Ehab Al-Shaer · Jinpeng Wei
Kevin W. Hamlen · Cliff Wang *Editors*

Autonomous Cyber Deception

Reasoning, Adaptive Planning, and Evaluation of HoneyThings

Springer

Autonomous Cyber Deception

Ehab Al-Shaer • Jinpeng Wei
Kevin W. Hamlen • Cliff Wang
Editors

Autonomous Cyber Deception

Reasoning, Adaptive Planning, and Evaluation of HoneyThings

Springer

Editors
Ehab Al-Shaer
Department of Software
& Information System
University of North Carolina Charlotte
Charlotte, NC, USA

Kevin W. Hamlen
Computer Science Department
University of Texas at Dallas
Richardson, TX, USA

Jinpeng Wei
Department of Software
and Information System
University of North Carolina
Charlotte, NC, USA

Cliff Wang
Computing and Information
Science Division
Army Research Office
Durham, NC, USA

ISBN 978-3-030-02109-2 ISBN 978-3-030-02110-8 (eBook)
https://doi.org/10.1007/978-3-030-02110-8

Library of Congress Control Number: 2018959750

This Springer imprint is published by the registered company Springer Nature Switzerland AG
The registered company address is: Gewerbestrasse 11, 6330 Cham, Switzerland

Preface

Why Cyber Deception? Cyberattacks have evolved to be highly evasive against traditional prevention and detection techniques, such as antivirus, perimeter firewalls, and intrusion detection systems. At least 360,000 new malicious files were detected every day, and one ransomware attack was reported every 40 s in 2017 (Chap. 10). An estimated 69% of breaches go undetected by victims but are spotted by an external party, and 66% of breaches remained undiscovered for more than 5 months (Chap. 10). Asymmetries between attacker and defender information and resources are often identified as root causes behind many of these alarming statistics. Cybercriminals frequently reconnoiter and probe victim defenses for days or years prior to mounting attacks, whereas defenders may only have minutes or seconds to respond to each newly emerging threat. Defenders seek to protect infrastructures consisting of thousands or millions of assets, whereas attackers can often leak sensitive information or conduct sabotage by penetrating just one critical asset. Finding ways to level these ubiquitous asymmetries has therefore become one of the central challenges of the digital age.

What Is Cyber Deception? Cyber deception has emerged as an effective and complementary defense technique to overcome asymmetry challenges faced by traditional detection and prevention strategies. Approaches in this domain deliberately introduce misinformation or misleading functionality into cyberspace in order to trick adversaries in ways that render attacks ineffective or infeasible. These reciprocal asymmetries pose scalability problems for attackers similar to the ones traditionally faced by defenders, thereby leveling the battlefield.

Cyber Deception Models Cyber deception can be accomplished in two major ways: (1) *mutation*, to frequently change the ground truth (i.e., the real value) of cyber parameters such as cyber configuration, IP addresses, file names, and URLs, and (2) *misrepresentation*, to change or corrupt only the value returned of cyber parameters to the attacker without changing the ground truth such as false fingerprinting, files, and decoy services. We therefore call the cyber parameters used

for deceiving the attackers HoneyThings. Using the concept of HoneyThings in both approaches expands the cyber exploration space for adversaries to launch effective attacks.

Cyber Deception 4D Goals Effective cyber deception aims to (1) *deflect* adversaries away from their goals by disrupting their progress through the kill chain; (2) *distort* adversaries' perception of their environment by introducing doubt into the efficacy of their attacks; (3) *deplete* their financial, computing, and cognitive resources to induce biased and error-prone decisions that defenders can influence; and (4) *discover* unknown vulnerabilities and new TTPs (tactics, techniques, and procedures) of adversaries while predicting the tactical and strategical intents of adversaries.

Book Overview In light of this vision, this book brings together recent research results pursuant to these goals, in four major parts:

Part I addresses in developing Cyber Deception Reasoning Frameworks and consists of three chapters:

- Chapter 1 presents a framework that uses deep learning and differential privacy techniques to generate deceptive data that is hard to differentiate from real data.
- Chapter 2 presents a framework and a research prototype for intelligent cyber deception agents that can make autonomous decisions on how to counter ongoing attacks and that integrate with active defense tools.
- Chapter 3 studies how honeypot deception can be made more effective when applied with variety and discusses the range of deception tactics that can be considered, such as random error messages, honey files with some convincing real data, and out-of-date vehicle positions.

Part II is about Dynamic Decision-Making for Cyber Deception, and it consists of two chapters:

- Chapter 4 models cyber deception as a hypergame in which attackers and defenders can have different perceptions toward a given situation and carries out case studies to examine how players' perception (or misperception) affects their decision-making in choosing a best strategy based on hypergame theory.
- Chapter 5 applies a series of game theory models to capture the strategic interactions between attackers and defenders, the multistage persistence, as well as the adversarial and defensive cyber deceptions.

Part III examines new approaches for network-based deception, spanning four chapters:

- Chapter 6 presents a new cyber deception framework that composes mutation, anonymity, and diversity to maximize key deception objectives (i.e., concealability, detectability, and deterrence) while constraining the overall deployment cost.
- Chapter 7 presents a highly dynamic network obfuscation and deception solution that overcomes limitations of existing solutions. Specifically, it mutates and

randomizes multiple aspects of network configurations simultaneously, leveraging network and host-level SDN, state-of-the-art virtualization techniques, and DNS deception.
- Chapter 8 examines how deceptive web service responses can be realized as software security patches that double as feature extraction engines for a network-level intrusion detection system, which can increase detection accuracy and adaptability due to the fast, automatic, and accurate labeling of live web data streams enabled by this approach.
- Chapter 9 presents a technique to contain the risks of compromising buggy IoT devices by creating a protection layer on top of the local network and providing fine-grained control over the communications of individual IoT devices in the network. It uses software-defined networking (SDN) technologies to realize device- and device-group-specific views of the network that reduce the attack surface against vulnerable devices in the network, contain effects of device infections in case of device compromise, and enforce effective measures for blocking unwanted release of contextual data.

Finally, Part IV discussed automated techniques for deceiving malware, consisting of two chapters:

- Chapter 10 presents a new analytics framework and tool that can analyze the malware binary and automatically extract *deception parameters* in order to enable the automated creation of cyber deception plans.
- Chapter 11 proposes a system for automatically extracting the system resource constraints from malware code and generating HoneyResource (e.g., malware vaccines) based on the system resource conditions.

In each book chapter, theoretical and experimental exercises for researchers and students are presented to deepen the understanding of the deception concepts and techniques presented in this book.

The investigations, discoveries, and experiences reported in these recent results open many potential avenues for future work. We recommend the following research directions:

- **Autonomy and Resiliency:** The high speed and complexity of modern cyberattacks demands cyber deceptions that are highly *autonomous* to be self-adaptive, and *resilient* to survive failures that might reveal deception assets or plans. Methods are needed that can clearly specify both the defender's mission and the attacker's mission and that can identify the theoretical foundations, objectives, levels, and risks of automation.
- **Modularity and Interdisciplinarity:** The innovation and deployment of cyber deception requires scientific and engineering foundations that make HoneyThings into plug-and-play commodities that are easy to instantiate, integrate, deploy, adapt, and maintain. Efforts are needed to close the presently large gap between engineering constraints and psychology theory. Mechanisms are also needed for sharing cyber deception data that can be used to study the psychological and cognitive influence of cyber deception on the adversary.

- **Quantitative Evaluation:** To make measurable scientific progress toward effective, deployable, deception-powered cyber defenses, the scientific community must establish accepted metrics and methodologies for evaluating proposed cyber deception techniques, frameworks, and systems. Such evaluations must go beyond mere anecdotal observations of effectiveness to obtain experimental results that are systematic, comprehensive, reproducible, and statistically valid and that afford "apples-to-apples" comparisons of competing research ideas.

Charlotte, NC, USA — Ehab Al-Shaer
Charlotte, NC, USA — Jinpeng Wei
Richardson, TX, USA — Kevin W. Hamlen
Durham, NC, USA — Cliff Wang
August 2018

The original version of this book was revised: Chapter authors have been added. The correction to this book is available at https://doi.org/10.1007/978-3-030-02110-8_12

Acknowledgments

We would like to thank the United States Army Research Office (ARO) for sponsoring the 2018 Workshop on HoneyThings: Autonomous and Resilient Cyber Deception, which created an opportunity for many interesting discussions about hot topics on autonomous cyber deception. We thank all authors of the chapters in this book for sharing their latest and greatest research results.

Finally, the editorial team thanks UNC Charlotte PhD students Md Sajidul Islam Sajid and Mohiuddin Ahmed for their help in preparing the final version of this book.

Contents

Part I
Cyber Deception Reasoning Frameworks

Chapter 1
Using Deep Learning to Generate Relational HoneyData

Nazmiye Ceren Abay, Cuneyt Gurcan Akcora, Yan Zhou, Murat Kantarcioglu, and Bhavani Thuraisingham

Abstract Although there has been a plethora of work in generating deceptive applications, generating deceptive data that can easily fool attackers received very little attention. In this book chapter, we discuss our secure deceptive data generation framework that makes it hard for an attacker to distinguish between the real versus deceptive data. Especially, we discuss how to generate such deceptive data using deep learning and differential privacy techniques. In addition, we discuss our formal evaluation framework.

Keywords Cyber deception · Differential privacy · Deep learning · Decoy deployment

1.1 Introduction

Deception techniques have been recently deployed in cybersecurity for achieving many important goals ranging from understanding the attacker intent to tricking attackers into spending resources and time on fake targets. For example, honeypots (e.g., [29]) have been proposed to provide deceptive targets (i.e., servers) for attackers. Similarly, Honeyfiles (e.g., [31]) have been proposed to lure attackers to spend time in searching files and potentially disclose their intent. Still, to our knowledge, none of the previous work tries to create deceptive "data" (i.e., HoneyData) to fool potential attackers. Unfortunately, lack of realistic deceptive data may make it easier for an attacker to detect deception. For example, without good HoneyData, it may be easier to spot a fake database hosted on a honeypot.

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

N. C. Abay · C. G. Akcora · Y. Zhou · M. Kantarcioglu (✉) · B. Thuraisingham
The University of Texas at Dallas, Richardson, TX, USA
e-mail: muratk@utdallas.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_1

Recent work looked into generating privacy-preserving synthetic relational data using differential privacy (e.g., [2]). The main purpose of this line of work is to preserve individual privacy while providing data utility. Therefore, it is not clear whether they could be applicable for generating good HoneyData. In the context of cyber deception, it is important that the HoneyData is indistinguishable from real data so that it can easily fool the attacker.

Creating deceptive data (i.e., HoneyData) has many challenges. For different settings, we may need different types of HoneyData. For example, to deceive an attacker and feed false information, deceptive technical plans (e.g., technical drawings of an airplane) could be generated. On the other hand, to make HoneyFiles more believable, fake text data could be added to such files. Since addressing all these different types of data requires different techniques, in this work, we focus on generating deceptive HoneyData that is relational data. The main differentiating factor for relational data is that the number of columns and the types of the columns in a given dataset are known in advance. Still, generating realistic relational HoneyData while not disclosing sensitive information is a significant challenge.

We need to answer questions, such as: (1) how to automatically generate relational HoneyData? and (2) how to measure whether the generated relational HoneyData is deceptive enough? In this work, we try to answer these questions by leveraging existing work in differentially private synthetic data generation and explore its effectiveness for generating relational HoneyData.

As a part of this work, we propose an important measure for understanding the effectiveness of HoneyData. Basically, given the available information, a potential attacker may not build an effective machine learning model to distinguish between real vs HoneyData. We evaluate the effectiveness of relational HoneyData on real datasets, and show under what conditions differentially private deep learning techniques could be used to generate relational HoneyData.

Remainder of this book chapter is organized as follows: Sect. 1.2 details related work. Section 1.3 gives preliminaries for our data generation technique in Deep learning and privacy preserving, and Sect. 1.4 explains our methodology. We report our experimental results in Sect. 1.5 and conclude with Sect. 1.6.

1.2 Related Work

Cyber deception mechanisms have been heavily studied to enhance the computer security. However, most of the existing techniques are not focused on generating deceptive data. Here, we review the existing cyber deception techniques with their limitations and strengths.

Honeypots are a prominent cyber deception mechanism to investigate and analyze the unauthorized intrusions [29]. Honeypots are designed as trap-based isolated systems that appear vulnerable to attackers. Legitimate users are not supposed to interact with them and any interaction with honeypots is considered an illicit attempt. While interacting with intruders, honeypots gather information of

them to disclose intruders' behavior for forensic analysis. Although honeypots are a notable cyber deception technique, they have limitations. Since honeypots are *fake environments*, they might fail to simulate the real services. As attackers become more sophisticated, they ensure their safety by using more advanced systems to distinguish "fake" and real system to avoid honeypots [18]. Moreover, honeypots might create irredeemable risks for the real user environment when the attacker can use honeypots as a bridge to the real user environment [4].

In addition to Honeypots, decoy injection mechanisms evolved to integrate real systems in aiding defensive computer deception. These mechanisms serve as a *decoy* to intruders to mitigate unauthorized threats by distracting attackers from a target that has sensitive information.

Yuill et al. [31] present an intrusion detection system that installs *decoy files* on file servers with enticing names to capture the attention of attackers. These decoy files are constantly monitored and when accessed by any intruder, the system will trigger an alarm to notify system administrator. However, in some cases, decoy files fail to influence the perception of attackers since published data (e.g., password file stolen from LinkedIn[1]) provide attackers insight to distinguish between real and fabricated data. They can enhance their technique and re-attack again. To circumvent attacker insight, Juels et al. [19] propose *Honeywords* to defend hashed password databases by generating "fake passwords" that seem real to attackers. In their work, they preserve N-1 "fake passwords" referred as honeywords for each legitimate user password in the database. If any of the honeywords is submitted for logging into databases, attack has been detected and system administrator is notified that database has been hacked. Although honeywords are useful to detect the unauthorized intruders, in some cases it may deteriorate system performance because each submitted password is compared with all previously generated honeywords which slows down the authentication process for legitimate users. Also, generating and preserving the honeywords increases the storage requirement N times. Still, this approach is only applicable for password setting.

Our approach proposes decoy data generation to fool attackers without degrading system performance. Although decoy files are used in a cyber defensive system to entice attackers, they may reveal sensitive information if care is not taken during data generation. To preserve individual privacy, the decoy files require sanitization of sensitive information. Dwork [10] proposes a data privacy model as ε-*differential privacy* to ensure the protection of private data from leakage by perturbing the data with random noise based on ε. Differential privacy has been implemented in a number of data analysis tasks, including regression models [9, 33], classification models [26, 30], and privacy-preserving data publishing [3, 6, 32]. In some cases, it is required to combine differentially private algorithms to formulate complex privacy solutions. To track the total privacy loss while executing these repetitive mechanisms, Abadi et al. [1] propose the advanced composition theorem known as

[1] https://www.cnet.com/news/linkedin-confirms-passwords-were-compromised/.

the *moment accountant* and verify that it has the best overall privacy bound in the literature. In this work, we also employ the moment accountant to bound privacy of the proposed technique to generate decoy files.

To balance both utility and user privacy, Rubin [25] introduces repetitive perturbation of the original data as a substitute to the original data. However, data generation may suffer from curse of dimensionality when the data has more than dozen attributes. To overcome the curse of dimensionality, Zhang et al. [32] present PRIVBAYES as a private generative model that decomposes high-dimensional data into low-dimensional marginals by constructing a Bayesian network. Afterwards, noise is injected into previously constructed low-dimensional marginals to ensure differential privacy and the synthetic data is inferred from these sanitized marginals. Acs et al. [3] model another generative approach to produce synthetic samples. First, the original data is partitioned into k clusters with private kernel k-means. Then, each previously clustered data is inputted to private generative neural networks to create synthetic data.

Park et al. [23] propose DPEM as a private version of the iterative expectation maximization algorithm. They combine differential privacy and expectation maximization algorithm to cluster datasets. Here, we use this approach to discover patterns in latent space. We observed an improvement in the performance of this technique when used with partitioning the original dataset into unique data label groups. Here, we use this modified version in our experiments [23] as DPEM^{+} and compare its results in the experiments section.

Similar to the clustering approach, Abay et al. [2] propose a new generative deep learning method that produces synthetic data from a dataset while preserving the utility of the original dataset. In [2], the original data is partitioned into groups, and then the private auto-encoder (a type of deep learning model) is employed for each group. Auto-encoder learns the latent structure of each group and uses expectation maximization algorithm to simulate them. In this work, we employ the same data generation model but we explore whether these techniques are applicable in the context of generating relational honeydata.

1.3 Background

This section provides a summary of deep learning and the principles of the differential privacy. Deep learning is utilized for the proposed approach. Differential privacy is applied to deep learning model to construct the private generative model to prevent the disclosure of sensitive data while generating honeydata.

1.3.1 Deep Learning

Deep learning is a representation learning-based machine learning technique that has been applied to image recognition and natural language processing where they have resulted in remarkable advances. The power of deep learning is based on

learning hierarchical concepts that allows the model to build complex concepts from the simpler ones [16]. Deep learning can be employed for addressing supervised, semi-supervised, or unsupervised tasks. Here, we employ unsupervised deep learning to form a generative neural network that samples honeydata.

Most deep learning models create complex networks that are formed with multilayer architectures. This multilayer network is a parametrized function that aims to fit any given input. To get optimal parameters to generalize input structure, our aim is to minimize the mismatching of error in the input, defined as loss function $\mathscr{L}(\theta)$, where θ is the set of network parameters. For each step in the optimization process, θ is updated with its gradient as follows:

$$\theta_{t+1} = \theta_t - \alpha \left(\frac{1}{|n|} \sum_{x_i \in D} \nabla_\theta \, \mathscr{L}(\theta; x_i) \right), \tag{1.1}$$

where D is the dataset with n records $x_i \in \mathbb{R}^d$. Deep learning models have complex networks usually formed with multilayer architectures that hinder the optimization. To circumvent this obstacle, Stochastic gradient descent (SGD) is used in the optimization [28].

1.3.2 Differential Privacy

Differential privacy is a mathematical formula that ensures privacy even if adversary has background knowledge [10]. Differential privacy adds random noise to the aggregated statistics to hinder impersonation attacks.

Theorem 1.1 *Mechanism $\mathscr{M}$ is a randomized real-valued function that satisfies (ε, δ) -differential privacy for some $\varepsilon > 0$ and $\delta > 0$ if for any adjacent datasets d, d', and for any subset of the output $S \subseteq Range(\mathscr{M})$ it holds that*

$$Pr[\mathscr{M}(d) \in S] \leq \exp(\varepsilon) \, Pr[\mathscr{M}(d') \in S] + \delta. \tag{1.2}$$

Adjacent datasets d, d' differ only in one tuple while rest is the same.

Mechanism $\mathscr{M}$ achieves (ε, δ) -*differential privacy* by perturbing the deterministic real-valued function f with random noise defined as follows:

$$\mathscr{M}(d) = f(d) + z, \tag{1.3}$$

where z is generated randomly from zero-mean Gaussian mechanism. Here, standard deviation of Gaussian mechanism is calibrated with σ and f's sensitivity s_f defined by the maximum of the absolute distance $\|f(d) - f(d')\|$ where d and d' are adjacent datasets. Relation among (ε, δ), σ, and s_f in Gaussian mechanism is given as $\sigma^2 \varepsilon^2 \geqslant 2 \ln 1.25/\delta s_f^2$ [13].

1.3.3 Differentially Private Composition Theorem

To analyze the privacy budget of our proposed work, we employ both sequential composition [11, 12] and advanced composition theorems [1, 8].

In our proposed work, while training the auto-encoder, we track the privacy loss at the end of each batch iteration. In the optimization phase, value of the current privacy loss ε' that has been spent on the private auto-encoder in a given iteration $t \in T$ is computed. Training ends when ε' reaches the final privacy budget ε.

According to moments accountant [1], deep learning network is (ε, δ)-differentially private if the privacy loss for any $\varepsilon' < k_1(|B|/n)^2 T$ is such that for some constants k_1, k_2:

$$\varepsilon' \geq k_2 \frac{|B|/n\sqrt{T \log 1/\delta}}{\sigma},$$

where T is the number of training steps and $|B|$ is the number of samples in mini-batch with a given privacy budget ε, delta δ, and standard deviation σ of the zero-mean Gaussian distribution.

1.4 Methodology

This section describes the details of our Differentially Private Synthetic Data Generation Model (DPSYN). We introduce the main algorithm and components of DPSYN.

1.4.1 Differentially Private Synthetic Data Generation Model

DPSYN has the primary purpose of generating synthetic data that is indistinguishable from the real data from the attacker's perspective given background knowledge. DPSYN also preserves the privacy by bounding the privacy loss with differential privacy. Abadi et al. [1] apply the moment accountant on differentially private deep learning. Here, we make several modifications to this work and extend it as a data generative model.

Figure 1.1 shows the fundamental steps of DPSYN. The dataset D contains a sequence of n training examples $(x_1, y_1), \ldots, (x_m, y_m)$ where $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$. Our learning approach partitions the dataset D into k groups denoted as $\{D_1, \ldots, D_k\}$. Partitioning of training examples is employed based on label $y \in \mathbb{R}$ associated with training example $x \in \mathbb{R}^d$. Group number k is identified by the unique label number. After partitioning the dataset into k groups $\{D_1, \ldots, D_k\}$, for each group private generative auto-encoder is constructed to generate synthetic data.

Algorithm 1 demonstrates the details of the proposed approach. The dataset that has sensitive information D is partitioned into k groups (Line 1) and those previously partitioned groups are used to construct the private generative auto-encoder (Line 4). This process is detailed later in Algorithm 2. Next, we obtain the private latent representation of the group (Line 5) with activation function $\mathbb{F}$ and inject it into a differentially private expectation maximization (DPEM) function. The DPEM function is detailed in [23]. The main task of DPEM is to detect different latent patterns in the encoded data and to generate output data with similar patterns. These patterns are decoded in Line 7, and appended to the synthetic data D'' (Line 8).

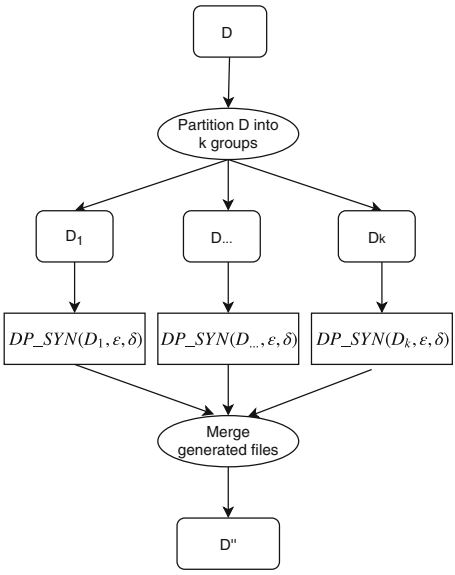

Fig. 1.1 Differentially private Synthetic data generation, DPSYN

Algorithm 1 DPSYN: Differentially Private Synthetic Data Generation

Require: D: $\{x_i, y_i\}_{i=1}^{m}$ where $x \in \mathbb{R}^d$ and $y \in \mathbb{R}$, α: learning rate; T: iteration number; ε: privacy budget; δ: Gaussian delta; σ: standard deviation; C: clipping constant.

$\{D_1 \dots D_k\} \leftarrow$ partition D into k groups
$D'' \leftarrow \{\}$
for $i \leftarrow 1$ **to** k **do**
  $\theta \leftarrow$ DP-Auto $(D_i, \alpha, \mathrm{T}, \varepsilon/2, \delta/2, \sigma, \mathrm{C})$ // see Algorithm 2
  $E' \leftarrow \mathscr{F}\,(X_i \cdot \theta)$ where $X_i \in D_i$
  $E'' \leftarrow$ DPEM $\left(E', \varepsilon/2, \delta/2\right)$ // see DPEM [23]
  ${D_i}' \leftarrow \mathscr{F}\left(E'' \cdot \theta^{\mathsf{T}}\right)$
  $D'' \leftarrow D'' \cup {D_i}'$
end
return D''

Algorithm 2 demonstrates the details of the DP-Auto model. Our private auto-encoder employs steps to improve the optimization process with gradient computation and clipping. While a gradient is computed for a batch in the standard stochastic training techniques, we compute the gradient for each training instance instead. This approach improves the optimization process since it reduces the sensitivity of the gradient present at each instance [15]. Norms of the gradients define the direction that optimizes the network parameters. However, in some deep networks, the gradients can be unstable and fluctuate in a large range. Such fluctuations can inhibit the learning process due to the increased vulnerability of the networks. To avoid this undesired situation, we bound norms of the previously computed gradients by a clipping constant C [24].

After clipping the gradients, noise is sampled from the Gaussian distribution with zero mean and standard deviation of σC and added to the previously clipped gradients (Line 8 in Algorithm 2). While training the auto-encoder, we track the privacy loss at the end of each batch iteration. As given in lines 2—2, we compute the value of current privacy loss ε' that has been spent on private auto-encoder in a given iteration $t \in T$. Training ends when ε' reaches the final privacy budget ε. If current privacy budget ε' is less than the final privacy budget ε, model parameters of the network are updated with the negative direction of the learning rate η multiplied by the averaged noisy gradients (Line 2 in Algorithm 2). And, current privacy budget ε' is updated by moments accountant technique in Line 2 in Algorithm 2. At the end of this step, the private auto-encoder outputs the model parameter θ based on final privacy budget ε (Line 2 in Algorithm 2).

Algorithm 2 DP-Auto: Differentially private auto-encoder

Require: α: Learning rate; T: iteration number; ε: privacy budget; δ: Gaussian delta; σ: standard deviation; C: clipping constant. $\mathscr{L}$ is the objective function

$\nabla\mathscr{L}$ is the gradient of objective function

initialize θ_0 randomly

$\varepsilon' = 0$

for $i \leftarrow 1$ **to** T **do**

  $B_t \leftarrow$ random batch

  $i_t \sim b$ where $x_{i_t} \in B_t$

  $z_{i_t} \sim \mathscr{N}(0, \sigma^2 C^2)$

  if $\varepsilon' < \varepsilon$ **then**

    $$\theta_{t+1} \leftarrow \theta_t - \alpha \cdot \left(\frac{1}{|B_t|} \sum_{i_t} \left(\nabla\mathscr{L}(\theta_t; x_{i_t}) + z_{i_t} \right) \right)$$

    $\varepsilon' \leftarrow$ calculate privacy loss with moments accountant

  end

end

return θ

1.5 Experiments

In this section, we explain our experimental setting and discuss our results. First we briefly introduce our datasets and detail parameter settings for the used machine learning models. Afterwards, we give our results for two cyber deception tasks: (i) *attacker with no synthetic knowledge* and (ii) *attacker with synthetic knowledge*.

Datasets We evaluate the proposed differentially private deep learning-based honeydata generation approach on four real datasets. The following is a brief description of each dataset:

(i) The **Diabetes** [21] dataset contains the information of **768 female patients** who are at least 21 years old. Each patient is classified as diabetic or non-diabetic. The dataset contains **8 features**.
(ii) The **Adult** [21] dataset contains the information of **45222 individuals**. The dataset shows whether the income of the individuals exceeds 50K US dollars. The dataset contains **15 features**.
(iii) The **BreastCancer(Diagnostic)** [21] dataset contains the information about whether a patient has breast cancer or not. It has **569 patient records** with **32 features**.
(iv) The **Spambase** [21] dataset contains **4601 emails**, each of which is labeled as spam or non-spam. Each instance has **58 attributes**.

