentation fault, the attacker might believe that a feature of the tool is not working. The attacker may then choose another feature of the tool by choosing the command line s_2. An alternative scenario is where an attacker downloads an additional tool which may reveal another malicious software repository.

substitute An adaptive honeypot is also capable to substitute a command. Attackers often download programs for there malicious activities. Examples are IRC bouncing tools, ssh brute force scanners, phishing server software, bots or rootkits. Let's assume an attacker downloaded an IRC bouncing tool. If an adaptive honeypot substitutes this tool with an with an SSH brute force scanner, the attacker might believe that he downloaded the wrong tool and downloads another IRC bouncing tool and he may disclose another malicious code repository.

insult An adaptive honeypot is also capable to insult an attacker. At the beginning of our early development of an adaptive honeypot it was questionable whether insults make sense. At first glance we believed that an attacker will immediately leave when an insult is displayed in his terminal. However, on a standard high-interaction honeypot based on a Linux operating system, connected users can communicate with each other using the command `wall`. Rudy administrators could also change error messages including insults. For instance the command `sudo` can be configured to display arbitrary messages. Attackers also sometimes do not know all possible behaviors about the used malicious programs. Nevertheless, we believe that an insult in the attacker's terminal induces a surprise effect for the attacker. When an attacker responds with an insult, a honeypot operator has revealed two important pieces of information. First it is highly probable that this attack was performed rather by a human being than an automated script. In this case the insult of an adaptive honeypot served as Reverse Turing test aiming to differentiate humans from machines [30]. Second, the attacker's origin can be identified. For instance, when adaptive honeypot is attacked from a German IP address and the attacker is swearing in Romanian it might be probable that this attacker compromised a German machine serving as rebound for further attacks and connection laundering.

5.3 Summary

This chapter has described the generic attacker behavior first set out in [181]. The context in which attackers penetrate a SSH server is described. Once attackers have obtained a command line shell, they enter commands in order to reach their attack goal. We define this kind of behavior as the advancing operation of an attacker. Having entered the system, attackers execute sequences of inputs. Each input usually corresponds to a command to execute a program on the system. We consider a honeypot to be a collection of programs and represent it with a hierarchical probabilistic automaton. A macro state in this automaton corresponds to an installed program. Each program is also an automaton, where each state corresponds to a command line argument. However, not all inputs can be associated with the execution of system commands: attackers often install programs for their own use. Attackers sometimes make typographical errors or type insults on the terminal. We therefore, added three special states denoted *insult*, *custom* and *empty* to the automaton in order to address such inputs. Inputs are classified into four types. Attackers can enter a valid command that corresponds to a system program installed during the setup of the honeypot. If the command is related to valid program that is not a system program, the attacker is executing a customized program. Attackers may also enter invalid commands. An invalid command is either a profanity or a typographic error. We describe a methodology for collecting these inputs on a high-interaction honeypot.

In the operating system kernel running the honeypot, process trees are extracted and are transformed into process vectors. In addition, a distinction is made between processes related to the system itself and those re-

lated to individual attackers. A process vector corresponds to a sequence of inputs from an attacker. We focused on Linux like operating systems because their source code is often freely available. However, the same information can be collected in a proprietary Microsoft operation system by using appropriate API functions [22] resulting in a nearly identical process tree structure. We formally describe how process trees can transformed into process vectors. An example showed that an adaptive honeypot interferes strategically with the execution of attackers' commands related to attackers in order to capture more information from them. An adaptive honeypot can allow program execution in which case it behaves like a normal high-interaction honeypot. It can also strategically block the execution of a program, substitute the program or insult the attackers, with the aim of revealing more information about them. In response to these actions an attacker can retry a command, select an alternative, type an insult or leave the honeypot. In the next chapter the interactions between attackers and the honeypots are modeled with three formal frameworks.

Chapter 6

Learning in Honeypot Games

We argue in this chapter that a new paradigm of adaptive honeypots can provide more intelligence than the established architectures. The major challenge was to define the context and automatically learn the best strategies for each contextual state. The organization of this chapter is shown in figure 6.1. The interactions with attackers are modeled with three formal frameworks and learning is done in these frameworks accordingly. The choice of learning a behavior is mainly due to the high algorithmic complexity required by these framework or unknown parameters which can be handled with learning approaches. The interactions between an adaptive honeypot and attackers are first modeled with game theory. We define games between an adaptive honeypot and an omnipresent attacker where each player has an own interest. For instance, an attacker want to reach her attack goal and a honeypot operator wants to reveal information from the attacker. In addition we assume no co-operation between attackers and adaptive honeypots. In these games payoffs for each player are computed with Monte Carlo simulations. These payoffs are then used to compute the optimal strategies for each player. A minor drawback of such an approach is that the payoffs have to be computed off-line and the payoffs may be out dated for a future operation. In a second step, we model an adaptive honeypot as an agent who is optimizing a reward signal. These kinds of interactions are formally modeled as a Markov Decision Processes. This approach enables us to abstract from the transition probabilities, previously learned in an off-line manner, but ignores the competitive nature off the players. In addition, a learning approach is used to approximate the optimal behavior because in practice not all of the parameters of the model are known in advance. In a third step, we model the interactions between attackers and the honeypot as a stochastic game because it permits to abstract from the attacker's transitions and in the meantime to include the competitive nature between attackers and the honeypot. As a learning approach, fast concurrent learning was selected due to its low algorithmic complexity which permits an implementation of an on-line adaptive honeypot.

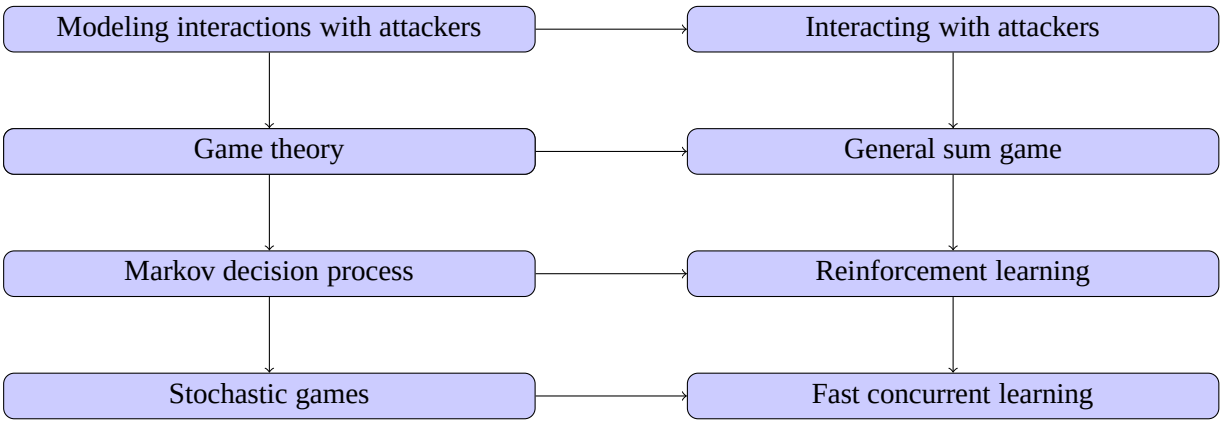


Figure 6.1: Learning in Games - Structure

6.1 Game Theory and High-Interaction Honeypots

In information security, game theory became applied [98], [144], [108] because it permits to model interactions among competitive players. However, neither game has been defined at system level directly facing attackers resulting in autonomous systems capable of interacting with attackers. Attackers penetrate the honeypot, modeled with a hierarchical probabilistic automaton, and enter commands which results into transitions in the automaton. Each command is associated with a reward or a cost. Similarly to Grossklags et al. [80] we assume that attackers want to achieve their goal as fast as possible and want to use as less effort as possible to reach their attack goal. Hence, 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 from real assets as long as possible. Similarly to Lye et al. [98], we consider attackers as one player because we are not interested to model individual attackers but an omnipresent attacker[1]. In this section we define two possible actions for the honeypot and three different actions for an attacker. We determine Nash equilibria [64], providing the optimal strategies for both the attacker and the honeypot.

We reuse the definitions and notations proposed by Amy Greenwald [64] in order to formally describe our games between attackers and the honeypot.

Let a 3-tuple $\Gamma = (N, (A_i, R_i)_{1 \leq i \leq n})$ be the game between the honeypot and the attacker, where

- N is a set of n players
- A_i is a finite strategy set ($a_i \in A_n$)
- $R_i : A \rightarrow \mathbb{R}$ is a payoff function, where $A = A_1 \times \cdots \times A_n$

The game between the attacker and the honeypot has two players. Thus, $N = \{honeypot, attacker\}$. The honeypot can block `sys_execve` 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's strategy with a 3-tuple ($Pr(Retry)$,$Pr(Alternative)$,$Pr(Quit)$) and the set A_a contains all these strategies. The notation $Pr(x)$ denotes the probability that a player is performing the action x described in chapter 5. The relation 6.1 holds for the attacker 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 games means that neither the honeypot nor the attacker can increase their expected payoffs assuming that neither player does not changes his strategy during the game.

$$Pr(Quit) + Pr(Retry) + Pr(Alternative) = 1 \tag{6.1}$$

Examples of attackers and honeypot strategies are depicted in figure 6.2. We observe that an attacker tries to invoke the command `nmap` (a popular network scanner). The honeypot might allow its 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 0.3). The execution of `nmap` was blocked and its probability was equally distributed among the transitions to the states `wget` and `uname`. Probabilities of attackers choosing the next command to be executed can be estimated from an operational high-interaction honeypot. 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) compose 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 the attacker ($Pr(Quit)$, $Pr(Retry)$,$Pr(Alternative)$).

[1] Detailed description is shown on page 66.

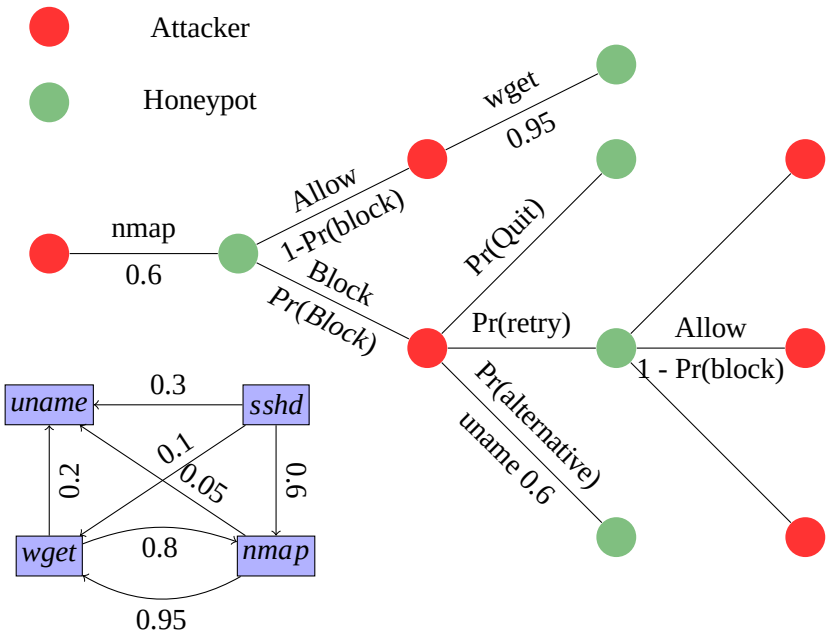


Figure 6.2: Honeypot Game

6.1.1 Defining Payoffs

Respectively to attacker and honeypot strategies, two honeypot games are proposed. The games are different with respect to the payoff computation.

Number of Transitions Similarly to Grossklags et al. [80], 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. Information, that is potentially useful for the honeypot will include tools used by attackers as well as their download sources. 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, denoted R_h^t returns -1 multiplied by the number of transitions. The less transitions attackers do, the less they are punished in terms of payoff. This game seems unfair at first glance regarding attackers. If we assume that attackers do not want to be discovered while they are performing 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 as little information as possible. Therefore, they try to minimize the number of transitions while still advancing in the attack.

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 6.2 and the payoff for the attacker is presented in definition 6.3.

$$R_a^p = \frac{Pr(path)}{Pr^*(path)} \tag{6.2}$$

$$R_h^p = 1 - \frac{Pr(path)}{Pr^*(path)} \tag{6.3}$$

The more the attacker path probability ($Pr(path)$) gets closer to the most probable path probability ($Pr^*(path)$), the payoff for the attacker (R_a^p) is likely to converge to 1. That is, if the attacker follows the most probable path, then the payoff for the attacker results in 1. In this case, the payoff of the honeypot (R_h^p) 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.

6.1.2 Computing Payoffs with Simulations

In order to compute the payoff values for all possible combination of strategies, a Monte Carlo simulation will be used. We have developed a simulator that uses bootstrap data obtained from an operational honeypot deployed over a period of 3 months. Due to computation and deployment constraints, simulations were necessary since, performing real life experiments for all possible behaviors would require 2684 different honeypots setups.

Attackers as well as the honeypot select their strategies according to a given 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 main simulator algorithm is presented in algorithm 1. In order to make the pseudo code more readable the probability Pr(block) is denoted P_{block}, the probability Pr(quit) is denoted P_{quit}, the probability $Pr(retry)$ is denoted P_{retry} and the probability $Pr(alternative)$ is denoted $P_{alternative}$. 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 (F_A), required by the probabilistic 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 denoted *steps*. When the simulation starts, the attacker enters the initial state shown in line 2. She is assumed to choose the next transitions on the most probable path. Due to the fact that the attacker was not blocked, she follows the same path. An attacker has a fixed goal. If this goal is reached, the simulation ends. Moreover, the number of transitions during a simulation is recorded and if this number exceeds a defined threshold the simulation ends. The rationale behind this is to avoid endless transitions. The attacker can retry a command or compute the next state and the step is recorded (line 15). The honeypot decides to block this step or to allow this step according the probability $Pr(Block)$. The attacker then decides whether to quit or continue the game according the probability $Pr(quit)$ (line 17). 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. Due to the fact that an attacker always computes the most probable path, the same path could not be selected due to the 0 probability transition. Of course this effect is undone for the next simulation round by loading the initial automaton. If the attacker decides to retry a command the state *alternative* is set to False, the loop ends and the next round starts. The decision to trigger an event is a function that takes as input a probability p_i then generates a random number x ($0 \leq x \leq 1$) according a uniform distribution. If x is below p_i then the value true is returned, false is returned otherwise.

The recovered payoffs enable to compute Nash Equilibria aiming to identify the best strategies profiles for each player. In a profile the honeypot can allow or block a program execution of an attacker according to a given probability. Hence, the action set of the honeypot is a set of tuples describing a blocking and allowing probabilities. As response an attacker can retry the program execution, look for an alternative solution or leave the honeypot according a probability. Therefore, the action set of an attacker contains 3-tuples

Algorithm 1 High-Interaction Honeypot Simulator

```
1:  procedure SIMULATE(src, dst, P_block, P_quit, P_retry, P_alternative)
2:      steps ← [ src ]
3:      Alternative ← True
4:      while src ≠ dst do
5:          LoadAutomaton()
6:          if currentRound > maxRound then
7:              maxRoundReached ← True
8:              return
9:          end if
10:         currentRound ← currentRound + 1
11:         if alternative then
12:             nextState ← GetNextState(src, dst)
13:             alternative ← False
14:         end if
15:         steps.add(nextState)
16:         if TriggerEvent(P_block) then
17:             if TriggerEvent(P_quit) then
18:                 quit ← True
19:                 return
20:             end if
21:             if TriggerEvent(P_alternative) then
22:                 alternative ← True
23:                 AdjustWeight(src, dst)
24:             else
25:                 alternative ← False
26:             end if
27:         else
28:             src ← nextState
29:         end if
30:     end while
31: end procedure
```

($Pr(Retry), Pr(Alternative), Pr(quit)$). The payoffs were computed via simulation. Among, the honeypot's and attacker's actions, strategy profiles can be generated which are combinations of all actions.

6.1.3 Leveraging Optimal Strategy Profiles

The key question is now which blocking probability leads to the highest expected payoff. After having recovered numerical values for the payoffs for each player, we determine Nash equilibria with state of the art algorithms. We have defined two games: a zero-sum game and a general sum game. In the zero sum game with the payoffs R_h^t and R_a^t, the honeypot's loss is the attacker's gain. While using the payoffs R_h^p and R_a^p the magnitude of the honeypot's gain is different than those from the attacker and we have a general sum game. As discussed in section 4.1, the algorithmic complexity is not entirely known for the determination of Nash equilibria. We address this problem with an experimental approach presented in section 8.3. 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 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 blocking probability β_0 is a given strategy of the honeypot that has been defined in advance. The probability α_0 results from the Nash equilibrium computation. This probability can be either 1 for a pure equilibrium or between 0 and 1 for a mixed equilibrium.

The reason for defining strategy sets with probabilities is arguable. One could imagine to define the strategy set for the honeypot with two elements, namely allow or block the execution of programs instead of defining discrete blocking probabilities. 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 [61]. 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 errors. The payoffs and transition probabilities may also change over time. In order to address these issues, the payoffs and the transition probabilities need to be periodically reestimated. Therefore we aim to abstract from this problem in the next section by choosing an approach to handle partially unknown parameters like these uncertain transition probabilities and payoffs.

6.2 Learning Honeypots Operated by Reinforcement Learning

Attackers who are executing commands on the honeypot are formally modeled with transitions in an hierarchical probabilistic automaton as it is defined in chapter 5. An adaptive honeypot can interfere with the attackers' actions by blocking the execution of strategical important programs. Transitions in the automaton are simulated and payoffs are derived. These payoffs are then used to compute the optimal strategy profiles founded with game theory. It is assumed that the attacker wants to stay on the path with the highest probability which is also included in a reward model. However, in practice it is difficult to get meaningful transition probabilities. A straightforward approach is to recover them from traces of a deployed high-interaction honeypot as it

is shown in chapter 8. However, there are no guarantees whether these probabilities are stationary. Another phenomenon is the diversity of attackers who are attacking the honeypot. Firstly, some attacks are fully automated. Secondly, attacks are performed by script kiddies, which are novice attackers, who do not understand all the details. Thirdly, some attackers are more experienced and understand all the details. All these kinds of attackers are considered as one collective player. A first step towards these uncertainties is the introduction of probabilities in the actions of each player. For instance, the probability 0.3 to block an attacker's program means that 30 program executions out of 100 should be blocked. Assuming, that a well balanced set of traces is present to bootstrap the automaton, the honeypot takes the right decisions on average. However, the uncertainty on the transition probabilities introduce erroneous payoffs for both players. This problem is tackled with a quantal response equilibrium. In this case a parameter λ is introduced which is the inverse of errors introduced on the payoffs. The parameter λ is echeloned in each quantal response equilibrium analysis.

The results of this analysis show how robust the Nash equilibria are facing faulty payoffs. In this section, a model is proposed to abstract from these uncertainties and to extract more information of an attacker rather than keeping him busy. This section describes an adaptive honeypot, denoted Heliza which allows information retrieval from attackers to be optimized by leveraging reinforcement learning algorithms.

According to Sutton [161], reinforcement learning is an automated method for goal-directed learning and decision making that works to maximize a numerical reward. An agent must discover which actions provide the most reward by trying them out. Rewards can be positive or negative and an agent by default tries to optimize its reward in the long run. A general overview of reinforcement learning is presented in [161] and is shown in figure 6.3. An agent operates in a specific environment and can perform various actions (a_t) at discrete time steps, denoted by the variable t. Each action results in a state change s_t and a reward is given for the selected action (r_t). A classical example is an agent that needs to find the exit of a maze. The agent can move north, south, east and west. Each position in the maze results in a distinct state. When the agent bumps to the wall or wants to make an impossible transition, the agent is punished. When the agent makes a valid movement no reward is given. However, if the agent finds the exit of the maze, it gets a positive reward.

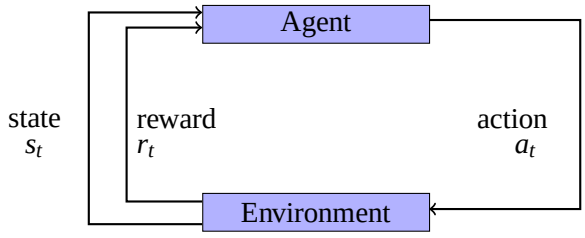


Figure 6.3: Reinforcement Learning Overview

In this section we consider an adaptive honeypot denoted Heliza which is dedicated to be attacked, and behaves like a learning agent which is continuously under attack. As described in chapter 5 attackers want to execute commands. Heliza has to take decisions as to allow or block these commands. Heliza is also able to forge outputs or insult an attacker. Following a decision, an attacker can enter another command, which results into a state change. Each state change is also linked with a reward. Attackers are constantly attacking Heliza resulting in sequence of inputs including program executions, typographic errors and insults.

6.2.1 Environment

Attackers compromising the system are modeled as its environment. The behavior of attackers is defined in chapter 5. They penetrate the system via SSH and provide input strings that are usually commands. For instance, they may inspect the system and then try to make it ready for their malicious purpose. A typical attack sequence is for instance `sshd` → `uname` → `wget` → `tar` → `custom`. Attackers may also enter empty commands, typographic errors or insults. As described in chapter 5 attackers enter sequences of strings denoted

$i_0 i_1 i_2 \ldots i_n$ where $i_n \in S^*$. The underlying finite state space is $S = \{s_1, s_2, \ldots, s_n\}$ is similar to the hierarchical probabilistic automaton presented in section 5. An input i_n is a system command if and only if it belongs to the set $L = \{s_1, s_2, s_3, \ldots, s_n\}$ which contains all bash commands [111] including system programs installed during the setup of the system. In addition we add three special states. $S = L \bigcup \{insult, custom, empty\}$. A transition to the `empty` state is made when an attacker hits the `ENTER` key on the system. Attackers sometimes do this to test whether the remote connection is still working. Attackers often install customized tools, which we designate the set C, like SSH brute force scanners, rootkits, local root exploits or phishing server software. Hence, $C \subseteq S^*$. This means that all valid programs on the honeypot not previously known are installed by attackers. After having successfully transferred them to the honeypot, they are valid programs on the honeypot and can be executed. Each input that is neither a program nor an ENTER keystroke typed by an attacker is considered to be an insult. Hence, the set of insults is $I = S^* - L - \{empty\} - C$. When an attacker wants to execute a custom installed program, a transition to the `custom` state is made. We define a relation $z_1 \subseteq C \times \{custom\}$. A string that is entered by an attacker and that is neither a program nor an ENTER keystroke is mapped into the state called `insult`, which is formally defined by the mapping $z_2 \subseteq I \times \{insult\}$.

6.2.2 Honeypot Actions

The added value of honeypots lies in their ability to learn from attackers or to reveal information about them. Heliza is adaptive and capable of taking actions in response to attacker actions. Heliza aims to collect attacker tools and to detect whether the attack is automated or manually performed. Heliza can also be tuned to keep attacker busy. Four actions $a_{1...4} \in A$ are possible for Heliza: It can behave like a standard high-interaction honeypot by allowing (a_1) command executions. When Heliza decides to block (a_2) a command entered by an attacker, it is not executed, but an error code is returned. Heliza can also substitute (a_3) commands. For instance, when the command `w` is executed, aiming to see how many users are logged in, Heliza lies and shows a previously generated content. Another, example is where an attacker executes the tool `wget` with the intention of downloading a customized tool. In this case, Heliza could lie and display a ”page not found” error, which may lead an attacker to reveal another malicious repository. An alternative possibility is to swap a few bytes in the downloaded payload. Provos et al. [132] showed that the lifetime of malicious code repositories can be very small (1 hour) and if an attacker is connecting to such a site we assume that it is not suspicious if we return a ”page not found” error. Heliza can also insult (a_4) attackers. This action mainly serves as Reverse Turing Test [30]. The purpose of such a test is to discover whether an action is being performed by a human being or an automated tool. The insult decision leads to a display of an insult in the terminal of an attacker. An attacker can respond with an insult. By doing so, it is highly likely that the attacker is a human being. Suppose that an attacker has downloaded a customized tool and wants to execute it. Heliza then replies: `Is this all what you want to do?` Some attackers immediately leave when they see a message like this. Obviously, in this case we cannot determine whether this attacker is a human or not. However, some attackers get overwhelmed by emotion and type insults on the terminal. In this case, Heliza can discover that the attacker is a human and can sometimes determine the native language of the attacker. Some attacks are automated and their reaction to insults depends on the capabilities of the script. Some scripts check error codes and the output of the executed command and take appropriate actions. Other scripts have no error handling and just continue the attack.

6.2.3 Rewards

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

Collecting Attacker Related Information

Alata et al. [3] described that attackers often install customized tools on high-interaction honeypots designed for their malicious purpose. Hence, the goal of our reward function is to collect as many attacker tools as

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

Table 6.1: Sample Attacker Session

possible. The more we are interested to discern the linguistic features of an attacker. However, the main focus is on customized tools installed by an attacker. The reward function with this purpose is defined in eq. 6.4 where i is the input string used by an attacker. Each input i of an attacker sequence of strings $i_0 i_1 \dots i_n$ is mapped with the states in the state space as it is described in section 6.2.1 $s_i \in S$. The normalized Levenshtein distance [62] is denoted l_d and the action taken by Heliza is denoted a_j. The merged set of custom commands and system commands is $Y = C \bigcup L$. If an attacker does a transition to a customized tool, Heliza gets the highest reward (1) and if an attacker executes a system related command the reward 0 is distributed. However, if an insult is entered, the Levenshtein distance between this string and all other programs (x) is computed and the minimal normalized distance is returned as reward.

$$r_t(s_i, a_j) = \begin{cases} 1 & \text{if } i \in C \\ \min\limits_{x \in Y}(l_d(i, x)) & \text{if } i \in I \\ 0 & \text{otherwise} \end{cases} \tag{6.4}$$

If the attacker just made a typographic error, the minimum normalized Levenshtein distance is low. Not much information about the attacker has been revealed except it is highly probable that this attacker is a human being. However, if the distance is close to 1, an attacker has entered a completely unknown input, which may be valuable for Heliza. The reason for normalizing the Levenshtein distance between 0 and 1, is that Heliza should focus on collecting tools than rather collecting attacker insults. The highest reward is still granted when a transition to a customized tool is made.

An example of an attacker session is presented in table 6.1 where the variable t represents discrete time steps. The column labeled with i_t shows the input provided by an attacker which is mapped to a state s_t in the state space. For a given transition made by an attacker, Heliza can take an *action* and get a *reward*. In this example, an attacker connected to the honeypot at time 0 and wants to get to the `sshd` state. Heliza allows this transition and gets a reward of 0. The attacker then wants to execute the command `ssudo`, which is classified as an insult. However, the attacker simply made a typographic error. For this simplified example, the Levenshtein distances are computed between the input `ssudo` and all the installed programs {`sshd`, `sudo`, `wget`}. The resulting set of Levenshtein distances is $\{1, 2, 5\}$. The minimal Levenshtein distance is 1, which means that only one character needs to be edited to get to the string `sudo`. Hence, the normalized Levenshtein distance becomes $\frac{1}{8}$. In step 2, the attacker notices the typographic error and enters the correct command. This time, Heliza blocks the command. The attacker decides in step 3 to download a local root exploit. Heliza decides to return a forged output stating that the requested page was not found. The attacker then selects another malicious repository and this download is allowed. The attacker wants to execute the local root exploit. A transition to the state `custom` is made because the program was not known during the honeypot setup. The reward 1 is returned because Heliza has collected a customized tool from an attacker. Heliza decides to print the text `Are you stupid enough to execute this crappy tool....` For the attacker this is usually a surprise, because she has to determine how this text emerged in her terminal. Is it simply an output of a compromised tool or coming someone else who is monitoring her? This could either be another attacker or a system administrator.

Sometimes, an attacker is overwhelmed by emotion in such a stress situation and types `I am not stupid dude, it is time for revenge ...`. This time, the normalized Levenshtein distance becomes $\frac{47}{145}$ which rewards the honeypot of having revealed an entire sentence from the attacker.

Keeping Attackers Busy

Cohen et al. [31] already discussed techniques aiming at increasing an attacker's workload. Thus, a straightforward reward is to take into account the delay between two successive commands expressed in seconds. A higher delay means a longer reaction time of the attacker in handling partial attack failures. We cannot determine what the attacker was doing in this reaction time. It could be that the attacker was looking up a solution in available information sources or she was busy with something else. Nevertheless, the delay between two successive commands is measurable for the honeypot. Hence, we define a function $\delta : S \times S \times A \times \mathbb{N} \rightarrow \mathbb{R}$. The reward function defined in eq. 6.5 returns the temporal difference needed to transit from the previous state to the current state by taking the action a_j at the time i.

$$r_d(s_i, a_j) = \delta(s_{i-1}, s_i, a_j, i) \tag{6.5}$$

6.2.4 Learning Agents

Formally, the interaction of the honeypot with attackers is described with a Markov Decision Process [161] which is composed of:

- A finite set of states S.
- A finite set of an agent's actions A.
- A transition function $T : S \times A \rightarrow PD(S)$, where $PD(S)$ is the probability distribution over the set S.
- The reward function $R : S \times A \rightarrow R$ defining the distributed rewards.

The set of states has already be defined in section 6.2.1 and is mainly derived from the hierarchical probabilistic automaton defined in chapter 5. It consists of installed programs on the honeypot including the special states *custom*, *insult* and *empty*. The modeled adaptive honeypot is an agent and has a set of actions A composed of the actions presented in chapter 5. An adaptive honeypot can allow or block the execution of a program. It can also substitute the execution of a program or insult an attacker. Hence $A = \{$ allow, block, substitute, insult$\}$. An adaptive honeypot is in a given state and takes an action which results in another state according to the probability distribution over the set S. Each action is also related to positive or negative reward according a reward function. Although this model permits to describe a transition function and a reward function, these parameters are often unknown and have to be learned.

Heliza is a learning agent, and attackers act as its environment. According to Sutton [161], an agent has the ability to perform a set of actions in various situations (states). Each action is awarded with a positive or negative reward. The purpose of reinforcement learning is to find the optimal policy to select the most promising actions in given states. Formally, a policy π is defined as a stochastic rule used by an agent to select actions [161]. Reinforcement learning is divided into two categories: off-line learning and on-line learning [161]. Monte Carlo methods are frequently used for off-line learning methods and time difference learning methods are used for on-line learning. Either method requires complete knowledge about the environment and both try to optimize received rewards. The purpose of a policy evaluation is to estimate the value of a given state s under a policy π [161]. However, in the context of adaptive honeypots, we are interested in evaluating state action pairs rather than states. An attacker whose rootkit execution has been blocked may chose another path in the hope to achieve the initial attack goal. The objective of Heliza is to incrementally discover the policy for choosing actions in given states, which is usual for on-policy methods [161]. Attackers connect to Heliza and

perform some malicious activity resulting in state transitions. In the state `exit`, they leave the honeypot which means that they have reached an absorbing state. This phenomenon gives the possibility of breaking down the learning method into episodes. The policy is being evaluated at the end of an episode. The State Action Reward State Action (SARSA) method is a straightforward method of on-policy learning method [161]. The general form is presented in eq. 6.6. The goal is to estimate the reward Q according to a policy, for a given state s_t and a given action a_t. Due to the fact that an environment is unknown for an agent, an explorer has to decide to explore or to exploit the learned knowledge. This is a fundamental problem in reinforcement learning. We used the ϵ-greedy explorer, shown in algorithm 2 because convergence to optimal Q values has been proved with such an explorer [152]. The environment is explored according to a random component ϵ and the environment is exploited according the learning rule shown in eq. 6.6. An estimation of Q at time t is augmented by the received reward r_t plus a discounted (γ) estimated future reward, taking into account a step size parameter (α). In practice, the rewards are set retroactively and in the adaptive honeypot scenario no discounting ($\gamma = 1$) is done because the beginning and the end of an episode are known. An episode begins when an attacker connects and ends when an attacker leaves. A default value of 0.05 is used as step size parameter (α) [143].

$$Q(s_t, a_t) \leftarrow Q(s_t, a_t) + \alpha[r_{t+1} + \gamma Q(s_{t+1}, a_{t+1}) - Q(s_t, a_t)] \tag{6.6}$$

Algorithm 2 Honeypot - Greedy Explorer

```
1:  function HONEYPOTGREEDYEXPLORER(state,actionSet,lastAttackerAction)
2:      action ← 0
3:      x ← random(0,1)
4:      if x < ϵ then                                          ▷ Explore
5:          i ← random(0,| actionSet |)
6:          action ← actionSet[i]
7:      else                                                   ▷ Greedy behavior
8:          mx ← 0
9:          for i ∈ actionSet do
10:             j ← StateMatrices[honeypot][state][lastAttackerAction][i]
11:             if j > mx then
12:                 mx ← j
13:                 action ← i
14:             end if
15:         end for
16:     end if
17:     ϵ ← ϵ × λ
18:     return action
19: end function
```

In order to avoid the problem of uncertain transition probabilities reinforcement learning can be used as an alternative to Monte Carlo simulations and static payoff computations. The advantage of reinforcement learning is that it embodies model-free approaches [82]. In this case a model does not have to be known in advance, meaning that the transition functions or reward functions do not have to be completely known. 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 [161].

Hence, the probability of blocking the program execution does not depend on the transition frequencies in the honeypot automaton. The previously discussed automaton induction problem (discussed in section 5.1.1) can be avoided because a closed loop is used to estimate rewards. Reinforcement learning can be more fine-grained than just determining the optimal probability to select a blocking probability. In reinforcement learning, 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. [11] 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.

6.3 Fast Concurrent Learning Honeypot

For single learning agents, Markov decision processes are popular for modeling the environment with an agent [73], [161]. However, this modeling framework is not suited when multiple players are taken into account. Stochastic games were proposed as extension to traditional Markov decision processes. According to Hu et al. [73], 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, $k = 1, 2$, $r^k : S \times A^1 \times A^2 \rightarrow \mathbb{R}$ 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 chapter 5. The state space S remains the same, containing the programs installed on the honeypot and the game is played among an adaptive honeypot and an omnipresent attacker. The actions of each player are the same than those for Heliza. The action set A^2 for the adaptive honeypot is {allow,block, substitute, insult} and the action set A^1 for the attacker is {continue, retry, alternative, insult, quit}.

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 [73]. 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 Bikramjit Banerjee et al. [11] 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.

6.3.1 Attacker and Honeypot Rewards

In a stochastic game each player has his own reward function. For the attacker we consider a similar reward function than the payoff model, defined for an attacker in section 6.1.1 concerning the general-sum game. We 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 that he 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 she is detoured from the shortest path targeting s^* and followed another path, denoted P'. In addition to the payoff model defined in section 6.1.1, 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, she tries a little bit more. If she is disturbed too much, we believe that the attacker's interest decreases until she gives up.

$$r^1 = \frac{|P^*|}{|P'|} \tag{6.7}$$

In section 6.2, two reward models have been suggested for calibrating a honeypot agent. The function defined in eq. 6.5 focuses on the delay involved in a transition and the function defined in eq. 6.4 on the inputs provided by an attacker. In this section we combine these two functions in a single function. This has as advantage that only a single agent needs to be operated. 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. 6.8 where l is the minimal Levenshtein distance [62] between the provided input c and all the known states (see eq 6.9). 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 [3]. 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.

$$r^2 = \delta \times (l + 1) \tag{6.8}$$

$$\forall s \in S \; l = \min Levenshtein(c, s) \tag{6.9}$$

6.3.2 Learning Honeypot 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 reinforcement learning. Hence, Hu et al. proposed Nash-Q [73]. 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. [11] prove that the minmax-Q and Nash-Q are equivalent in the purely competitive domains. However, the advantage of the minmax-Q algorithm that it is more resource-efficient.

Similarly to traditional reinforcement learning, the purpose of an adaptive honeypot is to incrementally learn the optimal policy for choosing actions in given states. The major difference, here is that the opponents actions are taken into account. In the context of adaptive honeypots we use the minmax learning proposed by Bikramjit et al. [11] 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 reinforcement learning and represents the discounting rate while α represents the learning rate [161]. 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. 6.10. The estimated rewards and the greedy explorer already presented in 6.2 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.

$$Q_k^{t+1}(s_t, a_t^k, a_t^{\bar{k}}) = (1 - \alpha_t)Q_k^t(s_t, a_t^k, a_t^{\bar{k}}) + \alpha_t[r_t^k + \gamma Q_k^t(s_t, a_{t+1}^k, a_{t+1}^{\bar{k}})] \tag{6.10}$$

In practice each player maintains a reward table per state which contains the estimated Q values. The actions of a player are identified with a natural number. The actions of the honeypot are $A_{honeypot} = \{0, 1, \ldots, N\}$ and the actions of an attacker are $A_{attacker} = \{0, 1, \ldots, M\}$. A row is identified with the numerical representation of a honeypot's action. A column is identified with the numerical representation of an attacker's action. When an attacker performs an action, the honeypot has to react regarding this action. This decision to exploit the acquired knowledge or to explore new choices is taken according an explorer. In this work the ϵ greedy explorer is considered and is shown in algorithm 2. An attacker is in a given state and enters a command. The honeypot can take an action defined in the action set for this state, which is a subset of the overall actions a honeypot is capable. For instance, when an attacker left the honeypot by closing her terminal, the honeypot can only allow this action. The ϵ -greedy explorer is called greedy explorer, because it emulates a greedy behavior. The explorer is parameterized with a parameter called ϵ. This parameter defines the exploring will of an agent. If this parameter is 0, then no exploration is done. If this parameter is 1 the explorer always tries new behaviors. The explorer generates a number between 0 and 1 according to a uniform random distribution. This number corresponds to the exploring probability. If the number is below ϵ, the explorer randomly selects an action. However, if this number exceeds ϵ, the explorer first checks the estimated reward table, then considers the attacker's desired action and performs a look-up of an action yielding the highest reward. This action is then returned. At the end the exploring parameter ϵ is decayed in order to ensure learning convergence. After having taken the action a reward is distributed and the corresponding reward table is updated respecting the learning rule defined in eq. 6.10.

6.4 Summary

This chapter proposes a new paradigm for adaptive high-interaction honeypots. The behavior of attackers who are attacking an adaptive high-interaction honeypot is modeled with a hierarchical probabilistic automaton presented in chapter 5. To this automaton, we added a special state *empty*. An attacker makes a transition to this state if she enters an empty command. The *unknown* state presented in chapter 5 is partitioned into the *custom* state and *insult* state. Every time an attacker executes a program that was installed by herself or another attacker, a transition is made to the state *custom*. An input corresponding to a profanity entered by an attacker or a typographical error, is mapped to the state *insult*. An attacker enters inputs to the automaton resulting into transitions in the automaton. The interaction between the honeypot and an attacker is modeled as a game where appropriate payoff functions model the behavior goals observed in the real world and from the literature. The best strategy profiles are derived from the Nash Equilibrium. We make the assumption that hackers are always rational although this might not be the case for all attackers. The results obtained provide practical solutions for designing adaptive high-interaction honeypots. The adaptability results from blocking one system call according to the calculated optimal blocking probabilities.

We first consider a Monte Carlo simulation which uses captured inputs by a previously-deployed high-interaction honeypot to parametrize our model. Our results have been published in [181]. However, transition probabilities remain uncertain and this method uses off-line computations. These problems were addressed by using a model-free learning approach, which applied reinforcement learning in the context of high-interaction honeypots.

The Heliza honeypot described in our paper in [182], optimizes a reward signal and defines attacking intruders as the environment. The behavior of Heliza has also be extended in order to reveal more information about an attacker rather than keeping her busy. Its behavior is defined in terms of several actions that may be taken: blocking, executing the command, returning errors, or insulting. The applicability of each action is

dependent on the context, which includes the command to be executed command and the history of commands. An on-line reinforcement algorithm was leveraged to map the context of actions to the action to be taken. This work can be used to develop a new generation of honeypots that exhibit learning capabilities and adaptability which reduces the risks involved in honeypot operation. Heliza could be used as information source for studies of the social backgrounds of attackers.

Considering attackers as the environment resolves the problem of transition probabilities, but ignores the competitive nature of the relationship between attackers and honeypots. Therefore, fast competitive learning in a stochastic game is considered equivalent to learning strategic decisions while facing attackers. The interaction of a honeypot with attackers is framed as 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 also assessed learning what an attacker can do in order to accommodate a broader model in which an attacker can also be adaptive. This approach is to be published in [177].

6.5 Limitations

Even with an adaptive honeypot, the risk of losing control during a high-interaction session still exists, attackers could also make profiling attacks on the honeypot, suggesting that further work on machine learning and self stabilization approaches in the context of adaptive honeypots would be desirable. An adaptive honeypot may interfere with some commands in command block in which all commands are normally successful or all fail. The case where only one fails appears suspicious to some attackers. Attackers could perform indirect attacks. To do this, they would install an attacking script and reconfigure the honeypot such that the system itself performs the attack. In this case only the process tree corresponding to the deployment can be observed. To counter such attacks, the process tree should be extended with additional information like file system knowledge. In addition, attackers could misuse commands to achieve their goals. For instance, an attacker could for instance use `perl` to list programs instead of using the command `ls`. Experienced attackers could replace programs on the honeypot and poison the learning process. The honeypot model could also be extended by system command semantics. More complex competitive learning algorithms could be explored, taking into account multiple, possibly colluding, attackers. Further research should be done concerning attackers that know how Heliza works and who try to poison its learning process.

Chapter 7

Honeypot Operation

As discussed in chapter 3, the exposure of an adaptive high-interaction honeypot is a risky operation. A short recapitulation of these risks including their mitigation techniques is presented in order to suggest additional mitigation or evaluation techniques needed for the experiments presented in chapter 8. In addition, we aim at the identification of additional information sources for enhancing the previous suggested adaptive honeypots.

Honeypots face real attackers and real damage may be the result. An overview of operation state-of-the art virtual high-interaction honeypot is presented in figure 7.1. On a given operating system, a virtual machine is installed to expose a vulnerable service to the Internet. Spitzner [154] recommended using redundant observation points to monitor the activities of attackers. The figure 7.1 identifies four layers of data capture. Firstly, all the network traffic directed and originated from the honeypot is captured and stored as network traffic captures with the aim of keeping all the network traces. Secondly, on the honeypot itself, all executed processes are monitored, and the information being stored in an external observation database in order to determine the executed commands. Thirdly, all the system calls executed in the virtual honeypot are recorded so as to determine the actions of individual programs executed by attackers. Finally, all the system calls related to the virtual machine are monitored, (see circle (4) in figure 7.1) in order to detect misbehavior of the virtual honeypot. An adaptive honeypot is a system that is exposed to attackers and should interact with attackers. The decision on whether to allow or deny an action is taken by the decision maker, represented by a star in figure 7.1. Each adaptive honeypot is operated in an isolated laboratory environment with an an exposed on the Internet. As discussed in this chapter, the collected data should be stored in a tamper-proof environment such that attackers cannot amend or delete it which is analyzed in this chapter. The honeypot is operated on a public IP address such that attackers can access it. Usually, a firewall is set up to protect third parties from collateral damage in which the honeypot is involved. In addition, the traffic is queued to ensure that a given bandwidth is not exceeded. However, as we show in the next section, these mitigation techniques are not sufficient. Due to the legal constraints on the honeypot operator, attackers should not be able to inflict damage on third parties. Despite the previously described autonomous behaviors, the system should also be monitored by a human honeypot operator. Thus, we propose two new visualization techniques. The first gives a broad overview of the network activity and the scope is labeled (1). The second (2) allows the nature of attacks against third parties to be determined. Therefore, prior the operation of adaptive honeypots additional contributions are proposed in order to guarantee monitoring.

The data collected from attackers is a valuable asset. The data has to be stored in a reliable way such that an attacker cannot alter or delete previously collected data. An attacker who manages to delete this data could wipe the traces of the actions she carried out on the adaptive honeypot. The assets of an adaptive honeypot are system resources like CPU power and memory. If an attacker manages to make an adaptive honeypot so busy that it can no longer operate correctly. The exhaustion of resources may result in data loss or in services becoming unavailable. An adaptive high-interaction honeypot suffering from resource exhaustion caused by one attacker cannot collect data on others. The most valuable asset of an adaptive high-interaction honeypot is its public IP address: it is needed to let attackers in, but it can also abused by attackers to attack third parties. Such abuse

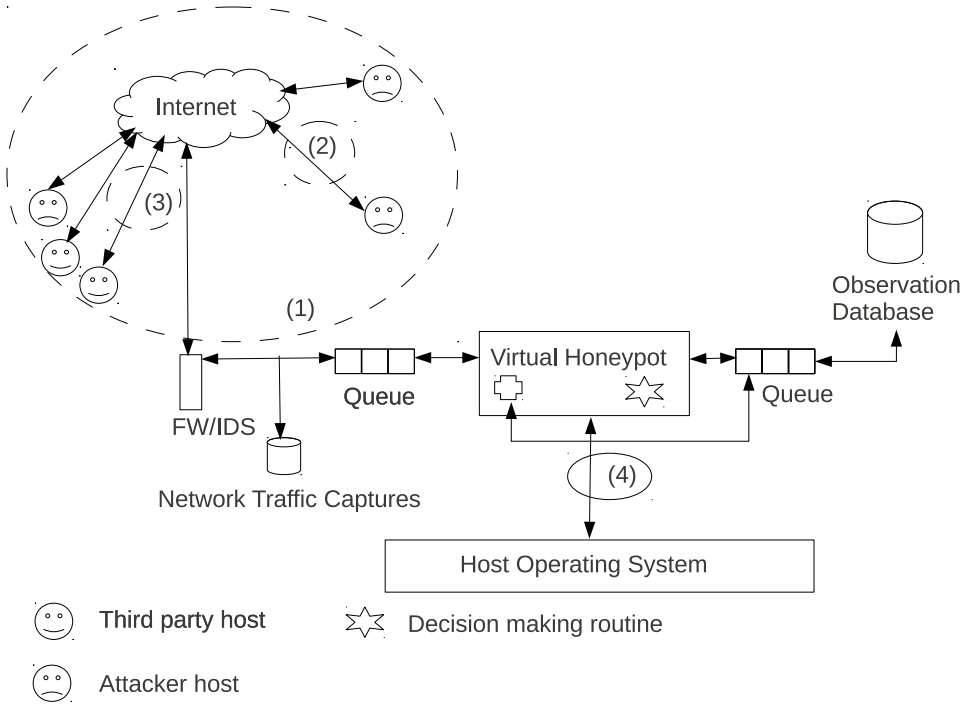


Figure 7.1: Overview of Honeypot Operation

would result in legal liabilities for the honeypot operator. Therefore, it is essential to have correct and usable traces. Hence, we focus on an evaluation of monitoring techniques, followed by technical recommendations for the implementation of adaptive honeypots.

7.1 Netflow Analysis

As shown in figure 7.1 a well-known dilemma for honeypot operators that of the Internet connectivity for a honeypot [154]: Should an attacker be capable of connecting to other hosts when she has entered the system? A honeypot operator is usually legally responsible for his IP addresses. An intruder who attacks third parties using the honeypot brings legal consequences for the honeypot operator. A honeypot becomes unattractive if attackers cannot connect to the Internet. On such a honeypot attackers quickly leave and nothing interesting can be observed. If all connections are are allowed, attackers quickly attack third parties, resulting in collateral damage and legal liabilities for the honeypot operator. All traffic incoming and outgoing from on a honeypot is considered suspicious by default[154]. Having compromised a honeypot, attackers often install customized and protected[1] tools which do not generate log information. Hence, network monitoring and, in particular, early pattern recognition of this kind of traffic is crucial. Besides state of the art mitigation techniques, like the use of network intrusion detection systems, firewalls, connection throttling and bandwidth reduction, visual aggregation helps the honeypot operator to get a wide overview of the networks involved in the attacks.

We have used Aguri [87] a flow-monitoring tool supporting IPv4 and IPv6 traffic. IP flows are monitored in near real time and are spatially aggregated. Aggregation is particularly useful to give an overview at the subnet layer rather than considering each individual flow. Spatial aggregation is realized by a special process which aggregates small flow entities into larger prefix-based trees. From these temporal and spatial aggregations of

[1] Using obfuscation and anti-reverse engineering techniques [183].

```
%!AGURI-1.0
%%StartTime: Tue Dec 01 13:54:12 2009 (2009/12/01 13:54:12)
%%EndTime:   Tue Dec 01 13:54:44 2009 (2009/12/01 13:54:44)
%AvgRate: 323.40Kbps

[src address] 1293591 (100.00%)
0.0.0.0/5 7531 (0.58%/99.22%)
     10.0.0.0/9     13545 (1.05%/30.79%)
                 10.4.0.13     237599 (18.37%)
             10.91.0.0/24 19625 (1.52%/10.09%)
                 10.91.0.22    110920 (8.57%)
                 10.91.1.4     16664 (1.29%)
   72.0.0.0/5  21618 (1.67%/37.09%)
                 74.125.79.91  202791 (15.68%)
                 74.125.79.93  214301 (16.57%)
                 74.125.79.99  27396 (2.12%)
                 74.125.79.104 13649 (1.06%)
                 83.231.205.49 324379 (25.08%)
                 83.231.205.50 73506 (5.68%)
::/0 10067 (0.78%/0.78%)
%LRU hits: 95.52% (1790/1874)  reclaimed: 0
```

Figure 7.2: Aguri Profile Representation

network traffic, the tool generates four different profiles providing summaries for incoming and outgoing network traffic in a tree-like structure. The first profile shows the source addresses, the second the destination addresses, while the third profile captures the source protocols and the last gives destination protocols. Temporal and spatial aggregations have a differing purposes. Temporal aggregation is more coarse-grained and similar to a summary of profiles, whereas spatial aggregation performs better for real-time monitoring.

A traffic profile generated by Aguri can be seen in Figure 7.2. It shows an extract of an Aguri profile for source traffic, summarizing traffic over 5 seconds. The profile is composed of a four-line header followed by the monitored network traffic in a tree-like structure that contains IP addresses, prefix lengths, the total number of bytes transferred and the volume compared to its sub-tree as percentage. The structures of the other profiles are similar. Aguri defines aggregation thresholds for each kind of profile. These present a dilemma. One the one hand, using a high-aggregation threshold results in a very high level overview. On the other, using a low aggregation threshold results in a large number of profiles. We preferred to use low aggregation thresholds, then look for similarities in these profiles.

The similarity metric for comparing two profiles is taken from [184]. The metric takes into account the layout of the tree and the aggregate traffic volume. The purpose of the comparison is to detect structural similarities that indicate activity such as brute-force attacks on third parties or the scanning of uncontrolled machines. We compute similarities on successive aggregated Aguri profiles and visualize these. To do this, we refer to the values of the kernel function, K, for Aguri trees presented in [184] which are assembled as a vector v that describes evolution of traffic using sliding window. This vector v is then mapped to a rectangle that is sequentially put into an image. The rectangle is then filled with a color derived from the kernel value K itself. The color of the rectangle describes the intensity of the evolution. We define an image as a two-dimensional space having an x- and y-axis. The discrete time is defined by $t = (t_0, t_1, t_2, \ldots, t_n)$. The time step, α, corresponds to an interval of α seconds between the export of successive Aguri trees, such that from time t_i to t_{i+1}, α seconds have elapsed. The first rectangle at the top left in our image, represents the kernel value K

for the first time period and is defined by the coordinate (x_0, y_0). The next kernel value at time step t_1 has the coordinates $(x_i + r, y_j)$, with r being the rectangle width. On reaching the end of a line, we force a line break by resetting x_i to 0 and incrementing y_j by the rectangle height r. The freshness Γ of a picture is defined in eq. 7.1. The size of the data window has an impact on the freshness of the images. A short window means a smaller and thus a fresher image, whereas a larger one means that the visualized traffic is more out of date, but at the same time gives a better overview. In our tool the freshness parameter can be specified by the honeypot operator. The graphical representation used in our work is somewhat similar to a Self Organizing Map [162] with the difference that no learning process and no training set is needed for the network traffic analysis.

$$\Gamma = \alpha \times width \times height \tag{7.1}$$

We use the similarities of our Aguri input obtained from the kernel function (defined in [184]) and map them to a color space defined by the Red – Green – Blue model (RGB) [36] (in eq. 7.2). Intuitively "black" represents the network traffic noise, while interesting patterns are shown with more intensive colors. We are particularly interested in detecting whether a given host is scanning other systems, and in tracking dominant and long-lasting TCP sessions. A dominant TCP session is a high bandwidth consuming one initiated by SSH brute-force attacks or IRC bouncing. Another interesting indicator is the amount of traffic targeting a given host. By focusing on a sequence of successively observed kernels we can normalize the kernel values between 0 and 1. We then multiply each K_i value by 2^{24} which has the effect of exploring the RGB colour space (represented in eq. 7.2). This adds a brightness factor B, considering a higher decimal precision of the kernel values[2] an intensity factor I, was added to linearly shift kernel values in the RGB space.

$$K_i' = \frac{K_j \times B}{\sum K_i \times B} \times 2^{24} + I \tag{7.2}$$

A RGB color is composed of 3 bytes. Each byte is used to represent the colors red, green and blue respectively and can be obtained by using a logical AND operation with respective bit masks. The lower eight bits of K_i' represent the color "blue", the next eight model the "green" and the high eight bits "red". This means when traffic fluctuates rapidly the similarity between two successive trees is quite low, meaning all bits are low resulting in a black color. Small similarities are displayed in bluish colors and high similarities in reddish colours. When all the bits are high (very high similarities) the color tends to "white". An observation of a white rectangle means that there is a persistent attack going on aimed at a particular target which has to be manually stopped in order to avoid real damage. Such an observation usually results in a shutdown of the honeypot experiment.

PeekKernelFlow is the outcome of our early prototyping. An overview is presented in figure 7.3. The current network topology of the honeypot is discovered by the Aguri. In the same time, all network traffic is captured with the tool tcpdump [79]. If no full-packet capture is available, we use nfdump [66] to provide Netflow records as input for the NetflowToAguri module. The honeypot operator configures the Aguri-tool to periodically export Aguri trees, which are then processed with the AguriProcessor module.

The components of the architecture are as follows:

Tcpdump Tcpdump is a popular network forensic tool implemented by van Jacobson et al. [79]. It is able to put a network interface into promiscuous mode resulting in the interception of all packets even those not addressed to the machine running tcpdump. The captured packets are buffered and can be summarized in text form or stored in a file. Tcpdump can also read previously captured traffic. Tcpdump is based on the library libpcap which implements the reception of packets in a binary form.

Nfdump Nfdump, developed by Haag [66] that reads Netflow records captured by a Netflow collector and displays them in a human-readable form that can be easily parsed.

[2]The parameter B could be canceled out. However, each value is multiplied and then casted to an integer. Thus, we can mitigate the casting error.

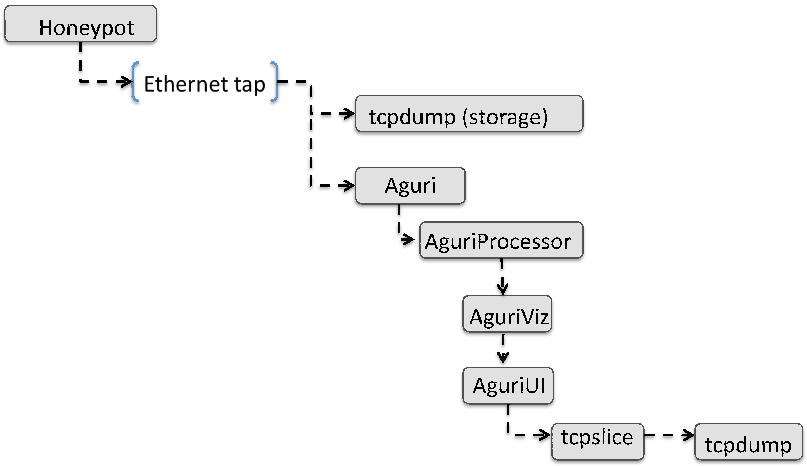


Figure 7.3: PeekKernelFlow - Architecture

Aguri Aguri [87], also uses the library libpcap to acquire network packets. From these packets, TCP flows are reconstructed and then stored in Patricia trees that take their network prefixes into account.

NetflowToAguri In this module, we implemented an adapter that converts captured Netflows into Aguri format such that the Aguri tool can process them.

AguriProcessor The AguriProcessor computes the kernel value K of two successive trees. The AguriViz-module then reads these kernels and presents them in a two-dimensional space, taking responsibility for the correct visualisation of the Aguri tree kernel functions. The honeypot operator uses the PeekKernelFlows User Interface, AguriUI. If an interesting pattern is observed in the display, the relevant network traffic can be extracted with tcpslice and piped to the tcpdump, which in this context is used to transform the captured network packets into a human readable form. We created a script, AguriProcessor which takes the output trees of Aguri as input, waits for at least two successive Aguri trees and then computes the kernel-value K, defined in [184], between these trees. The computed K-values are then used in AguriViz for further processing.

AguriViz We implemented a module AguriViz, which processes kernel values for visualisation and maps them into RGB format using the equations presented in 7.2.

AguriUI The AguriUI module is our implementation of a visual user interface for the honeypot operator. With this interface, he can adjust the monitoring setting and the parameters (image height, width, rectangle size, freshness parameter, brightness, intensity) of the generated images.

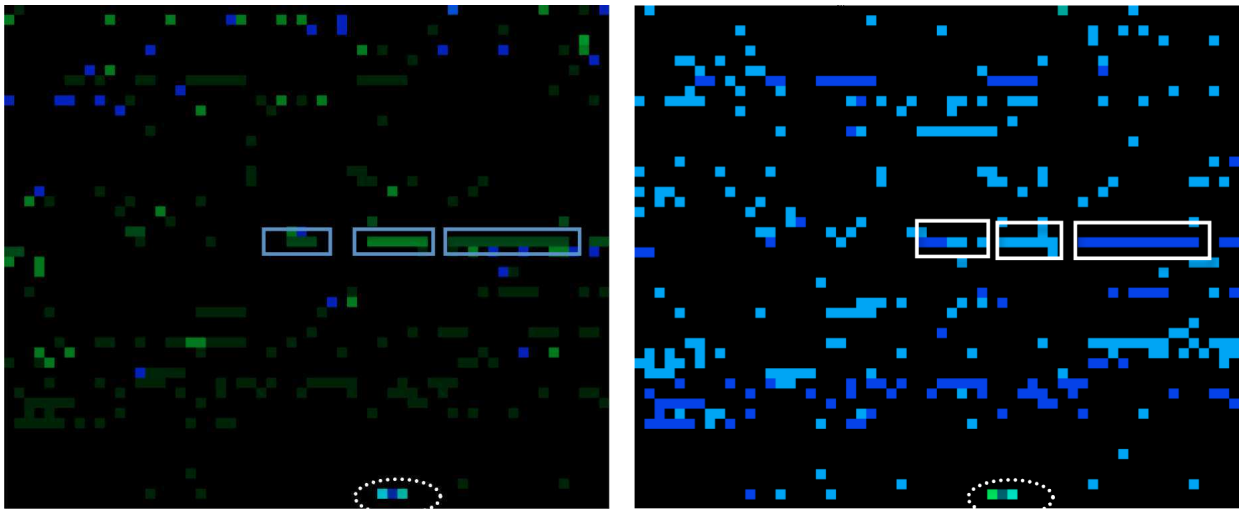
Tcpslice Tcpslice [118], developed by Paxson, is a program for extracting portions of packet-trace files according to records' time stamps.

Traces from a deployed high-interaction honeypot exposing a vulnerable SSH server on a public IP address, are used for this evaluation. The capture contains 24 hours of network traffic. All traffic related to this host is by definition suspicious and was recorded. The honeypot interacted with 47 523 different external addresses. We used Aguri's default time parameter (α = 5 seconds) which we call freshness parameter for the experiments and represented the visualization in a 24-bit color space. Choosing a larger freshness parameter α results in less timely observations. Table 7.1 summarizes statistical information for the data sets used.