Parameter Setting for Data Generation Our DPSYN technique generates synthetic data by using Deep Auto-encoders [5]. An auto-encoder is trained on n data points. Once a model is learned, the auto-encoder can be used to generate *any number of data points* (e.g., honeydata). For n training samples, we report the results of the privacy loss (i.e., the measure of potential leakage to an attacker) using differential privacy with (ε, δ) parameters that is computed from the noise level σ (see Sect. 1.3.3). We fix the δ as $\frac{1}{n}$ and compute the value of ε for each iteration $t \in T$. In moment accountant, we use several noise levels to obtain consistent results. The large noise level ($\sigma = 6.0$) is implemented for small $\varepsilon = 1.0$ and the small noise level ($\sigma = 4.0$) is implemented for large $\varepsilon \in \{2.0, 4.0\}$. In these settings with the increasing ε values, synthetic data generation techniques are perturbed less since small noise is added to these techniques.

In all synthetic datasets (i.e., the generated relational honeydata), biases are initialized to zero, while the initial values of the weights θ are randomly chosen from a zero-mean normal distribution with a standard deviation of 0.05. For each dataset, we form a new auto-encoder to generate its corresponding honeydata.

Parameter Setting for Machine Learning Models We employ four machine learning models in measuring the efficiency of our approach in synthetic data generation for cyber deception: One-class SVM [27], two-class SVM [17], Logistic Regression (LR) [22], and Random Forest (RF) [7]. We chose to employ these methods because they are widely used for classification tasks [20]. Furthermore, these machine learning models will be used to explore whether an attacker can distinguish between the real data vs the honeydata easily.

For the hyper parameter of one-class SVM, we experiment with kernel types $\{linear, poly, rbf\}$ and gamma values $\{1.0, 0.1, 0.01, 0.001\}$. We select the most consistent results of one-class SVM with different (ε, δ) pairs. For two-class SVM, we employ the LinearSVM [14].

Benchmark Techniques In all experiments, we compare DPSYN results with two state-of-the-art synthetic data generation techniques: PRIVBAYES [32] and DPEM$^+$ [23]. We run the experiments 10 times and report the average of the results.

1.5.1 Task 1: Cyber Deception for Attacker with No Honeydata Knowledge

In the first task, we assume that the attacker has knowledge about real data where we model the background knowledge as the number (i.e., $\{50, 100, 200, 400\}$) of data points known to the attacker. This approach is similar to the setting reported in [6]. In this scenario, the attacker does not have access to honeydata samples. We evaluate the quality of the generated honeydata by observing whether the attacker can distinguish the real vs honeydata by leveraging the obtained real data. In these experiments, the attacker employs a one-class SVM model that is built on the $\{50, 100, 200, 400\}$ real data points.

We measure the success of DPSYN by the attacker's failure to separate honeydata and real data by using the classifier. The test data is an equal mix of 50% real and 50% honeydata. *A honeydata generation technique achieves best results if the attacker's SVM model labels all synthetic data as real, which results in a 50% accuracy*.

Figure 1.2 shows the performance of techniques in deceiving the attacker on the four real datasets. In Fig. 1.2a, we show performance for various training number sizes 50, 100, 200, and 400. Each training set is used by increasing noise additions, which is calibrated to $\varepsilon = 1.0, 2.0$, and 4.0, where $\varepsilon = 4.0$ shows the least amount of added noise in the model. As training size increases, performance of all three techniques improves and accuracy values approach 50%. In Spambase and BreastCancer datasets, PRIVBAYES and DPEM$^+$ perform worse compared to Adult and Diabetes datasets. We hypothesize that PRIVBAYES and DPEM$^+$ are more

vulnerable to the curse of high dimensionality; both Adult and Diabetes datasets have less than 15 attributes, whereas BreastCancer and Spambase have more than 32 attributes.

DPSYN results are comparable with those of DPEM$^+$ and PRIVBAYES in the Adult dataset. In Diabetes and Spambase datasets, DPSYN has better accuracy. PRIVBAYES has its worst results in the BreastCancer dataset. In all figures, increasing ε values result in values closer to the desired 50% accuracy value.

1.5.2 Task 2: Cyber Deception for Attacker with Honeydata Knowledge

In the first task, the attacker had access to real data only. In this second task, the attacker has access to both real and honeydata, which may help the attacker in distinguishing honeydata. The knowledge of honeydata implies that attacker was fooled into accepting some honeydata previously; when the attacker fails while using the honeydata (e.g., could not use the honeydata for identity theft), he/she may start analyzing other stored data files to authenticate them.

The attacker employs three binary classification models (i.e., two-class SVM, LR, and RF) that are built on a mix of 50% real and 50% synthetic data. The classifier is trained to learn two labels: real and synthetic (i.e., honeydata). The classifier is tested on a mix of 50% real and 50% honeydata. The accuracy of a model is given as the percentage of correctly classified data points. For the best performance in honeydata generation, the attacker must wrongly classify all honeydata points as real, which results in a 50% accuracy.

In this set of experiments, we report our results for each of the three Machine Learning classification models separately. We begin by demonstrating the Random Forest results.

Figure 1.3 demonstrates accuracy of the three techniques for the Random Forest classifier. The honeydata generated by DPSYN is more similar to real data when compared to the synthetic data generated from PRIVBAYES for Diabetes, Spambase, and BreastCancer datasets. For the Adult dataset, DPEM$^+$ results are closer to the desired 50% level. DPSYN performs better than DPEM$^+$ for Diabetes and Spambase. The failure of DPEM$^+$ and PRIVBAYES on Spambase and Diabetes datasets is expected since the attacker can already distinguish the difference between synthetic and real data with one-class SVM (See Sect. 1.5.1). DPSYN exhibits remarkable improvement for the majority of the RF test cases when compared to DPEM$^+$ and DPSYN.

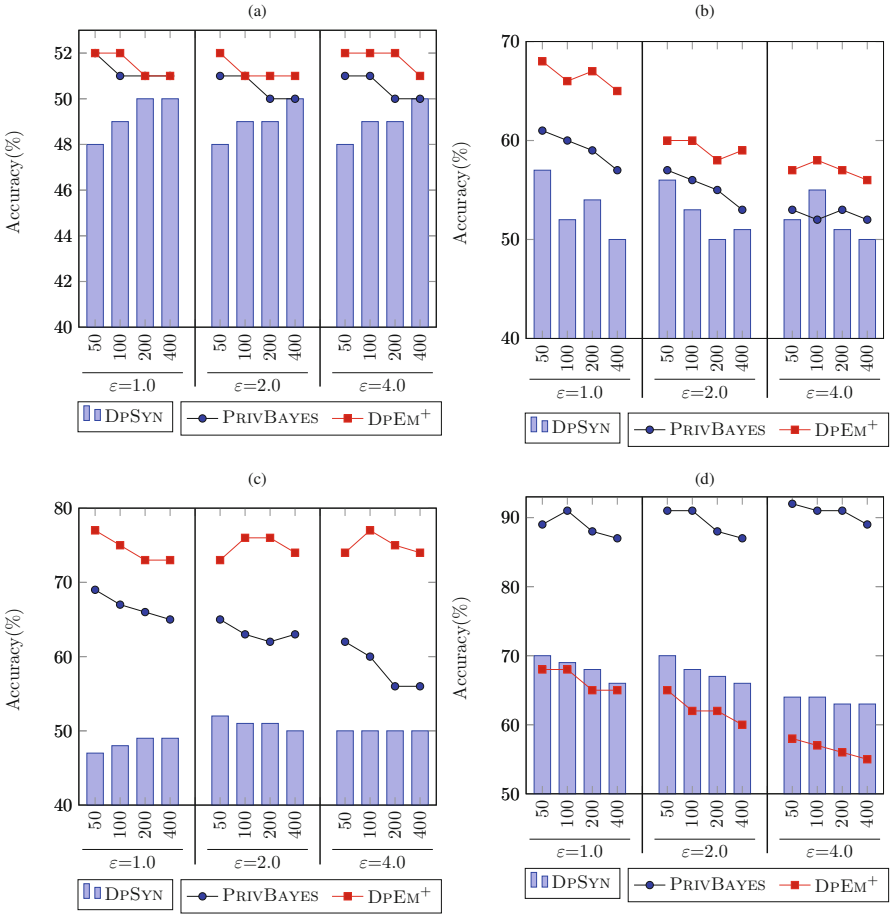

Fig. 1.2 Accuracy results of one-class SVM classifiers of attackers that are modeled on different percentages of real data with varying privacy budgets. The desired accuracy is 50%. (**a**) Adult. (**b**) Diabetes. (**c**) Spambase. (**d**) BreastCancer

The performance of the attacker with Logistic Regression (LR) is demonstrated in Fig. 1.4. The test results of LR are consistent with the other machine learning models previously shown in Figs. 1.3 and 1.2. In fact, on average, LR results are highly correlated (0.94) with RF results in all datasets. However, it is noticeable that attacker with RF is better able to distinguish between real and honeydata when compared to LR; accuracy levels are lower in the LR results. DPSYN outperforms PRIVBAYES in three out of four datasets. For Adult dataset, PRIVBAYES has slightly better results than DPSYN. For this set of experiments with LR, DPSYN results are similar to those of DPEM$^+$ in the Diabetes, Spambase, and BreastCancer datasets.

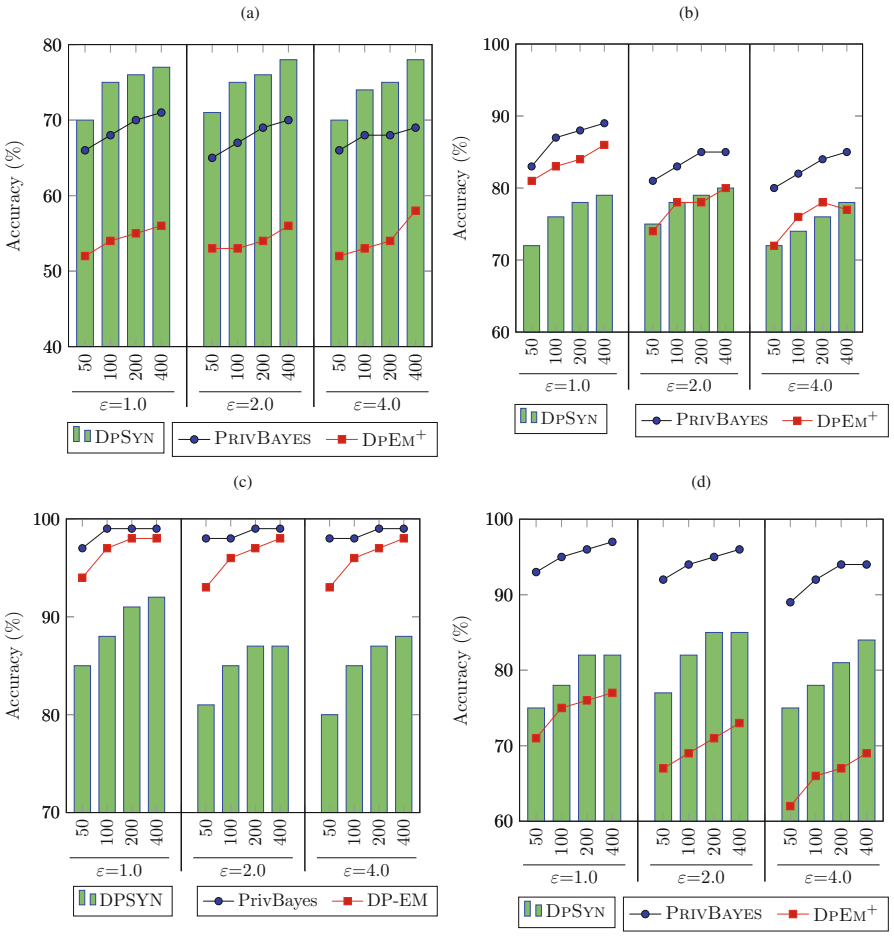

Fig. 1.3 Accuracy results of RF classifiers of attackers that are modeled on different percentages of real data with varying privacy budgets. The desired accuracy is 50%. (**a**) Adult. (**b**) Diabetes. (**c**) Spambase. (**d**) BreastCancer

The performance of the attacker with two-class SVM is demonstrated in Fig. 1.5. Results are consistent with those of LR and RF in Figs. 1.4 and 1.3. Only in the Adult dataset, we see a slight increase of difference between PRIVBAYES and DPEM$^+$ performances with the increase of training numbers.

In all Machine Learning models, our method DPSYN generates honeydata that has better indistinguishability (i.e., the attacker has less accuracy in distinguishing real vs honeydata) for cyber deception. With increasing training dataset size, the classifiers that could be used by the attacker perform better in all data generation techniques. Except for the Adult dataset, DPSYN outperforms PRIVBAYES significantly in all experimental settings. Compared to DPEM$^+$, DPSYN generates better or

comparable synthetic data for the Spambase, BreastCancer, and Diabetes datasets. For the majority of the test cases, increasing ε values harm the attacker machine learning model. However, increasing ε values result in less added noise which may cause the leakage of sensitive data. Hence, there is a trade-off between the quality of honeydata and the prevention of sensitive data leakage.

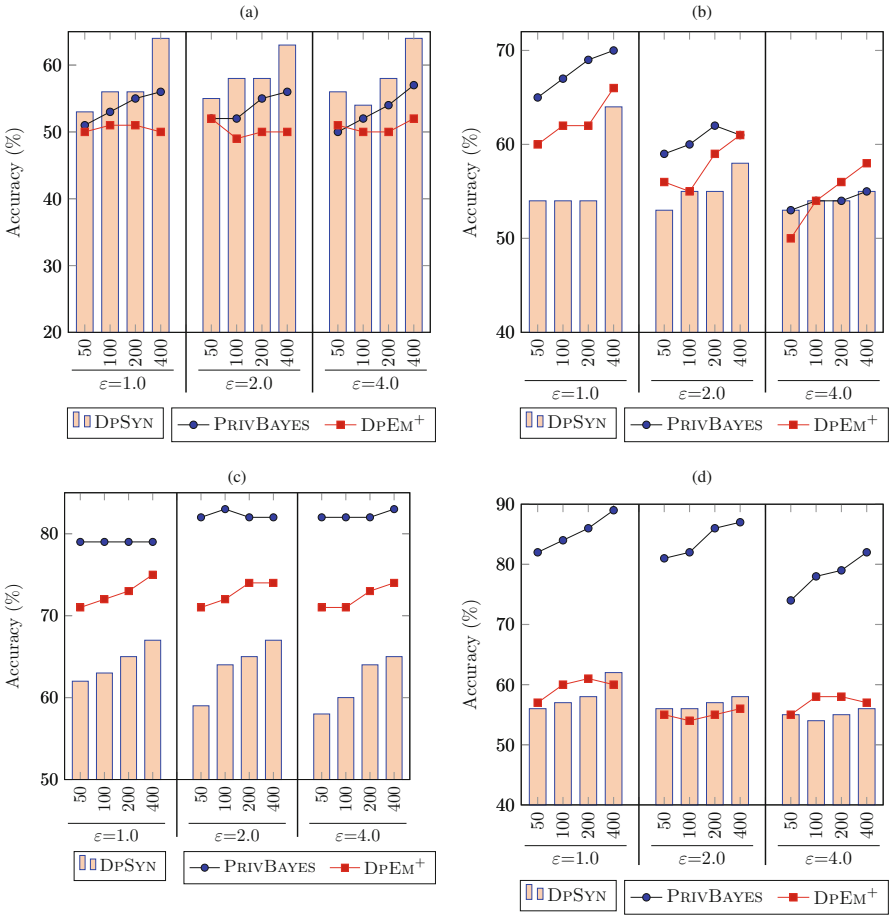

Fig. 1.4 Accuracy results of LR classifiers of attackers that are modeled on different number of real data with varying privacy budgets. The desired accuracy is 50%. (**a**) Adult. (**b**) Diabetes. (**c**) Spambase. (**d**) BreastCancer

1.6 Conclusions

In this book chapter, we explore the applicability of using privacy-preserving deep learning-based synthetic data generation techniques for creating HoneyData that can fool potential cyberattackers. We define a machine learning (ML)-based metric (i.e., the accuracy of any ML model in distinguishing real vs HoneyData) to measure the goodness of generated deceptive HoneyData. Although, our results indicate that existing techniques could be leveraged for HoneyData generation, care must be taken in setting the privacy parameters used in HoneyData generation.

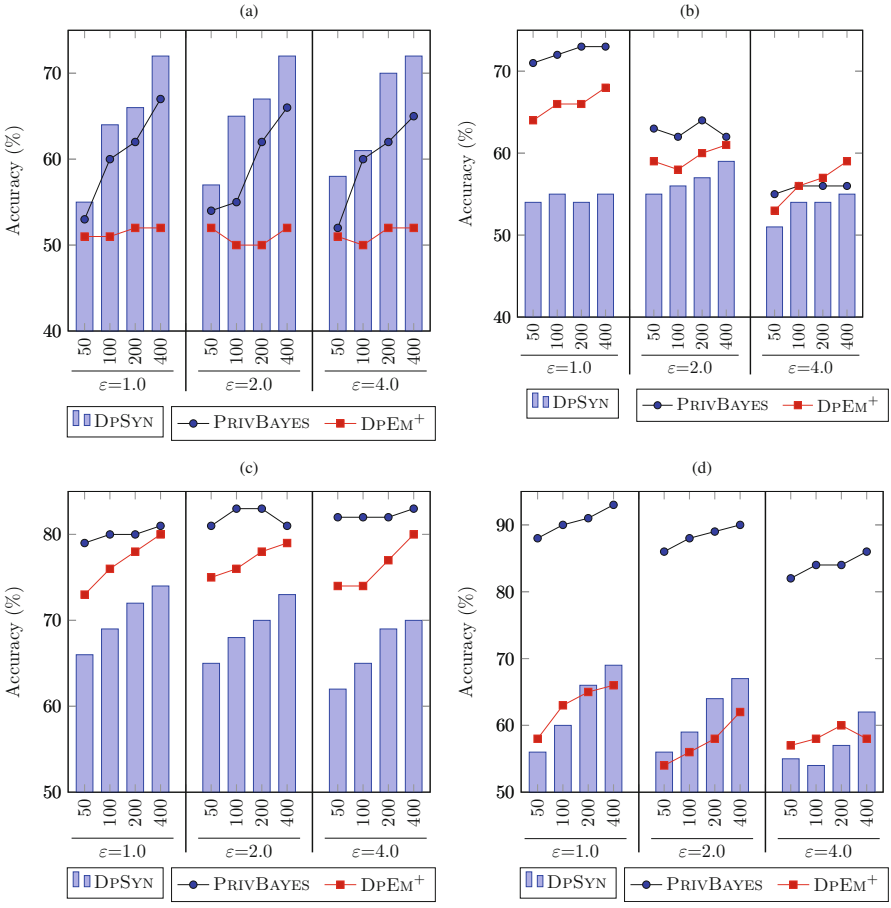

Fig. 1.5 Accuracy results of binary SVM classifiers of attackers that are modeled on different percentages of real data with varying privacy budgets. The desired accuracy is 50%. (**a**) Adult. (**b**) Diabetes. (**c**) Spambase. (**d**) BreastCancer

References

1. Abadi, M., Chu, A., Goodfellow, I., McMahan, H.B., Mironov, I., Talwar, K., Zhang, L.: Deep learning with differential privacy. In: Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 308–318. ACM (2016)
2. Abay, N.C., Zhou, Y., Kantarcioglu, M., Thuraisingham, B., Sweeney, L.: Privacy preserving synthetic data release using deep learning. The European Conference on Machine Learning and Principles and Practice of Knowledge Discovery in Databases (PKDD 2018) (2018)
3. Ács, G., Melis, L., Castelluccia, C., Cristofaro, E.D.: Differentially private mixture of generative neural networks. CoRR **abs/1709.04514** (2017). URL http://arxiv.org/abs/1709.04514
4. Almeshekah, M.H., Spafford, E.H.: Cyber security deception. In: Cyber Deception, pp. 23–50. Springer (2016)
5. Baldi, P.: Autoencoders, unsupervised learning, and deep architectures. In: Proceedings of ICML workshop on unsupervised and transfer learning, pp. 37–49 (2012)
6. Bindschaedler, V., Shokri, R., Gunter, C.A.: Plausible deniability for privacy-preserving data synthesis. Proceedings of the VLDB Endowment **10**(5), 481–492 (2017)
7. Breiman, L.: Random forests. Machine learning **45**(1), 5–32 (2001)
8. Bun, M., Steinke, T.: Concentrated differential privacy: Simplifications, extensions, and lower bounds. In: Theory of Cryptography Conference, pp. 635–658. Springer (2016)
9. Chaudhuri, K., Monteleoni, C.: Privacy-preserving logistic regression. In: Advances in Neural Information Processing Systems, pp. 289–296 (2009)
10. Dwork, C.: Differential privacy. In: Proceedings of the 33rd International Conference on Automata, Languages and Programming - Volume Part II, ICALP'06, pp. 1–12. Springer-Verlag, Berlin, Heidelberg (2006). DOI 10.1007/11787006_1. URL http://dx.doi.org/10.1007/11787006_1
11. Dwork, C., Kenthapadi, K., McSherry, F., Mironov, I., Naor, M.: Our data, ourselves: Privacy via distributed noise generation. In: Annual International Conference on the Theory and Applications of Cryptographic Techniques, pp. 486–503. Springer (2006)
12. Dwork, C., Lei, J.: Differential privacy and robust statistics. In: Proceedings of the forty-first annual ACM symposium on Theory of computing, pp. 371–380. ACM (2009)
13. Dwork, C., Roth, A., et al.: The algorithmic foundations of differential privacy. Foundations and Trends® in Theoretical Computer Science **9**(3–4), 211–407 (2014)
14. Fan, R.E., Chang, K.W., Hsieh, C.J., Wang, X.R., Lin, C.J.: Liblinear: A library for large linear classification. Journal of machine learning research **9**(Aug), 1871–1874 (2008)
15. Goodfellow, I.: Efficient per-example gradient computations. arXiv preprint arXiv:1510.01799 (2015)
16. Goodfellow, I., Bengio, Y., Courville, A.: Deep Learning. MIT Press (2016). http://www.deeplearningbook.org
17. Hearst, M.A., Dumais, S.T., Osuna, E., Platt, J., Scholkopf, B.: Support vector machines. IEEE Intelligent Systems and their applications **13**(4), 18–28 (1998)
18. Holz, T., Raynal, F.: Detecting honeypots and other suspicious environments. In: Information Assurance Workshop, 2005. IAW'05. Proceedings from the Sixth Annual IEEE SMC, pp. 29–36. IEEE (2005)
19. Juels, A., Rivest, R.L.: Honeywords: Making password-cracking detectable. In: Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security, pp. 145–160. ACM (2013)
20. Kotsiantis, S.B., Zaharakis, I., Pintelas, P.: Supervised machine learning: A review of classification techniques. Emerging artificial intelligence applications in computer engineering **160**, 3–24 (2007)
21. Lichman, M.: UCI machine learning repository (2013). URL http://archive.ics.uci.edu/ml
22. Nerlove, M., Press, S.J.: Univariate and multivariate log-linear and logistic models, vol. 1306. Rand Santa Monica (1973)

23. Park, M., Foulds, J., Chaudhuri, K., Welling, M.: Practical privacy for expectation maximization. CoRR **abs/1605.06995** (2016). URL http://arxiv.org/abs/1605.06995
24. Pascanu, R., Mikolov, T., Bengio, Y.: On the difficulty of training recurrent neural networks. In: International Conference on Machine Learning, pp. 1310–1318 (2013)
25. Rubin, D.B.: Discussion statistical disclosure limitation. Journal of official Statistics **9**(2), 461 (1993)
26. Rubinstein, B.I., Bartlett, P.L., Huang, L., Taft, N.: Learning in a large function space: Privacy-preserving mechanisms for SVM learning. arXiv preprint arXiv:0911.5708 (2009)
27. Schölkopf, B., Platt, J.C., Shawe-Taylor, J.C., Smola, A.J., Williamson, R.C.: Estimating the support of a high-dimensional distribution. Neural Comput. **13**(7), 1443–1471 (2001). DOI 10.1162/089976601750264965. URL https://doi.org/10.1162/089976601750264965
28. Song, S., Chaudhuri, K., Sarwate, A.D.: Stochastic gradient descent with differentially private updates. In: Global Conference on Signal and Information Processing (GlobalSIP), 2013 IEEE, pp. 245–248. IEEE (2013)
29. Spitzner, L.: Honeypots: tracking hackers, vol. 1. Addison-Wesley Reading (2003)
30. Vaidya, J., Shafiq, B., Basu, A., Hong, Y.: Differentially private naive Bayes classification. In: Proceedings of the 2013 IEEE/WIC/ACM International Joint Conferences on Web Intelligence (WI) and Intelligent Agent Technologies (IAT)-Volume 01, pp. 571–576. IEEE Computer Society (2013)
31. Yuill, J., Zappe, M., Denning, D., Feer, F.: Honeyfiles: deceptive files for intrusion detection. In: Information Assurance Workshop, 2004. Proceedings from the Fifth Annual IEEE SMC, pp. 116–122. IEEE (2004)
32. Zhang, J., Cormode, G., Procopiuc, C.M., Srivastava, D., Xiao, X.: Privbayes: Private data release via Bayesian networks. In: Proceedings of the 2014 ACM SIGMOD international conference on Management of data, pp. 1423–1434. ACM (2014)
33. Zhang, J., Zhang, Z., Xiao, X., Yang, Y., Winslett, M.: Functional mechanism: regression analysis under differential privacy. Proceedings of the VLDB Endowment **5**(11), 1364–1375 (2012)

Chapter 2
Towards Intelligent Cyber Deception Systems

Fabio De Gaspari, Sushil Jajodia, Luigi V. Mancini, and Giulio Pagnotta

Abstract The increasingly sophisticated nature of cyberattacks reduces the effectiveness of expert human intervention due to their slow response times. Consequently, interest in automated agents that can make intelligent decisions and plan countermeasures is rapidly growing. In this chapter, we discuss intelligent cyber deception systems. Such systems can dynamically plan the deception strategy and use several actuators to effectively implement the cyber deception measures. We also present a prototype of a framework designed to simplify the development of cyber deception tools to be integrated with such intelligent agents.

2.1 Introduction

The knowledge of attackers and the sophistication of cyberattacks are constantly increasing, as well as the complexity of the cyber domain. The result of this process is that expert human intervention, even if available, is not always fast enough to deal with the speed of cyberthreats. As a consequence, cyber deception strategies aimed at hindering attackers' progress and cyber defense agents that can make autonomous decisions are receiving an increasing amount of attention [9, 17]. An important part of cyber deception is *active defense* [15, 16]. Differently from classical, reactive systems such as firewalls, IPS, and IDS, active defense tools aim to hinder attackers' progress in a proactive manner, rather than responding if and when an attack is detected. One of the most well-known examples of active

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

F. De Gaspari · L. V. Mancini · G. Pagnotta
Sapienza University of Rome, Roma, RM, Italy

S. Jajodia (✉)
George Mason University, Fairfax, VA, USA
e-mail: jajodia@gmu.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_2

defense tools are honeypots [5, 8]: mock systems designed to lure attackers in order to study their behavior and restrict their access to the real production systems. Other active defense techniques, like honeypatches [6], trick attackers into believing that their exploit was successful, but transparently redirect him to an unpatched, heavily monitored decoy system. Fake login sessions [26], mock services, and port randomization [4] aim at confounding the attacker, compromising and slowing down the reconnaissance phase. Coupling such active defense tools with autonomous, intelligent agents has the potential of greatly improving cyber defense, reducing the reliance on human intervention in response to cyberattacks.

In this chapter, we discuss intelligent cyber deception agents that can make autonomous decisions on how to counter ongoing attacks, and their integration with active defense tools. We also discuss our design of an active defense framework that allows fast prototyping of active defense tools to be integrated directly into live, production systems. The framework uses a modular approach to add and remove active defense tools, and aims to provide seamless integration with the agent to provide sensing and actuating functions.

2.2 Preliminaries

In this section, we discuss intelligent cyber defense agents and the complexity of deploying them in the context of cyber defense. We also discuss active defense techniques and the advantages it brings with respect to traditional systems.

2.2.1 Intelligent Cyber Defense Agents

An intelligent agent is an entity which takes autonomous decisions based on the observations of the current world state through sensors, and which applies actions through actuators to achieve an end goal. Agents can rely on different methodologies to produce decisions, such as knowledge-based systems [10] or machine learning techniques [24]. While research and applications of intelligent agents is already underway in multiple fields, this is not the case in cyber defense. Indeed, the realm of cyber defense introduces a number of unique obstacles that are particularly challenging, such as the extreme complexity and size of the state space (i.e., the possible states of the world the agent is monitoring). Artificial neural networks [12] and deep learning techniques [18] can potentially help to overcome these challenges. However, research in this direction is still in its early stages and deep learning is mostly used to devise new attacks [13] or for the purpose of attack detection [22, 23], rather than to plan countermeasures. Moreover, deep neural networks are subject to a new type of attack known as adversarial examples [11, 19]. Such attacks could be exploited by attackers to target the decision-making process of the agent itself, tricking it into taking decisions that are detrimental for the system. For instance, if

the agent is designed to shut down a particular service under certain severe attack conditions, an attacker could potentially craft an adversarial example that causes the agent to misclassify the current world state and erroneously shutdown the service.

2.2.2 Active Defense

Active defense is a branch of cybersecurity aimed at actively hindering attackers' progress preemptively, rather than reactively as in traditional systems [15, 16]. Indeed, active defense tools are always active, and do not rely on detection of an attack in order to function. Active defense is related to cyber deception. Most active defense tools, in fact, heavily rely on deception techniques to confound attackers and slow down their progress. The most well-known instance of an active defense tool is the honeypot [20]. Honeypots are replicas of real systems, instrumented with logging and deception capabilities such as fake services. Honeypots are designed to look like attractive targets for attackers, in order to obtain as much information from adversarial interactions as possible. However, honeypots require complex configuration and it is very hard to hide their nature to attackers, reducing their effectiveness [21]. To mitigate this drawback, recent works aim to integrate deception capabilities directly into the real production systems themselves, avoiding the issue of camouflaging altogether [9]. These systems use techniques similar to those employed by honeypots, as well as other active defense tools such as honeyfiles [7, 25] and network randomization [4], in order to heavily slow down the attacker, while at the same time increase the chances of detection.

2.3 Towards Intelligent Cyber Deception Systems

Autonomous agents require sensors and actuators to respectively measure and alter the current world state. Active defense tools are designed to interact with attackers and collect important data regarding how the attacker interacts with the system. Therefore, such tools can be extremely effective sensors for the agent. For instance, fake services can provide information regarding what type of services the attacker is looking for, as well as how he interacts with such services. Logging honeyfiles access provides detailed information regarding which types of files are interesting to the attacker, and honeypatch sensors allow the agent to isolate specific exploits used during the attack. Moreover, active defense tools can also be used as actuators: the agent can use the data generated by the sensors to dynamically reconfigure the active defense tools, in addition to dynamically deploying new tools aimed to hinder the specific pattern of the current attack.

In our preliminary work [9], we proposed an automated, cyber deception system called Attackers Hindered by Employing Active Defense, or *AHEAD*. The AHEAD

architecture describes an autonomous agent that employs an array of active defense tools as both sensors and actuators. AHEAD is comprised of an autonomous agent, the *AHEAD controller*, which manages a cluster of active defense tools, the *AHEAD Pot* as illustrated in Fig. 2.1.

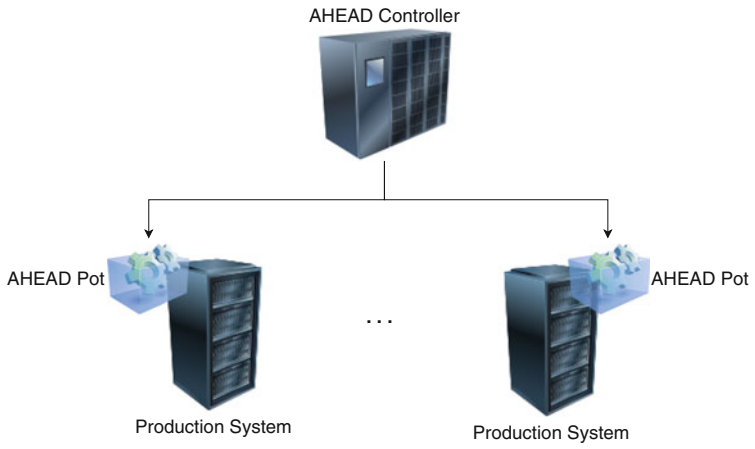

Fig. 2.1 Overview of the architecture of AHEAD

Differently from systems such as honeypots, the AHEAD Pot is deployed alongside the production services in a real system, rather than on a separate (virtual) machine. Using the AHEAD Pot to directly instrument production systems with deception capabilities allows to avoid the drawbacks of honeypots such as its ease of detection. Moreover, this design makes it harder for attackers to identify vulnerable services in the production systems, as well as providing the systems with advanced monitoring capabilities. In order to prevent the AHEAD Pot itself from becoming an attack vector, the pot is isolated from the production system through the use of container technology [2], as illustrated in Fig. 2.2. The use of containers, as well as mandatory access control techniques to limit the pot's access to the system, provides a layer of isolation and hardening against attacks directed at the pot itself. The AHEAD controller is responsible for planning the defense strategy during an attack, which is done based on the inputs from the active defense tools of the AHEAD Pot. The controller is also responsible for actuating the planned countermeasures through dynamic reconfiguration of the AHEAD Pot.

2.3.1 Usage Scenario

In this section, we describe a usage scenario of AHEAD. Let us consider an attacker who wants to attack some production systems on a target network. We distinguish the two scenarios depicted in Fig. 2.3: *(A)* a network protected by a classical

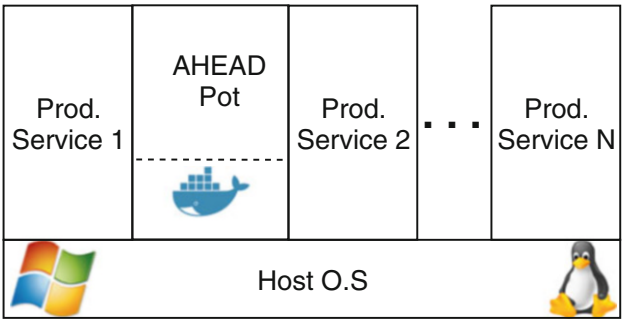

Fig. 2.2 Integration of the AHEAD Pot in the production system

honeypot and *(B)* a network protected with AHEAD. Before performing any attack, the attacker will have to perform reconnaissance on the network in order to identify valuable targets. Let us assume that, in order to reach this goal the attacker performs a network scan to identify existing systems and services.

(A) Classical Honeypot In scenario (A), the network scan will eventually reach the honeypot (step 1). At this point, if the configuration of the honeypot is realistic enough, the attacker will start attacking one of the available services provided by the honeypot. The attack is detected by the honeypot (step 2), and the Incident and Response Team (IRT) will be notified that something anomalous is going on in the network. Unfortunately, the attacker will eventually realize that the target is indeed a honeypot (step 3) and will move on to attacking one of the remaining systems (step 4). In this scenario, the limitation of the honeypot approach from the point of view of the IRT is that the interaction between the attacker and the honeypot is extremely limited in time, often in the order of seconds. Indeed, after the attacker leaves the honeypot and moves on to another system, the IRT loses the chance to monitor the attacker and devise a proper identification and defense strategy.

(B) AHEAD On the other hand, when AHEAD is employed, the attacker will have to sift through fake services and mock vulnerabilities (step 1) in order to try to compromise the production system, forcing him to interact with AHEAD for a considerably longer time (step 3.i). This provides the autonomous agent (or the IRT, if the agent is disabled) with considerably more time to act, and more information to decide how to counter the attack (step 2), as well as provide more material to analyze the strategy of the attacker after the attack has concluded (step 4). This additional information allows to improve the attribution of the attack and the security of the network and systems, adapting them to ever-evolving attack strategies. Moreover, AHEAD can also work as a deterrent. Indeed, if the attacker realizes that the real production system is heavily monitored and instrumented, he might also choose to forfeit the attack in order to protect himself (step 3.ii). In both cases, the network is protected.

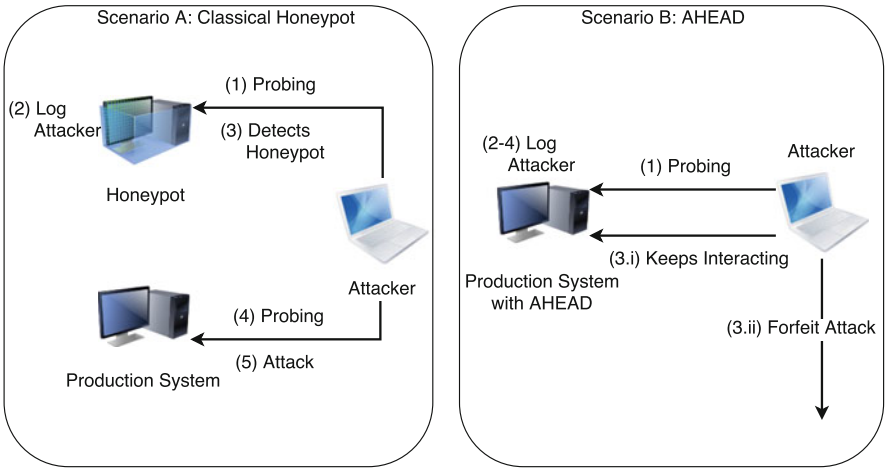

Fig. 2.3 Comparison between a classical honeypot system and AHEAD

2.3.2 The Architecture of the AHEAD System

From an architectural perspective, the AHEAD system is composed by two components: the *AHEAD Controller* and *AHEAD Pot*. The AHEAD Controller is the single point of interaction with the pots, and allows to manage the active defense tools deployed over a secure channel. The AHEAD Pot is the component effectively implementing the active defense countermeasures and is deployed on the production systems. In a real-world scenario, several AHEAD Pots are deployed in a corporate network, covering all components of the information system (see Fig. 2.4). The AHEAD Pots are encapsulated in a container and therefore do not interfere with the production services, while at the same time having a low, configurable overhead on the production system itself. Full automation of the security management of a network is a challenging task that requires gradual evolution and integration. Therefore, the AHEAD system is designed to be tightly integrated with pre-existing security information and event management (SIEM) systems, and can provide an admin interface for the security admin. The AHEAD Pots constantly send activity logs to the SIEM systems, allowing the IRT to improve other security components already deployed (e.g., intrusion detection/prevention systems, and firewall). The feedback from the AHEAD Pots is also used by the IRT to identify what additional active defense modules need to be deployed in the Pots themselves, so that the system can dynamically adapt to emerging threats. However, the final goal is for the system to be fully autonomous: the controller makes decisions on what tools should be deployed based on the current state of the world, which is reported by the tools of the pot itself as illustrated in Fig. 2.5.

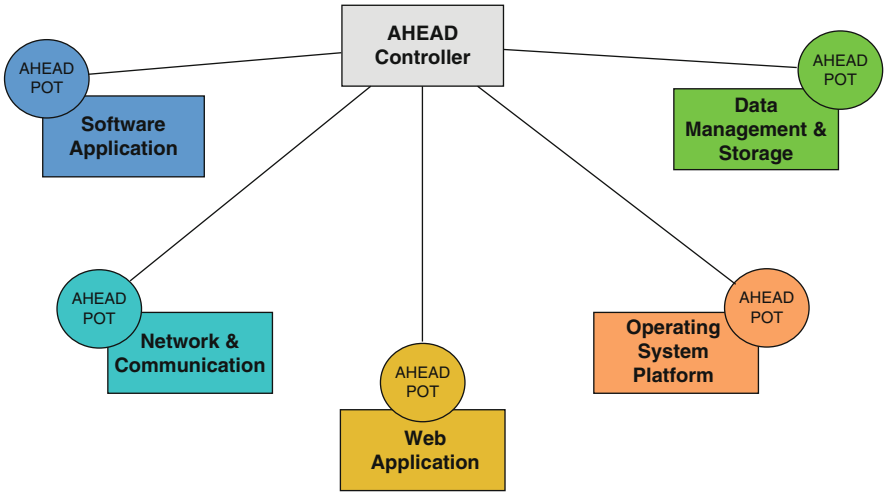

Fig. 2.4 Integration of AHEAD in the architecture of a typical corporate information system

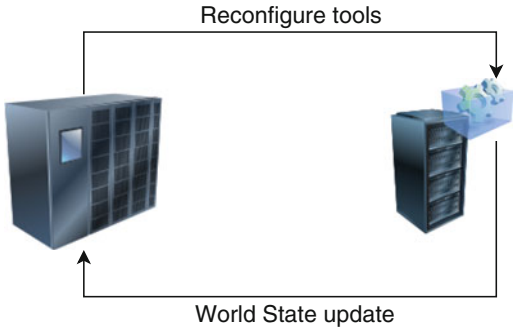

Fig. 2.5 Feedback loop of the AHEAD system. The Controller reconfigures the tools in the pot, based on the world view provided by the pot itself

2.4 Evolving the Pot: ADARCH

During the development of the AHEAD Pot prototype, we quickly realized that a unified framework for the development and integration of active defense tools was required. Existing active defense tools are implemented using a heterogeneous mix of programming languages and libraries, as well as differing architectural designs. Moreover, different tools tend to use different logging formats, sometimes custom-made, which complicates the interaction with the AHEAD Controller. Finally, having separate active defense tools makes it harder to present the attacker with a consistent view of the Pot, potentially creating side channels that allow the attacker to distinguish between services exposed by the Pot, and production services exposed by the real system. Integrating and maintaining such a diverse set of tools into a

coherent architecture would be complex and error-prone. Moreover, assessing the overall security of the system would be a daunting task, especially given the overlap in functionality between certain tools, and the code duplication ensuing from it.

In order to address the above-mentioned issues, we designed and implemented a new, cross-platform pot architecture, *ADARCH* (the Active Defense ARCHitecture), to facilitate the development of active defense tools that share a uniform architecture. The goal of ADARCH is to simplify the implementation of common functionalities of active defense tools, as well as to provide a uniform interface for the controller to interact with the tools. In particular, we identified two main functions that are used by multiple tools and that require simplification and unification: networking and logging. The first implementation of ADARCH aims to simplify and uniform the network flow management and concurrency across the tools, as well as to provide a common logging interface that enables the AHEAD Controller to more easily parse their output.

2.4.1 ADARCH Design

As we discussed in the previous section, one of the design goals of ADARCH is to simplify the development and integration of active defense tools. However, since the Pot is designed to be integrated into real production systems, it is also important to reduce the overhead introduced and the resource requirements as much as possible. To this end, we designed ADARCH and the new ADARCH Pot around a core software module written in C, and integrated a Python interpreter to facilitate the prototyping and development of active defense tools. Figure 2.6 provides a high-level overview of the ADARCH framework and the architecture of the ADARCH Pot. The core C module efficiently implements common functionalities required by multiple active defense tools, such as network connection and concurrency management, as well as provides an interface for the integrated python interpreter. Active defense tools developed with ADARCH are executed within the integrated Python interpreter, which provides them with access to the API exposed by the C core module. Moreover, ADARCH allows active defense tool developers to use a configuration file-based approach to instantiate required resources (e.g., port bindings) that are transparently handled by the ADARCH core, as well as to define triggers associated with particular functions of the tools (e.g., which function to call when a new connection is open on a port). Finally, ADARCH is designed to be cross-platform and to work with container technologies, such as Docker, to provide an additional layer of isolation to the underlying production system.

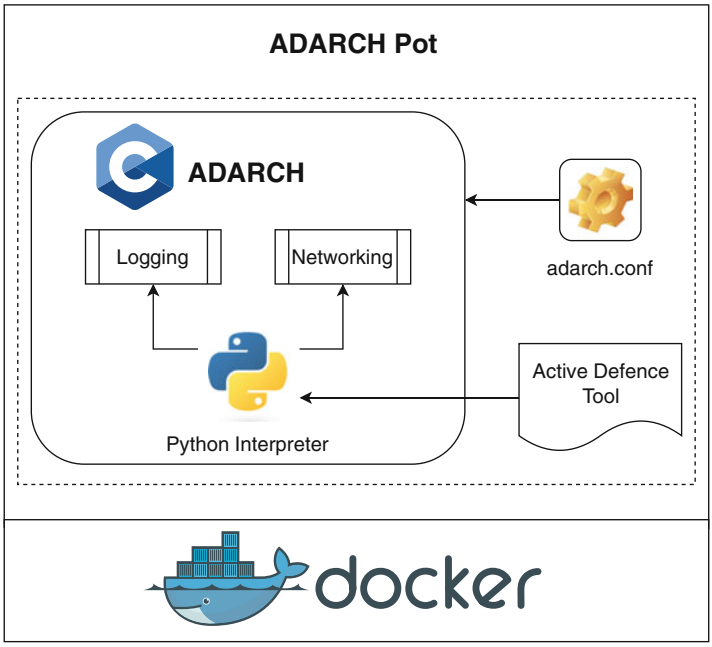

Fig. 2.6 High-level ADARCH architecture overview

2.4.2 Python Embedding and Extension

We chose Python as the programming language for the development of new active defense tools because of the large number of available libraries, the community support, and the ease of prototyping it provides. Moreover, several pre-existing active defense tools were already implemented in Python, so its choice also provides continuity for the developers. However, integrating Python with the C core of the framework to provide better efficiency and performance presented several challenges. In particular, it required overcoming the limitation of the Global Interpreter Lock (GIL) of the standard CPython interpreter, which heavily limits concurrent architectural designs [3]. The GIL prevents the Python interpreter from concurrently interpreting bytecode for different threads, effectively resulting in a sequential execution. This behavior is highly undesirable as it reduces the performance of the ADARCH Pot, and might even provide attackers with a side channel to differentiate the services exposed by the Pot from the production services [14]. To overcome these drawbacks, the interface between the Python interpreter and the C core of ADARCH was designed to manage the GIL in a fine-grained manner, releasing the lock whenever possible when an ADARCH API call is made, and trying to parallelize the execution of the various modules as much as possible.

The Python interpreter was also extended in order to expose the ADARCH's API to the tool developers. The extension of the interpreter required explicit managing

of the internal reference count of Python objects, which is used by the Python garbage collector to periodically cleanup unreferenced memory. While reference count management is generally well-documented and understood, special care was needed due to the concurrent nature of the ADARCH core. ADARCH's API, illustrated in Table 2.1, is currently minimalistic mainly due to the fact that it is updated as more active defense tools are integrated and developed into ADARCH. However, it can be easily extended to provide additional functionalities, such as filesystem management to implement integrity check tools, similarly to Artillery [1]. The API is encapsulated in a wrapper Python class, allowing developers to extend the standard API if needed.

Table 2.1 ADARCH API

API	Description
log.write	Helper function to write logs in a standard format
connection.send	Network wrapper to send data through an open connection
connection.shutdown	Network wrapper to close an open connection

2.4.3 Advantages of the ADARCH Framework

The ADARCH frameworks implements functions that are common to multiple active defense tools, such as logging and networking, and provides a transparent interface to the developer through the API and ADARCH configuration file. Allowing developers to focus on the core deception aspects of the tools, rather than having to deal with networking, threading, and synchronization, greatly simplifies and expedites the development process. Moreover, the integration of the Python interpreter allows developers to use a high-level language, further simplifying the prototyping and development of new tools, while at the same time maintaining high performance as a result of the C core module of ADARCH. The ADARCH framework is also cross-platform, working both under Linux and Windows systems, which means that active defense tools developed with ADARCH do not require additional work to be ported to different systems. Moreover, having multiple tools integrated and running within the same process space allows them to more easily share resources and interact with each other if required. Finally, ADARCH provides active defense tools with a standard format for logging, which allows for immediate integration of new active defense tools with the AHEAD Controller.

To assess the advantages of ADARCH over previous versions of the Pot, we re-implemented a popular active defense tool called Portspoof as an ADARCH module. Portspoof is an active defense tool which allows to simulate the signatures of a great number of network services. The goal of the tool is to hinder the discovery phase of an attack, forcing the attacker to perform a more thorough service scan,

and to generate much more traffic in the process. The original software, developed in C++, counts $3k$ lines of code to manage concurrent network connections using multiple threads. The corresponding ADARCH module is less than 100 lines of code written in tens of minutes and is functionally equivalent to the original tool. Moreover, our preliminary performance evaluation shows that the ADARCH tool introduces a slightly lower overhead to a production system than the original Portspoof. ADARCH extremely simplifies the development of active defense tools, heavily reducing the time required and the complexity of the code, while at the same time improving the maintainability and the security of the tools due to less code duplication. Finally, ADARCH allows to use a single, optimized instance of the Python interpreter for all active defense tools, rather than one instance per tool, further reducing system overhead when considering deployments of multiple tools.

2.5 Conclusions

In this chapter, we discussed autonomous, intelligent cyber agents and the challenges associated with their implementation. Moreover, we examine the architecture we presented in [9] and propose a new framework to develop and deploy active defense tools, ADARCH, and the new ADARCH Pot. The ADARCH framework allows to heavily simplify and expedite the development of new active defense tools and their integration in the AHEAD architecture, transparently implementing common functions required by multiple active defense tools. We discuss the advantages of ADARCH with respect to stand-alone implementations of active defense tools, and we compared the complexity of such stand-alone tools with the simplicity of an ADARCH module providing the same functionalities.

2.6 Exercises

In this section, we propose a list of exercises, in increasing order of difficulty, that can help familiarize students with the concepts presented in this chapter.

1. Identify and describe the disadvantages of the presented approach.
2. What are the trade-offs of using container technology for isolation vs. virtual machines?
3. Configure and deploy a simple honeypot on the Internet. Analyze how long attackers interact with the honeypot on average before realizing that it is not a real system.
4. What are the risks of installing active defense tools on live, production systems? How would you minimize these risks?
5. Write a simple active defense tool that can create trap files in the file system. Once opened, the files should trigger and log an alert.
6. Extend the tool described in the previous point to provide attribution capabilities.

Acknowledgements This work was partially funded by the Army Research Office under the grants W911NF-13-1-0421 and W911NF-15-1-0576, and by the Office of Naval Research under the grant N00014-15-1-2007.

References

1. Artillery. https://github.com/shoreditch-ops/artillery.
2. Docker platform. https://www.docker.com/.
3. Python Global Interpreter Lock. https://wiki.python.org/moin/GlobalInterpreterLock.
4. E. Al-Shaer. *Toward Network Configuration Randomization for Moving Target Defense*, pages 153–159. 2011.
5. 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*, SSYM'05, pages 9–9, 2005.
6. F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 942–953, 2014.
7. B. M. Bowen, S. Hershkop, A. D. Keromytis, and S. J. Stolfo. Baiting inside attackers using decoy documents. In *Security and Privacy in Communication Networks*, pages 51–70.
8. M. L. Bringer, C. A. Chelmecki, and H. Fujinoki. A survey: Recent advances and future trends in honeypot research. In *International Journal of Computer Network and Information Security*, IJCNIS, 2012.
9. F. De Gaspari, S. Jajodia, L. V. Mancini, and A. Panico. Ahead: A new architecture for active defense. In *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '16, 2016.
10. J. C. Giarratano and G. Riley. *Expert Systems: Principles and Programming*. Brooks/Cole Publishing Co., Pacific Grove, CA, USA, 1989.
11. I. J. Goodfellow, J. Shlens, and C. Szegedy. Explaining and Harnessing Adversarial Examples. *ArXiv e-prints*, 2014.
12. M. H. Hassoun. *Fundamentals of Artificial Neural Networks*. MIT Press, Cambridge, MA, USA, 1st edition, 1995.
13. B. Hitaj, P. Gasti, G. Ateniese, and F. Perez-Cruz. PassGAN: A Deep Learning Approach for Password Guessing. *ArXiv*, 2017.
14. R. Hund, C. Willems, and T. Holz. Practical timing side channel attacks against kernel space ASLR. In *2013 IEEE Symposium on Security and Privacy*, 2013.
15. S. Jajodia, K. A. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and S. X. Wang, editors. *Moving Target Defense II: Application of Game Theory and Adversarial Modeling*. Springer, 2013.
16. S. Jajodia, K. A. Ghosh, V. Swarup, C. Wang, and S. X. Wang, editors. *Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats*. Springer, 2011.
17. A. Kott, L. V. Mancini, P. Théron, M. Drašar, E. Dushku, H. Günther, M. Kont, B. LeBlanc, A. Panico, M. Pihelgas, and K. Rzadca. Initial Reference Architecture of an Intelligent Autonomous Agent for Cyber Defense. *ArXiv e-prints*, 2018.
18. Y. LeCun, Y. Bengio, and G. Hinton. Deep learning. *Nature*, 521:436 EP –, May 2015.
19. N. Papernot, P. McDaniel, S. Jha, M. Fredrikson, Z. B. Celik, and A. Swami. The limitations of deep learning in adversarial settings. In *2016 IEEE European Symposium on Security and Privacy (EuroS P)*, 2016.
20. N. Provos. A virtual honeypot framework. In *Proceedings of the 13th Conference on USENIX Security Symposium - Volume 13*, SSYM'04, 2004.

21. N. Provos and T. Holz. *Detecting Honeypots*, chapter in book: Virtual Honeypots: From Botnet Tracking to Intrusion Detection. Addison-Wesley Professional, 2007.
22. J. Saxe and K. Berlin. Deep neural network based malware detection using two dimensional binary program features. In *2015 10th International Conference on Malicious and Unwanted Software (MALWARE)*, 2015.
23. S. Seufert and D. O'Brien. Machine learning for automatic defence against distributed denial of service attacks. In *2007 IEEE International Conference on Communications*, 2007.
24. D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. van den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot, S. Dieleman, D. Grewe, J. Nham, N. Kalchbrenner, I. Sutskever, T. Lillicrap, K. Leach, Madeleineand Kavukcuoglu, T. Graepel, and D. Hassabis. Mastering the game of Go with deep neural networks and tree search. *Nature*, 529:484 EP –, Jan 2016. Article.
25. J. Yuill, M. Zappe, D. Denning, and F. Feer. Honeyfiles: deceptive files for intrusion detection. In *Proceedings from the Fifth Annual IEEE SMC Information Assurance Workshop, 2004.*, pages 116–122.
26. L. Zhao and M. Mannan. Explicit authentication response considered harmful. In *Proceedings of the 2013 New Security Paradigms Workshop*, NSPW '13, 2013.