In figure 7.4(a) and figure 7.4(b) we present the pictures generated by applying PeekKernelFlows to a 24-hour honeypot traffic capture. Figure 7.4(a) presents the analysis of the source profile, while figure 7.4(b)

Operation time	24 hours
Number of observed addresses	47523
Used bandwidth	64Kbit/s
Exchanged TCP packets	1183419
α (seconds)	5
Colors (bit)	24

Table 7.1: Aguri Visualization Data Set



(a) Source Profiles (b) Destination Profiles

Figure 7.4: Visualization of Aguri Profile Similarities

presents the destination profiles. The figures have a resolution of 1 200 x 1 000 pixels. This means that with a rectangle size of r = 20 and a freshness parameter α = 5 seconds, we have represented 3 000 Aguri trees in one picture, corresponding to approximately four hours of traffic.

We manually investigated some interesting patterns and noticed a minor design problem in Aguri. According to its user manual, the s-switch is used to output a summary every α seconds. However, by analyzing the Aguri trees we realized that the interval is not constant. After an investigation of the Aguri source code, we noticed that the start and end time are taking from the captured packets. This has as consequence that moments of silence, where no packets are transmitted, are not been taken into consideration, and so we detected that the time intervals varied by[3] $\alpha+\tau$. An active honeypot is continuously under a wide variety of attacks. Some attackers launch brute-force attacks against the honeypot, while others having already compromised the system, scan or control other targets. Both kinds of attackers generate a lot of network traffic-noise that is hard to investigate manually. In both images the background network traffic noise is represented by the “black”, which means that successive Aguri trees are completely different. The more the color tends to “white”, the more similar the successive Aguri trees. In Figure 7.4(a), four relevant patterns can be observed. Three successive green lines which have been framed by the rectangles can be seen. After a manual investigation of the recorded network traffic, we observed that these “green” lines represents SSH brute-force attacks, whereas the “colored” line (framed by the dashed ellipse) in the right bottom of the image represents scanning activities from our honeypot directed towards other victims. In the observed scanning activities, the attackers nearly used the entire bandwidth of the honeypot and continuously scanned entire sub-networks. This results in similar successive Aguri trees.

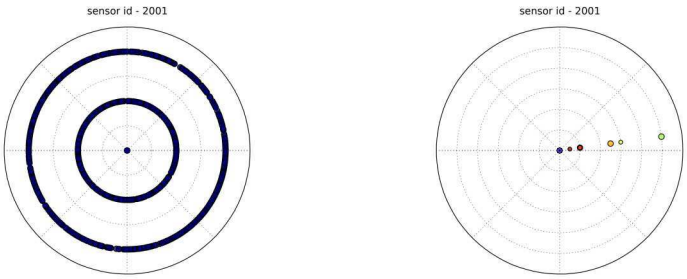
[3]If we assume that the packet p_1 has the time stamp t_1 and that the next packet p_2 has the time stamp t_2. The variable τ is defined as $t_2 - t_1 - \alpha$. Ideally, τ should be 0.

The colors of scanning activities are more light in color than dominant TCP sessions, which consist mostly of "dark" colors. This can be explained by the kernel function K, where the topological part has a greater weight than the volume part. The source profile does not focus on the exact target of the honeypot attacks. To obtain more fine-grained information about the targets, the destination profile represented in figure 7.4(b) can be used. In the destination profile analysis we can observe further patterns that are represented as segments due to the focus on destinations. We can observe how long attackers concentrated on a particular target and how much traffic was exchanged.

In this section, we also describe a visualization tool for temporal and spatially aggregated flows. The main idea is to track changes in the topology and volume on a network between successive time intervals in order to detect anomalous behaviors by which we mean harmful attacks directed towards third parties. Flows are captured for a given time interval in a special tree like structure (Aguri tree). We have introduced a similarity metric that leverages kernel functions defined over such tree structures and assessed its efficiency using a scenario of traffic captured from a high-interaction honeypot. A limitation of our approach is that an experienced attacker can poison our visualization technique by generating additional noise, so reducing the similarity between successive samples, with the result that the display becomes blacker, and hence less noticeable. We plan to improve the tool by increasing the Human Machine Interaction, by for example adding zoom features, integrated analysis and decision-making components. Another future work possibility is the implementation of "image transparency" for better tracking long-term evolution.

7.2 Network Activity Identification

The previously described visualization approach permits the identification of similarities in aggregated netflow records generated by a full capture. The Peekkernelflows tool enables a honeypot operator to identify similarities at sub net layer and to extract the related network packets. However, these network packets can easily have a large volume and it is not necessarily known which type of attack has been spotted. Hence, we propose an additional technique to identify patterns in a subset of captured packets. In essence, we use a polar representation in order to differentiate between attack patterns. Two typical attacks are presented in figure 7.5(a) and in figure 7.5(b). Both figures show a circle similar to a plan position indicator (PPI) which allows enemy vehicles to be located.



(a) Detecting a Scanning Attack (b) Detecting a Targeted Attack

Figure 7.5: Polar Attack Representation

The aim is to identify the manner and intensity with which attackers probe third parties' services in order to keep collateral damage as low as possible and to quickly identify attacks and select appropriate countermeasures. When attackers scan third party hosts by sending simple TCP SYN packets in order to discover running services, not much damage is done. However, if an attacker manages to penetrate another host from our honeypot severe damage could be done and these kind of attacks in particularly have to be avoided. The PPI is fed with TCP and UDP packets. The time stamp, the source port and destination ports of each packets are used. The source and destination port are normalized in the interval of $[0, 2\pi]$ and the result is an angle. Hence, the

angle of a point on the PPI corresponds to the ports associating with a network packet. In a set of captured packets the time stamp of each packet is analyzed. The time differences are used to determine the radius of a point on the PPI. The time is then normalized into a discrete series of time windows or slots. In each slot, the frequency f_i of each packet is counted. The number f_i is then normalized to 0 and 2^{24} and a color is the output. The slot number corresponds to the radius of a point. This kind of visualization allows three attacks families to be identified:

Port-scanning activities Such an attack makes a kind of concentric rings appear on the PPI. It is usually composed of dark points due to the low frequency of the use of individual ports. The rings result from the fact that a large number of ports is probed in a short time.

Successful Penetrations A long-lasting TCP session from our honeypot to another machine generates a large number of TCP packets having the same source port. This high frequency results in a light colored point.

Brute - force attacks Such an attack often involves a large number of connections to a given port in order to abuse the associated service. The establishment of connection to another host results in the appearance of a higher-numbered source ports. This produces a light colored spot in the first quadrant, assuming that a service with a port number below 128 is being targeted, followed by a large number of dark points in the third quadrant which represent the source ports.

The PPI visualization enables a honeypot operator to identify attack families directed towards third parties. The simplicity of mapping network packets on the PPI avoids the need to implement per host counters which quickly became exhausted when concurrent attacks are taking place on. A honeypot operator can also identify ongoing concurrent attacks on the same PPI. For the sake of illustration, the network scanner `nmap` was launched against another machine from our honeypot and the resulting concentric rings can be seen in figure 7.5(a). In figure 7.5(b), five successful TCP connections can be observed from a single source port. The major limitation of the PPI visualization is that traffic data must be pre-filtered, for instance with the Peekkernelflows, to ensure that there is no ambiguity on the target. Finally, a clever attacker could instrument the honeypot to use the same source port while doing her brute-force attack; the honeypot operator observes a successful attack on his PPI instead of observing a brute-force attack.

7.3 Full Network Capture Analysis

On a high-interaction honeypot, attackers can download tools and they can remove them after having used them. When a complete packet capture is made on the inbound interface of the honeypot, these tools are accessible in the network traces provided that the an attacker did not download them over an encrypted channel. The correct reassembly of network flows is a prerequisite for network intrusion detection systems and network forensic tools. The evasion of network intrusion detection systems was an active research topic in 1998 [133]. In our tests we discovered that some of the attacks outlined in the paper still evade network forensic tools, even when the best practices for honeypot operation are respected. In addition to classical attacks based on incomplete knowledge or unimplemented protocol features, we uncovered fundamental design flaws in popular network forensic tools. In essence a flow identifier problem causes intermixed streams which result in corrupted tools downloaded by attackers. Hence, it is not guaranteed that a honeypot operator can recover an attacker's tool despite it has been acquired over a clear text communication channel. Therefore, we start by formally describing the problem followed by an evaluation of popular network forensic tools. The root of the problem is the popular definition of a network flow, which assumes that an unidirectional IP flow is a set of IP packets that are characterized by a 5-tuple [29, 37, 41] (Source IP address, Source port, Destination IP address, Destination port, Protocol) $\in I$. The protocol parameter identifies the protocol is being used. Frequently used protocols are *TCP* and *UDP*. We define a mathematical relationship $\mathcal{F}$ between captured packets and flow identifiers shown in relation 7.3 where $\hat{P}$ is the set of captured packets where I is the set of flow identifiers.

$$(I, \mathcal{F}, \hat{P}) \subset I \times \hat{P} \tag{7.3}$$

The 5-tuple identifier is extended in NetFlow [29] by the addition of N other parameters such as ingress interface and type of service. Assuming that all parameters can be represented by numbers, the relationship $\mathcal{F}'$ between a general flow identifier and a set of captured packets is shown in relation 7.4[4].

$$(\mathbb{N}^N, \mathcal{F}', \hat{P}) \subset \mathbb{N}^N \times \hat{P} \tag{7.4}$$

In this section we focus on classical concept of TCP flows which are defined as a set of TCP packets identified by a 4-tuple (Source IP address, Source port, Destination IP address, Destination Port). The main purpose of TCP is to serve as a transport layer, which guarantees that a data stream is correctly transfered to a given destination over an unreliable network. In an unreliable network packets may be lost, duplicated or reordered [156]. The packets must be correctly reassembled in order to recover the sender's stream. Due to the diversity of TCP reassembly designs, let R be the set of reassembly functions. A reassembly function maps TCP packets to streams. The purpose of such a function is to recover the initial stream as emitted by the sender, from captured TCP packets. Let $\hat{P}$ be the set of captured TCP packets. $\hat{P} = \{p_1, \ldots, p_t, \ldots, p_T\}$. A packet that was captured at time t is designated p_t. A packet contains checksums in order to detect transmission errors. These checksums are verified with the function ω, which returns the value 1 if the packet has a correct checksum and 0 otherwise. The set of TCP packets with a correct checksum is defined in equation 7.5. In a network capture we only have the checksum information to see whether the packet is correct or corrupted, although it has been shown that checksums are not always reliable [159].

$$P = \{p_i \in \hat{P} \mid \omega(p_i) = 1\} \tag{7.5}$$

For a given set of captured packets, P, and a reassembly function $\rho \in R$ there is a set S that includes the data streams, recovered from TCP packets. A reassembly function $\rho \in R$ reassembles TCP packets and is defined in equation 7.6 such that $k, j \in \{1, 2, 3, \ldots, N\}$. The number j identifies the flow and the index k identifies the offset in the stream.

$$\begin{aligned} &\rho : P \to S \\ &\rho \mapsto f_k^j \end{aligned} \tag{7.6}$$

We also define a function σ, shown in equation 7.7, that maps TCP packets to TCP sessions. A TCP session starts with connection establishment and finishes with connection close, as it is described in [156]. The variable s_k is the set of TCP packets belonging to a TCP session. $s_k = \{p_i | \{(p_i, k)\} \subset \mathcal{F}\}$. Thus the set E contains subsets of TCP packets that belong to a TCP session.

$$\begin{aligned} &\sigma : P \to E \\ &\sigma \mapsto s_k \end{aligned} \tag{7.7}$$

Each session is mapped to a data stream, defined in function 7.8. Ideally the reassembled data stream should be identical to the data stream emitted by the sender.

$$\begin{aligned} &\eta : E \to S \\ &x \mapsto \eta(x) \end{aligned} \tag{7.8}$$

A reassembly function ρ is composed of the session function and the session mapping function, shown in proposition 7.9.

$$\begin{aligned} &P \to S \\ &\rho = \sigma \circ \eta \end{aligned} \tag{7.9}$$

[4] $\mathbb{N}$ is the set of natural numbers $\{0, 1, \ldots\}$.

TCP reassembly is a difficult task. Although a standard specification of the protocol, is given in RFC 793 [165], there are many implementations. Each reassembly tool has its own specification of a stream. The tcptrace tool considers that a stream constitutes a session, while tcpflow treats all matching tuples as belonging to a stream. Tcptrace and tcpflow put data sent from the sender to the receiver into one stream and data from the receiver to the sender in another. A stream recovered by wireshark, the successor of Ethereal [17], puts data sent by the sender and by the receiver in one stream.

High-interaction honeypots are frequently monitored by capturing incoming and outgoing the traffic. When we apply the traditional flow relation, defined in equation 7.3, we notice that the source/destination IP address and the destination port are constant for a given monitored resource. We are interested in the case where the same source port is reused. The set of packets that belong to multiple sessions using the same source port is described by equation 7.10.

$$\begin{aligned} &p_a, p_b \in P \\ &M_p = \{(\sigma(p_a) \neq \sigma(p_b)) \wedge (\rho(p_a) = \rho(p_b))\} \end{aligned} \tag{7.10}$$

The phenomenon of having multiple sessions per flow might be reducible to the birthday problem [102], assuming that the source port distribution is uniform. It is the probability P_b of finding at least two streams, belonging to same flow in a set of n streams. The applied birthday problem is formulated in equation 7.11, where $P_r = 2^{16} - 1024 - 1$ represents the TCP port range from which an operating system can choose a source port.

$$P_b = 1 - \frac{P_r!}{P_r^n (P_r - n)!} \tag{7.11}$$

As we have defined in equation 7.6, the payloads of the TCP packets are put in a stream at an offset defined in the TCP header. TCP headers indicate corrupted or duplicated packets and packets that have been received out of order. We need a verification process for reassembly functions in order to detect wrongly reassembled streams. This should meet two goals: it should estimate the accuracy of the reassembly of streams by comparing reassembled streams with a variety of independent tools; and it should help us to understand why streams have been reassembled differently by different functions. Therefore we aim to have methods to check streams. The input of these methods is composed of a reassembled stream and the raw captured packets. We put the payload of TCP packets in a vector $\overrightarrow{b}$ as shown in figure 7.6. This vector represents one TCP session. $p_i \in P, \overrightarrow{b} = (\sigma(p_i))$. The packets are put in the vector in order of arrival. A TCP packet p_i is a tuple (TCP header, and TCP payload).

We then consider a reassembled stream $\overrightarrow{d}$ as a vector of bytes, again shown in figure 7.6. From the vectors $\overrightarrow{b}$ and $\overrightarrow{d}$ we generate a matrix c, which serves to check the reassembly function. The column τ contains the time stamp attached during the capture. The variable Δ_H is the difference of the real packet length and the effective captured packet length. For packets that were completely captured, Δ_H is 0. The variable λ quantifies the TCP payload length. The number of occurrences of the packet's payload in the stream is specified by the variable f. The offset in the stream of the payload is described by the s' variable, followed by the q variable which is the sequence number given in the TCP packet. The column $\omega(p_i)$ indicates correct or incorrect checksums. Finally, the column g contains the TCP flags present in the TCP header. This matrix is sorted according to the sequence numbers and a matrix c' matrix is the result. Here we should mention that in the best case $\rho(p_i) = f_k^j$, $j = s'$ for a reassembly function. Two choices are possible if this statement does not hold: (i) the stream was not correctly reassembled or (ii) the payload of the TCP packet p_i appears more than once, i.e. $f > 1$. The computational complexity for establishing the matrix c is $O(n^2)$ because the frequencies of the TCP payloads in the stream are used.

Having presented the TCP reassembly model and TCP reassembly challenges, we wish to establish probabilities to quantify TCP reassembly errors. There are two types of potential error are presented (i) potential errors at the packet level and (ii) errors at the stream level. For a given reassembly tool it should first be checked whether it is necessary to compute these probabilities or whether they can be ignored due to a probably correct

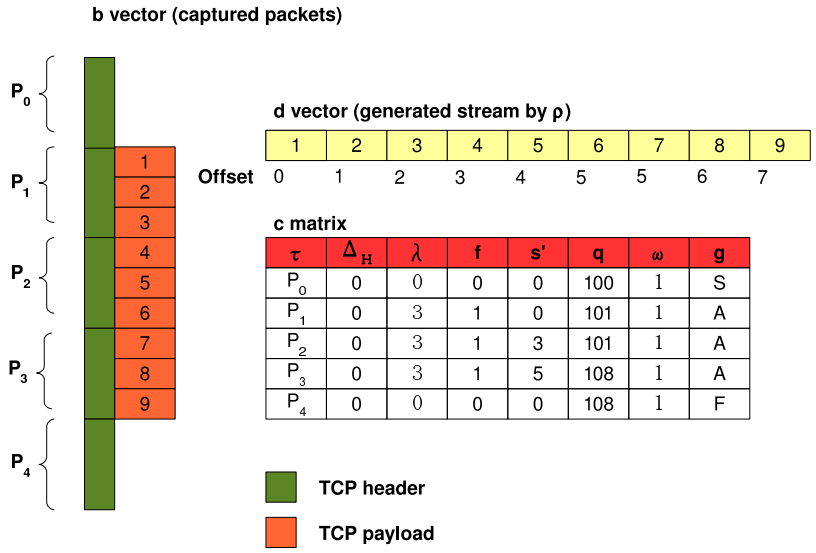


Figure 7.6: Detecting Corrupted Streams

software design. A potential cause of error at the packet level is a the possibility that a defective packet resulted in a wrongly reassembled stream. The probability of faulty checksums influencing stream reassembly is defined in equation 7.12.

$$P_c = \frac{|\{p_i \in \hat{P} \mid \omega(p_i) = 0\}|}{|\hat{P}|} \tag{7.12}$$

Some tools do not take into account IP fragmentation [156] and fragmented IP packets may impact the accuracy of reassembly. The probability of observing fragmented IP packets during the reassembly process is defined in equation 7.13 where ϕ is a function that returns 1 if a packet is fragmented and 0 otherwise.

$$P_f = \frac{|\{p_i \in P \mid \phi(p_i) = 1\}|}{|P|} \tag{7.13}$$

In order to avoid errors caused by fragmented IP packets or by faulty checksums it must be ensured that the reassembly function correctly handles such packets.

Potential errors at the stream level are quantified by the probability that a particular stream is corrupted. We have the set E containing TCP sessions and also the set of recovered streams. Ideally the number of reassembled TCP sessions should be the same as the number of recovered streams. The number δ represents the difference between the number of reassembled sessions and the number of recovered streams $\delta = \mid E \mid - \mid S \mid$. If δ is zero, no mismatch has been detected. If δ is negative, there must be TCP sessions that are present in the set but have not been reassembled. We call these sessions invisible. The probability of having such streams is defined in equation 7.14. If δ is positive, the cause is a spurious stream that was not generated from a session shown in equation 7.15.

$$\delta > 0 : I_p = 1 - \frac{\mid S \mid}{\mid E \mid} \tag{7.14}$$

$$\delta < 0 : A_p = 1 - \frac{\mid E \mid}{\mid S \mid} \tag{7.15}$$

Equation 7.15 and 7.14 give an idea of the number of streams that does not match with the number of sessions. However these equations might say that there is no error even in a situation where the two types of

error cancel each other out. An example is an invisible stream that cancels out a spurious stream. Therefore it is essential to check if the streams are consistent. The probability that multiple sessions are present in a TCP flow is defined in equation 7.16.

$$P_{spec} = \frac{|\{\sigma(p_i) \mid p_i \in \mathcal{M}_p\}|}{|I|} \tag{7.16}$$

The methodology proposed in figure 7.6 establishes a sorted matrix c' which can be used to detect incorrect reassembly in some cases. Cases where ambiguities are present can also be detected. Streams can be wrongly reassembled due to incompletely collected payloads or mixed packet payloads which are due to a faulty offset computation. the first step in the packet capturing process is to capture the complete packets in order to correctly reassemble a stream. The libpcap [79] library, used by the tcpdump provides two packet lengths. The *caplen* is the packet length as captured, and *len* is the length of the packet initially sent. The parameter $\Delta_H = len - caplen$ can be greater than zero, meaning that the packet was not completely captured. Incomplete packets that are used by reassembly functions cause gaps in reassembled streams due to the lack of captured information, with a probability show in equation 7.17.

$$P_{\Delta_H} = \frac{\mid \{p \in \sigma(p_i) | \Delta_H > 0\} \mid}{\mid P \mid} \tag{7.17}$$

Incorrect offsets can also be caused by software defects. In a correctly reassembled stream, no holes should be present. This can be formulated if relation 7.18 holding where $s'_0 = 0$.

$$i > 0 : s'_i = \lambda_{i-1} + s'_{i-1} \tag{7.18}$$

In order to use the output of a network forensic tool as evidence against someone we believe that it is mandatory to check firstly the captured input data and the capabilities of the tool. Secondly, output should validated by other independent tools. Two families of errors can emerge. Firstly it could be that the capture tool was not correctly calibrated, so that some packets are truncated. Streams might also be defectively reassembled due to ambiguities that result in implementation flaws. An additional requirement is that all analysis tools should have the same interpretation of flows, sessions and streams. We have proposed a TCP reassembly model and a stream verification methodology that can be used to derive and compute reassembly errors. We discovered that multiple sessions per flow cause problems in context where a resource is monitored over a long period of time as in high-interactive honeypots. In the next section, we are provoking the enumerated TCP reassembly errors aiming at an evaluation of popular network forensic tools.

7.3.1 Network Forensic Tool Analysis

The tools evaluated in this section are frequently used by the honeypot and security communities [17]. Many of them are vulnerable to TCP flow identification flaw resulting from their use of the 5-tuple.

Valgrind [110] dynamic execution analyzer was applied to the tools under evaluation in order to identify reports invalid pointer use and memory leaks as well as “use after frees”. Each tool was launched on a 512MB packet capture (PCAP) and was configured to extract the initial streams from the capture. The outcome is presented in table 7.2. It is a good sign that none of the tools made any invalid memory writes. These errors are particularly dangerous because in some cases they allow attackers to execute arbitrary code and thus infect the honeypot’s operator’s analysis machine. At first glance, the tools tcpflow and tcpick look clean. In tcpick, 16 bytes are not freed, but these 16 bytes are constant and do not depend on the capture. Tcpflow assumes that the encoded packet length always corresponds to the actual captured length. However, this is not always true. In some cases, the captured packet length is smaller than the expected packet length. The violation of this assumption results in invalid pointer operations leading to buffer overruns. Tcptrace also generates a large number of invalid file descriptors. This is mainly due to a lack of resource organization. Particularly in honeypot traffic, attackers are constantly scanning the honeypot resulting in a large number of concurrent

Error	Tcptrace	Tcpflow	Tcpick	
Invalid read s=4	5	0	0	occurrences
Invalid read s=1	2	11	0	occurrences
Definitely lost	345	0	16	bytes
Possibly lost	49152	0	0	bytes
Invalid file descriptors	36196	0	0	occurrences
Uninitialized	0	4	2	occurrences

Table 7.2: Valgrind Tests

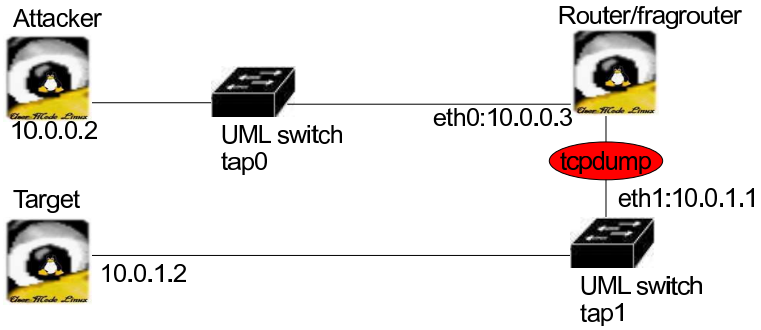


Figure 7.7: Fragroute - Attacks - Setup

TCP flows. Keeping every flow in memory is not an option if network traces of several GB are to be analyzed. Consequently, tcptrace opens a file descriptor per reassembled stream. The number of file descriptors is limited by the system, and when they run out, tcptrace simply continues without notifying the user. The result is that some sessions are not reassembled, allowing a clever attacker to wipe her traces by flooding the honeypot with TCP sessions.

The table 7.3 summarizes the test results of fragroute which is designed to test network intrusion detection systems [133]. The setup of the experiment is presented in figure 7.7. A stream is transmitted from an attacker to the victim. The attacker and the victim are on different networks. Instead of using a standard router to route the packets from one network to another, fragroute was used. All the network packets were captured by tcpdump and the resulting files were used to test the network forensic tools. Fragroute was instrumented to modify the packet streams to perform TCP man-in-the-middle attacks [133] (B1, T1, T5, F7). On the target machine, denoted victim, the stream from the attacker was stored. Each time a stream was transmitted from the attacker to the victim, the received stream was compared with the stored in order to ensure correct transmission despite the TCP attacks. For each type of attack a PCAP file was stored, if the stream was correctly transmitted. These PCAP files were analyzed with the network forensic tools. In addition, fragroute was disabled once, and the stream was transmitted via IPv6. Each column of table 7.3 describes a preset fragroute attack than can be enabled by the given command line switch. The symbol $\surd$ is used if the attack was successful and the symbol $\times$ is used if the attack failed. The versions of the forensic tools are summarized in table 7.4. Surprisingly, many of the old attacks still work today against state-of-the-art network forensic tools despite fragroute is being more than ten years old. Hence, network captures from high-interaction honeypots are not a reliable information source.

The fundamental problem of flow identification can theoretically be exploited by an attacker. We propose the *PCAP bomb* as proof of concept. A typical use case of the *PCAP bomb* is when an attacker suspects that all the traffic is being recorded. The attacker deploys the *PCAP bomb* by generating specially designed flows to an arbitrary host. Later analysis of the network traffic will trigger the *PCAP bomb* and destroy the analysis results. The name "*PCAP bomb*" is used for historical reasons inspired from the 42.zip bomb or compression bomb,

Attack	Tcpflow	Wireshark	Tcptrace	Tcpick
B1	√	√	√	√
T1	×	×	×	×
T5	×	×	×	×
F7	×	√	×	×
IPv6[5]	×	√	√	×

Table 7.3: Fragroute Tests

Tool	Version
Tcpflow	0.21-11
Wireshark	0.99.6a
Tcptrace	6.6.1-1.3
Tcpick	0.2.1

Table 7.4: Forensic Tool Versions

which were frequently used by attackers to disable anti-virus software [121]. The advantage of the *PCAP bomb* is that collateral damage is limited. An attacker does not require a lot of network bandwidth to deploy such a bomb, and network equipment is not hit. The full damage is done on the machine carrying out the forensic analysis.

An overview of the *PCAP bomb* is presented in figure 7.8. The idea is to create flow-tuple collisions. Assume that the first tuple has the lowest ISN (Initial Sequence Number), ISN_1, with the next flow having a larger ISN, ISN_2. The flow starting with ISN_2 is put in the same stream as the flow ISN_1 due to a stream identification collision, and starts at a much higher offset $ISN_2 - ISN_1$. The fake stream between the end of the first flow and the beginning of the next flow is filled either with random data or with zeros, depending on the implementation of the forensic tool. The proof of concept shell script needed to create a *PCAP bomb* is shown in figure 7.9. The bomb can be constructed using standard networking tools. The multi purpose relay tool `socat` is configured to reuse the same source port, is started in a loop in order to generate multiple flow-tuple collisions. The ISN difference is increased by using higher delays between successive flows: a time difference of 1 second between successive flows and the generation of 5 flow-collisions results on average in a stream length of 2GB. If an attacker generates 20 such collisions, meaning that she has generated 100 flows in total, she has been able to generate 40GB of traces from just 48KB bytes of raw TCP packets. Tcpflow, version 0.21-11, is vulnerable to this attack[6].

An attacker can also use a flow-tuple collision to hide a stream. A typical scenario involves an attacker who wants to download a malicious programs from a public-visible repository. The malicious software is accessible through an URL which the attacker wants to hide. To achieve this, the attacker establishes a dummy connection to the root of the repository and then reuses the same source port to trigger a flow-tuple collision. A honeypot operator using the "Follow stream" feature from the popular Wireshark tool, version 0.99.6a, sees an empty window instead of the program from the URL[7]. These preliminary tests showed that the network layer is not a perfectly reliable information source. Hence, the focus is put at the system layer in the next section.

7.4 User Mode Linux Tests

User Mode Linux (UML) is a customized Linux kernel that runs on top of a host kernel as a collection of non-privileged processes [42] and is frequently used to set up high-interaction honeypots [45, 70, 193]. User Mode Linux can be built from any recent Linux kernel (version >2.6). It is included in the default kernel as

[6] A proof is shown in the appendix chapter C.
[7] A proof is shown in the appendix chapter C.

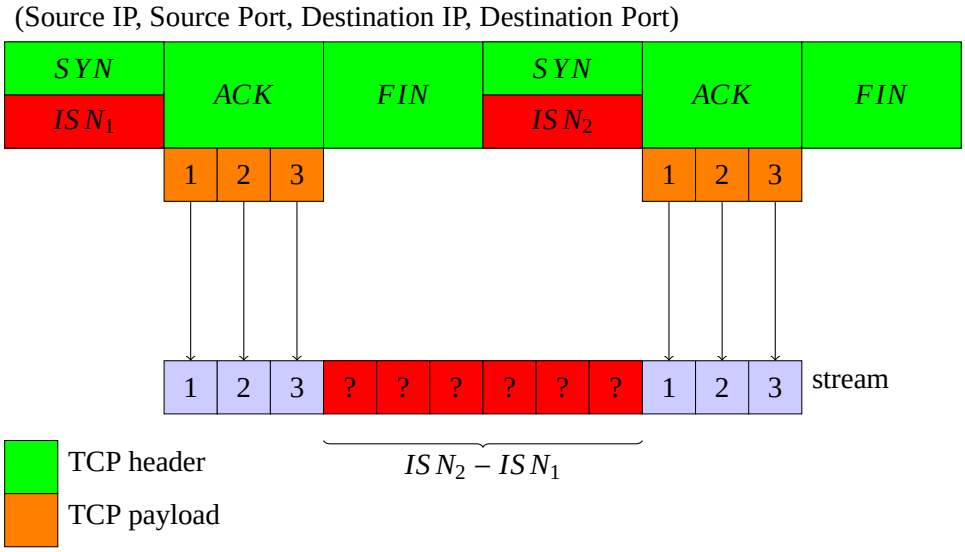


Figure 7.8: PCAP Bomb - Design

```
   tcpdump -i lo -s0 -w pcap-bomb.cap &
   i=1235
   while [ 1 ]; do
           j=0
           while [ \$j -lt 5 ]; do
                   cat req.txt | socat - tcp:localhost:80, \
                   sourceport=$i,reuseaddr
        sleep 1
        let j=$j+1
done
let i=$i+1
    done
```

Figure 7.9: PCAP Bomb - Proof of Concept in Bash

```
#include <sys/mman.h>
void main() {
mmap((void*) 0x10000, 1048576, PROT_NONE, MAP_PRIVATE | MAP_FIXED |
     MAP_ANONYMOUS | MAP_NORESERVE, -1, 0);
}
```

Figure 7.10: System Call of Death - Proof-of-Concept

```
Eeek! page_mapcount(page) went negative! (-1)
page pfn = 175
page->flags = 400
page->count = 1
page->mapping = 00000000
vma->vm_ops = 0x8227ae8
vma->vm_ops->fault = special_mapping_fault+0x0/0x60
Kernel panic - not syncing: BUG!
```

Figure 7.11: Kernel Output

a separate architecture. The UML kernel loads a dedicated file system image. In order to start, a 4GB file is filled with zeros, then this file is formated with an Ext3 file system. This file is then loop-back mounted on a mount-point of the host operating system. Because of the availability of tools like debootstrap [12] which automatically install all programs necessary for a Debian or Ubuntu operating system, we decided to install a minimalistic Ubuntu operating system in the file system image.

UML supports Copy-On-Write images (COW) [42]. The UML kernel takes two file system images. The first of these is static, including the operating system and fixed configuration files. During the operation of the UML kernel, all file system changes are written to the second file, with the result that the original operating system is not touched. Hence, heavily compromised systems can easily be restored. Another advantage of User Mode Linux, compared to real virtual machines like Qemu [18], is that the CPU is not emulated: CPU instructions triggered by an attacker are directly executed on the host hardware. User Mode Linux allows the creation of virtual hosts and routers on a single physical machine with negligible performance overhead because the hardware is used directly rather than being emulated. Despite these attractive advantages of UML, Holz et al. [70] report numerous techniques for identifying UML systems, so we went a step further. Our purpose is to determine whether, having detected a confining UML environment, an attacker could escape it or take control it. Even though when he had disabled kernel modules and `dev/kmem`, we discovered a that an attacker could execute the code presented in figure 7.10, which succeeds in destroying the honeypot by crashing the kernel as shown in figure 7.11. The kernel is unable even to synchronize data any more. This means that unsaved file system changes are not persisted. An attacker can take down the entire UML system as a non-privileged user. After identifying the vulnerabilities, we incrementally downgraded the UML kernel using the official Linux source code repository in order to determine scale of the impact of the proof of concept. Thirty kernels are vulnerable (see table 7.5) to the proof of concept presented in figure 7.10, representing a large population that could be taken down by attackers. As a mitigation technique, the system call can be patched such that the invalid arguments are rejected in advance. In addition, SELinux [142] or Apparmor [103] can be used as mitigation techniques. These components run in the kernel of the host operating system and are able to monitor processes running in user space. A policy can be established for the User Mode Linux processes such that only a limited set of system calls are allowed. Each policy violation results in process termination.

v2.6.27-rc9	v2.6.27	v2.6.27-rc1	v2.6.25-rc9
v2.6.27-rc8	v2.6.26-rc9	v2.6.26-rc2	v2.6.25-rc8
v2.6.27-rc7	v2.6.26-rc8	v2.6.26-rc1	v2.6.25-rc7
v2.6.27-rc6	v2.6.26-rc7	v2.6.26	v2.6.25-rc6
v2.6.27-rc5	v2.6.26-rc6	v2.6.25-rc5	v2.6.25-rc2
v2.6.27-rc4	v2.6.26-rc5	v2.6.25-rc4	v2.6.25
v2.6.27-rc3	v2.6.26-rc4	v2.6.25-rc3	v2.6.25-rc1
v2.6.27-rc2	v2.6.26-rc3		

Table 7.5: Vulnerable Linux Kernels

7.5 In vivo Malware Analysis

The states of the model presented in chapter 5 are identified with program names. However, attackers could modify these names or replace them with customized programs. Moreover, attacker could infect existing programs by adding malicious code. Therefore, we are looking forward for a more robust state representation based on system calls. In [185] we propose a unified approach for the analysis of both process-related information and system calls executed. This topic is of interests for several reasons. From a conceptual point of view, there is a need to represent two types of relevant information in the same model. Attack detection and analysis would be improved if two disparate types of information could be represented on the same model. The first type comprises relationships between processes, for instance the fact that one process launches another one. The second concerns the low-level inter-working between a host system and executed software. One possible information source can be provided by system calls performed during execution. There are at least two application domains for our approach: online host-level intrusion detection and malware analysis. A good host-level intrusion detector should be able to identify in real time that some user sessions are suspicious when, for instance, anomalies in user sessions occur. Such anomalies can be generated by sessions that launch processes that differ from those expected. We argue in this section that process-related information that also comprises relationships among processes can serve to build better host-level intrusion detectors. Malware analysis can also benefit from our work. Additionally, the detection and classification of malware can be improved by a combined process and system call mechanism. Obfuscated malware components might be revealed by exploring similarities in the structure of the underlying processes and system calls.

In chapter 5, process trees are extracted from the kernel operating the honeypot in order to determine the input sequences provided by an attacker. We enhance the process tree model, presented in chapter 5, and use tree kernels and graph-based kernels in order to to compare these process trees and so determine common behavior or anomalies with respect to legitimate SSH sessions.

7.5.1 Tree- and Graph-based kernels

Kernel methods are advanced mathematical tools for the classification (either supervised or unsupervised) of high-dimensional data. Their potential lies in the computation performed by a kernel function, which reveals the level of similarity between data sets. The input data is mapped into a higher dimensional space, such that the mapped data categories become separable and the distance (derived from a scalar product and corresponding to degree of similarity) in the mapped space can be directly derived from the original input space. In the recent past, kernel methods [168] have been shown to be very useful in solving problems from disparate domains such as bioinformatics or natural language processing.

The first contribution is a model that uses tree-based structures to evaluate process trees captured from live systems. An extension is based on graph kernels. Our second contribution is a method for capturing complex relationships among processes, executed system calls and other synthetic measures using the same objective.

Tree-based kernel functions

The method described in [106, 107] defines a tree kernel function based on the decomposition of a tree into its constituent substructures — subtrees or partial trees. In general, a kernel function is based on the inner product of input objects, which have been mapped onto a higher — dimensional vector space. In a tree kernel function, the similarity between two input trees T_1 and T_2 is measured by computing the number of common patterns (subtrees) between the two trees. This can be done without needing an exhaustive computation over the entire fragment space.

We have extended this kernel [32, 106] in order to capture host-related process information. For having a fragment set $F = \{ f_1, f_2, \dots \}$, the indicator function I_n is defined by [32] to be 1 if the target fragment f_i has its root in node n or 0 otherwise. The kernel function defined by [32, 106] is shown in eq. 7.19.

$$K(T_1, T_2) = \sum_{n_1 \in N_{T_1}} \sum_{n_2 \in N_{T_2}} \Delta(n_1, n_2) \tag{7.19}$$

where the sets of nodes of T_1 and T_2 are represented by N_{T_1} and N_{T_2} and the number of fragments common to n_1 and n_2 by eq. 7.20.

$$\Delta(n_1, n_2) = \sum_{i=1}^{|F|} I_i(n_1) I_i(n_2) \tag{7.20}$$

where Δ has the following recursive rule set defined in eq. 7.21,

1. $\Delta(n_1,n_2) = 0$ if productions at n_1 and n_2 are different
2. $\Delta(n_1,n_2) = 1$ if productions at n_1 and n_2 are the same and have only leaf nodes

$$\Delta(n_1, n_2) = \prod_{j=1}^{nc(n_1)} (\sigma + \Delta(c_{n_1}^j, c_{n_2}^j) \tag{7.21}$$

where $nc(n_1)$ represents the number of n_1 children, c_n^j the j-th child of node n and $\sigma \in \{0, 1\}$ is the parameter for evaluating subtrees and subset trees introduced by [106]. To include leaf nodes in the fragment space, [106] adds a condition to the recursive rule set. $\Delta(n_1,n_2) = 1$, if n_1 and n_2 are leaves and their associated symbols are equal. The similarity score is normalized in the kernel space by applying eq. 7.22.

$$K'(T_1, T_2) = \frac{K(T_1, T_2)}{\sqrt{K(T_1, T_1) \times K(T_2, T_2)}} \tag{7.22}$$

Graph kernels

On compromised machines attackers often install server software which are programs that run as daemon. Using the process tree model presented in chapter 5, only the deployment of the program and the program execution can be observed. However, all the actions being performed by the program itself are not handled with the process tree model. If an attacker installs a new server and reconnects to this server in order to continue his attack, then these steps are not observed with the process tree model. To handle the case of generalized graphs (Tree-based kernels can be applied only to connected acyclic graphs), we considered the more general functions presented in [84, 85], which introduce a family of kernel functions that addresses labeled graphs. A labeled graph is a directed graph G, a number of $|G|$ vertices and a labeling function. In the graph, each vertex has a unique index between 1 and $|G|$. The label of the vertex i is $v_i \in \Sigma_v$ and the edge label from i to j is $e_{ij} \in \Sigma_E$. A kernel function $K(G,G')$ between two graphs G and G' having both vertex and edge labels is introduced in [84]. The main building block for computing the kernel functions is the random walk. A random walk is given by the hidden sequence $h = (h_1,...h_l)$, where l is the length of h and has values between 1 and $|G|$. In the first step,

h_1 is established by the initial probability distribution p_s(h). After the i-th step the next node h_i is determined by the transition probability $p_t(h_i|h_{i-1})$, the random walk may end with probability $p_q(h_{i-1})$:

$$\sum_{j=1}^{|G|} p_t(h_j|h_i) + p_q(h_{i-1}) = 1 \tag{7.23}$$

For the labeled path h the posterior probability can be described as follows, where l is the length of h.

$$p(h|G) = p_s(h_1) \prod_{i=2}^{l} p_t(h_i|h_{i-1}) p_q(h_l) \tag{7.24}$$

The joint kernel K_z can be defined from two graphs G and G', using hidden sequences h and h'. During a random walk the visited labels might be as follows, $v_{h_1} e_{h_1} v_{h_2} e_{h_2} v_{h_3} e_{h_3} \ldots$

Two kernel functions can be defined using vertex and edge labeling respectively, $K(v,v')$ and $K(e,e')$. Both kernel functions are assumed to be nonnegative: $K(v,v')$, $K(e,e')\geq 0$. An example of a vertex label kernel is the identity kernel, where δ is a function returning value 1 if an argument holds, 0 otherwise.

$$K(v, v') = \delta(v = v') \tag{7.25}$$

If the labels used are defined in $\mathbb{R}^8$, the Gaussian kernel [146] is a good candidate:

$$K(v, v') = exp(-||v - v'||^2/2\sigma^2) \tag{7.26}$$

In general, the joint kernel is computed by the product of the label kernels. If the lengths of the two hidden sequences are equal that is $l=l'$, the following equation can be applied:

$$K_z(z, z') = K(v_{h1}, v'_{h'_1}) \prod_{i=2}^{l} K(e_{h_{i-1}h'_i}, e'_{h'_{i-1}h'_i}) \times K(v_{h_i}, v'_{h'_1}) \tag{7.27}$$

If the lengths of the sequences are not equal ($l\neq l'$), then $K_z(z,z')=0$.

7.5.2 The Process Tree Model

Process execution commands on a host can be monitored and processed. The results of the monitoring process are process trees. A process tree, formally defined in chapter 5, is redefined as an ordered pair $T = (V, E)$, where $V=(p_1,...,p_n)$ describes the set of nodes and $E=(t_1,...,t_j)$ represents the set of labeled edges. We distinguish between three different node types: *PID*, *Process name* and *System call*.

- A *PID* node refers to a process identifier in a Linux operating system
- A *Process name (pn)* refers to a the process name transmitted as argument to the system call `sys_execve`
- The *System call (sc)* node refers to a system call number implemented in the Linux kernel

In order to compute the tree-based kernel function afterwards, the types of nodes have to be of the same type. Nodes that have different types will have a similarity of 0. Further, we distinguish between three types of edges, *PID-to-PID*, *PID-to-pn* and *PID-to-sc*.

- The *PID-to-PID* edge refers to one process that has created another and is labeled with a time difference between the two nodes.

[8] $\mathbb{R}$ is the set of real numbers.

- The *PID-to-pn* represents a process has started a another program and is also labeled with the time difference.
- The last type of edge, *PID-to-sc*, describes how many times the given system call is executed, and is labeled with the number of executions. Such a link models the system calls (and their call pattern) executed by a process.

A main advantage of this proposed model is that cycles in the data structure can be avoided, even if the number of different nodes becomes very large.

An example of a process tree can be seen in figure 7.12. Its root is the initial process identifier (PID) 534, the privileged separated process [129] of *sshd*. Process 534 has created two other processes, process 1038 after three seconds and process 1031 after zero seconds. Process 1038 immediately executes the command *uname*. Process 1038 has also made a `sys_write` (4), `sys_execve` (11) and two `sys_read` (3) system calls, represented by boxes in the figure 7.12. On the other tree branch, process 1031 immediately executes the *bash* command, then creates process 1041 after a delay of eight seconds. This new process executes *wget* at once and ten seconds later creates a new process, which executes *ssh_brute* and *tar*. To avoid clutter we have not shown all system calls in the figure 7.12.

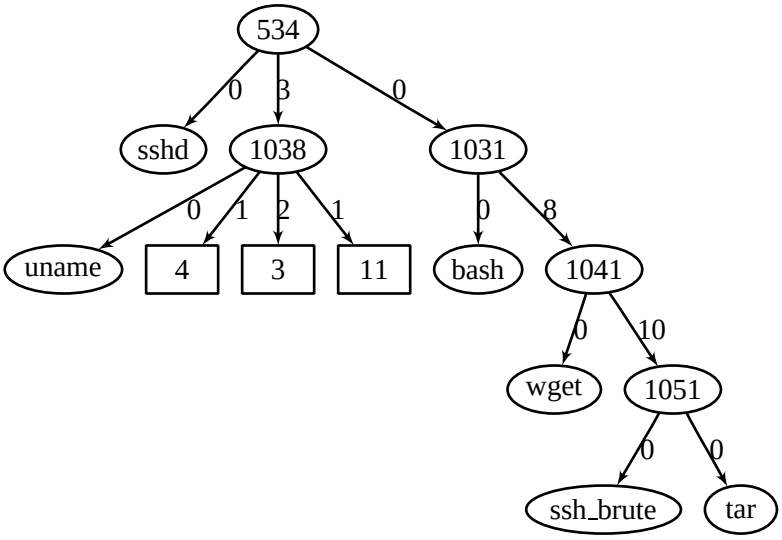


Figure 7.12: Example of a Process Tree

7.5.3 The Process Graph Model

The second concept is based on a labeled graph. A graph G can be described as a pair $T = (V, E)$, where V=(p_1(*name_proc*, *stat_d*),. . . , p_i(*name_proc*, *stat_d*)).

The set of nodes p is composed of two parameters, the process name parameter *name_proc* and the parameter for the corresponding probability distribution for system calls, *stat_d*, where *stat_d* = (x_1,...,x_n) is a set of distinct variables. *Stat_d* is another identifier for processes [187]. The inter-nodal differences, the distance between the sets of the probability distribution of the system calls of two nodes, can be established. This takes into account variants of the same malicious software or the cases where a malicious program tries to mimic already installed software. For instance, an attacker uses a bot *b*, adds customized features, and gets bot b_1. If another attacker uses the same bot *b*, extends it with other minor features and gets bot b_2, then the distance between bot b_1 and b_2 is probably small. However, if a malicious program tries to mimic another program, for example by executing out of a file named `sshd`, the distance between the legacy `sshd` and the fake `sshd` is likely to be quite large.

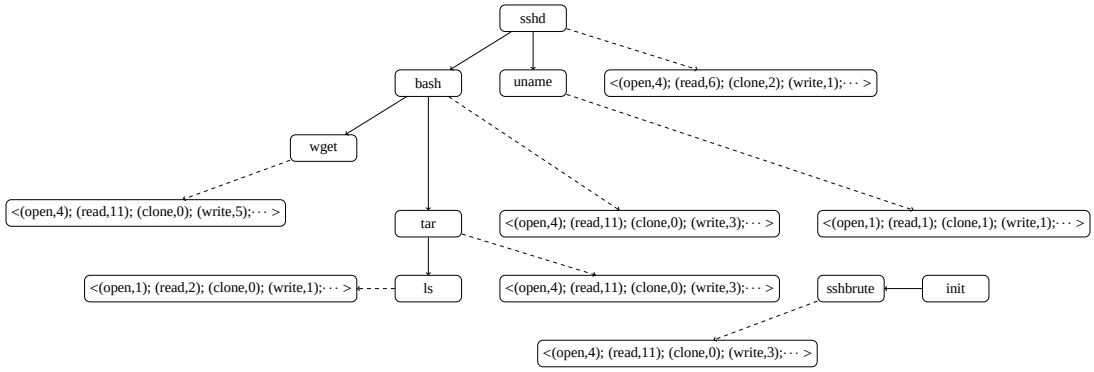


Figure 7.13: Example of a Graph Model

An example of a graph can be seen in figure 7.13. The privileged separated process `sshd` executes the processes `bash` and `uname`. Next, the program `wget` and `tar` are executed by `bash`. After the execution of `tar`, the command `ls` is executed, followed by the attacker tool `ssh_brute` which is then daemonized[9]. For the clarity, pseudo-edges, represented by dotted lines, are introduced to associate the probability distribution of system calls to each node.

In this section, published in [185], we have presented an extension to our process tree model initially introduced in chapter 5. We base our approach on supervised classification methods that leverage native graph/tree kernels. We propose two applicable kernel families: the first uses tree kernels and thus is easier to implement and has a better operational performance. The second family uses more general graph kernels. Although, in the latter case, the complexity of the method (algorithmic running times) increases, richer modeling capabilities are very promising. Our future work will address several remaining issues. We will look at further graph kernels and assess their applicability, especially with more complex and richer graph-related information, but we also planned to evaluate our solutions against earlier ones and to study interactions between multiple concurrent malicious programs. In the context of this PhD thesis we aimed with this contribution to explore potential enhancements of our process tree model. However, in the remaining we use the simplified version of the process tree model presented in chapter 5.

7.6 Implementation of Adaptive Honeypots

Various, adaption mechanisms are described in chapter 6. However, implementing a new honeypot from scratch for each of these is a tedious and error-prone task. Typically such implementations must be written in the C programming language, often required code change in kernel space. To overcome these difficulties, we have created a generic adaptation framework that provides building blocks for adaptive honeypots having the following requirements:

Kernel control Total control of the kernel is needed, in order to take decisions on allowing or blocking `sys_execve` system calls. If an attacker with sufficient privilege has compromise a machine she could potentially take control over the kernel – a worst-case scenario for the honeypot operator –.

Process monitoring Process execution must be tracked, so as to interfere only with system calls that are related to attackers. Preventing `sys_execve` calls related to the operating system may render the system unstable.

Opaque decisions An attacker should never be able to use measurements of process execution time to predict the decisions made by the honeypot. The honeypot should be reactive at all times even when decision

[9]The parent process of `ssh_brute` becomes the process `init`.

making involves computational overhead. Honeypot functionality should not introduce code paths that are overly complex and non-interruptible, as these may impact the stability of the entire operating system [97].

Flexible Many different games between attackers and high-interaction honeypots can be imagined. A system call can be blocked by returning different answers or error codes to an attacker. In the Linux kernel, version, 33 different error codes can be returned, for instance, permission denied, input output error, out of memory etc. Our games could be extended with a strategy to select appropriate return codes each time an attacker-related action is blocked. Therefore, it is preferable to have a flexible architecture which allows other games to be implemented without large development effort.

7.6.1 Adaptive Honeypot - Framework

An overview of our adaptive honeypot framework is shown in figure 7.14. We focus on a high-interaction honeypot exposing a vulnerable SSH service operated in a customized User Mode Linux (UML) [191]. Attackers are constantly scanning the Internet for new victims (step 0 in the figure). Once they have discovered the service they start launching a brute force attacks against the server in the hope of compromising a user-account. Nicomette et al. [112] figured out that specific attacker communities specialize in this task. The compromised accounts are then shared among other attacker communities, which can then get, a shell on User Mode Linux and can make their attack. In the background the SSH server on the User Mode Linux clones a privilege separated process [129] and its information is put into an output queue by the modified UML kernel (step 2). The AHA daemon, AHAD, fetches this message. For each message, the AHA daemon determines whether the process belongs to the system itself or to an attacker. If it belongs to system it is allowed by default. However, if it belongs to an attacker, AHAD passes the message to its intelligence routine for a decision. AHAD puts its decision as a reply message into an input queue (step 4). In the meantime, the UML kernel waits for τ milliseconds for a decision. After the pause, it polls for the reply message from the input queue (5). If it receives the decision within the predefined time frame τ, it implements the decision, if not the error `ENOMEM` is returned suggesting to the user that the system has run out of memory.

Execution performance is critical for AHAD and UML kernel. Therefore, a minimalistic design is used for AHAD and for the additional UML kernel functions. They should only fetch a message from the complementary subsystem and make or implement a decision. Messages only temporarily stay in the queues. The AHA_worker program takes messages from the queues after a fixed time interval and writes them in a log file. This program also cleans the queues. The Aha_eye program takes the raw reassembled messages[10] and generates a report which allows a honeypot operator to observe what is happening.

A generic adaptation mechanism is defined in chapter 5 which is summarized as:

allow The usual control flow of sys_execve is used.

block An error code is returned.

substitute The program arguments of the sys_execve system call are changed and the regular control flow is used.

insult The program arguments of the sys_execve system call are exchanged with an insulting program.

The modified UML exchanges messages with the AHA daemon. For each `sys_execve`, `sys_clone` and `sys_exit` system call, UML puts an export message into the output queue. Two directories presented the input and output queue. Each exported message has a simple key-value format, the fields are separated by the "=" character. The reason for using this format is that many fields have a variable length, potentially requiring additional memory management code in the Linux kernel. The alternative of copying everything to the stack is

[10] An example of a message is shown in the appendix in section D.3.

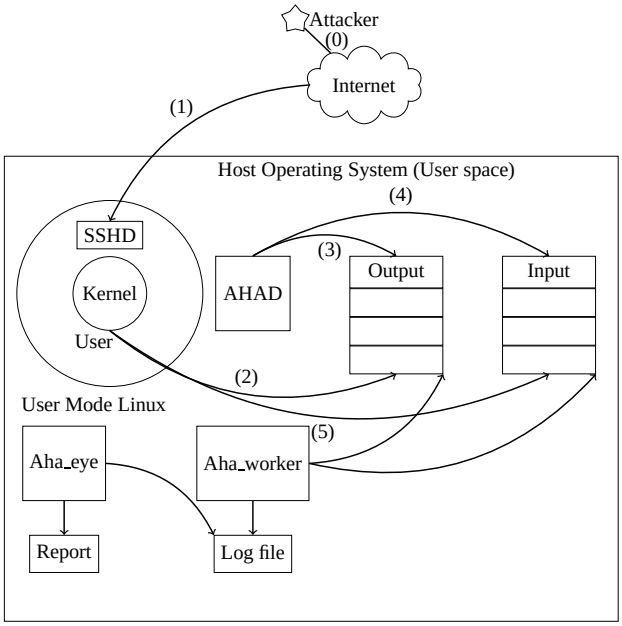


Figure 7.14: AHA - Architecture

not a viable technique, because stack length in kernel space is limited. Simply allocating memory with kmalloc is also tricky because page faults may occur during the creation of concurrent messages, possibly resulting in deadlocks. In order to correctly associate the decisions to the requests, a unique message is generated containing a unique filename in the output directory. The uniqueness of the file name avoids concurrency issues. Each message has a *t*ype key which is used to determine the system call created the message. A value of 1 indicates that `sys_execve` created the message, 2 indicates a `sys_clone` message and 3 indicates a `sys_exit` message. An export message has a variable length, which depends on the length of the file name of the program that is executed, the total length of the environment variables and the length of the process identifiers. There is a risk that the AHA daemon fetches incomplete messages. Therefore, the *d*one key is used to mark the end of the message. When the AHA daemon sees this key it knows that the message is complete. In appendix section D.3 we show a message created by the `sys_execve` system call. Besides the *type* and *done* keys, it contains a *file* key which corresponds to the absolute filename of the program that is scheduled to be executed. A program always has a command line argument: even if the user gives none, the system provides the program name as the first [97]. An *a*rgument key is set for each command line argument. The value for this key corresponds to the command line argument. A similar technique is used for the environment variables: each environment variable is identified with an *env* key denoted. Environment variables are particularly useful for identifying attackers. For instance, we can see what kind of terminal the attacker is using, the attacker's locale settings, and the source IP address and port. Messages resulting from `sys_clone` and `sys_exit` include only process identifier information. The process identifiers, represented by the keys *pid*, *ppid* and *rppid*, are essential in determining whether the program execution belongs to an attacker or the system itself.

The reply message, shown in figure 7.15 is composed of three integers. A non-zero value in any field means the corresponding action should be implemented. A value in the exit field represents the error that should be returned to the user. Examples of such errors are `EPERM=1` (Operation not permitted) and `ENOSPC=28` (No space left on the device). A complete error code list is part of the default kernel development tree (`include/asm-generic/errno-base.h`). If the field of insult is non-zero, it is passed as an argument to the `/sbin/insult` program installed on the honeypot. The argument is used to index an array of insults in the `/sbin/insult` program, which sends the corresponding message to an attacker's terminal. A non-zero value for the field substitute is used by the honeypot to index a hard-coded table of filenames for executable programs

```
1 struct ReplyMessage{
2     int block;
3     int substitute;
4     int insult;
5 };
```

Figure 7.15: Reply Message Structure

File	Function
arch/um/kernel/exec.c	sys_execve
arch/um/kernel/process.c	exit_thread
arch/um/sys-i386/syscalls.c	sys_clone
os-Linux/main.c	_init

Table 7.6: Modified Kernel Files

that can be run in place of that intended by the attacker.

When an export message is created, a unique message identifier generated in the user mode kernel is attached to it. This identifier is needed to correctly correlate match requests with decisions. To generate such a unique identifier in the UML kernel without adding large dependencies on other libraries, we exploited the traditional x86 architecture. With the RDTSC instruction in an inline assembly routine, we queried the time stamp counter on the CPU processor. The value of this counter changes with every instruction. Hence, these numbers are unique for code executed on a particular core of the CPU. The 64 bit value returned by RDTSC is converted to a hexadecimal string which is appended to the output directory path to create a full pathname.

7.6.2 Component Description

UML

Our modified UML can be built with a default UML configuration. We added a new header and C files in the directories, `arch/um/include/` and `arch/um/kernel` to the Linux kernel development tree, containing the adoption functions. Obviously, small changes also had to be made to the Linux kernel. The `sys_execve` wrapper for UML is hooked to export the executed program arguments, along with associated process identifier and environment variables. Besides these additional monitoring capabilities, the kernel polls replies from the AHA daemon and implements its decisions. The files that have been modified are presented in table 7.6. In addition to these files, we added a configuration directory containing configuration files which each contain a parameter. The reason for using a number of files instead of a single configuration file is that domain specific files can be parsed by few lines of C language code. In `main.c` the polling interval τ,which corresponds to the waiting time for a decision from the AHA daemon, is read from the configuration file `pollinterval`. A hook for the `sys_execve` wrapper has been added to the file `arch/um/kernel/exec.c`: the relevant section is shown in figure D.2 in the appendix D.3. This function puts a message in the output queue along with a unique message identifier which is used to ensure that decisions are correctly applied, even when several system calls are executed concurrently. The function then polls for a reply encapsulated in the flat C structure, shared between kernel and daemon, shown in figure 7.15.

Adaptive Honeypot Alternative Daemon

AHAD uses a library, AHAlib, containing the attacker monitoring and adoption mechanisms. The AHA daemon tracks the process hierarchy by following the process tree structure [104]. The relevant information is provided by the modified UML kernel. If the UML would confuse a decision for a system call initiated by the system itself with a decision for a system call related to an attacker may have dramatic consequences such as failures of the system. When an attacker connects to the honeypot, the SSH server clones a privilege separated process handling the attacker's connection [129]. The UML kernel notifies the AHA daemon about all the cloned processes and executed programs. If it sees that the program `/usr/sbin/sshd` clones a process, it knows that a new attacker has connected to the honeypot. The process id (PID) is then recorded in a list L containing all the users connected to the honeypot. This list contains the roots of the process subtrees belonging to SSH connections related to attackers. The AHA daemon creates a reply message stating that the system call should be allowed or blocked, substituted and whether an insult should be triggered. For each `sys_clone` or `sys_execve` message, the AHA daemon recursively looks up the corresponding process identifier in the recorded process tree. If a parent process identifier matches an element in the list L, the daemon knows that the process belongs to an attacker. If it does not match, the process belongs to the system itself and is allowed by default.

After having decided whether a process belongs to the system or to an attacker, the AHA daemon is capable to adopt itself to the attacker. This is the core of our research activities. Reimplementing a honeypot for each learning solution is not a very productive approach. Hence, we propose a generic framework containing the implementation of common features, like monitoring attackers, and differentiating between attackers and the system itself. The AHA daemon uses a library called AHAlib, which is a collection of four classes.

AHAActions This class contains all the necessary functions for communicating with the modified UML. It contains a functions to load and parse a message from the modified UML kernel. It includes functions to load, parse, and create reply messages.

KERNEL_ERRORS This class contains the errors that can be returned to an attacker's process. The error codes are taken from the default Linux development tree and are defined in the file `include/asm-generic/errno-base.h`.