Chapter 3
Honeypot Deception Tactics

Neil C. Rowe

Abstract Honeypots on computer networks are most effective when they use deception to fool cyberadversaries into thinking that they are not actual decoy intelligence collectors. Honeypot deception can be made more effective when applied with variety. We discuss the range of deception tactics of which honeypots can take advantage. Ideas can come from deception theory, and honeypot deceptions can benefit from planning and experimentation.

3.1 Introduction

Defensive cyber deception is increasingly used against adversaries in cyberspace, cyberattackers, and cyberspies [23]. Honeypots and honeynets [12] are an efficient way to test defensive deception tactics for computer systems, mobile devices, and networks. Honeypots can be defined as network nodes with no purpose beyond collecting network security intelligence. A honeypot testbed offers much flexibility since a computer or device that is not burdened with many routine tasks can be reconfigured more easily. Detailed scripting of some interactions with adversaries is possible in advance, and most automated attacks will not notice deceptive details since they are not looking for them and rarely encounter deception. Furthermore, minimizing links to the honeypot other than its address means that there will be little chance of it being encountered by normal non-malicious non-spying users.

Honeypots are helpful for both cyberattacker adversaries (to collect intelligence about attack methods) and cyberspy adversaries (to find out what they are interested

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

N. C. Rowe (✉)
U.S. Naval Postgraduate School, Monterey, CA, USA
e-mail: ncrowe@nps.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_3

in). Honeypots can be extended to "honeynets" which include network resources as well [22]. Defensive deceptions tested on honeypots can also be used on non-honeypots or "production" systems [6] to help protect them. And even if a honeypot is detected, that knowledge by adversaries may provide good protection for a "fake honeypot" [17], defined as any production system subsequently installed there, and it will receive fewer future attacks [26].

Deception enhances honeypots because cyberadversaries want to avoid honeypots. Honeypots do not have useful information, they record attack methods to enable later thwarting of them, and they have safeguards to prevent them from being exploitable as launch points for other activity. Thus, a honeypot will be more useful if it conceals its purpose and the purpose of its data.

3.2 Honeypot Deception Options

Many deception taxonomies have been proposed. Sztompka et al. "Trust" [20] is a classic analysis of trust; [7] outlines military deception, which is useful because cyberattacks are like low-level warfare; [9] gives practical advice on mounting deceptions in general; and [3] provides a recent taxonomy. These can provide a menu of choices from which to construct plausible and convincing deceptions. It has often been observed that deception is more effective when it is varied and surprising [9], so having a large menu is desirable.

Here are specific suggestions for honeypot deceptions based on the "semantic" taxonomy of [18]:

- Deception in superconcept: Honeypots can masquerade just as cyberadversaries do. They can pretend to be particular kinds of sites, concealing or deceiving as to who owns them, who uses them, whether they are servers, and their purposes [5].
- Deception in object: Honeypots can offer bait in the form of fake files and fake data within files [18]. In the military world, this is an accepted part of counterintelligence. Other organizations and businesses are increasingly recognizing that some counterintelligence is useful for learning about their adversaries in cyberspace.
- Deception in purpose: Honeypots can run scripted or program-based interactions to confuse and thwart cyberadversaries [21]. This is especially helpful with industrial-control-systems honeypots because they run quite specific processes.
- Deception in external precondition: When an adversary is asking it to do something dangerous, too revealing, or too hard to simulate, a honeypot can provide an excuse why not to do it [18]. There are many possible excuses, but a good one should still leave the adversary with hope to achieve their primary goals. This can be achieved, for instance, by giving a sense of progress to the adversary such as asking for a password or some other form of authorization to suggest that part of the previous request was acceptable.

- Deception in result: Lying about results of actions may also be effective when honeypots are asked to do something they do not want to do. The easiest lie is to say that an action has been performed, so that a cyberadversary will waste time trying to exploit the non-existent result, as by reporting non-existent network nodes [2]. A response can also simulate a vulnerability so the adversary thinks that their attack has succeeded [13]. The main difficulty with deception in result is in maintaining the consistency subsequently. This can require secondary excuses, such as claiming protection violations when trying to open a non-existent file. Alternatively, the honeypot may be able to model effects and be able to predict secondary effects, as when a honeypot simulates a cyber-physical system with a mathematical model. Even a crude simulation may fool automated adversaries.
- Deception in time-through: In time-critical situations, delaying an attack can be critical to getting resources to fight it. Deliberate delays are often not considered suspicious by cyberadversaries because unexpected delays occur frequently with networks and automated software processes [16, 18].
- Deception in time-from and time-at: Bait such as documents, messages, and logs can be assigned times to give the appearance of realistic activity.
- Deception in agent: Bait can refer to fake people, sites, and services.
- Deception in recipient and location-at: An adversary's activities can be routed to a "sandbox" site where they can be better controlled [5, 19].
- Deception in value: Arguments to adversary commands can be changed as a variant on deception in result. For instance, files can be stored in a different directory than specified, giving the excuse of a cloud-based system. Or the results of an operation can be deliberately modified to make them unusable or less usable, as for instance by deliberately changing the encoding scheme on some file that is downloaded.
- Deception in experiencer: Recording adversary interactions with the honeypot is central to their design.

Once a deception method has been chosen, honeypot deceptions should identify an object, a presentation method, a purpose, and a target.

- Deceptions can be distinguished as to the digital object with which they are associated. This can be a large object such as a computer, mobile device, or network, or it can be a particular directory, file, or packet. The importance of surprise in deception suggests putting it in unexpected objects like software patches [4] and databases [25].
- Deceptions can be distinguished by how they are presented. They can be overt (stated in verbal messages) or indirect (inferred from failures or unexpected consequences of commands). They can be single events or sequences forming a campaign, as with "second-order" deceptions where one deception, designed to be discovered, is a setup for a more complex deception.
- Deceptions can be distinguished as to whether their purpose is to encourage, discourage, or manipulate. Traditional honeypots want to encourage exploration by visitors so that they can collect more data. Hence, their deceptions should try to offer what the visitor wants, which is easier for a high-interaction honeypot that

offers or simulates an operating system. However, giving the user everything they want right away is not usually as good as giving it to them in small pieces so that they will reveal more of their methods. "Fake honeypots" can try to discourage visitors as a means of protecting themselves or other nodes on their network. Alternatively, honeypots can try to manipulate the user in other ways such as in trying to identify them and in deliberately making them angry. A sophisticated honeypot could use automated planning to better manipulate the user.

- We should also choose deceptions based on knowledge of our adversaries. The skill level of the adversary is often apparent, and deceptions can be tailored to it [14]. Most adversaries connecting to a honeypot are cybercriminals, and we can expect automated attacks from them with often a low level of sophistication and a tendency to be easily discouraged. Cybercriminals have many easy targets on the Internet, so if they cannot do much with our honeypot, they will give up quickly and move on. If we want to encourage them so as to better collect data on their attack methods, we need to provide at least simple forms of bait. On the other hand, a nation-state adversary will have less likelihood of being discouraged. They have teams of professionals with long-term plans involving both espionage and cyberattacks, and may be willing to explore even low-interaction honeypots thoroughly. These are the kinds of adversaries with which the honeypot can most effectively play games. Many of these adversaries have quotas to fulfill, so sophisticated bait can be very effective.

3.3 Some Example Tactics

Deception tactics like those described are mostly active honeypot tactics compared to the traditional passive tactics of waiting to be attacked and collecting data about it. Active tactics may provide intelligence more quickly to the defender. Many of these tactics involve generating data to give to adversaries. It can be displays, system or network configurations, error messages, misinformation, or files.

Short messages can be generated by stochastic context-free grammars where each rule has a probability of being used in a top-down expansion of a starting symbol [18]. For instance, a quick grammar for short error messages like "Error at 392257016" can generate a random error-describing string drawn from a set of common real error messages followed by "at" and a random number of 6–12 digits. It could be sent to the honeypot adversary whenever an excuse is needed to avoid doing something. Random error messages can be made unpredictable and thus hard for an adversary to anticipate. Most cyberattacks are automated, and unexpected responses can be difficult for them to handle.

Randomization can be used to generate files of random characters. These tend to look like encrypted files and adversaries could waste time trying to decrypt them. However, adversaries can be kept interested much longer if files contain convincing real data. This could occur if the data is almost real, as when it is real data slightly modified to be harmless, or real data that from a previous time period that is useless now. For instance, we have been running a honeypot that appears to present pages

from our school library [15] with real technical documents that are out-of-date, with the goal of seeing which documents that cyberspies are most interested in. To build this, we collected a representative set of technical documents on a range of current topics for the honeypot. A similar idea is giving out-of-date vehicle positions in military data files since timeliness is critical for planning of military operations.

A more dynamic approach to honeypot deception is using "software wrappers" to manage it [18]. Wrappers are code that intervenes before the entry into some software and after the exit from it. They are helpful for debugging and have good infrastructure support. Wrappers can implement deceptions based on their context. For instance, they could generate false error messages if asked to work with a suspicious resource, or if asked repeatedly about a resource, or if they receive what appears to be malicious code. They could also delay in such cases, or substitute a different executable for the one referenced. Wrappers permit a graduated response to the degree of a cyberthreat due to the range of options and parameters they have at their disposal.

Specialized honeypots such as those for cyber-physical systems can offer additional tactics for deception from their modeling of the physical system [18]. For instance, suppose a cyberattacker of a honeypot power plant sends commands intended to close valves supplying coolant to the plant. The honeypot has several options:

- It could give an error message and refuse to do anything. This is a typical tactic of low-interaction honeypots. That could encourage the adversary to try something else. It also might discourage the adversary when they have not envisioned alternatives.
- It could fail to do anything but not say so. This is another common tactic of low-interaction honeypots. This would discourage the adversary eventually, but for a while the honeypot could collect additional intelligence.
- It could simulate the effects of the command if the intended effect could be seen easily by the adversary. So if told to close a coolant value, it could report increasing temperatures in routine packets and eventually alarm messages. This will generally encourage the adversary, but only until their goals are achieved which may not be long. It would require building a simulation of the physical system, though perhaps only a partial one for the features adversaries are interested in. This can be done for simple sensor networks but will be more difficult for a complex power plant. However, if the intended effect should be confirmable by another source of information (such as causing catastrophic failure of the power plant), the honeypot could be discovered.
- It could simulate the effects of the command but slowly. For instance, it could demand a password, or respond slowly while giving periodic updates ("Working on your request"). This could actually be encouraging because many adversaries expect and like a challenge. Eventually, a password can be accepted to see what the adversary will do next.
- It could simulate some of the intended effect but appear to thwart it. For instance, it could simulate closing a value, then quickly opening it to suggest manual

intervention; temperatures would go up and then down. This will encourage the adversary to try again, perhaps with a different method that will offer new insights about them. This may be better than refusing or delaying execution of commands because the adversary had partial success and may feel invested in the process, and may be more inclined to increase their efforts. It also may encourage an adversary who likes the challenge of active opposition, and may increase their sense of self-worth.

- It could create some new effect that the adversary did not anticipate. For instance, it could send alarm traffic across the simulated network (suggesting that the adversary has been detected), or it could cause a valve to open wider instead of closing (suggesting that the adversary used the wrong argument in the command), or it could cause new generators to start up (suggesting that the adversary used the wrong command). This will encourage the adversary to try again with a different method, increasing the intelligence collected.

3.4 Deception as a Game

Some researchers have modeled cyber deception as a formal game and taken advantage of results in game theory [8, 18, 24]. Deception has some distinctive features compared to other games, however. Psychologists have noted that distrust propagates much better than trust: It takes only one untrustworthy act to cause someone to lose trust that took many acts to build up. Thus, honeypots should be very cautious in offering what might be considered obvious clues to deception. That means going to some length to conceal the honeypot mechanisms in software, as with many products of the Honeynet Project (www.honeynet.org). In addition, honeypots should be careful with what bait they offer. With automated attacks, the bait may not be inspected, but either low or high quantities of bait strongly indicate a honeypot and should be avoided.

The easy propagation of distrust has implications for deception taxonomies as well. Figure 3.1 shows part of a honeypot deception taxonomy. We can model a cyberadversary as initially sensing one or more of the leaves of this tree. As they experience more of the honeypot, they may generalize their knowledge to nodes above their initial observations in this tree. The degree to which this distrust is propagated across a link can be expressed as a conditional probability. It can be estimated from the semantic similarity of the two concepts, but alternatively if we have statistics on instances of these concepts in the real world (the "extensions" of the concepts), it can be estimated as the size of the overlap between the extensions. The simplest overlap measure is the Jaccard formula, the ratio of the size of the set intersection to the size of the set union. Note that these estimates should be based if possible on what we think the adversary knows, and can be aided by experiments with human subjects.

Different observations by the cyberadversary can propagate independently and reinforce a conclusion much in the way associations propagate with Bayesian

network models. So, if adversary sees both odd behavior and an unusual delay, they may realize that both are indicators of a honeypot. Eventually, an adversary will accumulate enough evidence to be confident that they are viewing a honeypot. Then, its value is decreased, since the adversary will likely leave and stop providing data, and may tell others. Thus, we should try to prevent that by minimizing the clues.

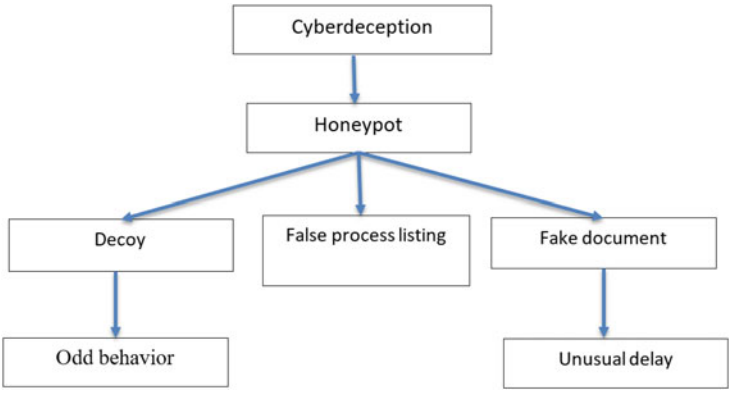

Fig. 3.1 Example deception taxonomy

Game theory is more useful when we have a set of honeypots (a honeynet) rather than just one. Then, we can test more tactics independently to see how well each works and what risks each entails [10]. Figure 3.2 shows an example decision tree for a honeynet. To analyze this, suppose that $\mathscr{P}_{hi}$ is the probability that the adversary will encounter a deception i when accessing the honeypot, $\mathscr{P}_{di}$ is the probability that deception where i succeeds in fooling the adversary, and b is the benefit to intelligence collection of each deception encountered (counting an opportunity for an additional deception when the adversary has been fooled twice). Assuming independence of the probabilities, the expected benefit is the sum of the expected costs over all leaf nodes n1 - n9:

$$\mathscr{P}_{h1}(1\text{-}\mathscr{P}_{di})\text{b} + \mathscr{P}_{h2}(1\text{-}\mathscr{P}_{d2})\text{b} + \mathscr{P}_{h3}(1\text{-}\mathscr{P}_{d3})\text{b}+$$
$$2\mathscr{P}_{h1}\mathscr{P}_{h2}\mathscr{P}_{d1}(1\text{-}\mathscr{P}_{d2})(2\text{b}) + 2\mathscr{P}_{h1}\mathscr{P}_{h2}\mathscr{P}_{d1}\mathscr{P}_{d2}(3\text{b})+$$
$$2\mathscr{P}_{h1}\mathscr{P}_{h3}\mathscr{P}_{d1}(1\text{-}\mathscr{P}_{d3})(2\text{b}) + 2\mathscr{P}_{h1}\mathscr{P}_{h3}\mathscr{P}_{d1}\mathscr{P}_{d3}(3\text{b})+$$
$$2\mathscr{P}_{h2}\mathscr{P}_{h3}\mathscr{P}_{d2}(1\text{-}\mathscr{P}_{d3})(2\text{b}) + 2\mathscr{P}_{h2}\mathscr{P}_{h3}\mathscr{P}_{d2}\mathscr{P}_{d3}(3\text{b})$$

We can calculate this for different honeypot and honeynet designs and choose the one with the highest average benefit. We should also recalculate periodically as adversaries readjust their tactics based on what they have seen.

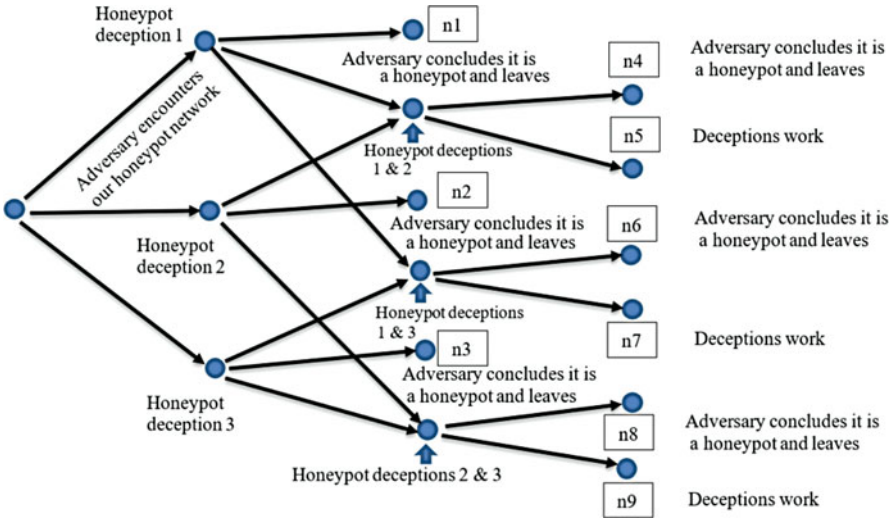

Fig. 3.2 Example decision tree for honeypot deceptions

3.5 Honeypot Experiments

Research is now conducting experiments with honeypot simulations [14] and live adversaries against honeypots [1, 13]. We have run honeypots for fifteen years [11, 18]. We encourage everyone doing network security to run them, as the hardware and software need not be up-to-date, they are easy to set up, and you often encounter surprises when you start monitoring them. We have run mostly ordinary machines as honeypots, plus a variety of honeypot software mostly from the Honeynet Project (www.honeynet.org) including Web-server, secure-shell, and industrial-control-system honeypots.

It often makes a considerable difference how a honeypot is configured. Small changes to the honeypot and its services can make major differences to the traffic observed. Also, the variety of traffic (not necessarily the volume) on a new honeypot is usually high at first and then decreases rapidly over a few weeks; a similar increase is observed after the honeypot is turned off for a while. Clearly, visitors are footprinting the machine and testing its susceptibility. This is useful for planning deceptions. A traditional honeypot that wants to encourage attacks should keep changing its appearance and keep going offline; a fake honeypot that wants to discourage attacks should avoid changing anything and try to stay online. The rate of change must be faster for low-interaction honeypots since they cannot keep a visitor's interest as long.

It is also useful to compare the same honeypot in different environments to see how much cyberadversaries exploit its context. Our experiments comparing a honeypot running at our school with a honeypot running the same hardware and software at a student's home, both using the same Internet service provider, showed

considerable differences in the traffic. This suggests that many adversaries routinely exploit Internet registry information, and that deceptions in this information or DNS data could aid honeypot effectiveness. Our current work is focused on industrial control systems with an emphasis on power plants. We are getting a significantly higher rate of traffic than with the conventional SSH honeypots we have run. This work is focusing on the simulation of processes as discussed in Sect. 3.3.

3.6 Exercises

1. Networked home-monitoring systems could be a possible target by cybercriminals for harassment or extortion purposes. A honeypot home-monitoring system not associated with a real home could collect intelligence on what cybercriminals are trying to do. Assume a design to control the heating, air conditioning, lighting, and alarm system of a house.
   a. Suggest possible deceptions in "result" that could be effective for such systems and explain how they would be implemented.
   b. Suggest possible deceptions in "object" different from "result" that could be effective for such systems and explain how they would be implemented.
   c. How could the honeypot respond to attempts to turn off all the lights in a way that could encourage further interaction?
   d. How could the honeypot respond to periodic attempts to modify parameters, such as every day, in such a way that the adversary will keep returning?
2. Consider the problem of measuring the effectiveness of a honeypot's deceptions.
   a. If we measure traffic volume, what should we compare to assess effectiveness?
   b. How could it be useful to measure traffic to nodes other than the honeypot to assess the effectiveness of the honeypot?
   c. If we measure changes made to the honeypot, what should we examine to assess effectiveness?

References

1. P. Aggarwal, C. Gonzalez, and V. Dutt. Looking from the hacker's perspective: Role of deceptive strategies in cyber security. In *2016 International Conference On Cyber Situational Awareness, Data Analytics And Assessment (CyberSA)*, pages 1–6, June 2016.
2. N. Al-Gharabally, N. El-Sayed, S. Al-Mulla, and I. Ahmad. Wireless honeypots: Survey and assessment. In *Proceedings of the 2009 Conference on Information Science, Technology and Applications*, ISTA '09, pages 45–52, New York, NY, USA, 2009. ACM.

3. M. H. Almeshekah and E. H. Spafford. Planning and integrating deception into computer security defenses. In *Proceedings of the 2014 New Security Paradigms Workshop*, pages 127–138, New York, NY, USA, 2014. ACM.
4. F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 942–953, New York, NY, USA, 2014. ACM.
5. K. Borders, L. Falk, and A. Prakash. Openfire: Using deception to reduce network attacks. In *SecureComm*, pages 224–233. IEEE, 2007.
6. F. De Gaspari, S. Jajodia, L. V. Mancini, and A. Panico. Ahead: A new architecture for active defense. In *Proceedings of the 2016 ACM Workshop on Automated Decision Making for Active Cyber Defense*, SafeConfig '16, pages 11–16, New York, NY, USA, 2016. ACM.
7. J. Dunnigan and A. Nofi. *Victory and Deceit, Second Edition: Deception and Trickery in War*. Writers Club Press, San Jose, CA, US, 2001.
8. C. D. Faveri, A. Moreira, and V. Amaral. Goal-driven deception tactics design. In *2016 IEEE 27th International Symposium on Software Reliability Engineering (ISSRE)*, pages 264–275, Oct 2016.
9. C. Fowler and R. Nesbit. Tactical deception in air-land warfare. *Journal of Electronic Defense*, 18(6), 1995.
10. D. Fraunholz and H. D. Schotten. Strategic defense and attack in deception based network security. In *2018 International Conference on Information Networking (ICOIN)*, pages 156–161, Jan 2018.
11. E. E. Frederick, N. C. Rowe, and A. B. G. Wong. Testing deception tactics in response to cyberattacks, In *Proceedings of the National Symposium on Moving Target Research*, Annapolis, Maryland, US, June 2012. Retrieved from http://faculty.nps.edu/ncrowe/rowe_mtr_realtimedecep.htm, March 10, 2018.
12. F. Girtler. *Efficient Malware Detection by a Honeypot Network*. AV Akademikerverlag, 2013.
13. X. Han, N. Kheir, and D. Balzarotti. Evaluation of deception-based web attacks detection. In *Proceedings of the 2017 Workshop on Moving Target Defense*, MTD '17, pages 65–73, New York, NY, USA, 2017. ACM.
14. S. Hassan and R. Guha. A probabilistic study on the relationship of deceptions and attacker skills. In *2017 IEEE 15th Intl Conf on Dependable, Autonomic and Secure Computing, 15th Intl Conf on Pervasive Intelligence and Computing, 3rd Intl Conf on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech)*, pages 693–698, Nov 2017.
15. S. F. McKenna. Detection and classification of web robots with honeypots, 2016. Retrieved from http://faculty.nps.edu/ncrowe/oldstudents/28Mar_McKenna_Sean_thesis.htm, March 3, 2018.
16. P. Pal, N. Soule, N. Lageman, S. S. Clark, M. Carvalho, A. Granados, and A. Alves. Adaptive resource management enabling deception (armed). In *Proceedings of the 12th International Conference on Availability, Reliability and Security*, ARES '17, pages 52:1–52:8, New York, NY, USA, 2017. ACM.
17. N. C. Rowe, E. J. Custy, and B. T. Duong. Defending cyberspace with fake honeypots, 2007.
18. N. C. Rowe and J. Rrushi. *Introduction to Cyberdeception*. Springer Publishing Company, Incorporated, 1st edition, 2016.
19. M. P. Stoecklin, J. Zhang, F. Araujo, and T. Taylor. Dressed up: Baiting attackers through end-point service projection. In *Proceedings of the 2018 ACM International Workshop on Security in Software Defined Networks & Network Function Virtualization, SDN-NFVSec@CODASPY 2018, Tempe, AZ, USA, March 19–21, 2018*, pages 23–28, 2018.
20. P. Sztompka. *Trust*. Cambridge University Press, London, UK, 1999.
21. J. Tammi, S. Rauti, and V. Leppänen. Practical challenges in building fake services with the record and play approach. In *SIN*, pages 235–239. ACM, 2017.

22. V. E. Urias, W. M. S. Stout, and H. W. Lin. Gathering threat intelligence through computer network deception. In *2016 IEEE Symposium on Technologies for Homeland Security (HST)*, pages 1–6, May 2016.
23. C. Wang and Z. Lu. Cyber deception: Overview and the road ahead. *IEEE Security Privacy*, 16(2):80–85, March 2018.
24. W. Wang, J. Bickford, I. Murynets, R. Subbaraman, A. G. Forte, and G. Singaraju. Catching the wily hacker: A multilayer deception system. In *2012 35th IEEE Sarnoff Symposium*, pages 1–6, May 2012.
25. M. Wegerer and S. Tjoa. Defeating the database adversary using deception - a MySQL database honeypot. In *2016 International Conference on Software Security and Assurance (ICSSA)*, pages 6–10, Aug 2016.
26. A. Zarras. The art of false alarms in the game of deception: Leveraging fake honeypots for enhanced security. In *48th IEEE International Carnahan Conference on Security Technology (ICCST)*, October 2014.

Part II
Dynamic Decision-Making for Cyber Deception

Chapter 4
Modeling and Analysis of Deception Games Based on Hypergame Theory

Jin-Hee Cho, Mu Zhu, and Munindar Singh

Abstract In this chapter, we discuss a deception game where attackers and defenders can have different perceptions towards a given situation. Although existing game theories have considered incomplete information to consider uncertainty, how players' different perceptions or misperceptions can affect their decision-making has not been fully addressed. In particular, we discuss *hypergame theory* which has been used to resolve conflicts under uncertainty. In this chapter, we examine how a player's perception (or misperception) affects their decision-making in choosing a best strategy based on hypergame theory. To deliver a concrete idea on how the attack–defense game can be modeled based on hypergame theory, we model a simple cybergame scenario and demonstrate an example probability model using Stochastic Petri Nets. Through the evaluation of the model, we show the experimental results to deliver insightful findings in terms of the relationships between perceptions by different players (i.e., an attacker or a defender), their chosen best strategies, and corresponding utilities. Lastly, we measure performance of the attacker and the defender in terms of *attack success probability* and *mean time to security failure*.

4.1 Introduction

Conflict situations are traditionally modeled based on game theory and decision theory, and more recently via hypergame theory. However, the applicability of each of these theories depends on the amount of information available. For example,

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

J.-H. Cho (✉)
Virginia Tech, Falls Church, VA, USA
e-mail: jicho@vt.edu

M. Zhu · M. Singh
North Carolina State University, Raleigh, NC, USA

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_4

game theory is useful to model a game in conflict with little information towards a situation (e.g., little knowledge about opponents). And, if complete information is known towards a game, decision theory can provide effective strategies for players to make decisions with maximum utility. If players are not fully aware of the situation with only partial information observed, each player will play a different game, which can be modeled based on hypergame theory [21]. Considering real-life situations in which we often face highly uncertain situations with only partial information, hypergame theory is designed to provide a form of analysis considering each player's subjective perception, misperception, and perceived uncertainty and accordingly their effect on their decision-making in choosing a best response [21].