ReplyMessage The purpose of this python class is to write a plain C structure. Obviously, the fields of the structure must be identical with the C structure included in the modified UML kernel (see figure 7.15), otherwise the decisions will be wrongly interpreted.

ProcessTrees The responsibility of this class is to track process identifiers in order to determine if an executed program belongs to an attacker or to the system itself. Each time the SSH server clones a process, the PID is put in a user list in a ProcessTree instance. A process dictionary is also included. This dictionary is keyed by a process identifier and each entry has an element that points to a parent process identifier. This information is augmented by an annotated process dictionary, which process identifiers serve as keys pointing to a third dictionary. This embedded dictionary contains the message creation time stamp and attacker SSH connection information including her source address. The most important function in this class is called `searchTree(pid,ppid)` which takes a process identifier and a parent process identifier as input and searches in the process tree for a match. If a parent process identifier cannot be found in the process tree, then it is assumed that the process belongs to the system itself and the value `False` is returned. However, if a match is found, the value `True` is returned, meaning that the parent process identifier belongs to an attacker. In this case, the current process identifier is also added to the process tree. The process look up is done recursively. In order to avoid an ever-growing process tree, this class also has a method to remove processes from both the process tree and the annotated process tree using information provided by the UML kernel instrumented `thread_exit` function.

Aha_Worker

Execution time is an essential consideration in the operation of an adaptive honeypot. Each system call needs to be acknowledged by the AHA daemon. If the system calls are suspended for a too long time, the system may become unstable and unusable. Hence, the AHA daemon should focus only on speedy decision making. This means that the queues can quickly fill up. Each message exchange results in creation of two files on the UML host file system. In order to avoid the accumulation of large number of files in a directory, the program AHA_worker daemon periodically checks the queue, selects outdated messages, merges them in a log file and removes them from the queues. Normally a message initiated by the UML should be acknowledged within 50ms. When a file is created in a queue, the file creation time is recorded. Each file that is older than 1 minute is merged in the log file and removed from the queue.

Aha_Eye

Attacker activity monitoring is an essential task during the operation of a high-interaction honeypot. Aha_Eye takes the merged exchanged messages as input and generates a report describing the sequences of program executions made by each attacker. In order to determine such a sequence, the process trees are inspected and recovered by Aha_Eye using AHAlib. Each time the SSH server clones a privilege separated process, p_0, an attacker has connected. All the programs executed during an SSH session belong to the process tree of an attacker, and so have the process p_0 as root. The sequences of executed programs from such a tree must to be recovered with the help of relative timestamps taken from the messages generated by the modified User Mode Linux. For a process tree of an attacker, that is a subtree of the overall process tree, the privilege separated process is remembered and considered as the root of the process tree related to an attacker. Each branch from the root to each leaf is inspected and each edge contains the time difference between these nodes. For each node on a given branch, the sum of the time differences is computed serving as index in the process execution sequence. An illustrative example is shown in figure 7.16. The SSH server cloned a privilege separated process, with the process identifier 4121. After 1 second this process clones another process with the identifier 4127. This process then executes the program `bash` with the program argument `bash` (first program argument set by default). This means that for this branch (marked in the left part of the figure) the sum of time differences is 1 + 1 + 0 = 2. After 2 seconds the processes with the identifier 4121 cloned another process with the identifier 4129 which executes after 3 seconds the program `bash`. For this branch (marked in the right part of the figure) the sum of time differences is 0 + 2 + 3 + 0 = 5. Due to the fact that 5 > 2 we know that the command `bash` was executed prior the program `uname`. The SSH server also exports SSH environment variables to its child processes which include the source IP address of the attacker, the source port and the used terminal.

7.7 Conclusions

The best practices [154] for setting up high-interaction honeypots advise that all the network traffic related to the honeypot should be recorded, for instance, by creating a network tap on the interface to which the honeypot is connected. Capturing all network traffic results in a large volume of data. In order to facilitate the monitoring operation, we propose two network traffic visualization techniques. Our PeekKernelFlows approach, of [184] gives an aggregated overview of overall network traffic based on Aguri. Aguri aggregates network traffic, resulting in aggregation profiles. We suggest a visualization approach of these profiles. A more detailed attack distinction is made with our polar representation. Assuming that an attacker on the honeypot could not take over the network switch and destroy the tap, the attacker could not tamper with the network traces. Even when an attacker cannot remove traces, we identified techniques she could use to obfuscate her traces. Her raw packets are still in the trace, but showing she could trigger faulty behavior of commonly-used network tools. We give a proof of concept in [179] showing how an attacker can hide a stream and how she could terminate the network analysis by force with our PCAP bomb. The essential problem we described in [178] can be reduced to an

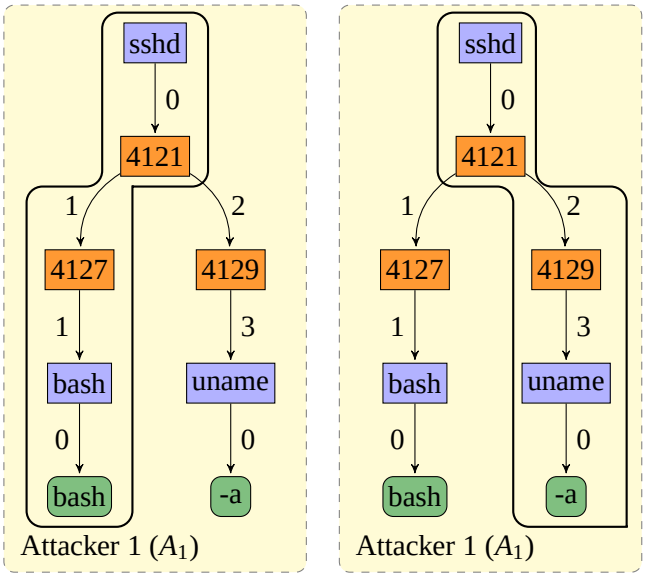


Figure 7.16: Recovering Attacker Sessions

imprecise definition of network flows that can be exploited by an attacker. Therefore, before performing the network analysis, the network traces must be preprocessed in order to eliminate such collisions.

Because a honeypot may not immediately recognize the significance of all the attacker related software which is often malware, our approach published in [185] suggests an in vivo malware analysis approach, where profiles of system calls are generated during an active attack. The proposed profiles can also be used to harden our automaton against state tampering made by an attacker.

Like a traditional high-interaction honeypot an adaptive high-interaction honeypot offers a complete operating system to attackers. However, an adaptive honeypot has also to interfere with commands run by an attacker. User Mode Linux runs as non-privileged processes on the host operating system and provides useful features for interacting with the host operating system and allows a variety of functions to quickly implement adaptation behaviors. As we showed in [180] an attacker can perform a forced take down of the system without any extra privileges. Hence, the mmap system call has to be patched in order to close the discovered vulnerability. The vulnerability could also be mitigated using Apparmor or SELinux.

Using these mitigation techniques, we propose an implementation of an adaptive honeypot framework in order to integrate a variety of learning algorithms into high-interaction honeypots without reimplementing the honeypot from scratch each time. Each system call related to the process execution of an attacker has to be acknowledged by a decision-making routine that implements the learning approach. The design and source code was published in [176] and can be checked out from our publicly available repository [174] which is an augmented clone of the Linux kernel git repository.

7.8 Limitations

Currently a few challenges needs to be addressed in the adaptive honeypot framework. First of all, an attacker could explicitly look for artifacts specific to the adaptive honeypot. Obviously, an adaptive honeypot is slower than a regular high-interaction honeypot, due to the necessary interprocess communication and decision-making. However, it is difficult for an attacker to make a timing attack on the decision-making routine of the

honeypot. This is mainly due to the constant polling interval, which is set to 50ms in the modified UML. An attacker could also look for the program `/sbin/insult`, because this tool is not installed on regular systems. An attacker could even remove this program. Additional work is needed to hide and protect this program by using traditional rootkit techniques [101]. An attacker could also do an indirect attack on the honeypot. She could first push an attack script to the honeypot and install a `cron` job that executes the script. An adaptive honeypot then only sees and interferes with the deployment of the script. When `crond` executes the script, the adaptive honeypot framework classifies its activity to system process executions and allows them by default. In order to tackle this problem, file system changes also need to be tracked. We could complement our approach by modifying the `tty_read` and `tty_write` functions to assess attackers' typing capabilities, for instance whether they use backspaces or cursor keys. Other interprocess communication techniques, like shared memory segments, named pipes or socket can be explored in order to optimize execution performance and avoid kernel deadlocks. Other protection mechanisms for User Mode Linux should also be explored to reduce the possibility of interference with the kernel. We could also adapt a virtual hardware layer of virtual machines to communicate with the adaptive honeypot daemon. However, this requires a larger development effort than modifying a Linux kernel and, as such, was out of scope in providing a proof of concept for adaptive honeypots.

Chapter 8

Experimental Evaluations

In this chapter are described the experiments related to adaptive high - interaction honeypots. We setup state-of-the art high-interaction honeypot and a low-interaction honeypot to recover traces from attackers. The traces from the high-interaction honeypots are used to to compute the optimal strategy profiles for attackers and the honeypot. In a next step, we evaluate adaptive honeypots driven by reinforcement learning and fast concurrent learning which have a higher adaptation degree than the adaptive honeypot which optimal behavior is defined through simulations.

8.1 Recovering High-Interaction Honeypot Traces

A high-interaction honeypot capable of recording `sys_execve` and `clone` system calls has been setup to induce our hierarchical probabilistic automaton described in chapter 5. The automaton instance serves as ground truth for the computation of Nash equilibria. The Linux kernel was directly patched in order to avoid detection by address arithmetic which is an attack described by McCarty [101]. The data collected in kernel space is directly transmitted to the hardware level in order to avoid it being accessible by an attacker. The honeypot is operated with the Qemu x86 emulator [18]. The kernel inside Qemu was modified such that process ids are logged. The host machine stores this data in a database. The honeypot has an additional network interface which transmits the system logs to a syslog-ng server. The default running service is a SSH server which serves as an entry point for attackers. It could be configured to use the PAM module `pam_permit`. In this case no password is required and this may be very suspicious for attackers. Therefore, the `pam_unix` module was modified. The patch is described in section D.1. It is responsible for password authentication in Linux operating systems. With the patch, the system asks for a password but then neglects all non-privileged user passwords. The attacker is asked a password and each password is accepted as valid password. This implementation choice is also resistant against password changes performed by attackers [135, 138], because the password is not checked anymore. In theory an attacker could also change the PAM modules during the operation of our honeypot but this phenomenon has not been observed. At the same time, some attackers installed their own shell in order to be sure that they are not using a shell with additional monitoring features. Furthermore, other attackers replaced the SSH server on the honeypot.

From this honeypot, we recovered the process trees related to attackers which are sub trees of the Unix process tree on the honeypot. These process trees were transformed in process vectors. Each vector corresponds to an attack. From the observed process vectors a hierarchical probabilistic automaton was created to drive the simulation. Our data sets and developed software are publicly available[1].

The honeypot was operated on one public IPv4 address and consisted of a Ubuntu Linux 7.10 operating system. The Linux operating system was ran 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 attackers and to mitigate the effects of an

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

attacker changing the password of a compromised account. Moreover, we want to mitigate the fact of attackers installing compromised SSH servers as it was reported by Raynal et al. [138]. We extended the Linux kernel, version 2.6.28-rc6 with the `sys_execve` and `clone` monitoring features.

During the operation of the honeypot, 637 successful ssh logins and 12140 ssh failures were observed. Despite the patched `pam_unix` module, a high number of ssh failures was discovered. 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, 12 additional user accounts have been created. Therefore, 25 non privileged user accounts existed in total. Attackers tested 1763 non existing accounts with different passwords which is another explanation for the high number of SSH failures. For the successful logins, 183 different IP addresses were recorded. 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. Whenever, the kernel was changed, the modified kernel was replaced with the original.

637 process trees were recovered. 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 by brute force attacks against the SSH server which were performed by some attackers using automated tools. Automated tools managed to break into the honeypot but left immediately. The maximum length of a process tree is due to bots that were installed on our honeypot. The bot master had long sessions with the bot in order to operate it. Due to data processing capabilities trees of length longer than 100 nodes, were not processed. The average number of nodes per process tree is 105 with a standard deviation of 231.

Each process tree was converted into a process vector aiming to extract the program sequences performed 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.

The hierarchical probabilistic automaton was set up using the recovered 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, the first command line argument (which corresponds to the program name in a Linux operating system) was removed. 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` [111]. 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 simulation 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. Figure 8.1 shows the evaluation of 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[2]). 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.

8.2 Recovering Low-Interaction Honeypot Traces

In order to recover traces from a low-interaction honeypot simulating a vulnerable SSH server, Hali was developed. Besides, the traces from an high-interaction honeypot, traces of a low-interaction honeypot are desired

[2]The longest process vector was removed due to data processing limitations.

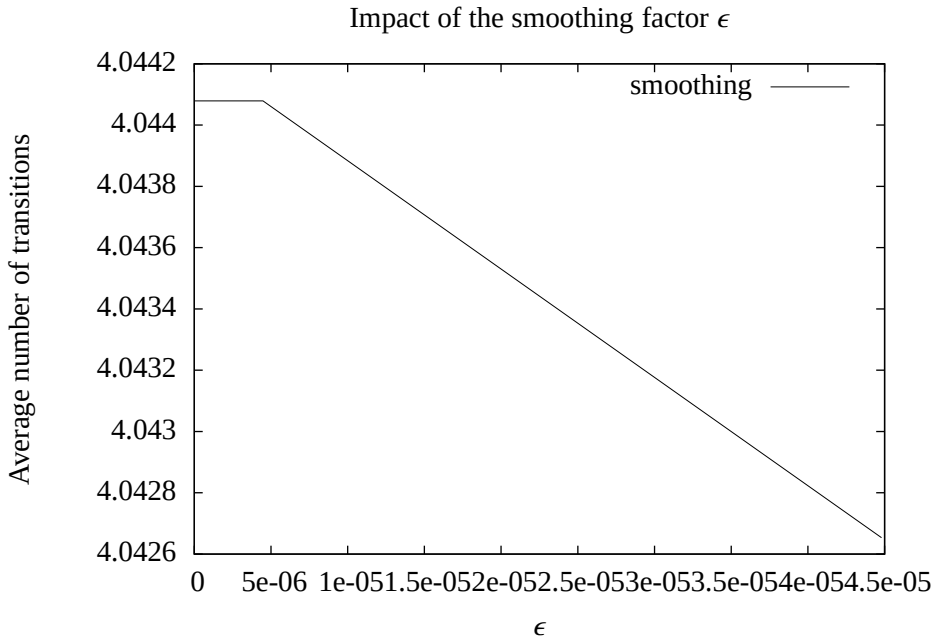


Figure 8.1: Evaluation of the Smoothing Factor ϵ

in order to compare our adaptive honeypots with a classical low-interaction and high-interaction honeypot. Hali is a pure python shell serving as low-interaction honeypot. Hali is a bluffing honeypot that reproduces attacker outputs from previously deployed honeypots. Hali is a shell implemented in python with ncurses [137] and with a memcached back-end [51]. The memcached back-end is a volatile server containing outputs from high-interaction honeypots. This means that the information is only kept in memory. Furthermore, memcached does not use a relational query language but only key-values can be stored or retried. This has as consequence that lookups are faster than on a persistent database. Furthermore, during the operation of Hali, Hali's knowledge can be extended without restarting the experiment. This is partially important because hosts with a large uptime are more interesting for attackers. An alternative implementation choice would be a shared memory segment. However, ready to use memcached clients exists that can be used by a honeypot operator to manage the database. Hali reads a command line from an attacker, queries a random pre-generated output from a memcached database. The input characters are logged to a syslog server [16]. Using this remote logging facilities, an attacker cannot interfere with existing logs. The architecture is shown in figure 8.2. Hali is foreseen to be installed in a User Mode Linux as default shell, with a patched `pam_unix` module. The User Mode Linux has two network interfaces. One with a public IP address (network interface tap0) where it is accessible via SSH, and one private (network interface tap1) which is used to export syslog messages and which is used for the memcached queries. The tool `tc` [75] is used to limit the throughput of the network interface having the public IP address. The network traffic is also filtered with an iptables [136] firewall in order to mitigate scanning activities performed by attackers. With a limited and filtered bandwidth, attacks towards third parties can be limited. These are best practices to operate honeypots [154]. An attacker connects to the UML (ssh user @ <public IP address>). Hali records the SSH environment variables via syslog. From these variables the source IP address and source port of the user is recovered which can be used to correlate network captures with SSH shell sessions. The correlation is based on the source IP address, source port, destination IP address, destination port and timestamp. Messages are sent to the syslog server that is reachable via the <management IP address>. The rationale to select two network interfaces is to avoid that syslog starts dropping messages while users are heavily probing the public available network interface. The user enters a command by typing characters. These

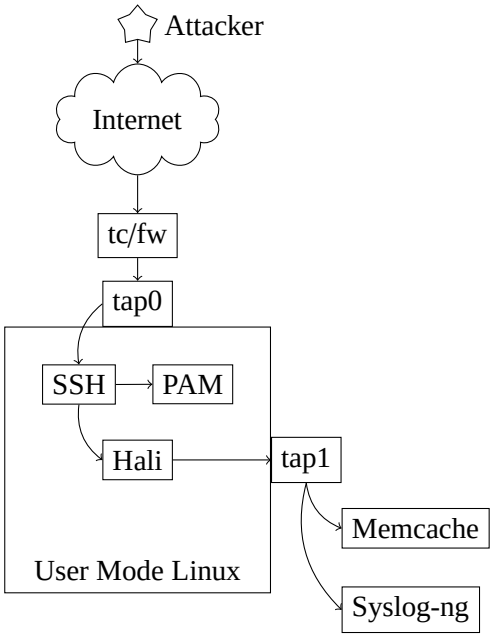


Figure 8.2: Hali Deployment Architecture

characters are also logged via syslog to the syslog server. When the user hits the key <Enter>, Hali selects an pre-generated output via memcached, where the server is also accessible from the <management IP address>. The selected output identifier is also logged via syslog. Hali is configured with a configuration file which is used by Hali and other tools (memcached) related to Hali. First of all, the User Mode Linux should be correctly configured, shown in architecture, with one public IP address and one management IP address. On the host of the User Mode Linux, GNU screen can be started. This has as advantage that a honeypot operator could remotely attach to each running process for its state inspection. In one window memcached can be started, in another one the User Mode Linux, and in anther one tcpdump to capture all the network packets. On the host machine syslog-ng can be installed, acting as syslog server. Inside the User Mode Linux syslog-ng could be configured to export messages to the host machine using the management network interface.

Hali was our second honeypot experiment and was operated from 2009-09-04 to 2009-12-19, on one public IPv4 IP address. Within this time frame 342 unique IP addresses were observed. These 342 hosts did 6087 SSH connections. The traces of Hali are not used for computing Nash equilibria but for evaluating the learning honeypots.

8.3 Computing Nash Equilibria

The hierarchical probabilistic automaton based on the data described in section 8.1 is used to simulate attacks in order to compute payoffs for each player. These payoffs are then used to determine the optimal strategy profiles by computing Nash equilibria. 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 6.1.

In a second step, we computed Nash equilibria using the game theory simulator Gambit [166]. Only mixed equilibria have been found. If we consider the first game (upper half of the table 8.1) 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 8.1), has also a mixed

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

Table 8.1: Gambit Simulation Results

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. The Nash equilibrium regarding the payoff computations R_a^t tells us that the attacker should leave the honeypot in most of the cases (60% and 80%)[3]. Playing a game respecting the payoff computation R_h^t is more attractive for attackers but less informative for the honeypot. Therefore, we decided to go for the payoff computation R_h^p. In that case, rational attackers leave in at least 30% of the cases and at most in 40% of the cases.

A first prototype of the adaptive honeypot framework (shown in chapter 7.6) was instrumented to use the optimal blocking probabilities defined in the Nash equilibria. The adaptive honeypot was deployed from 2010-01-19 until 2010-01-28 and in that short period, 31 attackers were observed. Each attacker is identified with an IP address. The polling interval τ was empirically set to 50ms. The system was still usable and the program AHAD has still enough time to react. In figure 8.3 we observe that the attackers enter more commands compared to a standard high-interaction honeypot. The comparison of the two process vector averages shows that attackers execute 3.45 commands more on the adaptive honeypot than on the regular honeypot. This is a gain of 55%. In both cases, for the initial high-interaction honeypot and the adaptive honeypot we removed the process vectors related to probes from brute force scanners just for the sake for this comparison. In case of a scan a single IP address is probing many user accounts. The original datasets remain untouched and one attacker from the 31 has been removed. The adaptive honeypot results may be premature due to the short period of operation. In addition, the experiments have been done at different dates. However, the purpose of this experiment was to see whether adaptive high-interaction honeypots work in real life and to determine whether attackers play and accept the challenge or if they just leave. A public available kernel development fork including adaptive honeypot features has been released[4] aiming to contribute to the security community.

8.4 Reinforcement Learning Driven Honeypots

In section 8.1 and 8.2, we operated a high- and a low-interaction honeypot to evaluate Heliza. Each honeypot was operated until 349 successful attacks have been observed.

In the early stages of our honeypot development, it was questionable whether attackers would react to insults. Such a reaction would be an immediate disclosure of personal information regarding attacker. Particularly interestingly, we observed 1011 insults from attackers. From a purely ethical point of view, we cannot print these insults in this chapter[5]. However, we can give some information (table 8.2) about the used language by attackers to insult Heliza. For most insults we were not able to discover the language attacker used. Some

[3] Assuming that the attacker aims at reaching the Nash equilibrium.

[4] git.quuxlabs.com

[5] The term insult may be misleading because it also includes typographic errors. As it has been defined in chapter 5 an insult is an input provided by an attacker that does neither correspond to a system program, nor to a program installed by an attacker, nor an empty command.

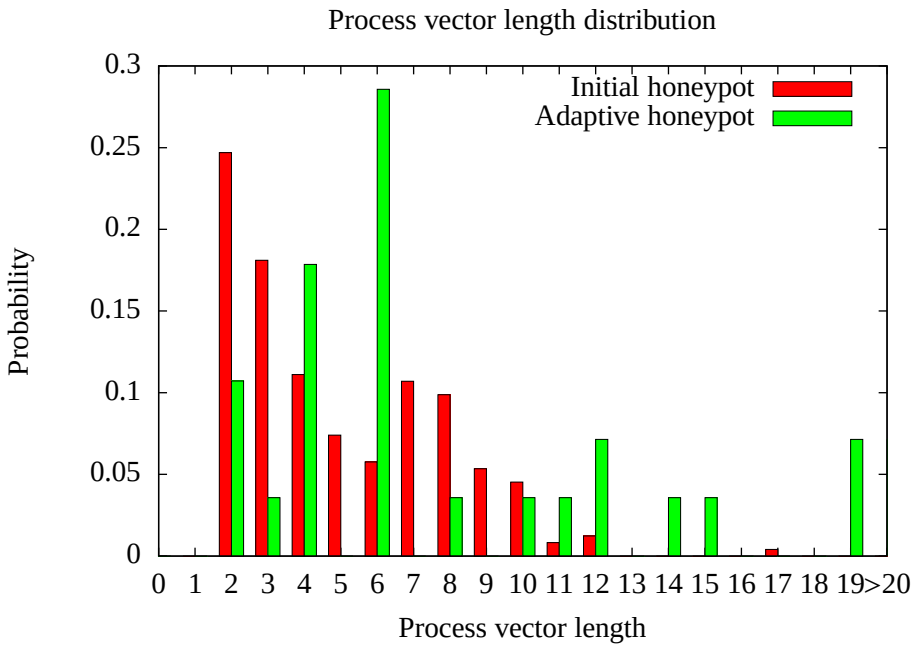


Figure 8.3: Process Vector Length Distribution

insults consisted of only one character or some random keystrokes. 17% of the insults were due to misspelled command, like the command `uanme` where we believed that the attacker wanted to type `uname`. From these attacker inputs it is highly probable that a human being was connected to Heliza, rather than an automated script assuming that most attackers test their malicious automated attacks before running them. Heliza always used the English language to insult attackers and, surprisingly, fewer than 10% of the returned insults were in English and 12% were in Romanian. Some attackers (5%) showed a sense of humor and replied with a smiley. The right part of table contains the top 10 commands entered by attackers after an insult of the Heliza. Table 8.4 shows the inputs and commands attackers have provided after they were insulted by Heliza. On the x-axis is presented the number of inputs an attacker provided and on the y-axis shows the amount of attackers. Heliza insulted 86 distinct attackers and 15% of them immediately left the honeypot. However, most of the attackers entered at least one command or an insult. After a manual investigation of the attacker input sequences, we noticed that some attackers believed that the insults are due to other attackers and not from the system itself. Even some attackers replied with the command `wall` which is used to display messages in all the terminals of the users that are connected. Some attackers just pressed enter to clean the terminal and repeated the command which explains that attackers preferred to continue the attack. Normally the attacker response time for the first insult is larger than the response time regarding another insult. After a while some attackers get annoyed and started to enter successive insults. For such a sequence of insults the delay between such successive insults is less than 2 seconds. Other attackers became curious and started to challenge Heliza in order to understand what is going on. It is worth to mention, that insults can give indication about other compromised machines. For instance, we observed some Romanian insults from German, French and Spanish IP addresses. In this case we assume that Romanian attackers have compromised these machines and used them as rebounds for attacking Heliza aiming connection laundering. The reactions of attackers regarding strategical blocks are also interesting. On average an attacker retries a command one time and there was an attacker who retried a command 116 times. After having done manual analysis of this attacker's traces we assume that this attacker tried to challenge Heliza in order to determine how the decisions are taken. The reaction of attackers namely if attackers continue their attack or if they get annoyed, their persistence facing resistance permit to draw a profile of attackers.

Command	Frequency	Language	Proportion
exit	15.77	Undefined	49.1%
ls	11.16	Typographic errors	17.1%
cd	9.95	Romanian	11.8%
uname	5.82	English	9.2%
ps	5.82	Smiley	5.3%
last	5.09	Slovak	5.3%
wget	4.61	Croatian	1.0%
id	4.36	Polish	1.0%
w	4.36	German	0.2%
others	33.06		

Table 8.2: Attacker Insult Analysis

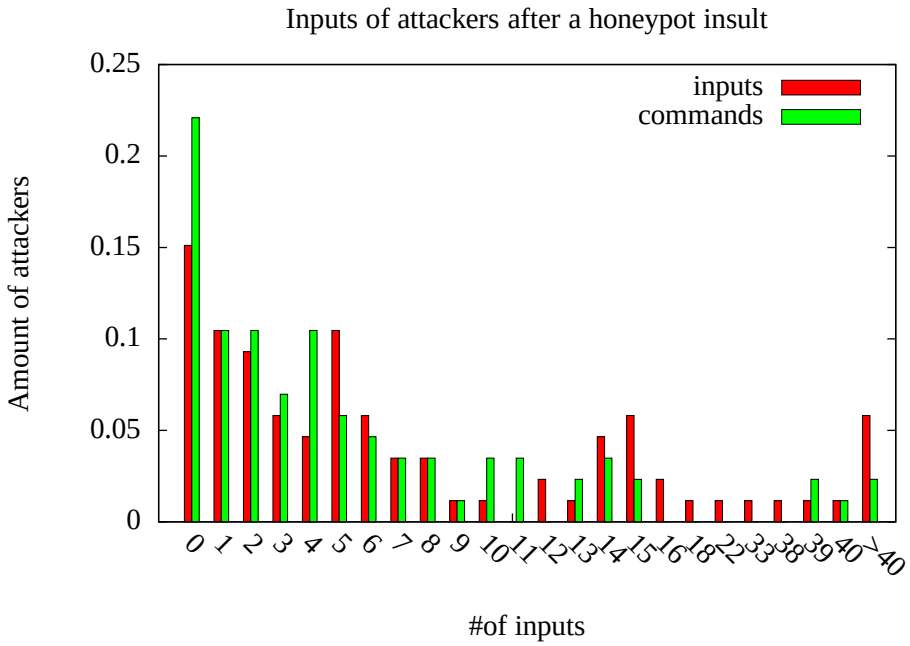


Figure 8.4: Inputs Entered by Attackers after an Insult

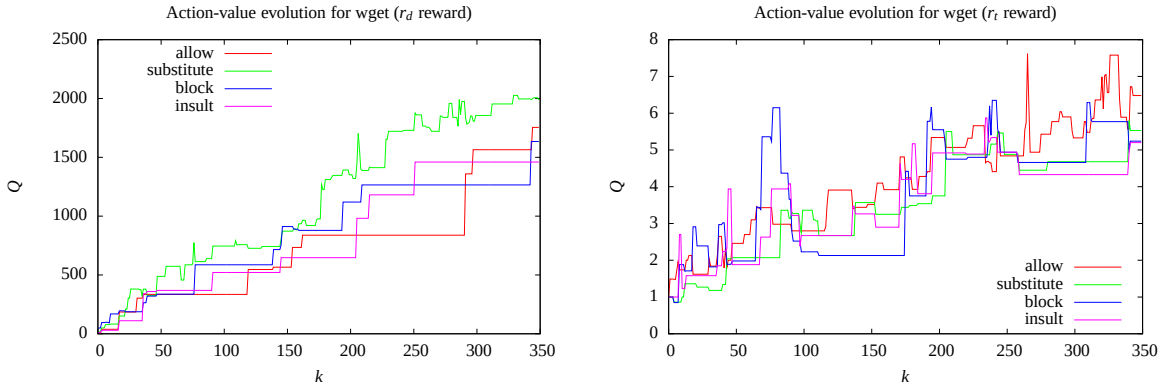
Heliza was configured with two reward functions defined eq. 6.4 and eq. 6.5. The honeypot environment Markov chain has 46 states. For space reasons, not all states can be discussed in detail in this chapter. Thus, we present only the most relevant and some general results. For the purpose of comparison and simplification each attacker connection to Heliza corresponds to an iteration k. Heliza incrementally ($k = k + 1$) computes an action value table describing the various states with the actions that provided the best rewards in the long term (estimated Q values defined in eq. 6.6). After the final attack ($k = 349$) a stripped action value table is shown in table 8.3. Generally the highest reward for a given state determines the action which should be taken for this state in the long run. This table is quite valuable for a honeypot operator who does not want to setup Heliza but rather simply wants to install static fake services. Despite following the two behaviors, some strategies for selecting an action for a given state are the same. For instance, when an attacker connects to Heliza, the action value table suggests to allow the command. An attacker who does not get a command prompt can hardly stay or install custom tools on Heliza. Some commands are frequently used by attackers to explore the compromised system. Heliza has decided to block the command `last` such that the attackers cannot detect other attackers on the system. The program `sudo` is a convenient way to get more user privileges and is often used for attacker maintenance work on Heliza. When Heliza insults an attacker, the attacker needs to investigate the situation, so spending more time on the system. The attacker needs to determine whether the system itself initiated the insult (i.e. provocative error messages configured by system administrators) or if the insult is due from other attackers concurrently connected to the system. However, if Heliza wants to collect tools, this command should be allowed, because it is often used for installing software on the system. If the purpose of Heliza is to collect attacker information, the command `wget` should be allowed[6]. However, if Heliza aims to waste an attacker's time, a forged output should be returned. Attackers usually download their tools as tarballs. Obviously, when transitions favoring custom-installed tools are desired, this transition should be allowed. If the purpose is to detain an attacker, the command *tar* should lie, such that the attacker needs to understand why the required tool is not working. From an implementation point of view the filename passed to the command tar is substituted with another filename.

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

Table 8.3: Final Action Values

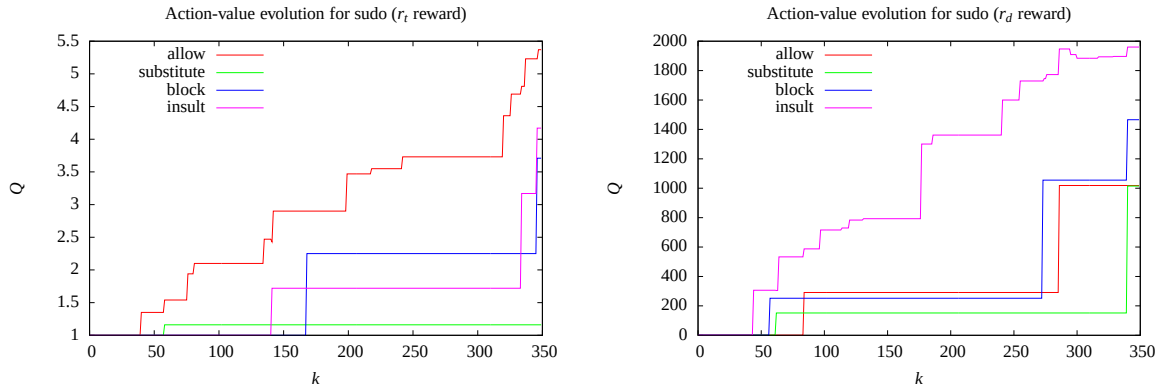
It has been proved formally that the SARSA always converges if each state is visited an infinity of times and if a greedy learning policy is used [152]. In practice, this means that we need to assess how many attackers are needed to compromise the system in order to have meaningful action value functions. We studied some relevant bash commands, `wget`, `sudo`. The results are presented in figure 8.5(a), 8.5(b), 8.6(a) and 8.6(b). The graphs 8.5(a) and 8.5(b) show estimated rewards for `wget`, and the graphs 8.6(a) and 8.6(b) for `sudo`. In the graphs 8.5(b) and 8.6(a), Heliza is configured to collect information; in the graphs 8.5(a) and 8.6(b), to waste

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



(a) r_d Reward (b) r_t Reward

Figure 8.5: Action-value Evolution for the State wget



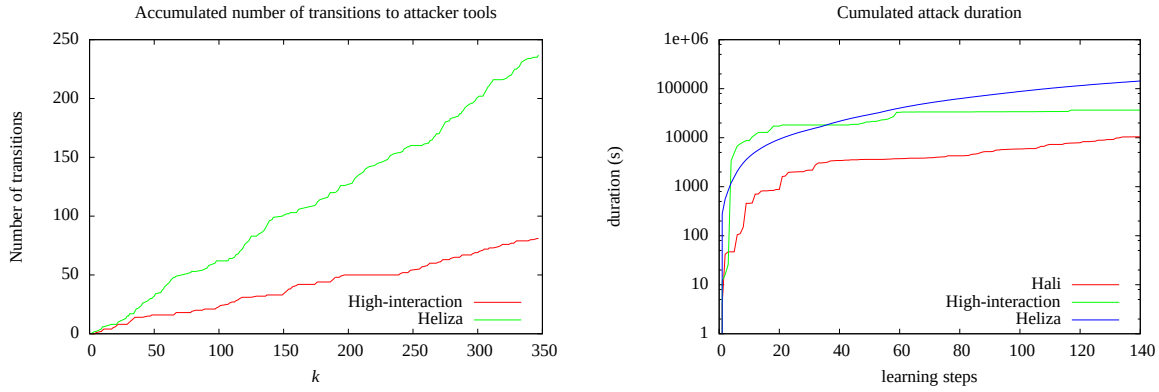
(a) r_t Reward (b) r_d Reward

Figure 8.6: Action-value Evolution for State sudo

attacker's time. Examining figure 8.5(b), we see that, by iteration 340, Heliza has learned that allowing `wget` is the best strategy for collecting information; in contrast to keep the attacker on time as long as possible, the graph 8.5(a) shows that substitution is identified as best by the 50th iteration. Similarly, Heliza learned after the 40th iteration that the execution of the `sudo` should be allowed when the purpose is to collect attacker related tools (figure 8.6(a)). In figure 8.6(b), Heliza learned at iteration 44 that attackers should be insulted for keeping them busy.

8.5 Honeypot Comparison

In this section we evaluate the performance of Heliza by comparing it with a standard high-interaction honeypot and a low-interaction honeypot called Hali which emulates behaviors for all the commands executed by attackers. Hali is a fake shell and was developed by ourselves (see section 8.2). The output of each command is forged. By default, the standard high-interaction honeypot allows all executions of programs; program executions are neither blocked nor substituted nor are attackers insulted. The purpose of this comparison is to determine whether Heliza reveals more information from attackers than typical low- or high-interaction honeypots. Figure 8.7(a) shows that attackers make more transitions to custom installed programs on Heliza than on a regular high-interaction honeypot when Heliza is configured to collect attacker-related information. The x-axis shows the iteration number k, on the y-axis is the cumulative number of transitions to custom installed



(a) Attacker Related-Information Collection

(b) Attack Duration

Figure 8.7: Honeypot Comparisons

commands by attackers is shown. The order of attacks may differ among the different honeypots. Alata et al. [3] reported that attacker launch automated attacks against a honeypot in a first step and come later back to perform the real attack. During the operation of a honeypot these two kinds of phenomena might be mixed. Hence, the cumulative number of transitions is considered in order to make the comparison more robust taking into account an equal set of attacks for each honeypot. For approximately the first 25 attackers Heliza has the same performance than a standard high - interaction honeypot in terms of transitions to attacker related programs. However after 347 successful attacks, Heliza provides an increase of 3 times in transitions to attacker related commands. This increase is partially due to blocked attacker-related programs. However, an analysis of the actions taken by Heliza for transitions to the execution of custom programs shows that 20% of the attacker-related programs have been blocked. These programs have been substituted in 26% of the cases. Heliza allowed transitions to these programs 21%. Finally, Heliza decided in 32% of the cases to insult an attacker when such a transition is made. The reason for not showing the comparison with Hali is that no installations of custom tools has been observed for the first 350 attackers. Obviously, if every command is forged, the installation process for attacker-tools fails. The figures 8.7(b) show the comparison results of Heliza with Hali and a standard high-interaction honeypot are shown in terms of attack duration (eq. 6.5). On Hali, attackers cannot install tools. Usually they try for a while and then give up. At first glance ($k < 30$), the standard high-interaction honeypot performs better than Heliza in terms of keeping the attacker busy. These short sequences of commands are often entered by automated scripts. If Heliza interferes with them they often fail. At the beginning of operation Heliza may take wrong decisions inducing attackers to leave. However, after approximately the 30th iteration Heliza keeps attackers longer than the standard high-interaction honeypot.

From these experiments we recovered honeypot traces and stored them in an SQLite database [117] which is freely available [175]. A honeypot trace is a chain of inputs provided by attackers and is composed of the following elements:

uid is a unique identifier which distinguishes different attacker sessions. An attacker session starts when the SSH server clones a privileged separated process [129] and ends when this process dies, from which it can be deduced that the attacker left the honeypot.

id is a numerical strictly monotonically increasing identifier that identifies the input that an attacker gives. This identifier is essential to establish the order of the inputs that an attacker has provided. Further details about recovering attacker sessions regarding concurrency issues between attackers and the system itself are presented in section 5.1.1.

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

General Statistics		Country Code	Proportion
Number of attacker sessions	349	RO	47%
Minimal attack duration	0s	DE	16%
Maximal attack duration	5849s	ES	10%
Average attack duration	162s	Unknown	4%
Stdev of attack duration	509s	LU	4%
Proportion of allow actions	31%	IT	4%
Proportion of block actions	22%	MK	4%
Proportion of substitute actions	30%	LB	3%
Proportion of insult actions	17%	NL	2%
		GB	1%
		BE	1%
		US	1%
		FI	1%
		AT	1%
		FR	1%

Table 8.4: Dataset Description

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

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

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

Table 8.4 (left part) describes general statistics about the honeypot experiments including attacker traces of the honeypot setups. We have observed 349 different successful logins on each SSH server. The shortest attack duration is between 0 and 1 seconds which is mainly due to automated brute-force tools that were run against the honeypots aiming to establish a list of successfully exploited user accounts. The maximal attack duration was 97 minutes. In this SSH session an attacker did heavy configuration work on the system and compiled large programs which took some time. For a standard high-interaction honeypot, the average attacker session lasted for approximately 3 minutes and most attacker session durations were below 3 minutes (83%). Heliza most frequently performed allow actions closely followed by the number of occasions on which it forged output for the attackers. Heliza explicitly blocked 31% of the inputs and deliberately insulted attackers in 17% of the cases. The country code corresponding to the IP addresses used by attackers was looked-up and the distribution is shown in the right part of table 8.4 (right part). Most attackers came from Romanian IP addresses. 16% of the attackers came from German IP addresses and 10% of the attackers had a Spanish IP addresses.

8.6 Fast Concurrent Learning

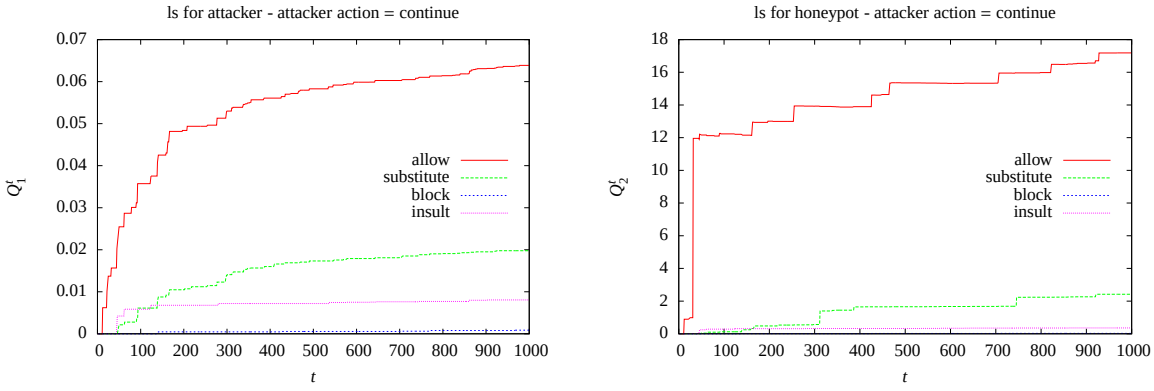
Heliza has some weaknesses to interact with learning attackers because their are considered as environment for Heliza. This problem is tackled with the adaptive honeypot in this section. The learning of attackers is hard to verify. Therefore, we used the honeypot traces described in section 8.1 and 8.2 as ground truth for learning evaluation and serve for setting up the honeypot automaton. 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 transitions between programs. In addition, the delays between

two successive programs were used, serving as reward for 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 [161]. Due to space reasons only a few relevant states are represented in figures 8.8(a), 8.8(b), 8.9(a) and 8.9(b). 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 8.8(a) and 8.8(b) 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 8.9(a)). 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 8.9(b)).

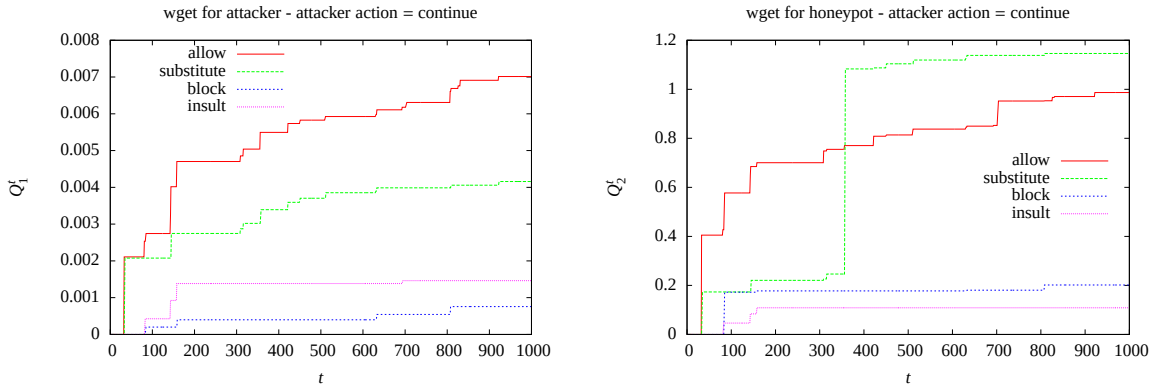
For each command entered, the honeypot needs to decide whether to allow this command to block the program execution, to substitute the output of the program or to insult the attacker. Q values are then recorded for each player. Figures 8.10(a) and 8.10(b) show the average Q-values, represented on the y-axis, for the honeypot in different states, shown on the x-axis. Rewards are distributed in a 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. In figure 8.10(a) 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. If the attacker made a typographic error, he may enter the intended program which is also considered as different program input. Hence, the highest average Q-value results in the selection of an an alternative path. Figure 8.10(a) 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.

We implemented a fast concurrent learning module for our AHA framework, which is publicly available [174]. 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, additional theoretical research needs to be done aiming at an assessment of learning techniques from empirical



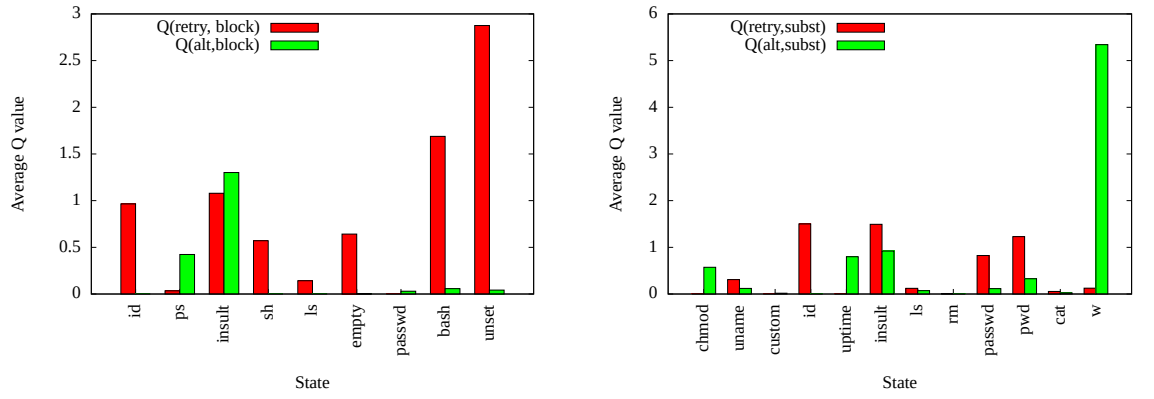
(a) Attacker's Perspective (b) Honeypot's Perspective

Figure 8.8: Q-value Evolution for the State `ls`



(a) Attacker's Perspective (b) Honeypot's Perspective

Figure 8.9: Q-value Evolution for the State `wget`



(a) Block Impact (b) Substitute Impact

Figure 8.10: Impact when the Honeypot Blocks or Substitutes Program Executions

data because we do not know which learning approaches real attackers use. If these challenges are solved, a larger dataset is needed for doing this analysis, but these preliminary results already confirm that our approach can be implemented and operated.

8.7 Conclusions

This chapter has described experiments related to our adaptive honeypots. We operated two state-of the art honeypots in order to to recover data of attackers. In our initial study, a high-interaction honeypot and a a low-interaction honeypot were set up for this purpose. The traces from the high-interaction honeypot were used to generate a hierarchical probabilistic automaton. This then served as the basis for computing payoffs via simulations with these payoffs, we could to compute optimal strategy profiles for each player. The initial aspect of adaptability of the honeypot to attacker is given by blocking one system call according to optimal blocking probabilities resulting from the Nash equilibria. These probabilities depend on the payoff functions used to model the objective of an adaptive honeypot.

We operated an adaptive honeypot with these blocking probabilities and observed that attackers typically enter three more commands than on a regular high-interaction honeypot. However, this approach has two major disadvantages. Firstly, the optimal blocking probabilities depend on the particular instance of our hierarchical probability automaton, and thus depend on recovered traces used to generate it. Secondly, we assume that attackers are always rational. This might not always be the case. Nicomette et. al. [112] described observations of script kiddies, who do not understand the details of an attack, but just try blindly to apply a particular attack technique. Thirdly, the adaptability of the adaptive honeypot is quite coarse grained. The optimal strategy profiles for an adaptive honeypot define only a probability of blocking or allowing the execution of a program, while ignores, the context of the execution. Attackers could easily perform a profiling attack on the adaptive honeypot by executing the same command several times. To address this issue, we operated an additional adaptive high-interaction honeypot driven by reinforcement learning. This approach, allowed us to estimate the best behaviors for a given state, which corresponds to an installed program on the honeypot. In addition we extended the adaption mechanisms of our adaptive honeypots by adding the possibility of substituting the execution of a program with another program, and of insulting the attacker. The outcome of this experiment is a final action-value table that defines an adaptive's honeypot optimal behavior in a given state.

A comparison of the adaptive honeypot with a regular honeypot shows that attackers perform **three times more transitions** (see figure 8.7(a)) to customized tools on the adaptive honeypot. In addition, an adaptive honeypot allows to observe the skills or social background of an attacker. Examples are whether attackers respond to insults, or are clever enough to find alternative solutions to reach their attack goal. However, the straightforward reinforcement learning approach defines attackers as the environment of the adaptive honeypot, and ignores the differing goals of the attackers and adaptive honeypots. It has the weakness that it ignores the fact that attackers could learn too. Consequently, the learning algorithm has a slow convergence to the optimal values for a given state. To counter this, we applied a fast concurrent reinforcement learning algorithm in order to achieve a better convergence for optimal behaviors in each state.

Chapter 9

Conclusions and Perspectives

9.1 Summary of the thesis

This thesis presents a new paradigm for adaptive high-interaction honeypots. An adaptive honeypot can strategically interfere with an attacker's actions in order to make her reveal more information about herself. The idea of manually interacting with an attacker was pioneered in the early nineties by Cheswick [26]. We did a similar experience and we observe that attackers have dedicated attack goals and that they want to reach. In order to reach this goal, an attacker has to follow a path (i.e. has to enter a sequence of commands). We identified three actions an attacker could perform when she is diverted from a direct attack path. Firstly, an attacker could retry the command, either unchanged or with different command line arguments. Secondly, an attacker could select an alternative path to achieve an attack goal. Thirdly, an attacker can simply leave the compromised system because it is unattractive or because she cannot find an alternative means of reaching the initial attack goal.

After having described the assumed objectives and actions of an attacker, this thesis described the utilities and actions of an adaptive honeypot. Firstly, the main goal of honeypots is to collect information about attackers such as the tools or programs they use. Secondly, a honeypot should keep attackers busy. If attackers quickly leave a honeypot this means it is unattractive to them in terms of bandwidth, CPU power or installed programs. Thirdly, an adaptive honeypot should strategically interfere with an attack, for instance, by blocking the execution of an attacker's customized program. By using an increasing resistance against attackers, an attacker's skills can be assessed. They may give up, retry commands or intelligently look for alternative solutions. Fourthly, attackers often often use already-compromised machines as stepping stones for their attacks. Consequently, an attacker's source IP address does not necessarily reveal her origin. Hence, an adaptive honeypot should reveal an attacker's linguistic skills. For instance, they may use a slang language or profanities that reveal their ethnic background.

An attacker and a honeypot have differing interests. The interaction between the honeypot and an attacker can be modeled as a game, where appropriate payoff functions model the goals observed in the real world. Rather than individual attackers, an omnipresent attacker is modeled. Thus, two players are considered. Attackers penetrate the honeypot with the purpose of reaching their attack goal. The system penetrated by attackers is modeled with a probabilistic hierarchical automaton. Once attackers have entered a system, they usually start to execute programs. Each program corresponds to a macro-state of the automaton. In addition, attackers may execute programs with various command line arguments. The effect of a particular command-line argument depends on the program. Hence, each program is modeled by a distinct automaton, where each state corresponds to a command-line argument. Attackers enter sequences of commands, i.e. tokens in the alphabet of the automaton, resulting in program invocation triggering transitions in the automaton. The transitions are described by transition probabilities.

We first used, traces from a high-interaction honeypot to estimate these transition probabilities and a number of states. We used the resulting automaton for performing Monte Carlo simulations in order to calculate payoffs for each player and derive the optimum strategies using the well known Nash equilibrium. However, such

equilibria depend on the attack traces used to generate the probabilistic hierarchical automaton. Therefore, in a further study we used reinforcement learning in conjunction with a model-free approach. This means that the honeypot learns the optimum behavior as consequence of its operation. The honeypot is modeled as an agent operating in an environment that aims to optimize a reward signal. An omnipresent attacker is modeled as acting in the honeypot's environment. However, this approach ignores the fact that players have different interests, and makes the possibility false assumption that the environment is stationary. Therefore, our final approach is to use a multi-agent learning approach to frame this problem. This has the further advantage of being efficient to implement.

9.2 Insights

The competitive nature between players has been modeled with a game between two players, and a major problem has been to find the optimal behavioral strategies for each player. Different reward functions have been proposed for each player, including measurable parameters. These include the number of transitions an attacker makes and the time differences between successive commands and inputs provided by attackers. This has the advantages that the honeypot can be operated autonomously and that the attacks do not have to be manually defined in contrast to previously-published approaches. Because it embodies transition probabilities, our model, although it is presented a as hierarchical probabilistic automaton, does follow traditional attack trees, in which an attacker can reach a set of states represented as nodes that are logically interconnected. A premise of such an attack tree is that these logical interconnections must be known in advance. This is not a requirement for our proposed adaptive honeypots.

Proof-of-concept studies have shown that more information can be gathered from attackers using adaptive honeypots than regular high-interaction honeypots. We presented the use case of adaptive Linux systems. However, as discussed in chapter 5, we believe that the paradigm of adaptive honeypots is not necessarily restricted to a given technology. Our experiments suggest that on an adaptive honeypot attackers make more transitions to using their own customized tools than on a regular high-interaction honeypot. In addition, we determined that, on average attackers stay longer on adaptive honeypots than on regular high-interaction honeypots. Another powerful tool for an adaptive honeypot is the reverse Turing test that can distinguish between human and automated attackers. Often when attackers believe that other attackers have insulted, they reply using insulting language themselves. If no insult had been sent to an attacker's terminal, this information would not have been captured. Hence, an adaptive honeypot is sometimes able to reveal the linguistic background of an attacker. As demonstrated in our experiments, adaptive honeypots are calibrated using payoffs or a learning rule. These can be tuned to collect attacker tools, which are valuable prizes for the anti-virus industry. Adaptive honeypots can be configured to just keep attacker busy and so to measure the resistance of attackers confronted by failures. Finally, they can also be calibrated to collect insults from attackers; some combination of insults and attacker-related programs. Taken together, insults, simulated command failure and the reverse Turing test serve as solid building blocks in the measurement of attacker behavior and the identification of ethic and cultural background. We have taken care, that adaptive honeypots can be operated in an automated way aiming to reduce human interventions. We have shown the example of adaptive honeypots in the context of Linux systems exposing vulnerable SSH servers. However, we believe that the adaptation paradigm is not restricted to those systems and may be applied to a larger family of honeypots. We believe that the paradigm of adaptive honeypots may serve in industry to systematically explore attackers behaviors. The retrieved information about attackers may also serve for an improvement of existing low-interaction honeypots.

9.3 Limitations

9.3.1 System Attacks

An adaptive honeypot only interferes with program executions directly triggered by an attacker. Attackers can make indirect attacks by installing a script on the honeypot for later execution by the system itself, for example as a *CRON* job. In this case, an attacker can disconnect from the honeypot and the *CRON* daemon continues the attack. The honeypot only observes and interacts with the commands related to the deployment of the script. All program executions initiated by the system are allowed by default. Hence, an attacker could bypass the adaptation mechanisms by disguising herself as the system. In order to counter this attack, the system model must be extended with additional information such as internal file system knowledge. A taint analysis must be made in order to track all pieces of information related to an attacker. Each program handling such pieces of information has to be considered, and by default should not be allowed.

A similar attack against the system is the installation of an additional remote control mechanism. The deployment of such a backdoor can be observed and taken into account by the honeypot. The backdoor is then set up using a program that accepts incoming connections on a dedicated port. When the attacker leaves the honeypot, she wants the backdoor to remain, meaning that the corresponding program must run as a daemon otherwise it will be terminated by the kernel. The attacker reconnects to the system through the backdoor and is presented with a shell. This results in program execution system calls. However, the honeypot believes that these system calls are related to the system itself and allows them by default. The problem in this case is that the assumption that attackers enter the system via SSH is violated. However, such an assumption is needed to guarantee a stable system. If the honeypot interferes with all system calls, the startup of the operating system would fail and could not operate. The deployment of special firewall rules does not necessarily help. An attacker could use the Browser Exploitation Framework (BeEF) as a backdoor so that the honeypot operator only observes outgoing HTTP traffic. When HTTP traffic is blocked the honeypot becomes unattractive for an attacker. Instead of creating a process tree when the SSH daemon clones a privileged process, a process tree could be created for an arbitrary programs that accept incoming connections on any network interface. However, this requires additional implementation effort in kernel space. The local network interface also has to be considered, because an attacker could implement a backdoor that accepts only incoming connections. She can then connects to this service through a legitimate SSH tunnel.

Moreover, an attacker could confound the states of the hierarchical probabilistic automaton by overwriting existing programs. This strategy could bypass the learning algorithms when a state-action pair has been optimized. For instance, suppose that a learning algorithm has determined that the program `ls` should always be allowed because future rewards are likely to be high. An attacker can replace the program `ls` with her customized attack tool and ensure that it is always possible to execute it. This attack can be mitigated by computing checksums of each installed program at each system call that results in a program execution. This checksum is then compared with the initial tamper-proofed checksums of the programs. However, this mitigation technique requires additional implementation effort in kernel space and results in a performance overhead.

Also, the Linux Pluggable Authentication Module (PAM) could be patched in a different way. A clever attacker might find a machine that has multiple standard accounts like test, admin, ftp etc with the same password, suspicious. A game making login more challenging for attackers, could be developed.

Kernel Attacks

The honeypot's interference with an attacker's commands takes place in kernel space. To circumvent this, an attacker could install her own kernel and reboot the system. A mitigation technique to such an attack is to instrument the virtual machine so that a reboot is translated into a power-off. When a honeypot operator sees that the honeypot had been switched off, he could check the checksums of the kernel binary and determine whether it has been modified. An attacker could alternatively install and load a kernel module, that modifies the honeypot adaptation code. In this case, an attacker modifies the system call table that points to the code of

the system calls. However, in our proposed design, the kernel code is directly modified such that the attacker must alter more than one instruction and addresses must be consequently realigned requiring a large effort on the part of an attacker. Furthermore, our honeypot relies on a monolithic kernel, where no additional modules can be loaded. Another possibility for modifying the kernel memory is through the special device `/dev/kmem`, but this feature could also be switched off. Finally, an attacker could use a kernel exploit in order to access kernel memory. This risk cannot be entirely eliminated because new kernel exploits emerge from time to time. Therefore, the kernel should always be kept aligned with the latest security patches. Due to the fact that our modifications have been made in parts of the kernel code that rarely change, updates can be accomplished by merging our kernel repository clone with the original, using `git`. An adaptive honeypot may interrupt some commands in command blocks where normally all commands would be successful or all would fail. The case where only one fails might be suspicious to some attackers. The introspection of virtual machines is praised to be bullet proof for constructing high-interaction honeypots [194]. However, assumptions must be made on interpretation of the observed data. An attacker capable of modifying the kernel could alter the system call order resulting in misinterpretation of the observed data. Our proposed design is not to be meant bullet proof however we defined the residual risk as acceptable for the evaluation of adaptive high-interaction honeypots.

9.3.2 Behavioral attacks

Chapter 5 presents a game between the honeypot and an omnipresent attacker. A major assumption of game theory is that expecting rational players in simple games. In case of attackers, taking this for granted for attackers is questionable. In practice, it is not always true. There are for instance, script kiddies who simply copy and paste commands to the compromised system without understanding them. There are also attackers who have good Windows skills but poor Linux skills [197]. The individual discipline of attackers may also vary [197]. In order to mitigate the rationality problem, a quantal response equilibrium analysis can be carried out in order to see how stable the Nash equilibria are [61]. When the traces are recovered to calibrate the game model, attackers could poison the transition probabilities by performing dummy automated attacks resulting a biased automaton. Additional research should be done on the hardening of automata in order to tackle this problem. Another assumption of the Nash equilibrium is that players do not deviate from the optimal strategy. However, attackers could make profiling attacks to reveal the strategies of the honeypot. As a result, they could change their strategy during the game in order to gain additional advantage. Such attacks have been partially addressed by using the model-free learning approach presented in section 6.2.