Let us first motivate hypergame theory as a useful approach to model interactions between an attacker and a defender and to analyze their best strategies (or responses) and associated utilities, particularly where deception is placed as a defense mechanism. In this book chapter, we would like to contribute a game-theoretic approach that is not well-studied in the state of the art related on cybersecurity. The specific motivations and the corresponding contributions are as follows:

1. Although a game based on partial observations has been studied in existing approaches [7, 24], different perceptions, misperception, and prior beliefs due to their limited observability or confusion caused by an opponent player have not been fully addressed yet. Therefore, in this book chapter, by providing modeling details on system security state, an attacker, and a defender based on hypergame theory, we would like to encourage readers to leverage the powerful capability hypergame theory can offer in terms of considering various types of defense techniques in both proactive (e.g., deception or moving target defense) and reactive (e.g., intrusion detection) mechanisms.
2. Even if the usefulness of hypergame theory has been recognized, relatively less matured, structured modeling details have been provided in the literature. This chapter aims to fill the gap. In this chapter, we demonstrate a game between an attacker and a defender where the game is designed based on hypergame theory and the defender aims to mislead the attacker's perception by using deception techniques.

The rest of this chapter is structured as follows. Section 4.2 gives a brief summary of deception techniques and game-theoretic approaches to model cybergames between attackers and defenders. Section 4.3 provides the key concepts of hypergame theory for readers to grasp their in-depth understanding in modeling a game between an attacker and a defender. Section 4.4 discusses an example scenario, an attacker's game, a defender's game, and their corresponding hypergame expected utilities (HEUs). Section 4.5 shows an example analytical model using Stochastic Petri Nets (SPN) in order to deliver more concrete ideas on modeling and analyzing an attack–defense game based on hypergame theory. Section 4.6 discusses experimental results and lessons learned from the results. Section 4.7 concludes this chapter.

4.2 Related Work

In this section, we provide an overview of related work in terms of deception techniques and game-theoretic defense approaches.

Deception Techniques Deception techniques are developed to mislead an attacker to achieve a defender's goal [26]. This technique can be used at any level of interactions between attackers and defenders in a wide range of conflicting situations. Due to their high applicability, various types of deception strategies have been considered to confuse decision-makers for military operations [26].

Deception techniques can be categorized into two types, *hiding the truth* and *providing false information*. Deception defense seeks to mislead attackers by intentionally presenting false or incomplete information, which can lead the attacker to misdiagnose the quality or relevance of information [3, 4]. Caddell [6] categorizes deception as *passive* vs. *active* where passive deception means hiding information whereas active deception means disseminating false information. In addition, Daniel and Herbig [10] explain the purpose of deception defense as increasing ambiguity (indicating uncertainty or confusion) for attackers to be misled for their decision-making.

Game-Theoretic Approaches to Deal with Advanced Persistent Threat (APT) Game-theoretic approaches have been proposed to deal with APT attacks [1, 11, 19, 24, 34]. Hu et al. [19] modeled a two-layer game considering both APT and inside attacks and investigated Nash equilibria under both static and dynamic actions by players. Similarly, Feng et al. [12] considered a three-player game where the players include an attacker, a defender, and an insider. Fang et al. [11] modeled different attack strategies based on attack paths that incur different levels of cost and benefit. Abass et al. [1] analyzed APT attack strategies and corresponding defense strategies for cloud computing environments based on the evolutionary game theory and developed a dynamic APT game where defenders can apply dynamic learning rules for attack behaviors. Van Dijk et al. [28] and Zhang et al. [34] modeled highly intelligent attackers that can have perfect knowledge towards defenders' behaviors and investigated the Nash equilibria in terms of the defender's responses to the attack strategies. Rass et al. [24] considered various aspects of uncertainty caused by APT behaviors such as unexpected attack responses to assessed risk, unknown incentives to performed attacks, and unknowns about a current system state.

Game-Theoretic Approaches with Deception Techniques Deception techniques are modeled in game-theoretic approaches. Yin et al. [33] modeled a Stackelberg attack–defense game where both players make decisions based on their observations, not the actual strategies taken by the players, and examined the benefit of using deception techniques. Similarly, Garg and Grosu [15] also considered an attack–defense game and modeled a deception technique with a honeynet where a defender aims to confuse an attacker. Carroll and Grosu [7] built a signaling attack–defense game, which is a non-cooperative game with incomplete information where each

player aims to take a best strategy, such as improving a honeypot by a defender or compromising a honeypot by an attacker.

Hypergame-Theoretic Approach Hypergame theory has been used to deal with players' subjective, imperfect perception, which can lead their beliefs to uncertain or non-optimized strategies that better reflect the real-world scenarios rather than assuming that all players play the *same* game with perfect (or imperfect) knowledge of each other. Although other game theories deal with uncertainty by considering probabilities that a certain event may happen, they assume that all players play the same game [27]. Gharesifard and Cortés [16, 17] studied how players' perception evolves based on hypergame theory. They developed a misperception function to measure differences between a player's perception and the ground truth payoff of other players' strategies. Kanazawa et al. [20] proposed *evolutionary hypergame theory* to develop an interpretation function that considers individuals' subjective perception in hypergame theory. Aljefri et al. [2] developed a misperception model for decision-making situations in conflict based on a first-level hypergame. Sasaki [25] proposed a new solution concept called *subjective rationalizability* by defining an agent's action as *subjectively rationalizable* when the agent believes that the action is a best response to the other agent's choices based on its perceived game.

Hypergame theory has been used to solve decision-making problems in military and adversarial environments [18, 31, 32]. Vane and Lehner [32] applied hypergame theory to plan tactical strategies under uncertain situations, while Vane [31] proposed a decision framework based on hypothesized simulated emergency situations. Likewise, House and Cybenko [18] applied hypergame theory to model a cyberattack–defense game that is affected by players' understanding of rules, expertise, and misperception. Putro et al. [23] proposed and investigated the procedures of adaptive learning to derive accurate strategies and preferences used by players in a hypergame context by using a genetic algorithm.

4.3 Hypergame Theory

For concreteness, we now briefly discuss hypergame theory and its fundamental concepts that underlie our deception game. Bennett [5] first proposed the concept of *hypergame theory* by questioning the assumption of *common perception* maintained by players in game theory. In real situations, players do not have common perceptions about a game. Instead, their understanding about the game may be radically different and each player plays a different game although multiple players may be involved. Hypergame theory provides a form of analysis in which a system consists of a set of games, each of which represents a game each player plays based on its own ideas towards a given situation.

Here, we discuss two levels of hypergames that can be used to analyze games differently perceived by multiple players [13]. Although hypergame theory can be applicable for n players, since we consider a deception game by two players, an attacker and a defender, we describe hypergame theory based on two players.

First-Level Hypergame Given two players, vectors of two players' preferences, denoted by V_p and V_q, define a game G that can be represented by [13]:

$$G = \{V_p, V_q\} \tag{4.1}$$

Note that V_p and V_q are player p's and player q's actual preferences (i.e., ground truth), respectively. If all players exactly know all other players' preferences, all players are playing the same game because their view towards the game is the same. However, in reality, complete information may not be available and accordingly each player will form its own view, which may be different from the opponent's view. Under this situation, player p can perceive player q's preferences based on its own perception, which causes differences between p's view and q's view. A game perceived by player p based on its perceived preferences towards q's preferences, V_{qp}, and the game perceived by player q based on its perceived preferences towards p's preferences, V_{pq}, can be given by:

$$G_p = \{V_{qp}\}, \quad G_q = \{V_{pq}\} \tag{4.2}$$

Hence, the first-level hypergame H perceived by each player is written by:

$$\mathbf{H}^1 = \{G_p, G_q\} \tag{4.3}$$

In a first-level hypergame, analysis is performed at the level of each player's perceived game because each player plays the game based on its perception. Even if the player does not know all outcomes of the game, the outcome can be stable for the player because the player may not unilaterally change its perception. If a game includes an unknown outcome, the unknown outcome is caused by the uncertainty. The stability of an outcome about a game is determined for each player even if each outcome is the return by each player's reaction towards the action by the opponent. An outcome is *stable* for p's game if the outcome is stable in each of p's perceived preference vectors, V_{qp}'s. The equilibrium of p's game is determined by the outcome p believes to resolve the conflict [13].

Second-Level Hypergame A second-level hypergame occurs when at least one player is aware of another player's misperception so that there exists a hypergame being played by other players. The hypergame perceived by p or q is denoted by [13]:

$$H_q = \{G_{pq}\}, \quad H_p = \{G_{qp}\} \tag{4.4}$$

Note that p or q may not know a game played by the opponent. Hence, p may not know how q interprets the game. A second-level hypergame is notated by:

$$\mathbf{H}^2 = \{H_p, H_q\} \tag{4.5}$$

Given a two-player game, the stability of each player's game is based on what the player perceived its own game and the other player' hypergame. Based on the stability of the outcome based on the perception of the other player's game, the equilibrium of the hypergame by each player is determined.

Hypergame Normal Form (HNF) Vane [29] provides a hypergame normal form (HNF) that can efficiently model hypergames based on players' beliefs and possible strategies of their opponents. HNF is formulated in a manner similar to the normal strategic form in game theory. HNF consists of the following four sections: (1) full game; (2) row-mixed strategies (RMSs); (3) column-mixed strategies (CMSs); and (4) belief contexts.

The **full game** is the grid form consisting of row and column strategies, which are associated with the utilities, $ru_{11}, \cdots, ru_{mn}$ and $cu_{11}, \cdots, cu_{mn}$ where n is the number of the column player's strategies and m is the number of the row player's strategies. The full game's grid form can be represented by:

$$\mathbf{U} = \begin{pmatrix} (ru_{11},\ cu_{11}) & \cdots & (ru_{1n},\ cu_{1n}) \\ \cdots & \cdots & \cdots \\ (ru_{m1},\ cu_{m1}) & \cdots & (ru_{mn},\ cu_{mn}) \end{pmatrix} \tag{4.6}$$

This full game strategies by the row player is denoted by R_0, whereas the full game strategies by the column player (believed by the row player) is represented by C_0.

Row-Mixed Strategies Row-mixed strategies (RMSs) are the strategies the row player comes up with based on its perception of the column player's strategies. A player's subgame is defined as a subset of the full game (i.e., a set of all possible strategies by all players) because the player may limit a number of strategies it wants to consider based on its own perceived situation. Therefore, depending on the situation, the player can choose the subgames to play. RMSs for the k-th subgame by a game a player perceives to play due to its limited observations and beliefs are given by:

$$RMS_k = [r_{k1}, \cdots, r_{km}], \quad \text{where } \sum_{i=1}^{m} r_{ki} = 1. \tag{4.7}$$

For example, in the rock–paper–scissors game, player p may consider either rock or scissors as a first subgame, RMS_1 is defined by $[r_{1,rock}, r_{1,scissors}] = [0.3, 0.7]$ where each probability that a particular strategy is chosen is estimated by player p's belief which may be formed based on learning from past experience.

Column-Mixed Strategies Column-mixed strategies (CMSs) are a column player's strategies believed by the row player for a k-th subgame and denoted by:

$$CMS_k = [c_{k1}, \cdots, c_{kn}], \quad \text{where} \sum_{j=1}^{m} c_{kj} = 1. \tag{4.8}$$

Similarly, as an example, in the rock–paper–scissors game, player p may predict q's strategies based on its belief as CMS_1 is defined by $[c_{1,rock}, c_{1,paper}] = [0.6, 0.4]$ where each probability that a particular strategy is chosen is obtained by player p's learning towards q's strategies based on the observations made towards q's playing.

Belief Contexts Belief contexts are the row player's belief probabilities that each subgame k will be played and are represented by:

$$P = [P_0, \cdots, P_K], \quad \text{where} \sum_{k=0}^{K} P_k = 1. \tag{4.9}$$

P_0 is the probability that the full game is played where the full game considers all possible strategies a player has based on the ground truth view of a situation. If the row player does not know whether a particular subgame k will be played (i.e., uncertainty), the unknown belief probability is treated simply as 0 and P_0 is computed by:

$$P_0 = 1 - \sum_{k=1}^{K} P_k \tag{4.10}$$

The row player's belief towards the column player's strategy j, denoted by S_j, is computed by:

$$S_j = \sum_{k=0}^{K} P_k c_{kj} \quad \text{where} \sum_{j=1}^{m} S_j = 1. \tag{4.11}$$

The summary of the row player's belief on m number of the column player's strategies is represented by:

$$C_{\sum} = [S_1, S_2, \cdots, S_m]. \tag{4.12}$$

Each of a row player's strategy can be evaluated based on the full game strategies by a column player to derive the utility for the worst-case column strategy (CMS_w) (i.e., lowest utility) or the expected utility (EU) for the column player's strategies

under each subgame. The hypergame expected utility (HEU) can be calculated based on $EU(\cdot)$, and the probability that a row player does not know for a given situation (i.e., uncertainty), denoted by g. Thus, g represents the level of uncertainty towards what is guessed about a given game, which significantly affects the degree of the EU of a given hyperstrategy by the row player. HEU for the given row player's strategy rs_i with the uncertainty g is given by [30]:

$$HEU(rs_i, g) = (1 - g) \cdot EU(rs_i, C_{\sum}) + g \cdot EU(rs_i, CMS_w), \tag{4.13}$$

where rs_i is a given strategy i by the row player. $EU(rs_i, C_{\sum})$ and $EU(rs_i, CMS_w)$ are computed by:

$$EU(rs_i, C_{\sum}) = \sum_{j=1}^{m} S_j \cdot u_{ij}, \quad EU(rs_i, CMS_w) = \sum_{j=1}^{m} c_{wj} \cdot u_{ij}. \tag{4.14}$$

When $g = 0$, meaning complete confidence (i.e., complete certainty) in a given strategy, strategies are selected like decision theory (DT). Conversely, if $g = 1$, implying that the row player is completely occupied with the fear of being outguessed (i.e., complete uncertainty because of being afraid that the chosen strategy is wrong), the strategy selection is made like traditional game theory (GT) [29].

4.4 Case Study

To provide more concrete ideas on modeling an attack–defense game based on hypergame theory, we introduce an example game scenario, associated modeling of an attacker and a defender, and their utility calculations based on hypergame theory.

We consider an enterprise system that consists of multiple components. We abstract out how each component is connected to each other. Instead, we specify how many components exist and their vulnerability levels. If the components have higher vulnerabilities, it can naturally lead to an attacker being able to compromise a system component with a higher probability changed. In this work, we don't assume that there is a recovery mechanism. Hence, once a component is compromised, it cannot be recovered. We leave the design of the recovery mechanism for future research. We assume that the system is equipped with a network-based IDS that can capture inside attackers after an attacker penetrated into the system.

The system can fail if either of the following conditions is met: (1) when more than one third of its components are compromised based on the concept of

Byzantine failure (i.e., breach of system integrity and availability) [14]; and (2) when a confidential information is leaked out to unauthorized parties (i.e., breach of data integrity), ultimately leading to the disruption of the entire system. Now, we discuss how an attacker or a defender plays a given game in the following sections.

We consider attack behaviors based on the characteristics of Advanced Persistent Threat (APT) attacks derived from the concept of cyber kill chain (CKC) [22]. APT has been emerged as one of the most serious threats that use advanced skills and has the nature of persistent threat with the long-term control of a targeted system. To give better ideas on the example scenario, we describe the well-known life cycle of APT consisting of the following stages:

- *Reconnaissance* is a stage for attackers to gather basic information of a target in order to investigate the victim before deploying an attack and prepare various tools for different attack vectors as well.
- *Delivery* represents a stage for attacker delivering their exploit to the target. Familiar actions include social engineering attack (e.g., phishing email) or water-hole attack (i.e., attacker seeks to compromise users by infecting websites or software that members of target are known to visit) [8].
- *Exploitation* describes a stage after an attacker's penetration into a victim. Once a malware gets inside a targeted organization network, the downloaded malware is eventually installed and activated. A security channel between the compromised system and Command and Control (C&C) is created to support future attacks. In addition, the attacker stealthily continues to collect system information, including security configurations, installed software versions, and directory listings of network shared folders.
- *Data collection and exfiltration* are the stages in which attackers leak and transfer the collected information to multiple external servers.

In the present scenario, we simply consider the three stages: (1) reconnaissance; (2) delivery; and (3) exploitation and data collection and exfiltration. To make the system fail, the attacker can perform critical attacks such as compromising system components or leaking out confidential information. We combine the last two stages because an inside attacker can freely perform either compromising any vulnerable system component based on available information or leaking obtained information out. For simplicity, we just call the third stage in our category as "exploitation" which allows the attacker to perform both types of attacks.

4.4.1 Attacker's Game

The attacker has four types of strategies as attack behaviors but considers a subset of them depending on the stage it is in. The example four types of attack strategies

include attacks in each of three stages and a stealthiness attack which does not perform any attack with a strategic reason not to be detected soon until it gains more intelligence. To be specific, we assume that an attacker is *subjectively rational* to choose its best strategy based on its perception to maximize its HEU by considering the following strategies where attack strategy i is denoted by AS_i:

- **AS_1**: This strategy can be used when an attacker just started to perform attacks so it is not in a target system yet. Or, it can be applied when an attacker is in the reconnaissance stage with the purpose of gaining intelligence before it goes to a next step. This attack strategy is called *reconnaissance attack*. The attacker aims to identify a targeted system by scanning its attack surface. In this stage, the attacker is an outsider and consumes its resources to identify a vulnerable targeted system.
- **AS_2**: This strategy can be used when an attacker is in the delivery stage with the success of the reconnaissance attack. This attack strategy is called *delivery attack*. In this attack, the attacker aims to successfully deliver the exploit to the targeted system by performing attacks such as social engineering attacks (e.g., phishing attacks). The successful outcome of this attack leads the attacker to penetrate into the system, making it an inside attacker.
- **AS_3**: This strategy can be applied when an attacker is in the exploitation stage. With this strategy, the attacker stays stealthy in the system not to be detected by the system (i.e., an IDS). We call this attack *stealthiness attack*. If a disseminated patch is real to patch system vulnerabilities, this strategy may have the attacker lose its chance to launch an attack to compromise vulnerable components or leak out confidential information. However, if the attacker aims to perform attacks later and to avoid exposing a risk of being caught by the IDS, it chooses this strategy.
- **AS_4**: This strategy can be applied when an attacker is in the exploitation stage. But unlike AS_3, the attacker actively performs attacks by compromising vulnerable components or leaking confidential information out. This attack is called *exploitation attack* or the *exfiltration attack*. If a real patch is disseminated, the attacker can leverage the defense strategy in order to identify vulnerable components more easily. However, if a fake patch is disseminated and it is deceived by the fake patch, it will waste its time or energy to perform their attacks.

Table 4.1 Attacker's subgames

Subgame	Stage in CKC	Attack strategies	Defense strategies
1	Reconnaissance	AS_1, AS_2	DS_1, DS_2, DS_3, DS_4
2	Delivery	AS_2, AS_3	DS_1, DS_2, DS_3, DS_4
3	Exploitation	AS_3, AS_4	DS_1, DS_2, DS_3, DS_4

In this work, we design an attacker's subgames in terms of which stage an attacker is in the CKC. In Table 4.1, we summarize what attack strategies an attacker will consider to deal with types of defense strategies it believes to be taken by a defender depending on its stage in the CKC. We don't restrict the types of defense strategies an attacker considers but restricting only the attack strategies the attacker can take in terms of feasibility that the strategy can be effective based on its stage in the CKC. Although the attacker considers all four defense strategies in its subgames, its belief in whether a particular defense strategy is chosen is subjectively determined based on its perception. Note that which subgame is played is determined based on which stage an attacker is located in. In this work, we consider a dynamic selection of a subgame to be played at time t because the attacker's stage in the CKC can change the attacker's decision on which subgame it wants to play. That is, each game (i.e., an attacker chooses its best strategy based on its HEU, while a defender chooses its best strategy based on its HEU) is played at each time t. Hence, when a certain subgame, say k, is selected, we can say $P_k = 1$ and the probabilities of all other subgames are set to zero. But at a different time point, a different subgame may be played by each player.

The **attacker's goal** is to maximize its utility where the **attacker's utility** increases with: (1) increasing attack success probability; (2) decreasing attack cost; and (3) decreasing probability of being detected by the IDS, implying high stealthiness. The *attacker's utility* for j strategy with the defender's i strategy, cu_{ij}, considers the gain and loss from the following:

- Whether a given attack strategy j can contribute to attack success by exploiting system vulnerabilities, denoted by $AEU(DS_i, j)$, referring to the *attack effect* when a defender takes DS_i strategy;
- Whether a chosen strategy exposes high vulnerabilities to be detected by the IDS, denoted by $ASU(DS_i, j)$, indicating the *detection vulnerability* when DS_i is taken by the defender; and
- How much cost the chosen strategy j incurs, denoted by $ACU(DS_i, j)$, which refers to the *attack cost* with DS_i strategy taken by the defender.

Then, the **attacker's utility**, u_{ij}^A, is computed by:

$$u_{ij}^A = w_1 \frac{AEU(DS_i, j)}{u_{max}} + w_2 \frac{u_{max} - ASU(DS_i, j)}{u_{max}} + w_3 \frac{u_{max} - ACU(DS_i, j)}{u_{max}} \tag{4.15}$$

where $w_1 + w_2 + w_3 = 1$ and each utility score associated with attack effect, detection vulnerability, and attack cost ranges over $[u_{min}, u_{max}] = [0, 3]$ as an integer, where a higher score represents higher degree. Each weight w_i is determined based on attackers' objectives. Note that we used a weighted form to consider multiple objectives in this utility function, which is a well-known scalarization multi-objective optimization function [9]. For the detection vulnerability, $ASU(DS_i, j)$, it

means the degree that an attacker is detected by an IDS. Note that an attacker's HEU is computed based on three components: (1) attack utilities when i's attack strategy is played under j's defense strategy (i.e., u_{ij}^A's); (2) the attacker's belief towards a defense strategy j (i.e., S_j^A as in Eq. 4.11); and (3) the degree of uncertainty, g_A, which is considered based on the attacker's capability to perceive the security state of a given system (see Eq. 4.17).

4.4.2 Defender's Game

A defender is assumed *subjectively rational* to maximize its HEU based on its perception towards a given situation and an attacker. In this work, we assume that an intrusion detection system (IDS) executes continually and can detect any malicious entities. Additionally, the defender considers a *defender strategy*, denoted DS_i, for strategy i, as follows:

- **DS$_1$**: A defender disseminates a fake patch in order to lure an attacker to a honeypot. This strategy can often be used when the system is not highly vulnerable so it can more actively protect the system, instead of patching or changing a platform under more vulnerable states. To this end, the defender needs to consider cost for building and maintaining the honeypot as the part of its defense cost. If the attacker recognizes the honeypot as fake, it will not actively access the targeted system component based on the patch to save its resource. This defense is to deal with inside attackers in the exploitation stage, where the attackers are legitimately present within the system and aim to exploit vulnerabilities of system components.
- **DS$_2$**: A defender disseminates a real patch when some system components have vulnerabilities. The vulnerable components (e.g., software vulnerability) must be patched to mitigate the risk of the vulnerabilities that may be exploited by attackers. If an attacker believes in this real patch, it may perform a zero-day attack, which may lead to a certain level of damage (or risk) to the system. If the attack is successful, the attacker can compromise vulnerable components or leak out confidential information obtained from the vulnerable target entity to unauthorized outside attackers.
- **DS$_3$**: A defender can perform a moving target defense (MTD) technique such as platform migration by changing operating systems or databases, IP mutation, or network topology change to affect attack paths, when the system is detected as more vulnerable. In this work, we consider a platform migration as an MTD. If this defense strategy is deployed, an attacker's intelligence collected towards the given system is wiped out and the attacker needs to start from the reconnaissance stage.

Table 4.2 Defender's subgames

Subgame	System security state	Defense strategies	Attack strategies
1	High ($N_h \geq 90\% \ \wedge$ No information leak-out)	DS_1, DS_4	AS_1, AS_2, AS_3, AS_4
2	Mid ($80\% \leq N_h < 90\% \ \wedge$ No information leak-out)	DS_1, DS_2, DS_3	AS_1, AS_2, AS_3, AS_4
3	Low ($N_h < 80\% \ \wedge$ No information leak-out)	DS_2, DS_3	AS_1, AS_2, AS_3, AS_4

Note: N_h refers to the fraction of healthy system components over the total number of initially given components. Therefore, even if all compromised components are detected by an IDS, the loss of available assets is also counted to consider a system failure because it will affect mission performance capability (e.g., not properly providing normal services or functionalities due to lack of critical components)

- **DS_4**: If the defense cost is a major concern or the system is secure enough, a defender may choose not performing any additional defense mechanisms except running its IDS. During this defense strategy, if a reconnaissance attack or delivery attack is successful, the system becomes vulnerable, leading to a compromise of components that have exposed vulnerabilities. Otherwise, the system can maintain system operations with minimum resource consumption to ensure security (i.e., running a periodic IDS).

A defender can choose a subgame based on its perceived system security state. As the system failure conditions are discussed based on two conditions (i.e., either more than one third of components are compromised or a critical, confidential information is leaked out), the defender will choose an appropriate action based on its perceived system state. Table 4.2 summarizes the defender's subgame depending on its perceived system state and its corresponding strategies towards attack strategies. As for the attacker's subgame, at each time t, the defender will choose a particular subgame based on its perceived system state. This means that the defender will choose a particular subgame k with $P_k = 1$ which implies that other subgames will not be chosen. But since this game is a repeated game, the defender will choose a different subgame at a different time point because the system state will keep changing over time.

The **defender's goal** is to maximize the gaps between the ground truth system state based on defense strategies performed and the system state perceived by the attacker so that the attacker does not choose its optimal strategies based on the ground truth system state and accordingly an accurate view towards the given game. The **defender's utility** increases with: (1) increasing system lifetime (e.g., increasing mean time to security failure, or MTTSF), (2) decreasing system vulnerabilities, and (3) decreasing defense cost. The *defender's utility* for attacker's strategy j with defender's strategy i, u_{ij}^D, is computed based on the following:

- Whether given defense strategy i can contribute to mitigating or preventing the effect of attack strategy AS_j (i.e., attack success), which is denoted by $DEU(AS_j, i)$ and called *defense effect*;

- How much given defense strategy i will expose system vulnerabilities upon AS_j, represented by $DSU(AS_j, i)$; and
- How much cost the defense strategy incurs by taking given defense strategy i upon AS_j, denoted by $DCU(AS_j, i)$ and it is called *defense cost*.

Similar to the formulation of the attacker's utility function, the **defender's utility**, u_{ij}^D, is formulated by:

$$u_{ij}^D = w_1 \frac{DEU(AS_j, i)}{u_{max}} + w_2 \frac{u_{max} - DSU(AS_j, i)}{u_{max}} + w_3 \frac{u_{max} - DCU(AS_j, i)}{u_{max}} \tag{4.16}$$

where $w_1 + w_2 + w_3 = 1$ and $0 \leq w_i \leq 1$. Each weight, w_i, is determined based on system objectives and each utility factor is ranged in $[u_{min}, u_{max}] = [0, 3]$ as an integer where a higher score represents higher degree (Table 4.3).

4.4.3 Estimation of HEUs

In this section, given the scenario described above, we discuss how the HEU of an attacker or a defender can be estimated by following the key concepts of hypergame theory. First of all, we need to figure out how to compute the degree of perceived uncertainty by each player, denoted by g, in Eq. 4.13. In the given example, we simply estimate the degree of an attacker's perceived uncertainty, denoted by g_A, in terms of whether an attacker is able to correctly detect a system state or effectively perform a chosen attack or not. Suppose that given the attacker's capability, denoted

Table 4.3 Full game in the attack-defense deception game

	C_0	c_{01}	c_{02}	c_{03}	c_{04}
R_0		AS_1	AS_2	AS_3	AS_4
		Reconnaissance attack	Delivery attack	Stealthiness attack	Exploitation attack
r_{01}	DS_1				
	Disseminate a fake patch	(u_{11}^A, u_{11}^D)	(u_{12}^A, u_{12}^D)	(u_{13}^A, u_{13}^D)	(u_{14}^A, u_{14}^D)
r_{02}	DS_2				
	Disseminate a real patch	(u_{21}^A, u_{21}^D)	(u_{22}^A, u_{22}^D)	(u_{23}^A, u_{23}^D)	(u_{24}^A, u_{24}^D)
r_{03}	DS_3				
	Perform an MTD	(u_{31}^A, u_{31}^D)	(u_{32}^A, u_{32}^D)	(u_{33}^A, u_{33}^D)	(u_{34}^A, u_{34}^D)
r_{04}	DS_4				
	No defense operations	(u_{41}^A, u_{41}^D)	(u_{42}^A, u_{42}^D)	(u_{43}^A, u_{43}^D)	(u_{44}^A, u_{44}^D)

by P_a, g_A is simply estimated by $1 - P_a$ where P_a can be computed based on the level of its intelligence that can be useful to perform a planned attack. As an example, we measure it as a function of the effective monitoring time and estimate g_A by:

$$g_A = 1 - P_a \quad \text{where} \quad P_a = e^{-1/T_{ef}^A} \tag{4.17}$$

This equation implies that as the effective monitoring time, T_{ef}^A, increases (e.g., more monitoring without experiencing any changes of system configuration by a defender's MTD triggered or without being caught), the attacker can make decisions to choose a best strategy with high confidence. Then, the attacker's high confidence minimizes its perceived uncertainty. We can similarly model the degree of a defender's perceived uncertainty, denoted by g_D, and measure it by:

$$g_D = 1 - P_d \quad \text{where} \quad P_d = e^{-1/T_{ef}^D} \tag{4.18}$$

Here, T_{ef}^D can be captured based on the level of intelligence the defender has gained towards attack behaviors. For example, T_{ef}^D can be obtained based on how long the defender has observed attack behaviors or how many anomalous behaviors the system has detected as intrusions. In Sect. 4.5, we show how to estimate T_{ef}^A and T_{ef}^D by evaluating the developed SPN model.

To obtain HEU of each player (see Eq. 4.13), each player needs to estimate two EUs over C_Σ and CMS_w for a given strategy of a row player (either an attacker or a defender). To calculate the HEUs, we use the pre-defined utility values (i.e., u_{ij}^A's and u_{ij}^D's) and associated beliefs of the row player towards what strategy the opponent player will choose to play (i.e., c_{kj}'s in Eq. 4.11). To distinguish an attacker's c_{kj}'s (i.e., the attacker's beliefs about which j defense strategy is played in the attacker's subgame k) from a defender's c_{kj}'s (i.e., the defender's beliefs about which j attack strategy is played in the defender's subgame k), we denote them by c_{kj}^A's and c_{kj}^D's, respectively. The default values used for the utilities of the attacker and defender are shown in Table 4.6. The default values for c_{kj}^A's and c_{kj}^D are also shown in Table 4.7. Again, in the ideal case, we may want to derive these values dynamically at runtime depending on changes made at a given situation.

An attacker's HEU (AHEU) is estimated based on Eq. 4.13 and is given by:

$$AHEU(AS_i, g_A) = (1 - g_A) \cdot EU(AS_i, C_\Sigma^A) + g_A \cdot EU(AS_i, CMS_w^A) \tag{4.19}$$

where AS_i refers to attack strategy i and g_A is the degree of perceived uncertainty by the attacker (see Eq. 4.17). $EU(AS_i, C_\Sigma^A)$ and $EU(AS_i, CMS_w^A)$ are obtained by:

$$EU(AS_i, C_\Sigma^A) = \sum_j^m S_j^A \cdot u_{ij}^A, \quad EU(AS_i, CMS_w^A) = \sum_j^m c_{wj}^A \cdot u_{ij}^A. \tag{4.20}$$

Note that the attacker's belief towards DS_j is $S_j^A = \sum_{k=0}^K P_k^A c_{kj}^A$ where P_k^A refers to the probability that an attacker plays subgame k and c_{kj}^A is the attacker's belief

that j strategy will be played by a defender under subgame k. CMS_w^A is a set of an attacker's belief that minimizes its utility based on its chosen strategy and also corresponding strategies to be played by a defender.

Similarly, a defender's HEU (DHEU) is obtained based on Eq. 4.13 and is estimated by:

$$DHEU(AS_i, g_A) = (1 - g_D) \cdot EU(DS_i, C_\Sigma^D) + g_D \cdot EU(DS_i, CMS_w^D) \quad (4.21)$$

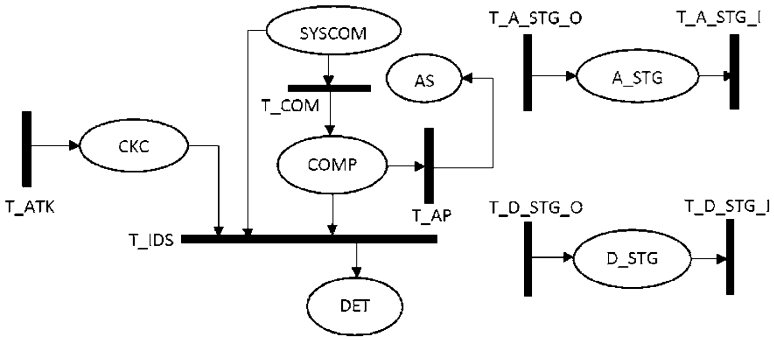

Fig. 4.1 SPN model

where DS_i refers to attack strategy i and g_D is the degree of perceived uncertainty by the attacker (see Eq. 4.18). $EU(DS_i, C_\Sigma^D)$ and $EU(DS_i, CMS_w^D)$ are obtained by:

$$EU(DS_i, C_\Sigma^D) = \sum_j^m S_j^D \cdot u_{ij}^D, \quad EU(DS_i, CMS_w^D) = \sum_j^m c_{wj}^D \cdot u_{ij}^D. \quad (4.22)$$

Note that the defender's belief towards AS_j is $S_j^D = \sum_{k=0}^K P_k^D c_{kj}^D$ where P_k^D refers to the probability that a defender plays subgame k and c_{kj}^D is the defender's belief that j strategy will be played by an attacker under subgame k. CMS_w^D is a set of a defender's belief that minimizes its utility based on its chosen strategy and also corresponding strategies to be played by an attacker.

4.5 Example Analytical Model Using Stochastic Petri Nets

In this section, we show an example deception game between an attacker and a defender based on the key concepts of hypergame theory as a case study. As a modeling tool, we use an analytical model, called *Stochastic Petri Nets* (SPNs),

whose underlying model is Markov or Semi-Markov where an event's arrival time follows the Poisson process and the inter-arrival time between events is exponentially distributed. We show the SPN model we developed for the example scenario of the deception game described in Sect. 4.4. In an SPN model, an oval is called a "place," which refers to a system state when a particular event is triggered. When the place has a token, this means that the system is at a particular state. A black bar is called "Transition" which indicates a "transition rate" the system changes from one state to another. This models the time taken for a particular event to occur. The notation $mark(place\_name)$ means the number of tokens in a particular place with the given place name. Now, we describe each SPN subnet to describe each event and the impact on a system state when the event occurs. Note that $mark(place\_name)$ refers to the number of tokens in the given place. The system described in Fig. 4.1 consists of N components which are initialized as the number of tokens in place $SYSCOM$. Now, we will discuss how each subset (or event) occurs in terms of corresponding transition rates and tokens taken from or to places to describe the proposed game between an attacker and a defender.

Cyber Kill Chain An attacker can be one of the first three stages (i.e., reconnaissance, delivery, or exploitation) in the CKC in order to perform an attack. The number of tokens in place CKC (i.e., $mark(CKC)$) refers to which stage the attacker is in where $mark(CKC) = 1, 2$, or 3 indicates reconnaissance, delivery, or exploitation stage, respectively. Since the attacker is regarded as an inside attacker when it is in the exploitation stage (i.e., $mark(CKC) == 3$), the inside attacker can perform an attack and its attack behaviors are described below in the Attack Success event.

Attack Strategy An attacker's four strategies are modeled by the number of tokens in place A_STG where the number of tokens refers to each attack strategy i (i.e., i for AS_i). The attacker chooses a best strategy based on its estimated maximum AHEU with the rate $T\_A\_STG\_O$ as long as it is not caught by an IDS having $0 < mark(CKC) < 4$ where $mark(CKC)$ indicates which stage the attacker is in the CKC. Since the attacker may take a different action depending on the stage it is in the CKC (i.e., $mark(CKC)$), in order for a new action to be chosen, the current action is flushed out by the rate $T\_A\_STG\_I$. Note that the attacker's capability is closely related to the measure of perceived uncertainty, g_A, that affects AHEU where P_a is captured by e^{-1/T_{ef}^A} (see Eq. 4.17) where

$$T_{ef}^A = mark(CKC) + 2 \text{ if } mark(A\_STG) == 4 \wedge mark(D\_STG) == 2; \tag{4.23}$$

$$T_{ef}^A = 1 \text{ if } mark(CKC) > 0 \wedge mark(D\_STG) == 3;$$

$$T_{ef}^A = mark(CKC) + 1 \text{ otherwise.}$$

Here, the number of tokens in place D_STG refers to the defense strategy ID i for DS_i where $mark(D\_STG) == 2$ means that the defender takes DS_2 (i.e., disseminating a real patch) which allows the attacker to have better intelligence or leverage the real patch to perform a zero-day attack. $mark(D\_STG) == 3$ means that the defender takes DS_3 by executing an MTD, which lowers down the attacker's capability because its information obtained from the previous reconnaissance activities is not useful anymore. Note that higher T_{ef}^A indicates a higher capability, and vice versa.

Defense Strategy Similar to the attack strategies, a defender's four strategies are considered in place D_STG. When the defender chooses a best strategy based on its estimated maximum DHEU, the number of tokens corresponding to a chosen strategy's ID (i.e., i for DS_i) goes to D_STG with the rate $T\_D\_STG\_O$. The defender chooses a new strategy with the rate $T\_D\_STG\_I$, taking all tokens in D_STG out from place D_STG. The defender's DHEU is highly affected by its perceived uncertainty, $g_D = 1 - P_d$ where P_d is computed by e^{-1/T_{ef}^D} (see Eq. 4.18) where

$$T_{ef}^D = mark(CKC) + mark(DET). \tag{4.24}$$

Similar to T_{ef}^A, higher T_{ef}^D means a better capability for the defender.

Intrusion Detection A given system triggers an IDS periodically with the rate T_IDS, which contributes the only cost even if the defender chooses the "no defense" strategy (i.e., DS_4). The place DET refers to the number of accumulated compromised components detected by the IDS. The performance of the IDS is affected by the false positives incurred by the IDS, P_{fn}. The IDS can detect either an inside attacker which is in the stage of exploitation (i.e., $mark(CKC) == 3$) or a compromised system component. But, the place DET captures only the number of compromised components, not the number of tokens in CKC.

Attack Success The considered main two attacks include: (1) the attacker can compromise a vulnerable system component (or a node in a network) to breach system integrity and availability; and (2) the attacker can leak out confidential information to unauthorized parties to breach data integrity and confidentiality. The system fails when: (1) more than one third of components are compromised or detected (i.e., $1/3 > (mark(DET) + mark(COMP))/N$, following the concept of Byzantine Failure [14], where N is the number of initial tokens N in $SYSCOM$; or (2) confidential information is leaked out by an attacker or compromised nodes (i.e., $mark(AS) > 0$) with the rate T_AP.

The key design parameters, their meanings, and their default values used in this experiment are summarized in Table 4.4. All the detailed descriptions of transition rates, their meanings, and their enabling functions (i.e., to set when the transition is enabled or disabled) in our SPN model are summarized in Table 4.5.

Table 4.4 Key design parameters, their meanings, and their default values

Param.	Meaning	Value
N	Number of components of a system	50
P_v	Probability of a given node being vulnerable	0.5
P_{fn}	Probability of incurring false negatives	0.05
P_{fp}	Probability of incurring false positives	0.05
T_{as}	Inter-arrival time of an attacker taking an action	10 s
T_{ds}	Inter-arrival time of a defender taking an action	10 s
T_{nas}	Inter-arrival time of an attacker moving to a next action	1 h
T_{nds}	Inter-arrival time of a defender moving to a next action	1 h
T_a	Inter-arrival time of an attacker being at a certain stage of the CKC	1 h
T_s	Inter-arrival time of an attacker attempting to leak out a confidential information	1 h
T_c	Inter-arrival time of an attacker attempting to compromise a vulnerable system component	1 h
T_d	Inter-arrival time of an IDS being triggered	1 h
w_1, w_2, w_2	Weights to consider three aspects of utilities (i.e., effect, vulnerability, and cost)	0.4, 0.3, 0.3

Table 4.5 Transition names, meanings, rates, and enabling conditions

Name	Meaning	Rate	Enabling conditions
T_ATK	Timed transition of an attacker being at a certain stage of the CKC	$1/T_a$	Enable when $isSF() == 0 \wedge mark(CKC) < 3$; disable otherwise
T_COM	Timed transition of an attacker compromising a vulnerable system component	$(P_a \cdot P_v)/T_c$	Enable $isSF() == 1 \wedge mark(CKC) == 3$; disable otherwise
T_AP	Timed transition of an attacker breaching data integrity	$(P_a \cdot P_v)/T_s$	Enable when $isSF() == 0$; disable otherwise
T_IDS	Timed transition of an IDS being triggered with a given interval	$(1 - P_{fn})/T_d$	Enable $isSF() == 0 \wedge (mark(CKC) == 3 \vee mark(COMP) > 0)$; disable otherwise
$T\_A\_STG\_O$	Timed transition of an attacker taking a chosen best action	$1/T_{as}$	Disable when $isSF() > 0 \vee mark(A_STG) > 0 \vee mark(CKC) == 0$; enable otherwise
$T\_A\_STG\_I$	Timed transition of an attacker preparing to take an action	$1/T_{nas}$	Disable $isSF() > 0 \vee mark(A\_STG) == 0$; enable otherwise
$T\_D\_STG\_O$	Timed transition of a defender taking a chosen best action	$1/T_{ds}$	Disable $isSF() > 0 \vee mark(D\_STG) > 0$; enable otherwise
$T\_D\_STG\_I$	Timed transition of a defender preparing to take an action	$1/T_{nds}$	Disable when $isSF() > 0 \vee mark(D\_STG) == 0$; enable otherwise

Note: $isSF()$ returns 1 when $(mark(COMP) + mark(DET)/N > 1/3) \vee (mark(AS) > 0) \vee (N_{total} > N)$ where $N_{total} = mark(SYSCOM) + mark(DET) + mark(COMP) + mark(AS)$; 0 otherwise. P_a and P_d are obtained from Eqs. 4.17 and 4.18, respectively

Table 4.6 Attack and defense utility values used for the experiments

u_{ij}^A	DS_1	DS_2	DS_3	DS_4	u_{ij}^D	AS_1	AS_2	AS_3	AS_4
AS_1	(2, 2, 2)	(3, 2, 2)	(1, 2, 3)	(2, 2, 2)	DS_1	(3, 1, 2)	(3, 1, 2)	(3, 0, 2)	(3, 1, 2)
AS_2	(2, 2, 2)	(3, 2, 2)	(2, 2, 3)	(2, 2, 2)	DS_2	(2, 1, 2)	(2, 1, 2)	(2, 0, 2)	(2, 3, 2)
AS_3	(1, 3, 1)	(1, 3, 1)	(1, 3, 1)	(1, 3, 1)	DS_3	(3, 0, 3)	(3, 0, 3)	(3, 0, 3)	(3, 1, 3)
AS_4	(3, 3, 2)	(3, 3, 2)	(3, 3, 3)	(3, 3, 2)	DS_4	(0, 2, 1)	(0, 2, 1)	(0, 0, 1)	(0, 2, 1)

Note: u_{ij}^A or u_{ij}^D is estimated $(AEU(DS_i, j), ASU(DS_i, j), ACU(DS_i, j))$ or $(DEU(DS_i, j), DSU(DS_i, j), DCU(DS_i, j))$ as shown in Eqs. 4.15 and 4.16, respectively

Table 4.7 Attack and defense utility values used for the experiments

Subgame k for an attacker	c_{k1}^A	c_{k2}^A	c_{k3}^A	c_{k4}^A	Subgame k for a defender	c_{k1}^D	c_{k2}^D	c_{k3}^D	c_{k4}^D
1	0.25	0.25	0.25	0.25	1	0.6	0.2	0.1	0.1
2	0.1	0.3	0.4	0.2	2	0.3	0.3	0.1	0.3
3	0.4	0.2	0.4	0	3	0.1	0.2	0.1	0.6

Note: c_{kj}^A's or c_{kj}^D's are the beliefs of each party (either an attacker or a defender) about which strategy will be played by the opposite player in given subgame k, respectively

The utilities an attacker or a defender perceives at a given situation substantially affect each player's decision-making process. Also, how the attacker or the defender perceives a given situation, such as what strategy an opponent player will choose (i.e., c_{kj}^A's or c_{kj}^D), is critical to estimating their corresponding HEU (i.e., AHEU or DHEU). However, to show simple example trends observed based on our proposed SPN model, we assume that those utilities (i.e., u_{ij}^A's, u_{ij}^D's) and the corresponding beliefs (i.e., c_{kj}^A's, c_{kj}^D's) are given as static, as summarized in Tables 4.4 and 4.7, respectively. How to obtain the utilities and beliefs dynamically using other learning techniques (e.g., machine learning or belief models) should be discussed in a separate work.

We consider two metrics to capture the performance by a defender and an attacker. The performance of the defender is obtained by MTTSF, measuring the *mean time to security failure*, which is defined based on $isSF()$. MTTSF is measured by:

$$MTTSF = \sum_{i \in S} r_i \int_{t=0}^{\infty} P_i(t)dt \tag{4.25}$$

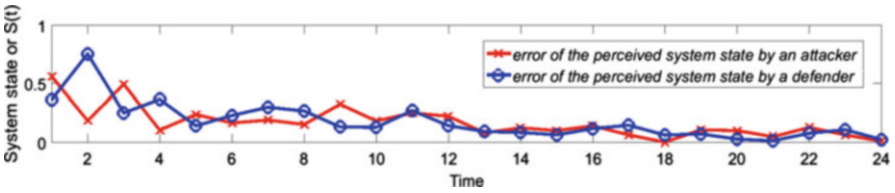

Fig. 4.2 Errors in system security states as perceived by the attacker and the defender

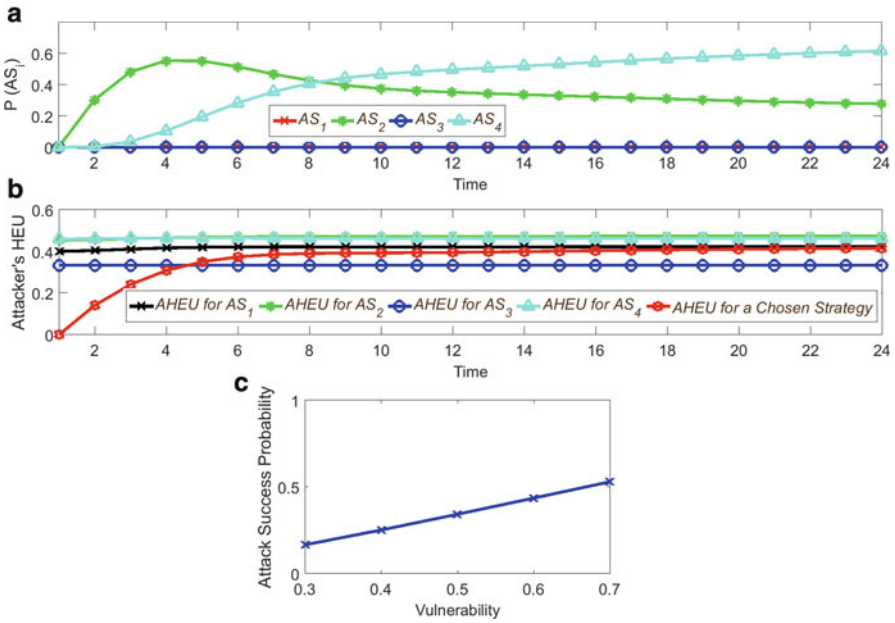

Fig. 4.3 (**a**) Attacker's strategies, (**b**) AHEUs, and (**c**) attack success probability (P_{AS})

where S denotes the set of all states, r_i (reward) is 1 for the absorbing states having $isSF() == 0$ and 0 for $isSF() > 0$, and $P_i(t)$ is the probability of state i at time t.

The performance of the attacker is measured based on the attack success probability (ASP) which is obtained in the developed SPN model by:

$$P_{AS} = \frac{\sum_{i \in S} SF_i \int_{t=0}^{\infty} P_i(t)dt}{MTTSF} \tag{4.26}$$

where S denotes the set of all states i's, SF_i (reward) is set to 1 for $isSF() > 0$ (see Table 4.5), and $P_i(t)$ is the probability of state i at time t.

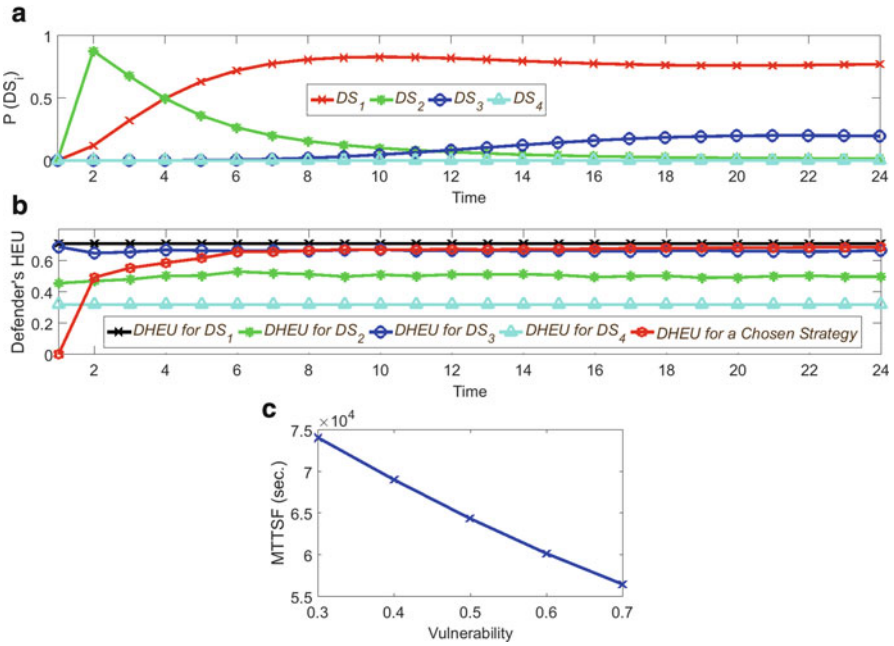

Fig. 4.4 (**a**) Defender's strategies, (**b**) DHEUs, and (**c**) MTTSF

4.6 Experiments and Discussion

In this section, we present our experimental findings based on the evaluation of the developed SPN model and discuss overall trends observed and key insights obtained from this case study.

System Security States Perceived by an Attacker and a Defender and Their Ground Truth Figure 4.2 shows the errors that are introduced by estimating the difference between the ground truth system view and the view by either an attacker or a defender over time, respectively. The system security state (SS) is captured based on the number of healthy components over the total number of the components initially given (i.e., $mark(SYSCOM)/N$). How to perceive the SS by the attacker or the defender is affected by its respective capability to detect the ground truth information. As discussed in Eqs. 4.17 and 4.18, their abilities to detect the ground truth SS are affected by how long they have made effective monitoring to detect the behaviors of the opponent player, which were considered in deriving their capabilities, P_a and P_d, respectively. As shown in Fig. 4.2, both the attacker and the defender are evolving their abilities to accurately detect the SS (i.e., showing lower differences).

Attacker's Strategies, AHEUs, and Attack Success Figure 4.3 shows the attacker's chosen strategy and the corresponding AHEU. Figure 4.3a shows the

probabilities that each attack strategy is chosen by the attacker over time. For the attacker, AS_1 and AS_3 are not attractive because they do not yield better utilities than AS_2 and AS_4. In particular, when the attacker is in the reconnaissance stage, it always prefers trying delivering an attack to penetrate into the system, instead of taking more time in scanning the system surface because it won't be able to use all the collected information if the defender chooses an MTD (i.e., DS_3). In addition, when the attacker is in the stage of exploitation, instead of performing passive attack by being stealthy without performing any attacks, it always tries to access any patch disseminated and tries to penetrate into any vulnerable system component. Over time, since the attacker is more likely to be in the exploitation stage, AS_4 is even more preferred than AS_3. Figure 4.3b shows AHEU for each attack strategy and the AHEU obtained by the chosen strategy. It is obvious that the chosen strategy is not the best strategy based on the ground truth AHEUs by other strategies. This is because the attacker is limited in its perception due to its inability to perceive all the ground truth. Hence, its decision is not optimal, which is natural because the attacker is distracted by the defender's deception or MTD strategies (i.e., DS_1 or DS_3), aiming to increase confusion or uncertainty for the attacker. Lastly, in Fig. 4.3c, we simply varied the probability that a system component (or a node) is vulnerable, denoted by P_v, from 0.3 to 0.7 with 0.1 increment. It is natural to observe higher P_{AS} as P_v increases.

Defender's Strategies, DHEUs, and MTTSF Figure 4.4 shows the defender's best strategy chosen and the corresponding DHEU. Figure 4.4a shows the probabilities that the defender chooses each strategy over time. Overall, DS_1 (i.e., disseminating a fake path as a deception technique) is the most favorable because it has high effect with less cost. The second most favorable strategy is DS_3 (i.e., triggering an MTD) because it is highly effective although it incurs high cost. DS_2 (i.e., disseminating a real patch) is favored in the beginning while it is not chosen anymore over time because the defender chooses DS_3 more due to its higher utility when the system becomes more vulnerable. Obviously, DS_4 is not attractive at all because it won't protect the system sufficiently although it incurs minimum cost (i.e., cost to trigger an IDS). This trend is well-explained in Fig. 4.4b, as DS_1 shows the highest DHEU. The chosen strategy generates the second highest HEU compared to the highest (i.e., DS_1) because the defender's perception is not perfect as well as the defender restricts its available strategies based on its own prior belief under a given subgame (i.e., c^D_{kj} in Table 4.7). Similar to P_{AS}, when P_v varies from 0.3 to 0.7 with 0.1 increment, we observe that MTTSF decreases as P_v increases.