Since the semantics of the programs are not considered, an attacker could abuse existing programs to achieve particular goals. Instead using the program `ls`, to list a directory an attacker could use the command `echo *` and obtain the same result. When a learning approach is used, the immediate rewards may be assigned to the wrong state, possibly slowing down convergence to optimal values. A solution to this problem is to manually define super states [15] on top of the existing hierarchical probabilistic automaton that take the semantics of each program into account. Program profiles defined in chapter 7 could be used to identify a program instead of using the program file name. Markovian environments with partially visible states [189] could be used instead of simple Markov decision processes.

9.4 Future Work

In this thesis, the interactions of attackers with a honeypot are formally modeled an evaluated. Attackers are assumed to be omnipresent. We elected to not consider individual attackers. However, game and learning models that take into account swarms of attackers could be explored colluding in order to discover their individual role during an attack.

9.4.1 Alternative Honeypot Designs and Feature Extensions

Our adaptive honeypot paradigm builds on top of User-Mode-Linux. As described in section 9.3, this design choice leads to traditional high-interaction problems. Virtual machine introspection [59] could be used in order to ensure that an attacker cannot modify observations and decision-making routines. An alternative solution would be to extend mid-interaction honeypots. Jose Antonio Coret [34] implemented Kojoney, an SSH honeypot purely in Python so reducing the operational risk. The author reimplemented a sub-set of frequently used programs in Python. The Kippo honeypot is another mid-interaction honeypot implemented entirely in python. This honeypot includes a fake file system in which attackers believe that they can create files. These are particularly interesting for a honeypot operator. Such a mid-interaction honeypot could be extended with the proposed adaptation mechanisms. However, an open problem is to make attackers believe that they are executing their own programs. Jose Antonio Coret [34] distinguished human from automated attacks by applying heuristics to an attacker's keystroke dynamics. Such dynamics could also be included in our proposed reward models.

Before a widespread deployment of adaptive honeypots, there is a need for a detailed performance analysis. However, an individual analysis has to be made for each learning algorithm. The proposed adaptive honeypots are only capable of insulting in English. Language modules could be added such that the honeypot could swear in different languages or even in the attacker's native language. Our research activities have focused on adaptive high-interaction honeypots exposing a vulnerable SSH service. The paradigm of adaptive honeypots could be extended to other types of services like email and the web [200]. We considered games between attackers and honeypot operators at the operating system level. However, we could model adaptive network routers specialized in collecting attackers' tools based on game theory and reinforcement learning. We could also investigate the presentation of adaptive client honeypots as autonomous services. Client honeypots were not addressed in this thesis; instead of emulating a vulnerable service, they mimic a careless user visiting dubious web sites. A major research challenge in this area is to identify rogue web-sites using a learning approach. The crawling technique typically used for this purpose consists of recursively visiting web sites. However, this method is often inefficient, and costly in terms of resources due to the large volume of data that must be processed. [134]. Autonomous and collaborating swarms of agents driven by reinforcement learning [69] may be a research track that can tackle this problem. The problem of the in-vivo analysis of malicious software has been briefly mentioned. However, we believe that additional malware analysis research should be done in this area.

9.4.2 Additional Honeypot - Attacker System Games

Once an attacker enters the system, she starts to execute commands that result in program executions. Our current adaptive honeypot can allow this execution, substitute the execution with another program or insult the attacker. The honeypot can also block the program execution. Chapter 7 suggest that an arbitrary exit code could be returned. However, in the evaluations, the `Permission Denied` exit code was used. However, examining the Linux kernel's header file defining error types it is clear that many other errors could be returned. A straightforward extension of the honeypot game would be to integrate different exit codes into the game model. As a result, the action space of the honeypot would increase in proportion to the number of possible errors. At system-call level, attackers also create, read and write files. An adaptive honeypot could also interfere with these system calls. Again, this would dramatically increase the action space of the honeypot. Current adaptive honeypots only consider programs and ignore their command line arguments in their learning algorithms and computing the Nash equilibrium. Arguments could also be included, again increasing the action space of the honeypot. Such increases, impact algorithmic complexity of learning algorithms and the computation of the Nash equilibrium, but would permit the modeling of finer-actions for the honeypot.

Attackers penetrate the system through the SSH service. A patch for the Unix authentication module has been suggested for two reasons: firstly, to make it easier for attackers to access to the system; secondly to avoid the possibility of an attacker changing an account to lock others out. However, a clever attacker performing

a brute-force attack against the system will notice that several accounts have the same password and become suspicious. A formal game could be modeled in which the honeypot has to discover the best strategies for letting more experienced attackers in while discouraging novices.

In this thesis, strategic games have been modeled between attackers and adaptive honeypots. However, other types of games could be modeled, such as Stackleberg games [151] or bluffing games [19]. In a Stackleberg game, leaders and followers are defined. Bayesian games [67] could also be modeled, allowing other types of equilibria to be investigated, among them the Bayesian equilibria, which would provide a link between adaptive honeypots and Bayesian learning.

Cooperative Adaptive Honeypots

Formally, games defined between attackers and honeypots are games represented in the normal form, and are either zero-sum or general-sum games which model just one honeypot under attack. Groups of honeypots sharing data among themselves could be modeled. The shared data might concern vulnerabilities and content exposed to attackers. In addition, strategies could be exchanged among honeypots. Hence, Stackelberg games where one honeypot takes a leaders role and the others follows could be developed. Cohen's deception techniques [31] could be automated and distributed. Each honeypot would offer a given number of services and relay real services. The leader would then discover the services that detain attackers for the most time, and share this information with other honeypots in order to keep a larger attacker group busy. By introducing the idea of having multiple honeypots into the reinforcement learning area, collaborative learning or hierarchical learning techniques [14] could be used to calibrate cooperative adaptive honeypots.

Author's publications

Scientific Awards

- *Best Paper Award* at 4th International Conference on Network and System Security. Melbourne, September, 2010.
- *Best Student Paper Award* at the 11th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS 2009) in Lyon, France, 2009.

Publications

Scientific Journal Publications

- Gérard Wagener, Radu State, and Alexandre Dulaunoy. Malware behaviour analysis. Journal in Computer Virology, 4(4):279, 2008.
- Gérard Wagener, Radu State, Alexandre Dulaunoy and Thomas Engel. Heliza: Talking Dirty to the Attackers. Journal in Computer Virology, 2010.

Accepted Journal Publications

- Gérard Wagener, Radu State, Alexandre Dulaunoy and Thomas Engel. Playing with Your Enemy: A Game Theoretical Approach for High Interaction-Honeypots. ACM Journal of Transactions on Autonomous and Adaptive Systems (TAAS), 2011.

Conference Publications

- Gérard Wagener, Radu State, Alexandre Dulaunoy. Malware Behaviour Analysis. In proceedings of the 2nd International Workshop on Theory of Computer Viruses (TCV 2007), May 2007, Nancy, France.
- Gérard Wagener, Alexandre Dulaunoy, Radu State. Automated Malware Behaviour Analysis. Presented at hack.lu, October 2007.
- Gérard Wagener, Alexandre Dulaunoy and Thomas Engel. An Instrumented Analysis of Unknown Software and Malware Driven by Free Libre Open Source Software. In proceedings of SITIS, November 2008.
- Gérard Wagener, Alexandre Dulaunoy and Thomas Engel. Towards an estimation of the accuracy of TCP reassembly in network forensics. Proceedings of the Second International Conference on Future Generation Communication and Networking, pages 273-278.
- Gérard Wagener, Radu State, Alexandre Dulaunoy and Thomas Engel. Self adaptive high interaction honeypots driven by game theory. In the 11th International Symposium on Stabilization, Safety, and

Security of Distributed Systems (SSS), volume 5873 of Lecture Notes in Computer Science, pages 741-755 Springer, Lyon, 2009.

- Cynthia Wagner, Gérard Wagener, Radu State and Thomas Engel. Malware analysis with graph kernels and support vector machines. In 4th International Conference on Malicious and Unwanted Software (Malware 2009), pages 63-68, Montreal, October, 2009.
- Cynthia Wagner, Gérard Wagener, Radu State, Alexandre Dulaunoy and Thomas Engel. Breaking Tor Anonymity with Game Theory and Data Mining. In the 4th International Conference on Network and System Security. Melbourne, September, 2010.
- Cynthia Wagner, Gérard Wagner, Radu State Alexandre Dulaunoy and Thomas Engel. PeekKernelFlows: Peeking into IP flows. In the 7th International Symposium on Visualization for Cyber Security, Ottawa, 2010.
- Gérard Wagener, Alexandre Dulaunoy, Radu State and Thomas Engel. AHA - Adaptive High-Interaction Honeypot Alternative. Presented at hack.lu, October, 2010.
- Gérard Wagener, Radu State, Alexandre Dulaunoy and Thomas Engel. Adaptive and Self-Configurable Honeypots. In the 12th IFIP/IEEE International Symposium on Integrated Network Management, Dublin, 2011.

Talks

- Malware Reverse Engineering. LIASIT Seminar, Luxembourg, April, 2006.
- Attacking the TCP Reassembly Plane of Network Forensics Tools. IT-Underground XI, Warsaw, October 2008.
- Gérard Wagener, Frédéric Raynal, Alexandre Dulaunoy, Christophe Kyvrakidis. Detecting User Mode Linux Honeypots is fine ... but it's better to crash them. Presented at hack.lu in the barcamp, Luxembourg, October 2008.
- Game-Theoretic Honeypot Control. Future Challenges in Network Security, Prague, June 2010.

Open Source Contributions

Discovered and Reported Bugs

Table 9.1 describe the bugs. The bugs have been reported to the software maintainers. The first column describes the destination where a given bug was filed. The official bug tracker from the Linux kernel[1] has the code K and the bug tracker Launchpad[2] is denoted L.

Destination.	Bug Identifier	Bug Description
L	289983	Tcptrace is vulnerable against some of the fragrouter attacks
L	245531	Use of uninitialized bytes during TCP reassembly (patch proposal)
L	252604	Stack overflow in the bvi package
L	256122	DOS vulnerability in tcpflow
L	289976	Segmentation fault on tcpick with fragmented IP packets
L	364688	Tcpick uses wrong timestamps in the output
K	11974	UML crashing as non-root with a specific mmap

Table 9.1: Reported Vulnerabilities

Developed Projects

Project Name	Project Description	Reference
ANNE	Malware Sandbox	http://sourceforge.net/projects/anne/
FIW	Highlevel Debugger	http://git.quuxlabs.com
AHA	Adpative High-Interaction Honeypot	http://git.quuxlabs.com

[1] https://bugzilla.kernel.org
[2] https://launchpad.net/

Bibliography

[1] Pieter Abbeel, Adam Coates, Morgan Quigley, and Y. Ng Andrew. An application of reinforcement learning to aerobatic helicopter flight. In *Advances in Neural Information Processing Systems*, pages 1–8. MIT Press, 2007.

[2] Éric Alata, Ion Alberdi, Vincent Nicomette, Philippe Owezarski, and Mohamed Kaâniche. Internet attacks monitoring with dynamic connection redirection mechanisms. *Journal in Computer Virology*, 4(2):127–136, 2008.

[3] Éric Alata, Vincent Nicomette, Mohamed Kaâniche, Marc Dacier, and Matthieu Herrb. Lessons learned from the deployment of a high-interaction honeypot. In *Dependable Computing Conference*, pages 39–46, 2006.

[4] K. G. Anagnostakis, S. Sidiroglou, P. Akritidis, K. Xinidis, E. Markatos, and A.D Keromytis. Detecting targeted attacks using shadow honeypots. In *Proceedings of the 14th conference on USENIX Security Symposium*, volume 14, Berkeley, CA, USA, 2005. USENIX Association.

[5] Spiros Antonatos, Kostas Anagnostakis, and Evangelos Markatos. Honey@home: A New Approach to Large-Scale Threat Monitoring. In *Proceedings of the 2007 ACM workshop on recurring malcode*, pages 38–45, New York, NY, USA, 2007. ACM.

[6] Paul Baecher and Markus Koetter. Dionaea catches bugs. `http://dionaea.carnivore.it/`. Last accessed, February 2011.

[7] Paul Baecher, Markus Koetter, Thorsten Holz, Maximillian Dornseif, and Felix Freiling. The Nepenthes Platform: An Efficient Approach to Collect Malware. In Diego Zamboni and Christopher Kruegel, editors, *Recent Advances in Intrusion Detection*, volume 4219 of *Lecture Notes in Computer Science*, pages 165–184. Springer Berlin / Heidelberg, 2006.

[8] George Bakos. Tiny Honeypot. `http://freshmeat.net/projects/thp/`. Last accessed December 2010.

[9] Edward Balas. Sebek: Covert Glass-box Host Analysis. *;Login Magazine*, 28(6), December 2003.

[10] Edward Balas and Camilo Viecco. Towards a Third Generation Data Capture Architecture for Honeynets. In *6th IEEE Information Assurance Workshop*, pages 21–28. IEEE, 2005.

[11] Bikramjit Banerjee, Sandip Sen, and Jing Peng. Fast Concurrent Reinforcement Learners. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence*, pages 825–830, San Francisco, CA, USA, 2001. Morgan Kaufmann Publishers Inc.

[12] Daniel Barlow. Building your own live CD. *Linux Journal*, 2005, April 2005.

[13] Andrew G. Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(1-2):41–77, 2003.

[14] Andrew G. Barto and Sridhar Mahadevan. Recent Advances in Hierarchical Reinforcement Learning. *Discrete Event Dynamic Systems*, 13(4):341–379, October 2003.

[15] Frédérique Bassino, Marie-Pierre Béal, and Dominique Perrin. Super-State Automata and Rational Trees. In *Proceedings of the Third Latin American Symposium on Theoretical Informatics*, LATIN '98, pages 42–52, London, UK, 1998. Springer-Verlag.

[16] Mick Bauer. Paranoid penguin: syslog configuration. *Linux Journal*, 2001, December 2001.

[17] Richard Bejtlich. *The Tao of Network Security Monitoring: Beyond Intrusion Detection*. Addison-Wesley Professional, 2004.

[18] Fabrice Bellard. QEMU, a Fast and Portable Dynamic Translator. In *USENIX Annual Technical Conference*, pages 41–46, Berkeley, CA, USA, 2005. USENIX Association.

[19] Richard Bellman. On games involving bluffing. *Rendiconti del Circolo Matematico di Palermo*, 1:139–156, 1952. 10.1007/BF02847783.

[20] Steven M. Bellovin. There Be Dragons. In *Proceedings of the Third Usenix Unix Security Symposium*, pages 1–16, September 1992.

[21] Ken Binmore. *Playing for Real*. Oxford University Press, 2007.

[22] Bill Blunden. *The Rootkit Arsenal: Escape and Evasion in the Dark Corners of the System*. Jones and Bartlett Publishers, Inc., USA, 2009.

[23] Herman Anthony Carneiro and Eleftherios Mylonakis. Google trends: a web-based tool for real-time surveillance of disease outbreaks. *Clinical infectious diseases: an official publication of the Infectious Diseases Society of America*, 49(10), October 2009.

[24] Brian E. Carpenter. Observed relationships between size measures of the internet. *SIGCOMM Comput. Commun. Rev.*, 39:5–12, March 2009.

[25] George Chamales. The Honeywall CD-ROM. *IEEE Security and Privacy*, 2:77–79, 2004.

[26] Bill Cheswick. An Evening with Berferd in Which a Cracker is Lured, Endured and Studied. In *Proceedings of the USENIX Conference*, pages 163–174. USENIX Association, 1992.

[27] William R. Cheswick, Steven M. Bellovin, and Aviel D. Rubin. *Firewalls and Internet Security; Repelling the Wily Hacker*. Addison-Wesley, Reading, Massachusetts, second edition, 2003.

[28] Eric Chien. The New Generation of Targeted Attacks. `http://www.raid2010.org/files/EricChien.pptx`, 2010. Keynote at Recent Advances in Intrusion Detection.

[29] Benoît Claise. Cisco Systems NetFlow Services Export System, Oct 2004. RFC 3954.

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

[31] Fred Cohen. A note on the role of deception in information protection. *Computers & Security*, 17(6):483–506, 1998.

[32] M. Collins and N. Duffy. Convolutional Kernels for Natural Language. *Advances in Neural Information Processing Systems 14*, 2002. MIT Press.

[33] Jonathan Corbet, Alessandro Rubini, and Greg Kroah-Hartman. *Linux Device Drivers, 3rd Edition*. O'Reilly Media, Inc., 2005.

[34] Jose Antonio Coret. Kojoney — A honeypot for the SSH Service. `http://kojoney.sourceforge.net`. Last accessed December 2010.

[35] Richard W. Cottle, Jong-Shi Pang, and Richard E. Stone. *The Linear Complementary Problem*. Academic Press, 1992.

[36] M.F. Cowlishaw. Fundamental Requirements for picture presentation. In *Proceedings of the Society for picture presentation*, volume 26, pages 101–107, 1985.

[37] Mark Crovella and Balachander Krishnamurthy. *Internet Measurement*, chapter Issues in capturing data, pages 101–102. John Wiley & Sons Ltd, 2006.

[38] Marc Dacier. Leurré.com: a worldwide distributed honeynet, lessons learned after 4 years of existence. In *Terena Networking Conference*, Bruges, Belgium, May 2008.

[39] Marc Dacier, Corrado Leita, Olivier Thonnard, Hau Pham, and Engin Kirda. Assessing Cybercrime Through the Eyes of the WOMBAT. In Sushil Jajodia, Peng Liu, Vipin Swarup, and Cliff Wang, editors, *Cyber Situational Awareness*, volume 46 of *Advances in Information Security*, pages 103–136. Springer US, 2010. 10.1007/978-1-4419-0140-8_6.

[40] David Dagon, Xinzhou Qin, Guofei Gu, Wenke Lee, Julian Grizzard, John Levine, and Henry Owen. HoneyStat: Local Worm Detection Using Honeypots. In *Recent Advances in Intrusion Detection*, volume 3224 of *Lecture Notes in Computer Science*, pages 39–58. Springer Verlag, 2004.

[41] Abhishek Das, David Nguyen, Jospeh Zambreno, Gokhan Memik, and Alok Chouldhary. An FPGA-Based Network Intrusion Detection Architecture. *Information Forensics and Security*, 3(1):118–132, Mar 2008.

[42] Jeff Dike. *User Mode Linux*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2006.

[43] Artem Dinaburg, Paul Royal, Monirul I. Sharif, and Wenke Lee. Ether: malware analysis via hardware virtualization extensions. In Peng Ning, Paul F. Syverson, and Somesh Jha, editors, *ACM Conference on Computer and Communications Security*, pages 51–62. ACM, 2008.

[44] Kevin Dooley. *Designing Large Scale Lans*. O'Reilly Media, November 2001.

[45] George W. Dunlap, Samuel T. King, Sukru Cinar, Murtaza A. Basrai, and Peter M. Chen. ReVirt: Enabling Intrusion Analysis through Virtual-Machine Logging and Replay. In *In Proceedings of the 2002 Symposium on Operating Systems Design and Implementation (OSDI)*, pages 211–224, 2002.

[46] Thomas E. Carrol and Daniel Grosu. A Game Theoretic Investigation of Deception in Network Security. In *Proceedings of 18th International Conference on Computer Communications and Networks*, pages 1–6. IEEE, 2009.

[47] Jon Erickson. *Hacking: The Art of Exploitation 2nd Edition*. No Starch Press, 2nd edition, February 2008.

[48] Kevin Fairbanks. *Forensic framework for honeypot analysis*. PhD thesis, Georgia Institute of Technology, April 2010.

[49] Éric Filiol. *Computer Viruses: from theory to applications (Collection IRIS)*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2005.

[50] A.M. Fink. Equilibrium in a Stochastic n-Person Game. *Journal of science of the Hiroshima university*, 28:89–93, 1964.

[51] Brad Fitzpatrick. Distributed caching with memcached. *Linux Journal*, 2004, August 2004.

[52] Mark Fleischer. The measure of pareto optima. applications to multi-objective metaheuristics. In *Evolutionary Multi-Criterion Optimization. Second International Conference, EMO 2003*, pages 519–533. Springer, 2003.

[53] Foster and C. James. *Metasploit Toolkit for Penetration Testing, Exploit Development, and Vulnerability Research*. Syngress Publishing, 2007.

[54] Stefan Frei, Martin May, Ulrich Fiedler, and Bernhard Plattner. Large-scale vulnerability analysis. In *LSAD '06: Proceedings of the 2006 SIGCOMM workshop on Large-scale attack defense*, pages 131–138, New York, NY, USA, 2006. ACM.

[55] Drew Fudenberg and Jean Tirole. *Game Theory*. MIT Press, August 1991.

[56] Drew Fudenberg and Jean Tirole. Perfect Bayesian equilibrium and sequential equilibrium. *Journal of Economic Theory*, 53(2):236 – 260, 1991.

[57] Fyodor. Remote OS Detection via TCP/IP Stack Fingerprinting, October 1998.

[58] Luca M. Gambardella and Marco Dorigo. Ant-Q: A Reinforcement Learning approach to the traveling salesman problem. In *Proceedings of the ML-95, 12th international conference on machine learning*, pages 252–260. Morgan Kaufmann, 1995.

[59] Tal Garfinkel and Mendel Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Network and Distributed Systems Security Symposium (NDSS)*, pages 191–206, 2003.

[60] Jan Göbel. Amun: A Python Honeypot. Technical Report TR-2009-008, University of Mannheim, 2009.

[61] Jacob Goeree, Charles Holt, and Thomas Palfrey. Regular Quantal Response Equilibrium. *Experimental Economics*, 8(4):347–367, December 2005.

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

[63] Google. Google Scholar. `http://scholar.google.lu/intl/en/scholar/about.htm`.

[64] Amy Greenwald. Matrix Games and Nash Equilibrium, 2007. Lecture.

[65] Roger Grimes. *Honeyd Service Scripts*, pages 167–188. Springer, 2005.

[66] P. Haag. nfdump. `nfdump.sourceforge.net`.

[67] John C. Harsanyi. Games with Incomplete Information Played by "Bayesian" Players, I-III. Part I. The Basic Model. *Management Science*, 14(3):159–182, 1967.

[68] Brian Hay and Kara Nance. Forensics examination of volatile system data using virtual introspection. *SIGOPS Oper. Syst. Rev.*, 42(3):74–82, 2008.

[69] Iima Hitoshi and Kuroe Yasuaki. Swarm Reinforcement Learning Method Based on an Actor-Critic Method. In *Simulated Evolution and Learning*, volume 6457 of *Lecture Notes in Computer Science*, pages 279–288. Springer Berlin / Heidelberg, 2010.

[70] Thorsten Holz and Frédéric Raynal. Detecting Honeypots and Other Suspicious Environments. In *6th IEEE Information Assurance Workshop*, United States Military Academy, West Point, 2005.

[71] Honeypot Background. http://www.honeyd.org/background.php.

[72] Michael Howard, Jon Pincus, and Jeannette Wing. *Measuring Relative Attack Surfaces*, pages 109–134. Springer, 2005.

[73] Junling Hu and Michael P. Wellman. Multiagent Reinforcement Learning: Theoretical Framework and an Algorithm. In *In Proceedings of the Fifteenth International Conference on Machine Learning*, pages 242–250. Morgan Kaufmann, 1998.

[74] Junling Hu and Michael P. Wellman. Nash Q-Learning for General-Sum Stochastic Games. *JOURNAL OF MACHINE LEARNING RESEARCH*, 4:1039–1069, 2003.

[75] Bert Hubert, Thomas Graf, Gregory Maxwell, Remco Van Mook, Martijn Van Oosterhout, Paul B. Schroeder, Jasper Spaans, and Pedro Larroy. *Linux Advanced Routing & Traffic Control HOWTO*, April 2004.

[76] Choi Hyunyoung and Varian Hal. Predicting the Present with Google Trends. http://www.google.com/googleblogs/pdfs/google_predicting_the_present.pdf.

[77] iDefense. MultiPot. http://labs.idefense.com/files/labs/releases/previews/multipot/.

[78] Omar Ismail, Masashi Etoh, Youki Kadobayashi, and Suguru Yamaguchi. A Proposal and Implementation of Automatic Detection/Collection System for Cross-Site Scripting Vulnerability. In *Proceedings of the 18th International Conference on Advanced Information Networking and Applications*, volume 2 of *AINA '04*, Washington, DC, USA, 2004. IEEE Computer Society.

[79] V Jacobson, C Leres, and S McCanne. tcpdump. http://www.tcpdump.org.

[80] Grossklags Jens, Christin Nicolas, and Chuang John. Secure or insure?: a game-theoretic analysis of information security games. In *Proceeding of the 17th international conference on World Wide Web*, WWW '08, pages 209–218, New York, NY, USA, 2008. ACM.

[81] Chow Jim, Pfaff Ben, Garfinkel Tal, Christopher Kevin, and Rosenblum Mendel. Understanding data lifetime via whole system simulation. In *Proceedings of the 13th conference on USENIX Security Symposium - Volume 13*, SSYM'04, Berkeley, CA, USA, 2004. USENIX Association.

[82] Leslie P. Kaelbling, Michael L. Littman, and Andrew W. Moore. Reinforcement Learning: A Survey. *Journal of Artificial Intelligence Research*, 4:237–285, 1996.

[83] Ionna Kantzavelou and Sokratis Katsikas. Playing Games with Internal Attackers Repeatedly. In *Proceedings of 16th International Conference on Systems, Signals and Image Processing*, pages 1–6, Los Alamitos, CA, USA, June 2009. IEEE Computer Society.

[84] H. Kashima, K. Tsuda, and A. Inokuchi. Marginalized kernels between labeled Graphs. In *Proceedings of the Twentieth International Conference on Machine Learning, ICML-2003, Washington DC*, 2003.

[85] H. Kashima, K. Tsuda, and A. Inokuchi. *Kernels for Graphs*. By B.Schoelkopf, K.Tsuda, J.-P.Vert, The MIT Press, illustrated edition, 2004. chapter 7, pp.155-170, ISBN-100262195096.

[86] Doya Kenji. Reinforcement Learning in Continuous Time and Space. *Neural Computation*, 12(1):219–245, 2000.

[87] Cho Kenjiro, Kaizaki Ryo, and Kato Akira. Aguri: An Aggregation-Based Traffic Profiler. In *COST 263: Proceedings of the Second International Workshop on Quality of Future Internet Services*, pages 222–242, London, UK, 2001. Springer-Verlag.

[88] Georgios Kontaxis, Iasonas Polakis , Spiros Antonatos , and Evangelos P. Markatos . Experiences and Observations from the NoAH Infrastructure. *Computer Network Defense, European Conference on*, pages 11–18, 2010.

[89] Christian Kreibich and Jon Crowcroft. Honeycomb: creating intrusion detection signatures using honeypots. *Computer Communication Review*, 34(1):51–56, 2004.

[90] Michael L. Littman. Markov games as a framework for multi-agent reinforcement learning. In *Proceedings of the eleventh international conference on machine learning*, pages 157–163. Morgan Kaufmann, 1994.

[91] Corrado Leita, Ken Mermoud, and Marc Dacier. Automatic handling of protocol dependencies and reaction to 0-day attacks with scriptgen based honeypots. In *Recent Advances in Intrusion Detection*, volume 4219 of *Lecture Notes in Computer Science*, pages 185–205. Springer Verlag, 2006.

[92] C. E. Lemke. Equilibrium Points of Bimatrix Games. *Journal of the Society for Industrial and Applied Mathematics*, 12(2):413–423, 1964.

[93] Connie Li and Taufik Parsioan. Profiling Honeynet Attackers. In *Proceedings of the Class of 2006 Senior Conference on Natural Language Processing*, pages 19–26, Swarthmore, Pennsylvania, 2006.

[94] Linux Kernel Documentation. `http://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt`. ip-sysctl.txt.

[95] Tom Liston. Home page of "LaBrea". `http://labrea.sourceforge.net`.

[96] Michael L. Littman and Csaba Szepesvári. A Generalized Reinforcement-Learning Model: Convergence and Applications. In *In Proceedings of the 13th International Conference on Machine Learning*, pages 310–318. Morgan Kaufmann, 1996.

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

[98] Kong-Wei Lye and Jeannette M. Wing. Game strategies in network security. *International Journal of Information Security*, 4(1):71–86, February 2005.

[99] Federico Maggi and Stefano Zanero. Analysis of the state-of-the-art. `http://wombat-project.eu/workpackages/wp2-analysis-of-state-of-the-a/`. ICT-216026-WOMBAT.

[100] O. L. Mangasarian. Equilibrium Points of Bimatrix Games. *Journal of the Society for Industrial and Applied Mathematics*, 12(4):778–780, 1964.

[101] Bill McCarty. The Honeynet Arms Race. *IEEE Security and Privacy*, 1(6):79–82, 2003.

[102] E. H. McKinney. Generalized Birthday Problem. *American Mathematical Monthly*, 73:385–387, 1966.

[103] Bauer Mick. Paranoid penguin: AppArmor in Ubuntu 9. *Linux Journal*, 2009, September 2009.

[104] Mark Mitchell and Alex Samuel. *Advanced Linux Programming*. New Riders Publishing, Thousand Oaks, CA, USA, 2001.

[105] David Moore, Geoffrey M. Voelker, and Stefan Savage. Inferring Internet denial-of-service activity. In *Proceedings of the 10th conference on USENIX Security Symposium*, volume 10, Berkeley, CA, USA, 2001. USENIX Association.

[106] A. Moschitti. Efficient Convolution Kernels for dependency and Constituent Syntactic Trees. In *Proceedings of the 17th European Conference on Machine Learning, Berlin, Germany*, 2006.

[107] A. Moschitti. Making Tree Kernels practical for Natural Language Learning. In *Proceedings of the 11th International Conference on EACL, Trento, Italy*, 2006.

[108] Garg Nandan and Grosu Daniel. Deception in honeynets: A game-theoretic analysis. In *Information Assurance and Security Workshop, 2007. IAW '07. IEEE SMC*, pages 107–113, 2007.

[109] John Nash. *Non-cooperative games*. PhD thesis, Princeton University, May 1950.

[110] Nicholas Nethercote and Julian Seward. How to shadow every byte of memory used by a program. In *VEE '07: Proceedings of the 3rd international conference on virtual execution environments*, pages 65–74, New York, NY, USA, 2007. ACM.

[111] Cameron Newham, J. Vossen, Carl Albing, and J.P. Vossen. *Bash Cookbook: Solutions and Examples for Bash Users*. O'Reilly Media, Inc., 2007.

[112] Vincent Nicomette, Mohamed Kaâniche, Éric Alata, and Matthieu Herrb. Set-up and deployment of a high-interaction honeypot: experiment and lessons learned. *Journal in Computer Virology*, pages 1–15, 2010.

[113] Gupta Nirbhay. Improving the effectiveness of deceptive honeynets through an empirical learning approach, 2002. Paper presented at the 2002 Australian Information Warfare and Security Conference.

[114] NIST. Recommended security controls for federal information systems and organizations, 2010. NIST Special Publication 800-53.

[115] Adam J. O'Donnell. When Malware Attacks (Anything but Windows). *IEEE Security and Privacy*, 6:68–70, 2008.

[116] Aleph One. Smashing The Stack For Fun And Profit. *Phrack*, 7(49), November 1996.

[117] Michael Owens. Embedding an SQL database with SQLite. *Linux Journal*, 2003(110), 2003.

[118] Vern Paxson. tcpslice. `ftp://ftp.ee.lbl.gov/tcpslice.tar.gz`.

[119] Jing Peng and Ronald J. Williams. Incremental multi-step Q-learning. In *Machine Learning*, volume 22, pages 226–232, 1994.

[120] Kalyan S. Perumalla and Srikanth Sundaragopalan. High-Fidelity Modeling of Computer Network Worms. *Computer Security Applications Conference, Annual*, 0:126–135, 2004.

[121] Scott Piper, Mark Davis, and Sujeet Shenoi. Countering Hostile Forensic Techniques. In Martin Olivier and Sujeet Shenoi, editors, *Advances in Digital Forensics II*, volume 222 of *IFIP Advances in Information and Communication Technology*, pages 79–90. Springer Boston, 2006.

[122] Michalis Polychronakis, Kostas G. Anagnostakis, and Evangelos P. Markatos. Network-level polymorphic shellcode detection using emulation. *Journal in Computer Virology*, 2(4):257–274, 2007.

[123] Ryan Porter, Eugene Nudelman, and Yoav Shoham. Simple Search Methods for Finding a Nash Equilibrium. In *Games and Economic Behavior*, pages 664–669, 2004.

[124] Georgios Portokalidis and Herbert Bos. Eudaemon: Involuntary and On-Demand Emulation Against Zero-Day Exploits. In *Proceedings of ACM SIGOPS EUROSYS'08*, pages 287–299, Glasgow, Scotland, UK, April 2008. ACM SIGOPS.

[125] Georgios Portokalidis, Asia Slowinska, and Herbert Bos. Argos: an emulator for fingerprinting zero-day attacks for advertised honeypots with automatic signature generation. *SIGOPS Oper. Syst. Rev.*, 40(4):15–27, 2006.

[126] J. Postel and J. Reynolds. Telnet protocol specification, May 1983. RFC 854.

[127] Shawn Powers. Virtual Interfaces: When One IP Isn't Enough. *Linux Journal*, September 2009.

[128] Niels Provos. A virtual honeypot framework. In *SSYM'04: Proceedings of the 13th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2004. USENIX Association.

[129] Niels Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *SSYM'03: Proceedings of the 12th conference on USENIX Security Symposium*, Berkeley, CA, USA, 2003. USENIX Association.

[130] Niels Provos and Thorsten Holz. *Virtual honeypots: from botnet tracking to intrusion detection*. Addison-Wesley Professional, 2007.

[131] Niels Provos, Panayiotis Mavrommatis, Moheeb Abu Rajab, and Fabian Monrose. All your iFRAMEs point to Us. In *Proceedings of the 17th conference on Security symposium*, Berkeley, CA, USA, 2008. USENIX Association.

[132] Niels Provos, Dean Mcnamee, Panayiotis Mavrommatis, Ke Wang, Nagendra Modadugu, and Google Inc . The ghost in the browser: Analysis of web-based malware. In *In Usenix Hotbots*, 2007.

[133] Thomas H. Ptacek and Timothy N. Newsham. Insertion, evasion, and denial of service: Eluding network intrusion detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, January 1998.

[134] M.T. Qassrawi and Hongli Zhang. Client honeypots: Approaches and challenges. In *4th International Conference on New Trends in Information Science and Service Science (NISS)*, pages 19–25, May 2010.

[135] Daniel Ramsbrock, Robin Berthier, and Michel Cukier. Profiling Attacker Behavior Following SSH Compromises. In *DSN '07: Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*, pages 119–124, Washington, DC, USA, 2007. IEEE Computer Society.

[136] Michael Rash. *Linux firewalls*. No Starch Press, San Francisco, CA, USA, first edition, 2007.

[137] Eric S. Raymond. ncurses. *Linux Journal*, 1995, September 1995.

[138] Fréderic Raynal, Yann Berthier, Philippe Biondi, and Danielle Kaminsky. Honeypot Forensics Part I: Analyzing the Network. *IEEE Security and Privacy*, 2(4), 2004.

[139] Fréderic Raynal, Yann Berthier, Philippe Biondi, and Danielle Kaminsky. Honeypot Forensics, Part II: Analyzing the Compromised Host. *IEEE Security and Privacy*, 2, 2004.

[140] Jörg Rech. Discovering trends in software engineering with Google trend. *SIGSOFT Softw. Eng. Notes*, 32(2), March 2007.

[141] Ripe. `http://www.ripe.net/`.

[142] Smalley S., Vance C., and Salamon W. Implementing SELinux as a Linux security module. NAI Labs Report #01-043, NAI Labs, Dec 2001. Revised May 2002.

[143] Tom Schaul, Justin Bayer, Daan Wierstra, Sun Yi, Martin Felder, Frank Sehnke andThomas, and Jürgen Schmidhuber. PyBrain. *Journal of Machine Learning Research*, 2010.

[144] Stephan Schmidt, Tansu Alpcan, Sahin Albayrak, Tamer Basar, and Achim Müller. A Malware Detector Placement Game for Intrusion Detection. In Javier Lopez and Bernhard M. Hämmerli, editors, *Critical Information Infrastructures Security (CSITIS), Second International Workshop*, volume 5141 of *Lecture Notes in Computer Science*, pages 311–326. Springer, October 2008.

[145] Bruce Schneier. Attack trees. *Dr. Dobbs Journal*, 24(12), December 1999.

[146] B. Schoelkopf and J. Smola. *Learning with Kernels*. The MIT Press, illustrated edition, 2002. chapter 1-3, pp. 1-78, ISBN-100262194759.

[147] Christian Seifert, Ian Welch, and Peter Komisarczuk. Taxonomy of Honeypots. Technical Report CS-TR-06/12, School of Mathematical and Computing Sciences, PO Box 600 Wellington New Zealand, June 2006.

[148] Linn I. Sennott. Average Cost Optimal Stationary Policies in Infinite State Markov Decision Processes with Unbounded Costs. *Operations Research*, 37(4):626–633, 1989.

[149] Lloyd Shapley. A note on the Lemke-Howson algorithm. *Mathematical Programming Study*, 1:175–189, 1974.

[150] L.S. Shapley. Stochastic games. *Proceedings of the National Academy if Sciences of the United States of America*, 39(10):1095–1100, October 1953.

[151] M. Simaan and J.B. Cruz Jr. On the Stackelberg Strategy in Nonzero-Sum Games. *Journal of Optimization Theory and Applications*, 11(5):533–555, May 1973.

[152] Satinder P. Singh, Tommi Jaakkola, Michael L. Littman, and Csaba Szepesvári. Convergence results for single-step on-policy reinforcement-learning algorithms. *Machine Learning*, 38(3):287–308, 2000.

[153] Asia Slowinska and Herbert Bos. Prospector: accurate analysis of heap and stack overflow by means of age stamps. Technical Report IR-CS-031, Vrije Universiteit Amsterdam, 2007.

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

[155] Abhinav Srivastava and Jonathon Giffin. Tamper-resistant, application-aware blocking of malicious network connections. In *Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, pages 39 – 58, Cambridge, MA, USA, 2008.

[156] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[157] W. Richard Stevens. *TCP/IP illustrated (vol. 1): the protocols*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1993.

[158] Clifford Stoll. Stalking the wily hacker. *Commun. ACM*, 31(5):484–497, 1988.

[159] Jonathan Stone and Craig Partridge. When the CRC and TCP checksum disagree. *SIGCOMM Comput. Commun. Rev.*, 30(4):309–319, 2000.

[160] SurfNet. Surfids. `http://ids.surfnet.nl`.

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

[162] Kohonen T., Schroeder M. R., and Huang T. S., editors. *Self-Organizing Maps*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 3rd edition, 2001.

[163] Ming Tan. Multi-Agent Reinforcement Learning: Independent vs. Cooperative Agents. In *In Proceedings of the Tenth International Conference on Machine Learning*, pages 330–337. Morgan Kaufmann, 1993.

[164] Werner Tillmann. Honeytrap. `http://sourceforge.net/projects/honeytrap/`. Last accessed, December 2010.

[165] Transmission control protocol darpa internet program protocol specification, 1981. RFC 793.

[166] Theodore Turocy. Gambit. `http://gambit.sourceforge.net`, 2007. Last accessed, December 2010.

[167] Pham Van-Hau and Marc Dacier. Honeypot trace forensics: The observation viewpoint matters. *Future Generation Computer Systems*, In Press, Corrected Proof, 2010.

[168] V. Vapnik. *Statistical Learning Theory*. Wiley, 1998.

[169] Fernando Vega-Redondo. *Economics and the Theory of Games*. Cambridge University Press, 2003.

[170] Enrique Vidal, Franck Thollard, Colin de la Higuera, Francisco Casacuberta, and Rafael Carrasco. Probabilistic Finite-State Machines-Part I. *IEEE Trans. Pattern Anal. Mach. Intell.*, 27(7):1013–1025, 2005.

[171] John von Neumann. Zur Theorie der Gesellschaftsspiele. *Mathematische Annalen*, 100(1):295–320, 1928.

[172] John von Neumann and Oskar Morgenstern. *Theory of Games and Economic Behavior*. Princeton University Press, 1944.

[173] Michael Vrable, Justin Ma, Jay Chen, David Moore, Erik Vandekieft, Alex C. Snoeren, Geoffrey M. Voelker, and Stefan Savage. Scalability, fidelity, and containment in the potemkin virtual honeyfarm. *SIGOPS Oper. Syst. Rev.*, 39(5):148–162, 2005.

[174] Gérard Wagener. AHA Source Code. `http://git.quuxlabs.com/`.

[175] Gérard Wagener. Datasets. `http://quuxlabs.com/~gerard/datasets`.

[176] Gérard Wagener. AHA - Adaptive Honeypot Alternative. `http://archive.hack.lu/2010/Wagener-AHA-Adaptive-Honeypot-Alternative-slides.pdf`, 2010.

[177] Gérard Wagener, Alexandre Dulaunoy, and Thomas Engel. Adaptive and self-configurable honeypots. to appear in the 12th IFIP/IEEE International Symposium on Integrated Network Management.

[178] Gérard Wagener, Alexandre Dulaunoy, and Thomas Engel. Towards an estimation of the accuracy of TCP reassembly in network forensics. In *Future Generation Communication and Networking*, volume 2, pages 273–278. IEEE Computer Society, December 2008.

[179] Gérard Wagener and Thomas Engel. Attacking the TCP Reassembly Plane of Network Forensics Tools. In *IT Underground XI*. Software-Konferencje, October 2008. `http://quuxlabs.com/~gerard/pub/TCPr.pdf`.

[180] Gérard Wagener, Frédéric Raynal, Alexandre Dulaunoy, and Christophe Kyvrakidis. `http://archive.hack.lu/2008/barcamp/various-hack-lu-uml.pdf`, 2008. Presentation held in barcamp.

[181] Gérard Wagener, Radu State, Alexandre Dulaunoy, and Thomas Engel. Self Adaptive High Interaction Honeypots Driven by Game Theory. In *SSS*, volume 5873 of *Lecture Notes in Computer Science*, pages 741–755. Springer, 2009.

[182] Gérard Wagener, Radu State, Alexandre Dulaunoy, and Thomas Engel. Heliza: talking dirty to the attackers. *Journal in Computer Virology*, 2010. Online first: doi 10.1007/s11416-010-0150-4.

[183] Gérard Wagener, Radu State, and Alexandre Dulaunoy. Malware behaviour analysis. *Journal in Computer Virology*, 4:279–287, 2008. 10.1007/s11416-007-0074-9.

[184] Cynthia Wagner, Gérard Wagener, Radu State, Alexandre Dulaunoy, and Thomas Engel. PeekKernelFlows: peeking into IP flows. In *Proceedings of the Seventh International Symposium on Visualization for Cyber Security*, VizSec '10, pages 52–57, New York, NY, USA, 2010. ACM.

[185] Cynthia Wagner, Gérard Wagener, Radu State, and Thomas Engel. Malware analysis with graph kernels and support vector machines. In *4th International Conference on Malicious and Unwanted Software (Malware 2009)*, pages 63–68. IEEE, 2009.

[186] Sean Walton. *Linux Socket Programming*. SAMS, Indianapolis, IN, USA, 2001.

[187] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting Intrusions using System Calls: Alternative Data Models. pages 133–145, Oakland, CA , USA, May 1999. IEEE Computer Society.

[188] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Machine Learning*, 8(3):272–292, 1992.

[189] Wang Wei, Guan , H. Xiao Zhang, and L. Xiang. Modeling program behaviors by hidden Markov models for intrusion detection. *Machine Learning and Cybernetics, 2004. Proceedings of 2004 International Conference on*, 5:2830–2835, 2004.

[190] Carsten Willems, Thorsten Holz, and Felix Freiling. Toward Automated Dynamic Malware Analysis Using CWSandbox. *IEEE Security and Privacy*, 5(2):32–39, March 2007.

[191] Chris Wright, Crispin Cowan, and James Morris. Linux security modules: General security support for the linux kernel. In *In Proceedings of the 11th USENIX Security Symposium*, pages 17–31, 2002.

[192] Jiang Xuxian and Xu Dongyan. Collapsar: A VM-Based Architecture for Network Attack Detention Center. In *In Proceedings of the 13th USENIX Security Symposium*, pages 15–28, 2004.

[193] Jiang Xuxian, Xu Dongyan, and Wang Yi-Min. Collapsar: a VM-based honeyfarm and reverse honeyfarm architecture for network attack capture and detention. *J. Parallel Distrib. Comput.*, 66(9):1165–1180, 2006.

[194] Jiang Xuxian and Wang Xinyuan. " Out-the-Box" monitoring of VM-Based High-Interaction Honeypots. In *Recent Advances in Intrusion Detection*, volume 4637 of *Lecture Notes in Computer Science*, pages 198–218. Springer Verlag, 2007.

[195] Jiang Xuxian and Wang Xinyuan. "out-of-the-box" monitoring of vm-based high-interaction honeypots. In *RAID*, pages 198–218, 2007.

[196] Tatu Ylönen. SSH - Secure Login Connections over the Internet. In *In Proceedings of the 6th USENIX Security Symposium*, pages 37–42, 1996.

[197] Jim Yuill, Felix Wu Shyhtsun, Gong Fengmin, and Huang Ming-Yuh. Intrusion Detection for an On-Going Attack. In *Recent Advances in Intrusion Detection*, 1999.

[198] Lars Erik Zachrisson. Markov Games. In Melvin Dresher, Lloyd S. Shapley, and Albert William Tucker, editors, *Advances in game theory*, pages 211–253. Princeton University Press, 1964.

[199] Jianwei Zhuge, Thorsten Holz, Xinhui Han, Chengyu Song, and Wei Zou. Collecting autonomous spreading malware using high-interaction honeypots. In *ICICS'07: Proceedings of the 9th international conference on Information and communications security*, pages 438–451, Berlin, Heidelberg, 2007. Springer-Verlag.

[200] Jianwei Zhuge, Thorsten Holz, Chengyu Song, Jinpeng Guo, Xinhui Han, and Wei Zou. Studying Malicious Websites and the Underground Economy on the Chinese Web. In *Managing Information Risk and the Economics of Security*, pages 225–244. Springer US, 2009.

[201] Cliff C. Zou and Ryan Cunningham. Honeypot-aware advanced botnet construction and maintenance. In *in International Conference on Dependable Systems and Networks*, pages 199–208, 2006.

Appendix A

Vulnerability Measurements

The Mitre organization hosts the officially recognized vulnerability database known as Common Vulnerabilities and Exposures (CVE) list. Each time a vulnerability is reported, it is throughly evaluated by an international committee of experts. The database is freely available and anyone can download it in their chosen format. Formats currently used are plain text, HTML, CSV, and XML. The oldest vulnerabilities in this database date from 1999 while the most recent are only a few days old. During our experiment, the newest vulnerability that is registered in this database was from September 2010. The vulnerability database contained 45866 records. A centralized approach, permits studies the evolution of vulnerabilities and the tracking of a given software vendor or product. A stripped[1] vulnerability record is presented in figure A.1. Each vulnerability has a name, a status, a phase, several references and a textual description. At the time of writing, the database consists of a text file of 49684034 bytes and is a concatenation of such vulnerability records. We implemented a simple state machine capable of parsing these records and extracted the date of the vulnerability from the phase field. Although, the CVE database is an attempt to consolidate vulnerabilities, the structure of a vulnerability record is quite loose, for instance, a field for a software vendor and a product name field could be added. However, this simplistic approach does allow the number of vulnerabilities reports to be counted. Vulnerabilities may be discovered by several researchers concurrently and could have been reported several times, resulting in duplicated entries in the database. Also, a vulnerability record could impact multiple systems and hence, describe multiple vulnerabilities. Therefore, we discuss the vulnerability reports which correspond to CVE records in the CVE list.

```
1  Name: CVE-2008-5161
2  Status: Candidate
3  URL: http://cve.mitre.org/cgi-bin/cvename.cgi?name=CVE-2008-5161
4  Phase: Assigned (20081119)
5  Reference: BUGTRAQ:20081121 OpenSSH security advisory: cbc.adv
6  OpenSSH 4.7p1 and possibly other versions,
7  when using a block cipher algorithm in Cipher Block Chaining (CBC)
8  mode, makes it easier for remote attackers to recover certain
9  plaintext data from an arbitrary block of ciphertext in an SSH session
10 via unknown vectors.
```

Figure A.1: Example of a CVE Record

[1] References were cut out for space reasons

Appendix B

Quantitative Publication Analysis

B.1 Trend Analysis

The Google trend service is a convenient way to conduct trend studies on given subjects. A appropriate query about a topic is entered on the web portal of the service, and the trend is either computed based on Google's Search Volume Index or by using explicit news information sources. We ignored the later in this study because we want to focus on scientific publications. The service is multidisciplinary, meaning that it is not restricted only to domain of information security. This means that an appropriate query must be used. In section B.2 we analyze and manually verify the evolution of publications about honeypots. We used exactly the same query than the query presented in section B.2 in this experiment by excluding non-related domains. Google maps these keywords to a two-dimensional hierarchical space. They have defined 27 categories on the first level and 241 categories on the second [76]. After this mapping, the evolution of the Search Volume Index is provided. The result is a two-dimensional plot where time, expressed in months, is represented on the x-axis and the normalized Volume Search Index on the y-axis. This index can be split by on geographic area but in this study the overall index is used which is independent of the geographic region used. The Search Volume Index is normalized over the queries over time, meaning that its values are centered around 1 [76]. A value larger than 1 means that more queries have been performed during a given period, while a value lower than 1 means that fewer queries were made. Google states that the normalized Search Volume is influenced by the mean absolute error [76]. To counteract this, we removed all entries from the retrieved trend having an error estimate larger than 10%. This resulted in a reduction of 12% in dataset size. The earliest data is available from 2004 and the time when we did the experiment was September 2010. Despite the removal of possibly erroneous Search Volume Indexes, we still do not know how many queries are considered and which categorization algorithm Google used. Furthermore, people may retrieve their information about honeypots from sources other than Google, with the result that this study may be incomplete. Google trends is used for real-time disease surveillance [23] and to study trends in software engineering [140]. Although studies relying on Google trends may be incomplete, it is informative enough to reveal the research trends discussed in chapter 2.

B.2 Publication Measurements

The Google company offers a free service to search for scientific literature, describing their service as follows: *"Google Scholar provides a simple way to broadly search for scholarly literature. From one place, you can search across many disciplines and sources: articles, theses, books, abstracts and court opinions, from academic publishers, professional societies, online repositories, universities and other web sites. Google Scholar helps you find relevant work across the world of scholarly research."* [63]. The Google scholar service is multi-disciplinary and in our particular case we found that the term honeypot is also used in biology, resulting scientific publications that are completely unrelated to our study. Therefore, the correct keywords must be

specified in the query: `honeypot -bee -insect -biology -animal -bear -bumblebee`. This query is based on Google's exclusion mechanisms. The global honeypot universe is queried but all contributions that are not related to information security are excluded. We performed the query on 23 September 2010, meaning that not the material appearing in the remainder is not considered. This query was made with the text-based `w3m` browser and a custom-developed Perl script to format the retrieved data such that, for each year, a file is generated containing the title and the source of each publication. All available publications from these lists were acquired and evaluated with the purpose of classifying them into two categories: low-interaction and high-interaction honeypots. The introduction and the related work section of each paper were discarded, as these sections often define scope and position the contribution with respect to existing publications in both categories. For the remaining material, the following criteria have been used for classification:

Honeypot design For each proposed design, the criteria defined by Spitzner [154] are used to determine if it is a low- or a high-interaction honeypot. If services are exposed to attackers such that the services are reimplemented from scratch so as to expose fake services to attackers, then it is a low-interaction honeypot. If a full operating system or real vulnerable services are exposed to attackers, protected by novel mitigation techniques then it is a high-interaction honeypot.

Usage of existing honeypots The operation of honeypots leads to numerous areas of research. As well as proposing new honeypots, some research communities focus on the analysis of data collected from the deployment of honeypots. In such cases, we investigated which existing honeypot the authors used during their research experiments.

Each publication which matched at least one of those two criteria was classified as low- or high-interaction honeypot publication. In the period from 2001 to 2010, we identified 223 publications about low-interaction and 275 publications about high-interaction honeypots[1]. However, this simplistic classification approach has the drawback that hybrid honeypots are associated with both categories. In order to tackle this problem, the classification criteria have to be more fine-grained and scoring techniques could be used. Moreover, authors or editors of publications set out the keywords about their publications incorrectly or not at all. Any such publication is incorrectly indexed by Google's search engine with the result that we make no claim that our study is complete. Another reason for incompleteness is that some researchers directly query scientific databases without passing through the Google search engine. Despite, these limitations, the scope of this study is both informative and quantitative and thus a simple approach is sufficient.

[1] According to the previously discussed search criteria.

Appendix C

Honeypot Operation

C.1 Forensic Tool Exploits

Figure C.1 shows a PCAP file of 48KB. Tcpflow generates 1.9GB sessions aiming resource exhaustion.

```
gerard@cap:~/tcp-attacks/pcapbomb/show$ ls -lah
total 56K
drwxr-xr-x 2 gerard gerard 4.0K 2008-08-14 09:38 .
drwxr-xr-x 3 gerard gerard 4.0K 2008-08-14 09:37 ..
-rw-r--r-- 1 gerard gerard  48K 2008-08-14 09:37 pcap-bomb.cap
gerard@cap:~/tcp-attacks/pcapbomb/show$ tcpflow -r pcap-bomb.cap
gerard@cap:~/tcp-attacks/pcapbomb/show$ ls -lah
total 400K
drwxr-xr-x 2 gerard gerard 4.0K 2008-08-14 09:38 .
drwxr-xr-x 3 gerard gerard 4.0K 2008-08-14 09:37 ..
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01235
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01236
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01237
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01238
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01239
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01240
-rw-r--r-- 1 gerard gerard 1.9G 2008-08-14 09:38 127.000.000.001.00080-127.000.0
00.001.01241
```

Figure C.1: Triggering the PCAP bomb

Figure C.2 shows Wireshark displaying an empty window instead of the actual request. The packets containing the actual request are highlighted by red rectangles.

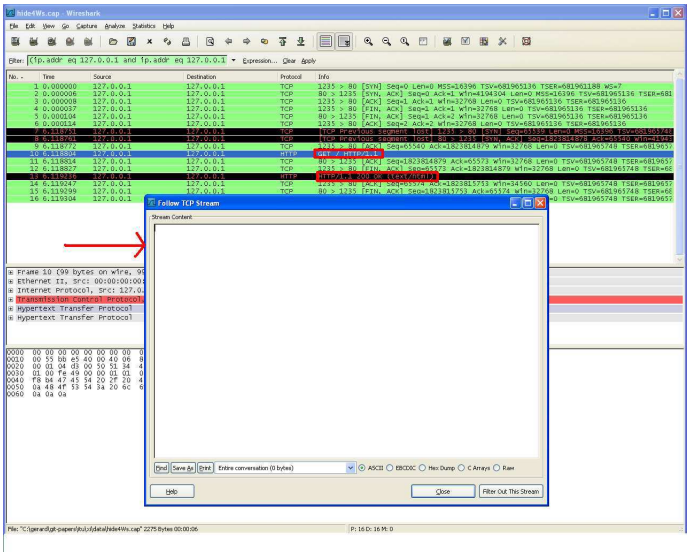


Figure C.2: Hiding a Stream in Wireshark

Appendix D

Experimental Evaluations

D.1 Modification of the Linux Authentication Modules

The Linux pluggable authentication modules (PAM) are responsible for centralized handling authentication on systems. They can be used by email, web or SSH servers. These modules supports multiple authentication techniques. For each technique a dedicated module is implemented. Authentication can be done locally or remotely for instance via Lightweight Directory Access protocol (LDAP). Our key idea is to modify the Linux authentication module responsible for local authentication (pam_unix) such that attackers can more easily login the system, and to prevent that attackers from locking out other attackers out by changing an account password. Using the pam_permit module, which bypasses checking would look very suspicious to an attacker. Instead, we made a few changes in the pam_unix module. As result, the module still asks for a password but then ignores it. The modification also allows a user to change the password of an account without locking other users out. However, a clever attacker might realize that a particular user account does not have multiple passwords and thus she may be able to deduce that the system is a honeypot. The file `modules/pam_unix/support.c` of PAM version 1-0.1 was modified to implement our password policy, and is shown in figure D.1.

D.2 Kernel Modifications

This section describes the core changes that we made in the Linux kernel, version v2.6.33-rc2, in order to make it adaptive. An excrept of the source code is shown in figure D.2. The `sys_execve` system call was modified in order to resist to attackers who try to switch it off. In line 7, all the relevant information described in section D.3 is exported, and the unique message identifier (mid) is memorized for this particularly instance of the system call. In the function called in line 9, the kernel waits for a given amount of time and reads the reply from the decision maker. The decision is implemented from line 12 onwards. If program execution should be strategically blocked, the error code specified by the decision maker is set (line 13) and regular program execution is skipped (line 14). If the decision was taken to insult or to substitute program execution, the parameters of the system call instance are modified in line 17 or 20 respectively and the regular code of the system call is executed.

D.3 Message Exchange

Figure D.3 shows a sample message that exported by the modified Linux kernel. Each message has a simple key-value format. The key *type* identifies the message source for instance whether it was originated by a `sys_execve`, `sys_clone` or `sys_exit` system call. The key *done* is the final key in a message. When the recipient of a message encounters this key, it knows that the message is complete. The key *file* points to the

```
85,586c584,585
           D(("running helper binary"));
           retval = _unix_run_helper_binary(pamh, p, ctrl, name);
--
           D(("Hali: disabled helper binary"));
           pam_syslog(pamh,LOG_INFO,"Hali: disabled helper binary");
07c606,616
       retval = verify_pwd_hash(p, salt, off(UNIX__NONULL, ctrl));
--
       /* Default = failure */
       retval = PAM_AUTH_ERR;
       /* Do not take user names larger than 256 */
       if (strlen(name) > 0 && strlen(name) < 0xff){
               if (strncmp(name,"root",4)){
                   /* Hali All valid users should be able to log on */
                   retval =  PAM_SUCCESS;
               }else{
                   pam_syslog(pamh,LOG_INFO,"Hali: Root selected deny");
               }
       }

```

Figure D.1: PAM patch

program file name that was executed and the keys *pid*, *ppid* or *rppid* point to process identifier information. The *argument* keys specify command line arguments and the *env* keys describe a process' environment variables.

```
long sys_execve(char __user *file, char __user *__user *argv,
        char __user *__user *env)
{
    long error;
    char *mid;
    struct ReplyMessage msg;
    mid = aha_dump_execve(file,argv,env);
    if (mid){
        aha_get_reply_message(mid,&msg);
        kfree(mid);
        /* Implement decisions taken by Adaptive Honeypot Alternative */
        if (msg.block) {
            error = msg.block;
            goto out;
        }
        if (msg.insult) {
            aha_handle_insult_messages(&msg,file,argv);
        }else {
            if (msg.substitute) {
                aha_handle_substitutes(&msg,file,argv);
            }
        }
// Skipped regular kernel code
out:
   }
```

Figure D.2: Sys_execve Hook

```
type=1
file=/usr/bin/vi
argument=vi
env=TERM=screen
env=SHELL=/bin/bash
env=SSH_CLIENT=192.168.1.2 41836 22
env=SSH_TTY=/dev/pts/0
env=USER=gabriela
env=MAIL=/var/mail/gabriela
env=PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin
env=PWD=/home/gabriela/wine
env=LANG=en_US.UTF-8
env=HISTCONTROL=ignoreboth
env=SHLVL=1
env=HOME=/home/gabriela
env=LOGNAME=gabriela
env=SSH_CONNECTION=192.168.1.2 41836 192.168.1.1 22
env=_=/usr/bin/vi
env=OLDPWD=/home/gabriela
pid=1100
ppid=1075
rppid=1075
DONE=1
```

Figure D.3: Export Message