4.7 Conclusion

In this chapter, we studied the key concepts of hypergame theory and how those can be used to model an attack-defense game which considers subjective beliefs and perceptions (or misperception) of each player in a given dynamic situation

where a defender uses deception techniques as one of the strategies to increase uncertainty for an attacker. We discussed how each player's decision utility can be estimated based on hypergame theory and how its uncertainty can significantly affect its expected utility. In addition, we showed an example modeling study based on Stochastic Petri Nets, which enables us to model the proposed deception game and to capture necessary measures and metrics in order to observe the behaviors of the attacker and the defender. We hope this work can provide guidance in modeling and analyzing realistic attack-defense games where various types of uncertainty exist in real-world situations. Utilities by each player are critical to making best decisions. How to determine or capture dynamically changing utilities and each player's beliefs towards an opponent's strategy is vital and should be investigated further in order to advance the techniques of game-theoretic modeling and analysis.

4.8 Exercise Problems

1. Discuss how hypergame theory, conventional game theory, and decision theory differ from each other. In particular, discuss the differences in terms of the perceived uncertainty by each player (refer to Eqs. 4.17 and 4.18).
2. Answer the following questions based on our example attack-defense game scenario in Sect. 4.4. The attacker is in the exploitation stage (i.e., $mark(CKC) = 3$). The system security state perceived by the defender is 0.95. The IDS has detected 5 system components as compromised (i.e., $mark(DET) = 5$). The questions are: (1) What are the uncertainty levels perceived by the attacker and the defender (i.e., g_A, g_D), respectively? (2) Which subgame will be played by the attacker or the defender, respectively? (3) What is the respective hypergame expected utility of the attacker and the defender (i.e., AHEU and DHEU)?
3. In the example attack-defense game scenario in Sect. 4.4, the defender's fourth strategy, DS_4, means that the defender does not perform any defense operation except running the IDS. We simply considered no utility gained from DS_4 in terms of its effect. This is shown in the fourth row in u_{ij}^D of Table 4.6 where the first elements in each vector with three utilities are set to zeros (i.e., $(\mathbf{0}, 2, 1)$, $(\mathbf{0}, 2, 1)$, $(\mathbf{0}, 0, 1)$, $(\mathbf{0}, 2, 1)$). But considering the effect of the IDS running periodically, if we change it to 1 (i.e., $(\mathbf{1}, 2, 1)$, $(\mathbf{1}, 2, 1)$, $(\mathbf{1}, 0, 1)$, $(\mathbf{1}, 2, 1)$), how does the defender's choice of its best strategy change when the defender perceives that the system state is 92% secure?
4. In the example attack-defense game scenario in Sect. 4.4, the attacker's view is fairly conservative about the defender's strategy in that there will be a higher level of defense if it performs active attacks. This trend is reflected in the attacker's beliefs towards which defense strategy j will be played under subgame k (i.e., c_{kj}^A in Table 4.7). Now, let's change the attacker's beliefs in a more optimistic way. The attacker believes that the system is vulnerable, but the defender may not necessarily run any additional defense mechanisms except the IDS. To reflect the

changes in the attacker's belief, let's change c_{kj}^{A} for the attacker's subgame 3 (i.e., the last row of c_{kj}^{A} in Table 4.7) from (0.4, 0.2, 0.4, 0) to (0, 0.4, 0.2, 0.4). How does this change in the subgame affect the attacker's HEU and its best strategy to choose?

References

1. Abass AAA, Xiao L, Mandayam NB, Gajic Z (2017) Evolutionary game theoretic analysis of advanced persistent threats against cloud storage. IEEE Access 5:8482–8491
2. Aljefri YM, Bashar MA, Fang L, Hipel KW (2017) First-level hypergame for investigating misperception in conflicts. IEEE Transactions on Systems, Man, and Cybernetics: Systems PP(99):1–18
3. Almeshekah MH, Spafford EH (2016) Cyber security deception. In: Cyber Deception - Building the Scientific Foundation, Springer, pp 25–52
4. Bell JB, Whaley B (1991) Cheating and Deception. Transaction Publishers
5. Bennett P (1977) Toward a theory of hypergames. Omega 5(6):749–751
6. Caddell JW (2004) Deception 101-primer on deception. Tech. rep., DTIC Document
7. Carroll TE, Grosu D (2011) A game theoretic investigation of deception in network security. Security and Communication Networks 4(10):1162–1172
8. Chen P, Desmet L, Huygens C (2014) A study on Advanced Persistent Threats. In: Proceedings of the IFIP International Conference on Communications and Multimedia Security, Springer, pp 63–72
9. Cho JH, Wang Y, Chen IR, Chan KS, Swami A (2017) A survey on modeling and optimizing multi-objective systems. IEEE Communications Surveys Tutorials 19(3):1867–1901
10. Daniel DC, Herbig KL (1982) Strategic Military Deception. Pergamon
11. Fang X, Zhai L, Jia Z, Bai W (2014) A game model for predicting the attack path of apt. In: 2014 IEEE 12th International Conference on Dependable, Autonomic and Secure Computing (DASC), IEEE, pp 491–495
12. Feng X, Zheng Z, Hu P, Cansever D, Mohapatra P (2015) Stealthy attacks meets insider threats: a three-player game model. In: IEEE Military Communications Conference (MILCOM 2015), IEEE, pp 25–30
13. Fraser NM, Hipel KW (1984) Conflict Analysis: Models and Resolutions, North-Holland Series in System Science and Engineering, vol 2. A. P. Sage, North-Holland
14. Gaertner FC (2003) Byzantine failures and security: Arbitrary is not (always) random. Tech. rep., EPFL
15. Garg N, Grosu D (2007) Deception in honeynets: A game-theoretic analysis. In: IEEE SMC Information Assurance and Security Workshop (IAW'07), IEEE, pp 107–113
16. Gharesifard B, Cortés J (2010) Evolution of the perception about the opponent in hypergames. In: Proceedings of the 49th IEEE Conference on Decision and Control (CDC), pp 1076–1081
17. Gharesifard B, Cortés J (2012) Evolution of players' misperceptions in hypergames under perfect observations. IEEE Transactions on Automatic Control 57(7):1627–1640
18. House JT, Cybenko G (2010) Hypergame theory applied to cyber attack and defense. In: Proceedings of the SPIE Conference on Sensors, and Command, Control, Communications, and Intelligence (C3I) Technologies for Homeland Security and Homeland Defense IX, vol 766604
19. Hu P, Li H, Fu H, Cansever D, Mohapatra P (2015) Dynamic defense strategy against advanced persistent threat with insiders. In: 2015 IEEE Conference on Computer Communications (INFOCOM), IEEE, pp 747–755
20. Kanazawa T, Ushio T, Yamasaki T (2007) Replicator dynamics of evolutionary hypergames. IEEE Transactions on Systems, Man, and Cybernetics - Part A: Systems and Humans 37(1):132–138

21. Kovach NS, Gibson AS, Lamont GB (2015) Hypergame theory: A model for conflict, misperception, and deception. Game Theory 2015:Article ID 570639, 20 pages
22. Okhravi H, Rabe MA, Leonard WG, Hobson TR, Bigelow D, Streilein WW (2013) Survey of cyber moving targets. Tech. Rep. 1166, Lexington Lincoln Lab, Massachusetts Institute of Technology
23. Putro US, Kijima K, Takahashi S (2000) Adaptive learning of hypergame situations using a genetic algorithm. IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans 30(5):562–572
24. Rass S, König S, Schauer S (2017) Defending against advanced persistent threats using game-theory. PloS one 12(1):e0168675
25. Sasaki Y (2014) Subjective rationalizability in hypergames. Advances in Decision Sciences 2014(Article ID 263615):7 pages
26. Sharp WL (2006) Military deception. Tech. rep., joint Publication 3-13.4
27. Tadelis S (2013) Game Theory: An Introduction. Princeton University Press
28. Van Dijk M, Juels A, Oprea A, Rivest RL (2013) Flipit: The game of "stealthy takeover". Journal of Cryptology 26(4):655–713
29. Vane R (2006) Advances in hypergame theory. In: Proceedings of the Workshop on Game-Theoretic and Decision Theoretic Agents, Hakodate, Japan
30. Vane RR (2000) Hypergame Theory for DTGT Agents. American Association for Artificial Intelligence
31. Vane RR (2005) Planning for terrorist-caused emergencies. In: Proceedings of the Winter Simulation Conference
32. Vane RR, Lehner PE (1999) Using hypergames to select plans in adversarial environments. In: Parsons S, Wooldridge MJ (eds) Proceedings of the 1st Workshop on Game Theoretic and Decision Theoretic Agents, pp 103–111
33. Yin Y, An B, Vorobeychik Y, Zhuang J (2013) Optimal deceptive strategies in security games: A preliminary study. In: Proc. of AAAI
34. Zhang M, Zheng Z, Shroff NB (2015) A game theoretic model for defending against stealthy attacks with limited resources. In: International Conference on Decision and Game Theory for Security, Springer, pp 93–112

Chapter 5
Dynamic Bayesian Games for Adversarial and Defensive Cyber Deception

Linan Huang and Quanyan Zhu

Abstract Security challenges accompany the efficiency. The pervasive integration of information and communications technologies (ICTs) makes cyber-physical systems vulnerable to targeted attacks that are deceptive, persistent, adaptive, and strategic. Attack instances such as Stuxnet, Dyn, and WannaCry ransomware have shown the insufficiency of off-the-shelf defensive methods including the firewall and intrusion detection systems. Hence, it is essential to design up-to-date security mechanisms that can mitigate the risks despite the successful infiltration and the strategic response of sophisticated attackers.

In this chapter, we use game theory to model competitive interactions between defenders and attackers. First, we use the static Bayesian game to capture the stealthy and deceptive characteristics of the attacker. A random variable called the *type* characterizes users' essences and objectives, e.g., a legitimate user or an attacker. The realization of the user's type is private information due to the cyber deception. Then, we extend the one-shot simultaneous interaction into the one-shot interaction with asymmetric information structure, i.e., the signaling game. Finally, we investigate the multi-stage transition under a case study of Advanced Persistent Threats (APTs) and Tennessee Eastman (TE) process. Two-sided incomplete information is introduced because the defender can adopt defensive deception techniques such as honeyfiles and honeypots to create sufficient amount of uncertainties for the attacker. Throughout this chapter, the analysis of the Nash equilibrium (NE), Bayesian Nash equilibrium (BNE), and perfect Bayesian Nash equilibrium (PBNE) enables the policy prediction of the adversary and the design of proactive and strategic defenses to deter attackers and mitigate losses.

Keywords Bayesian games · Multi-stage transitions · Advanced Persistent Threats (APTs) · Cyber deception · Proactive and strategic defense

---

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

L. Huang · Q. Zhu (✉)
Department of Electrical and Computer Engineering, New York University, Brooklyn, NY, USA
e-mail: qz494@nyu.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_5

5.1 Introduction

The operation of the modern society intensively relies on the Internet services and information and communications technologies (ICTs). Cybersecurity has been an increasing concern as a result of the pervasive integration of ICTs as witnessed in Fig. 5.1. Every peak of the yellow line corresponds to a cyberattack[1] and both the frequency and the magnitude which represents the scope of influence have increased, especially in recent years. For example, the Domain Name System (DNS) provider Dyn has become the targeted victim of the multiple distributed denial-of-service (DDoS) attacks in October 2016. The Mirai malware has turned a large number of IoT devices such as printers and IP cameras to bots and causes an estimate of 1.2 Tbps network flow. More recently in May 2017, the WannaCry ransomware has attacked more than 200,000 computers across 150 countries, with total damages up to billions of dollars.

One way to contend with the cyberattacks is for the defenders to set up firewalls with pre-defined rules to prevent their internal network from the untrustworthy network traffic. Moreover, defenders can use intrusion detection systems [3] to detect a suspected malicious activity when an intrusion penetrates the system. These defensive methods are useful in deterring naive attacks up to certain degree. However, the unequal status between the attacker and the defender naturally gives the attacker an advantage in the game. An attacker succeeds by knowing and exploiting one zero-day vulnerability, while a defender can be successful only when he can defend against all attacks. Moreover, attacks evolve to be increasingly sophisticated and can easily challenge the traditional defense mechanisms, i.e., intrusion prevention, detection, and response.

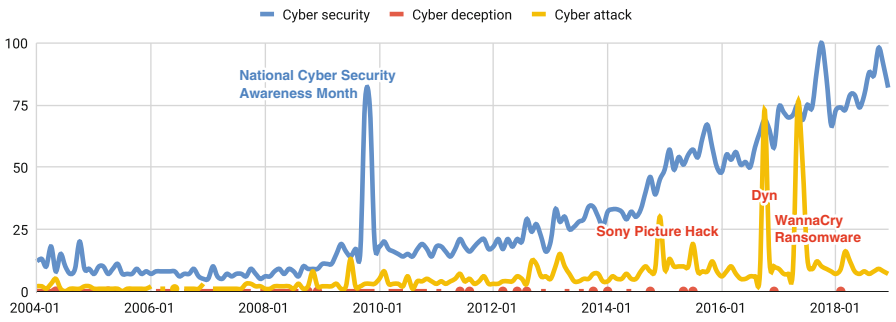

Fig. 5.1 The search results of three keywords, i.e., the cybersecurity (in blue), the cyber deception (in red), and the cyberattack (in yellow) in the USA from Jan. 2004 to Aug. 2018 via the Google Trends. Compared with the blue and yellow line, the cyber deception which endows attackers an information advantage over the defender requires more investigations. Numbers on the y-axis represent the search frequency normalized with respect to the highest point on the chart for the given region and time. A value of 100 is the peak popularity

[1] https://en.wikipedia.org/wiki/List_of_cyberattacks.

Cyber deception is one way to evade the detection [14, 21, 32]. As defined in [16], the deception is either the prevention from a true belief or a formulation of a false belief. In the cybersecurity setting, the first type of deception corresponds to a stealthy attack where the virus can behave to be legitimate apparently and remain undetected. For example, if a strategic attacker knows the pre-defined rules of the firewalls or the rule-based intrusion detection system, they can adapt their behaviors to avoid triggering the alarm. In the second type, for example, hackers can launch "sacrificial attacks" to trick the defender into a false belief that all viruses have been detected and repelled [6]. The adversarial cyber deception introduces the information asymmetry and poses attackers in a favorable position. A defender is analogous to a blind person who competes with a sighted attacker in a well-illuminated room.

To tilt the information asymmetry, the defender can be reactive, i.e., continuously consummates the intrusion prevention and detection system capable of stealthy and deceptive attacks. This costly method is analogous to curing the blindness. Defensive deception, however, provides an alternative to the costly rectifications of the system by deliberately and proactively introducing uncertainties into the system, i.e., private information unknown to the attacker. This proactive method is analogous to turning off the light and providing every participant, especially the attacker with sufficient amount of uncertainties. For example, a system can include honeypots that contain no information or resource of value for the attackers. However, the defender can make the honeypot indistinguishable from the real systems by faking communication and network traffic. Since a legitimate user should not access the honeypot, the activities in the honeypot reveal the existence as well as characteristics of that attack.

The cyberattacks and defenses are the spear and shield, the existence of attackers motivates the development of defensive technologies, which in turn stimulates advanced attacks that are strategic, deceptive, and persistent. In this chapter, we model these competitive interactions using game theory ranging from complete to incomplete information, static to multi-stage transition, and symmetric to asymmetric information structures.

5.1.1 Literature

Deception and its modeling are emerging areas of research. The survey [20] provides a taxonomy that defines six types of defensive deception: perturbation via external noises, moving target defense (MTD), obfuscation via revealing useless information, mixing via exchange systems, honey-x, and the attacker engagement that uses feedback to influence attackers dynamically. MTD [13] can limit the effectiveness of the attacker's reconnaissance by manipulating the attack surface of the network. The authors in [29] combine information—and control—theory to design an optimal MTD mechanism based on a feedback information structure while

[15, 17] use the Markov chain to model the MTD process and discuss the optimal strategy to balance the defensive benefit and the network service quality.

Game-theoretic models are natural frameworks to capture the adversarial and defensive interactions between players [7, 9, 10, 18, 19, 25, 28, 30–32]. There are two perspectives to deal with the incomplete information under the game-theoretic setting, i.e., the robust game theory [1] that conservatively considers the worst-case and the Bayesian game model [8] that introduces a random variable called the *type* and the concept of Bayesian strategies and equilibrium. Signaling game, a two-stage game with one-sided incomplete information has been widely applied to different cybersecurity scenarios. For example, [32] considers a multiple-period signaling game in the attacker–defender resource allocation. The authors in [22] combine the signaling game with an external detector to provide probabilistic warnings when the sender acts deceptively. The recent work of [12] has proposed a multi-stage Bayesian game with two-sided incomplete information that well-characterizes the composite attacks that are advanced, persistent, deceptive, and adaptive. A dynamic belief update and long-term statistical optimal defensive policies are proposed to mitigate the loss and deter the adversarial users.

5.1.2 Notation

In this chapter, the pronoun "he" refers to the user denoted by P_2, and "she" refers to the defender as P_1. Calligraphic fonts such as $\mathscr{A}$ represent a set. For $i \in \mathscr{I}$, notation "$-i$" means $\mathscr{I} \setminus \{i\}$. Take $I := \{1, 2\}$ as an example, if $i = 1$, then $-i = 2$. If $\mathscr{A}$ is a finite set, then we let $\triangle\mathscr{A}$ represent the set of probability distributions over $\mathscr{A}$, i.e., $\triangle\mathscr{A} := \{p : A \mapsto R_+ | \sum_{a\in\mathscr{A}} p(a) = 1\}$.

5.2 Static Game with Complete Information for Cybersecurity

Game theory has been applied to cybersecurity problems [4, 18, 21, 23, 24, 27, 28] to capture quantitatively the interaction between different "players" including the system operator, legitimate users, and malicious hackers. As a baseline security game, the bi-matrix game focuses on two non-cooperative players, i.e., an attacker P_2 aiming at compromising the system and a defender P_1 who tries to prevent systems from adverse consequences, mitigate the loss under attacks, and recover quickly and thoroughly to the normal operation after the virus' removal.

Each player $P_i, i \in \{1, 2\}$ can choose an action a_i from a finite set $\mathscr{A}_i$ and $m_i := |\mathscr{A}_i|$ is the number of actions P_i can choose from. The value of the utility $J_i(a_1, a_2) \in \mathscr{R}^{m_1\times m_2}$ for each player i depends collectively on both players' actions as shown in Table 5.1. As stated in the introduction, targeted attacks can investigate

the system thoroughly, exploit vulnerabilities, and obtain the information on the security settings including the value of assets and possible defensive actions. Thus, the baseline game with complete information assumes that both players are aware of the other player's existence, action sets, and payoff matrices. However, each player will not know the other player's action before making his/her decision. Example 5.1 considers a nonzero-sum complete-information security game where the attacker and the defender have conflicting objectives, i.e., $\exists a_1 \in \mathscr{A}_1, a_2 \in \mathscr{A}_2, J_1(a_1, a_2) + J_2(a_1, a_2) \neq 0$. For scenarios where the defender does not know the utility of the attacker, she can assume $J_2(a_1, a_2) = -J_1(a_1, a_2), \forall a_1 \in \mathscr{A}_1, a_2 \in \mathscr{A}_2$ and use the zero-sum game to provide a useful worst-case analysis.

Table 5.1 Utility bi-matrix (J_1, J_2) of the static secure game, i.e., $J_1 = [0, -r_1; 0, r_3]$, $J_2 = [0, r_2; 0, -r_4]$. P_1 is the row player and P_2 is the column player. Both players are rational and aim to maximize their own payoffs

$P_1 \setminus P_2$	NOP	Escalate
Permit	$(0, 0)$	$(-r_1, r_2)$
Restrict	$(0, 0)$	$(r_3, -r_4)$

Example 5.1 Consider the game in Table 5.1. Attacker P_2 can either choose action $a_2 = 1$ to escalate his privilege in accessing the system or choose No Operation Performed (NOP) $a_2 = 0$. Defender P_1 can either choose to restrict $a_1 = 1$ or allow $a_1 = 0$ a privilege escalation. The value in the brackets $(\cdot, \cdot)$ represents the utility for P_1, P_2 under the corresponding action pair, e.g., if the attacker escalates his privilege and the defender chooses to allow an escalation, then P_2 obtains a reward of $r_2 > 0$ and P_1 receives a loss of $r_1 > 0$. In this example, no dominant (pure)-strategies exist for both players to maximize their utilities, i.e., each player's optimal action choice depends on the other player's choice. For example, P_1 prefers to allow an escalation only when P_2 chooses the action NOP; otherwise P_1 prefers to restrict an escalation. The above observation motivates the introduction of the mixed-strategy in Definition 5.1 and the concept of Nash equilibrium in Definition 5.2 where any unilateral deviation from the equilibrium does not benefit the deviating player. □

Definition 5.1 A mixed-strategy $\sigma_i \in \triangle\mathscr{A}_i$ for P_i is a probability distribution on his/her action set $\mathscr{A}_i$. □

Denote $\sigma_i(a_i)$ as P_i's probability of taking action a_i, then $\sum_{a_i \in \mathscr{A}_i} \sigma_i(a_i) = 1, \forall i \in \{1, 2\}$ and $\sigma_i(a_i) \geq 0, \forall i \in \{1, 2\}, a_i \in A_i$. Once player P_i has determined strategy σ_i, the action a_i will be a realization of the strategy. Hence, each player P_i under the mixed-strategy has the objective to maximize the expected utility $\sum_{a_1 \in \mathscr{A}_1} \sum_{a_2 \in \mathscr{A}_2} \sigma_1(a_1)\sigma_2(a_2) J_1(a_1, a_2)$. Note that the concept of the mixed-strategy includes the pure-strategy as a degenerate case.

Definition 5.2 A pair of mixed-strategy (σ_1^*, σ_2^*) is said to constitute a (mixed-strategy) Nash equilibrium (NE) if for all $\sigma_1 \in \triangle\mathscr{A}_1, \sigma_2 \in \triangle\mathscr{A}_2$,

$$\sum_{a_1\in\mathscr{A}_1}\sum_{a_2\in\mathscr{A}_2}\sigma_1^*(a_1)\sigma_2^*(a_2)J_1(a_1,a_2) \geq \sum_{a_1\in\mathscr{A}_1}\sum_{a_2\in\mathscr{A}_2}\sigma_1(a_1)\sigma_2^*(a_2)J_1(a_1,a_2),$$

$$\sum_{a_1\in\mathscr{A}_1}\sum_{a_2\in\mathscr{A}_2}\sigma_1^*(a_1)\sigma_2^*(a_2)J_1(a_1,a_2) \geq \sum_{a_1\in\mathscr{A}_1}\sum_{a_2\in\mathscr{A}_2}\sigma_1^*(a_1)\sigma_2(a_2)J_1(a_1,a_2).$$

□

In a finite static game with complete information, the mixed-strategy Nash equilibrium always exists. Thus, we can compute the equilibrium which may not be unique via the following system of equations:

$$\sigma_1^* \in arg\max_{\sigma_1}\sum_{a_1\in\mathscr{A}_1}\sum_{a_2\in\mathscr{A}_2}\sigma_1(a_1)\sigma_2^*(a_2)J_1(a_1,a_2),$$

$$\sigma_2^* \in arg\max_{\sigma_2}\sum_{a_1\in\mathscr{A}_1}\sum_{a_2\in\mathscr{A}_2}\sigma_1^*(a_1)\sigma_2(a_2)J_1(a_1,a_2).$$

The static game model and equilibrium analysis are useful in the cybersecurity setting because of the following reasons. First, the strategic model quantitatively captures the competitive interaction between the hacker and the system defender. Second, the NE provides a prediction of the security outcomes of the scenario which the game model captures. Third, the probabilistic defenses suppress the probability of adversarial actions and thus mitigate the expected economic loss. Finally, the analysis of the equilibrium motivates an optimal security mechanism design which can shift the equilibrium towards ones that are favored by the defender via an elaborate design of the game structure.

5.3 Static Games with Incomplete Information for Cyber Deception

The primary restrictive assumption for the baseline security game is that all game settings including the action sets and the payoff matrices are of complete information to the players. However, the deceptive and stealthy nature of advanced attackers makes it challenging for the defender to identify the nature of the malware accurately at all time. Even the up-to-date intrusion detection system has the false alarms and misses that can be fully characterized by a receiver operating characteristic (ROC) curve plotted with the true positive rate (TPR) against the false positive rate (FPR). To capture the uncertainty caused by the cyber deception, we introduce a random variable called the *type* to model the possible scenario variations as shown in Example 5.2.

Table 5.2 Utility bi-matrix when user P_2 is either adversarial $\theta_2 = \theta^b$ or legitimate $\theta_2 = \theta^g$

$\theta_2 = \theta^b$	NOP	Escalate
Permit	$(0, 0)$	$(-r_2, r_2)$
Restrict	$(0, 0)$	$(r_0, -r_0)$

$\theta_2 = \theta^g$	NOP	Escalate
Permit	$(0, 0)$	(r_1, r_1)
Restrict	$(0, 0)$	$(-r_1, -r_1)$

Example 5.2 Consider the following static Bayesian game where we use two discrete values of the type $\theta_2 \in \Theta_2 := \{\theta^b, \theta^g\}$ to distinguish the user P_2 as either an attacker $\theta_2 = \theta^b$ or a legitimate user $\theta_2 = \theta^g$. The attacker can camouflage to be a legitimate user and possess the same action set $\mathscr{A}_2$, e.g., both attacker and legitimate can request to escalate the privilege $a_2 = 1$. However, since they are of different types, the introduced utilities $\bar{J}_i(a_1, a_2, \theta_2), i \in \{1, 2\}$ are different under the same action pair(a_1, a_2) as shown in Table 5.2. For example, the privilege escalation has a positive effect on the system when the user P_2 is legitimate, yet will harm the system when P_2 is an attacker. Since the defender does not know the type of the user due to the cyber deception, we extend the Nash equilibrium analysis of the complete-information game to Bayesian Nash equilibrium in Definition 5.3 to deal with the type uncertainty. Since P_2 knows his type value to be either θ^g or θ^b, his mixed-strategy $\bar{\sigma}_2 : \Theta_2 \mapsto \triangle\mathscr{A}_2$ should be a function of his type value. Thus, with a slight abuse of notation, $\bar{\sigma}_2(a_2, \theta_2) \geq 0, \forall a_2 \in \mathscr{A}_2, \forall \theta_2 \in \Theta_2$ is the probability of taking action a_2 under the type value θ_2. Clearly, the mixed-strategy is a probability measure, i.e., $\sum_{a_2 \in \mathscr{A}_2} \bar{\sigma}_2(a_2, \theta_2) = 1, \forall \theta_2 \in \Theta_2$. Suppose that P_1 manages to know the probability distribution of the type $b_1^0 \in \triangle\Theta_2$, e.g., defender P_1 believes with probability $b_1^0(\theta^g)$ that user P_2 is of a legitimate type and $b_1^0(\theta^b)$ that P_2 is of an adversarial type. Similarly, we have $\sum_{\theta_2 \in \Theta_2} b_1^0(\theta_2) = 1$ and $b_1^0(\theta_2) \geq 0, \forall \theta_2 \in \Theta_2$. □

Definition 5.3 A pair of mixed-strategy $(\sigma_1^*, \bar{\sigma}_2^*(\cdot))$ is said to constitute a (one-sided) mixed-strategy Bayesian Nash equilibrium (BNE) if

$$\sum_{\theta_2 \in \Theta_2} b_1^0(\theta_2) \sum_{a_1 \in \mathscr{A}_1} \sum_{a_2 \in \mathscr{A}_2} \sigma_1^*(a_1) \bar{\sigma}_2^*(a_2, \theta_2) \bar{J}_1(a_1, a_2, \theta_2) \geq$$

$$\sum_{\theta_2 \in \Theta_2} b_1^0(\theta_2) \sum_{a_1 \in \mathscr{A}_1} \sum_{a_2 \in \mathscr{A}_2} \sigma_1(a_1) \bar{\sigma}_2^*(a_2, \theta_2) \bar{J}_1(a_1, a_2, \theta_2), \forall \sigma_1(\cdot).$$

and

$$\sum_{a_1 \in \mathscr{A}_1} \sum_{a_2 \in \mathscr{A}_2} \sigma_1^*(a_1) \bar{\sigma}_2^*(a_2, \theta_2) \bar{J}_2(a_1, a_2, \theta_2) \geq$$

$$\sum_{a_1 \in \mathscr{A}_1} \sum_{a_2 \in \mathscr{A}_2} \sigma_1^*(a_1) \bar{\sigma}_2(a_2, \theta_2) \bar{J}_2(a_1, a_2, \theta_2), \forall \theta_2 \in \Theta_2, \forall \bar{\sigma}_2(\cdot, \theta_2).$$

□

Note that the binary type space Θ_2 can easily extend to finitely many elements to model different kinds of legitimate users and hackers who bear diverse type-related payoff functions. Since the type distinguishes different users and characterizes their essential attributes, the type space can also be a continuum and interpreted as a normalized measure of damages or the threat level to the system [11]. Moreover, the defender P_1 can also have a type $\theta_1 \in \Theta_1$, which forms a static version of the two-sided dynamic Bayesian game as shown in Sect. 5.4.2. Theorem 5.1 guarantees the existence of BNE regardless of extensions mentioned above.

Theorem 5.1 *A mixed-strategy BNE exists for a static Bayesian game with a finite type space. For games with a continuous type space and a continuous strategy space, if strategy sets and type sets are compact, payoff functions are continuous and concave in players' own strategies, then a pure-strategy BNE exists.*

5.4 Dynamic Bayesian Game for Deception and Counter-Deception

Followed from the above static Bayesian game with one-sided incomplete information, we investigate two types of dynamic games for cyber deception and counter-deception. The signaling game is two-stage and only the receiver has the incomplete information of the sender's type. The two-sided dynamic Bayesian game with a multi-stage state transition in Sect. 5.4.2 can be viewed as an extension of the signaling game. The solution concept in this section extends the BNE to the perfect Bayesian Nash equilibrium (PBNE).

5.4.1 Signaling Game for Cyber Deception

We illustrate the procedure of the signaling game as follows:

- An external player called the Nature draws a type θ_2 from a set $\Theta_2 := \{\theta^1, \theta^2, \cdots, \theta^I\}$ according to a given probability distribution $b_1^0 \in \triangle\Theta_2$ where $b_1^0(\theta^i) \geq 0, \forall i \in \{1, 2, \cdots, I\}$ and $\sum_{i=1}^{I} b_1^0(\theta^i) = 1$.
- The user P_2 (called the sender) observes the type value θ_2 and then chooses an action a_2 (called a message) from a finite set of message space $\mathscr{A}_2$.
- The defender P_1 (called the receiver) observes the action a_2 and then chooses her action $a_1 \in \mathscr{A}_1$.
- Payoffs $(\bar{J}_1(a_1, a_2, \theta_2), \bar{J}_2(a_1, a_2, \theta_2))$ are given to the sender and receiver, respectively.

5.4.1.1 Belief Formulation

Since the receiver P_1 has incomplete information about the sender's type, she will form a belief $b_1^1 : \mathscr{A}_2 \mapsto \triangle\Theta_2$ on the type θ_2 based on the observation of the sender's message a_2. As a measure of the conditional probability, the belief b_1^1 satisfies $b_1^1(\theta^i|a_2) \geq 0, \forall i \in \{1, 2, \cdots, I\}, \forall a_2 \in \mathscr{A}_2$ and $\sum_{\theta_2 \in \Theta_2} b_1^1(\theta_2|a_2) := \sum_{i=1}^{I} b_1^1(\theta^i|a_2) = 1, \forall a_2 \in \mathscr{A}_2$.

5.4.1.2 Receiver's Problem

For every received message a_2, receiver P_1 aims to optimize her expected payoffs under her belief $b_1^1(\cdot|a_2)$, that is:

$$\max_{a_1 \in \mathscr{A}_1} \sum_{\theta_2 \in \Theta_2} b_1^1(\theta_2|a_2) \bar{J}_1(a_1, a_2, \theta_2). \tag{5.1}$$

As a result, the receiver's (pure)-strategy is given by the mapping $\hat{a}_1 : \mathscr{A}_2 \mapsto \mathscr{A}_1$. Thus, the receive P_1's action is the outcome of the mapping, i.e., $a_1 = \hat{a}_1(a_2)$.

5.4.1.3 Sender's Problem

For every type $\theta_2 \in \Theta_2$ that the Nature picks for P_2, sender P_2 should pick a message $a_2 \in \mathscr{A}_2$ that maximizes the following utility with the anticipation of receiver's action $a_1 = \hat{a}_1(a_2)$, that is:

$$\max_{a_2 \in \mathscr{A}_2} \bar{J}_2(\hat{a}_1(a_2), a_2, \theta_2). \tag{5.2}$$

Hence, the sender's (pure)-strategy is given by the mapping $\bar{a}_2 : \Theta_2 \mapsto \mathscr{A}_2$ and P_2's action under the type value θ_2 is $a_2 = \bar{a}_2(\theta_2)$. The sender's strategy $\bar{a}_2$ is called a pooling strategy if he chooses the same message a_2 independent of the type given by the Nature, and is called a separating strategy if the mapping $\bar{a}_2$ is injective. For all other feasible mappings, $\bar{a}_2$ is called a semi-separating strategy.

5.4.1.4 Mixed-Strategy Receiver and Sender's Problem

We can extend the pure-strategy to the mixed-strategy $\hat{\sigma}_1 : \mathscr{A}_2 \mapsto \triangle\mathscr{A}_1$ for receiver P_1 and the same $\bar{\sigma}_2 : \Theta_2 \mapsto \triangle\mathscr{A}_2$ defined in Sect. 5.3 for sender P_2. After observing sender's message a_2 as a realization of the mix-strategy $\bar{\sigma}_2$, receiver P_1 assigns probability $\hat{\sigma}_1(a_1, a_2)$ to her action a_1 with the feasibility constraint

$\sum_{a_1 \in \mathscr{A}_1} \hat{\sigma}_1(a_1, a_2) = 1, \forall a_2 \in \mathscr{A}_2$ and $\hat{\sigma}_1(a_1, a_2) \geq 0, \forall a_1 \in \mathscr{A}_1, a_2 \in \mathscr{A}_2$. The expected objective functions for both players under the mixed-strategy are defined as follows:

$$\begin{aligned} &\max_{\hat{\sigma}_1(\cdot)} \sum_{\theta_2 \in \Theta_2} b_1^1(\theta_2|a_2) \sum_{a_1 \in \mathscr{A}_1} \hat{\sigma}_1(a_1, a_2) \bar{J}_1(a_1, a_2, \theta_2), \forall a_2 \in \mathscr{A}_2. \\ &\max_{\bar{\sigma}_2(\cdot)} \sum_{a_1 \in \mathscr{A}_1} \hat{\sigma}_1(a_1, a_2) \sum_{a_2 \in \mathscr{A}_2} \bar{\sigma}_2(a_2, \theta_2) \bar{J}_2(a_1, a_2, \theta_2), \forall \theta_2 \in \Theta_2. \end{aligned} \tag{5.3}$$

5.4.1.5 Belief Consistency

Since the message a_2 is a function of the type θ_2, the observation of the message should reveal some information of the type. Thus, the receiver updates the initial belief $b_1^0(\cdot)$ to form the posterior belief $b_1^1(\cdot|a_2)$ via the Bayesian rule:

$$\begin{aligned} &b_1^1(\theta_2|a_2) = \frac{b_1^0(\theta_2)\bar{\sigma}_2(a_2|\theta_2)}{\sum_{\theta_2 \in \Theta_2} b_1^0(\theta_2)\bar{\sigma}_2(a_2|\theta_2)}, \text{ if } \sum_{\theta_2 \in \Theta_2} b_1^0(\theta_2)\bar{\sigma}_2(a_2|\theta_2) > 0, \\ &b_1^1(\theta_2|a_2) = \text{any probability distributions, if } \sum_{\theta_2 \in \Theta_2} b_1^0(\theta_2)\bar{\sigma}_2(a_2|\theta_2) = 0. \end{aligned} \tag{5.4}$$

Serving as a particular case, the receiver and the sender's problem under the pure-strategy should also satisfy the Bayesian update of the belief. Note that although P_1 can observe the message a_2 which is a realization of $\bar{\sigma}_2$, she cannot directly update her belief via (5.4) if the signaling game is only played once. However, (5.4) contributes to the PBNE of the signaling game in Definition 5.4, serving as the belief consistency constraint.

Definition 5.4 A pure-strategy perfect Bayesian Nash equilibrium of the signaling game is a pair of strategies $(\hat{a}_1^*, \bar{a}_2^*)$ and belief $b_1^{1,*}$ that satisfy (5.1), (5.2), and (5.4). A mixed-strategy perfect Bayesian Nash equilibrium of the signaling game is a pair of strategies $(\hat{\sigma}_1^*, \bar{\sigma}_2^*)$ and belief $b_1^{1,*}$ that satisfy (5.3) and (5.4). □

The reader may already realize that we can use signaling game to model the same cyber deception scenario in Example 5.2 only with the difference of the asymmetric **information structure**, i.e., the defender P_1 has a chance to observe the behavior of the user P_2 before making her decision. The information asymmetry results in the following changes. First, P_1's mixed-strategy $\hat{\sigma}_1(a_2)$ is a function of her observation, i.e., P_2's action a_2. Second, instead of directly taking an expectation over the initial belief b_1^0, defender P_1 obtains a posterior belief b_1^1 that is consistent with the new observation a_2. Third, the type of belief can affect the PBNE even under the cheap-talk setting when utilities of both players are independent of the message. Finally, if there is only one type with a known b_1^0, which means that the

type value becomes common knowledge, the signaling game becomes a Stackelberg game with leader P_2 and follower P_1.

5.4.2 *Multi-stage with Two-Sided Incomplete Information*

The deceptive techniques adopted by the attacker make it challenging for the defender to correctly identify the type of the user even observing the manifested behavior as shown in Example 5.2. To tilt the information asymmetry, we can either continue to develop the intrusion detection system to increase the TPR with decreased FPR or refer to defensive deception techniques to create a sufficient amount of uncertainties for the attackers. Use of defensive and active deception as a counter-deception technique will disorient and slow down the adversarial infiltration because attackers have to judge the target's type, i.e., whether it is a real valuable production system or a well-pretended honeypot. Therefore, we introduce a two-sided incomplete information Bayesian game model with a multi-stage state transition for advanced attacks such as Advanced Persistent Threats (APTs) which infiltrate stage by stage.

5.4.2.1 Two-Sided Private Types

This section discusses the scenarios where not only the user P_2 has a type the defender P_1 also has a private type $\theta_1 \in \Theta_1$ to distinguish a system's different levels of sophistication and security awareness. For example, the defender's type space can be binary $\Theta_1 := \{\theta^H, \theta^L\}$ where θ^H represents a defender who is well-trained with a high-security awareness and also supported by advanced virus detection and analysis systems. Thus, she may refer to the log file with a higher frequency and more likely to obtain valuable information through the behavior analysis. Thus, once the attacker requests for privilege escalation and P_1 restricts and inspects the log file, a higher reward and a higher penalty are introduced under a high-type defender θ^H than a low-type defender θ^L, i.e., $r_0 = r_3 \cdot \mathbf{1}_{\theta^L} + r_4 \cdot \mathbf{1}_{\theta^H}$ where $r_4 > r_3 > 0$ as shown in Table 5.3.

Table 5.3 Utility bi-matrix when user P_2 is either adversarial $\theta_2 = \theta^b$ or legitimate $\theta_2 = \theta^g$ and defender P_1 is either of high type $\theta_1 = \theta^H$ or of low type $\theta_1 = \theta^L$

$\theta_2 = \theta^b$	NOP	Escalate
Permit	$(0, 0)$	$(-r_2, r_2)$
Restrict	$(0, 0)$	$(r_0, -r_0)$

$\theta_2 = \theta^g$	NOP	Escalate
Permit	$(0, 0)$	(r_1, r_1)
Restrict	$(0, 0)$	$(-r_1, -r_1)$

Two aspects motivate us to introduce a random variable as the defender's type, i.e., the user P_2 only knows the prior probability distribution over the type space Θ_1 yet not the value/realization of P_1's type. On the one hand, the

modern cyberinfrastructure networks have become increasingly interdependent and complicated, so it is hard to evaluate the system payoff accurately even given both players' actions. On the other hand, the adoption of defensive deception techniques brings uncertainties and difficulties for the user, especially attackers to evaluate the system setting. Therefore, we model the uncertainties by letting the utility function be a function of the type, which is a random variable.

5.4.2.2 A Scenario of Advanced Persistent Threats

A class of stealthy and well-planned sequence of hacking processes called Advanced Persistent Threats (APTs) motivates the multi-stage transition as well as two strategic players with two-sided incomplete information [12, 31]. Unlike the non-targeted attackers who spray a large number of phishing emails and pray for some "phools" [2] to click on the malicious links and get compromised, nation-sponsored APTs have sufficient amount of resources to initiate a reconnaissance phase to understand their targeted system thoroughly and tailor their attack strategies with the target. Multi-stage movement is an inherent feature of APTs as shown in Fig. 5.2. The APTs' life cycle includes a sequence of stages such as the initial entry, foothold establishment, privilege escalation, lateral movement, and the final targeted attacks on either confidential data or the physical infrastructures such as nuclear power stations and automated factories. APTs use each stage as a stepping stone for the next one. Unlike the static "smash and grab" attacks who launch direct attacks to obtain one-shot reward and then get identified and removed, APTs possess a long-term persistence and stage-by-stage infiltration to evade detection. For example, APTs can stealthily scan the port slowly to avoid hitting the warning threshold of the IDS. APTs hide and behave like legitimate users during the escalation and prorogation phases to deceive the defender until reaching the final stage, launch a "critical hit" on their specific targets and cause an enormous loss.

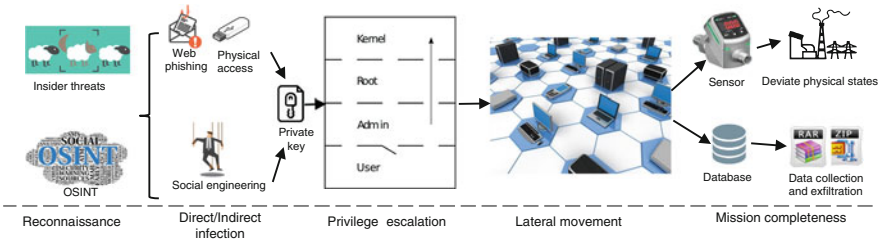

Fig. 5.2 The multi-stage life cycle of APTs forms a tree structure. During the reconnaissance phase, the threat actor probes the system and obtains intelligence from open-source information or insiders. The infection can be either directly through the web phishing and the physical access or indirectly through social engineering to manipulate the employees and then obtain a private key. Then, APTs gain the foothold, escalate privilege, propagate laterally in the cyber network, and finally either cause physical damages or collect confidential data

The classical intrusion prevention (IP) techniques such as the cryptography and the physical isolation can be ineffective for APTs because APTs can steal full cryptographic keys by techniques such as social engineering. Stuxnet, as one of the most well-known APTs, has proven to be able to successfully bridge the air gap between local area networks with the insertion of infected USB drives. Similarly, the intrusion detection (ID) approach including [5] can be ineffective if APTs acquire the setting of the detection system with the help of insiders. Moreover, APTs operated by human expert can analyze system responses and learn the detection rule during their inactivity, thus deceive the system defender and evade detection. Additionally, APTs can encrypt the data as well as their communication content with their human experts. A well-encrypted outbound network flow will limit the effectiveness of the data loss prevention (DLS) system which detects potential data exfiltration transmissions and prevents them by monitoring, detecting, and blocking sensitive data.

Hence, besides traditional defensive methods, i.e., IP, ID, DLS, it is essential to design strategic security mechanisms to capture the competitive interaction, the multi-stage multi-phase transition, as well as the adversarial and defensive deception between the APTs and advanced defenders. As shown in Table 5.3, the advanced defender with a private type is deceptive and increases the attacker's uncertainty. The defender is also adaptive because she forms and updates the belief of the user's type according to the observation of the user's actions as shown in (5.5).

5.4.2.3 Multi-stage State Transition

As shown in Fig. 5.2, the APT attacker moves stage by stage from the initial infection to the final target without jumps of multiple stages in one step. There are also no incentives for the attacker to go back to stages that he has already compromised because his ultimate goal is to compromise the specific target at the final stage. Therefore, we model the APT transition as a multi-stage game with a finite horizon K. Each player $i \in \{1, 2\}$ at each stage $k \in \{0, 1, \cdots, K\}$ can choose an action a_i^k from a stage-dependent finite set $\mathscr{A}_i^k$ because the feasible actions are different for each player at different stages. The history $h^k := \{a_1^0, \cdots, a_1^{k-1}, a_2^0, \cdots, a_2^{k-1}\} \in \mathscr{H}^k$ contains the actions of both players up to stage $k-1$ and can be obtained by reviewing system activities from the log file. Note that user's actions $a_2^k \in \mathscr{A}_2^k$ are the behaviors that are directly observable such as the privilege escalation request and the sensor access in the case study of Sect. 5.4.2.8. Since both legitimate and adversarial users can take these activities, a defender cannot identify the user's type directly from observing these actions. On the other hand, the defender's action a_1^k will be mitigation or proactive actions such as restricting the escalation request or monitoring the sensor access. These proactive actions also do not directly disclose the system type.

State $x^k \in \mathscr{X}^k$ representing the status of the system at stage t is the sufficient statistic of the history $\mathbf{h}^k$ because a Markov state transition $x^{k+1} = f^k(x^k, a_1^k, a_2^k)$ contains all the information of the history update $\mathbf{h}^k = \mathbf{h}^{k-1} \cup \{a_1^k, a_2^k\}$. Unlike the history, the cardinality of the state does not necessarily grow with the number of

stages. The function f^k is deterministic because history is fully observable without uncertainties. The function f^k is also stage-dependent and represents different meanings. For example, in Sect. 5.4.2.8, the state at the second last stage represents the current privilege level, while at the final stage, the state indicates which sensors have been compromised.

5.4.2.4 Behavior Mixed-Strategy and Believe Update

According to the information available at stage k, i.e., history h^k and his/her type θ_i, player i takes a behavioral mixed-strategy $\sigma_i^k : \mathscr{H}^k \times \Theta_i \mapsto \triangle\mathscr{A}_i^k$ with the available information as the input of the function. Note that $\sigma_i^k(a_i^k|h^k, \theta_i) \in \Sigma_i^k := \{\sigma_i^k(a_i^k|h^k, \theta_i) \geq 0 : \sum_{a_i^k \in \mathscr{A}_i^k} \sigma_i^k(a_i^k|h^k, \theta_i) = 1\}$ is the probability of taking action a_i^k given h^k, θ_i for all stage $k \in \{0, 1, \cdots, K\}$.

To correspond to the challenge of incomplete information of the other player's type, each player i forms a belief $b_i^k : \mathscr{H}^k \times \Theta_i \mapsto \triangle\Theta_{-i}$ that maps the available information h^k, θ_i to the distribution over the type space of the other player. Likewise, $b_i^k(\theta_{-i}|h^k, \theta_i)$ at stage k is the conditional probability mass function (PMF) of the other player's type θ_{-i} and $\sum_{\theta_{-i} \in \Theta_{-i}} b_i^k(\theta_{-i}|h^k, \theta_i)d\theta_{-i} = 1, \forall k \in \{0, 1, \cdots, K\}, \forall h^k \in \mathscr{H}^k, \theta_i \in \Theta_i, i \in \{1, 2\}$.

Assume that each player i knows the prior distribution of the other player's type, i.e., b_i^0 according to the historical data and the statistical analysis. If no prior information is available, a uniform distribution is an unbiased estimate. Since the multi-stage model provides a sequential observation of the other player's action a_{-i}^k which is a realization of the mixed-strategy σ_{-i}^k, player i's belief of the other's type can be updated via the Bayesian rule, that is:

$$b_i^{k+1}(\theta_{-i}|[h^k, a_i^k, a_{-i}^k], \theta_i) = \frac{b_i^k(\theta_{-i}|h^k, \theta_i)\sigma_{-i}^k(a_{-i}^k|h^k, \theta_{-i})}{\sum_{\theta_{-i} \in \Theta_i} b_i^k(\theta_{-i}|h^k, \theta_i)\sigma_{-i}^k(a_{-i}^k|h^k, \theta_{-i})}. \tag{5.5}$$

Note that the one-shot observation of the other player's action does not directly disclose the type because of the deception. However, since the utility function in Sect. 5.4.2.5 is type dependent, the action made by the type-dependent policy will serve as a message that contributes to a better estimate of the other's type. The accuracy of the belief will be continuously improved when more actions are observed.

5.4.2.5 Utility Function and PBNE

At each stage k, J_i^k is the utility that depends on the type and the action of both players, the current state x^k, and some external random noise w_i^k with a known distribution. We introduce the external noise to model other unknown factors that

could affect the value of the stage utility. The existence of the external noise makes it impossible for each player i to directly acquire the value of the other's type θ_{-i} based on the combined observation of input parameters $x^k, a_1^k, a_2^k, \theta_i$ plus the output value of the utility function J_i^k. In the case study, we consider any additive noise with a 0 mean, i.e., $J_i^k(x^k, a_1^k, a_2^k, \theta_i, \theta_{-i}, w_i^k) = \tilde{J}_i^k(x^k, a_1^k, a_2^k, \theta_i, \theta_{-i}) + w_i^k$, which leads to an equivalent payoff over the expectation of the external noise $E_{w_i^k} J_i^k = \tilde{J}_i^k, \forall x^k, a_1^k, a_2^k, \theta_i, \theta_{-i}$.

One significant improvement from the static game to the dynamic game is that each player i has a long-term objective to maximize the total expected payoff $U_i^{k':K}$. For example, attackers of APTs may sacrifice the immediate attacking reward to remain stealthy and receive more considerable benefits in the following stages, e.g., successfully reach the final target and complete their mission. Define $\sigma_i^{k':K} := [\sigma_i^k(a_i^k|h^k, \theta_i)]_{k=k',\cdots,K}$ and the cumulative expected utility $U_i^{k':K}$ sums the expected stage utilities from stage k' to K as follows:

$$\begin{aligned} & U_i^{k':K}(\sigma_i^{k':K}, \sigma_{-i}^{k':K}, h^{K+1}, \theta_i) \\ & := \sum_{k=k'}^{K} E_{\theta_{-i} \sim b_i^k, \sigma_i^k, \sigma_{-i}^k, w_i^k}[J_i^k(x^k, \sigma_i^k, \sigma_{-i}^k, \theta_i, \theta_{-i}, w_i^k)] \\ & = \sum_{k=k'}^{K} \sum_{\theta_{-i} \in \Theta_{-i}} b_i^k(\theta_{-i}|h^k, \theta_i) \sum_{a_i^k \in \mathscr{A}_i^k} \sigma_i^k(a_i^k|h^k, \theta_i) \\ & \cdot \sum_{a_{-i}^k \in \mathscr{A}_{-i}^k} \sigma_{-i}^k(a_{-i}^k|h^k, \theta_{-i}) \tilde{J}_i^k(x^k, a_i^k, a_{-i}^k, \theta_i, \theta_{-i}). \end{aligned} \tag{5.6}$$

Similar to the PBNE of the signaling game, the PBNE of multi-stage Bayesian game defined in Definition 5.5 requires a K-stage belief consistency. Since the equilibrium may not always exist, an ε-equilibrium is introduced.

Definition 5.5 In the two-person K-stage Bayesian game with two-sided incomplete information and a cumulative payoff function $U_i^{k':K}$ in (5.6), a sequence of strategies $\sigma_i^{*,k':K} \in \prod_{k=k'}^{K} \Sigma_i^k$ is called the ε perfect Bayesian Nash equilibrium for player i, if b_i^k satisfies the consistency constraint (5.5) for all $k \in \{0, 1, \cdots, K-1\}$ and for a given $\varepsilon \geq 0$,

$$U_1^{k:K}(\sigma_1^{*,k:K}, \sigma_2^{*,k:K}, h^{K+1}, \theta_1) \geq \sup_{\sigma_1^{k:K}} U_1^{k:K}(\sigma_1^{k:K}, \sigma_2^{*,k:K}, h^{K+1}, \theta_1) - \varepsilon.$$

$$U_2^{k:K}(\sigma_1^{*,k:K}, \sigma_2^{*,k:K}, h^{K+1}, \theta_2) \geq \sup_{\sigma_2^{k:K}} U_2^{k:K}(\sigma_1^{*,k:K}, \sigma_2^{k:K}, h^{K+1}, \theta_2) - \varepsilon.$$

If $\varepsilon = 0$, we have a perfect Bayesian Nash equilibrium.

5.4.2.6 Dynamic Programming

Given any feasible belief at every stage, we can use dynamic programming to find the PBNE in a backward fashion because of the tree structure and the finite horizon. Define the value function as the utility-to-go function under the PBNE strategy pair, that is:

$$V_i^k(h^k, \theta_i) = U_i^{k:K}(\sigma_i^{*,k:K}, \sigma_{-i}^{*,k:K}, h^{k+1}, \theta_i).$$

Let $V_i^{K+1}(h^{K+1}, \theta_i) := 0$ be the boundary condition of the value function, we have the following recursive system equations to solve the PBNE mixed-strategies $\sigma_1^{*,k}, \sigma_2^{*,k}$ for all stage $k = \{0, 1, \cdots, K\}$:

$$\begin{cases} V_1^{k-1}(h^{k-1}, \theta_1) = \sup_{\sigma_1^{k-1} \in \Sigma_1^{k-1}} E_{\theta_2 \sim b_1^{k-1}, \sigma_1^{k-1}, \sigma_2^{*,k-1}} \\ [V_1^k([h^{k-1}, a_1^{k-1}, a_2^{k-1}], \theta_1) + \tilde{J}_1^{k-1}(x^{k-1}, a_1^{k-1}, a_2^{k-1}, \theta_1, \theta_2)]; \\ V_2^{k-1}(h^{k-1}, \theta_2) = \sup_{\sigma_2^{k-1} \in \Sigma_2^{k-1}} E_{\theta_1 \sim b_2^{k-1}, \sigma_1^{*,k-1}, \sigma_2^{k-1}} \\ [V_2^k([h^{k-1}, a_1^{k-1}, a_2^{k-1}], \theta_2) + \tilde{J}_2^{k-1}(x^{k-1}, a_1^{k-1}, a_2^{k-1}, \theta_1, \theta_2)]. \end{cases} \tag{5.7}$$

Under the assumption of a Markov mixed-strategy $\tilde{\sigma}_i^t(a_i^k | x^k, \theta_i) \equiv \sigma_i^k(a_i^k | h^k, \theta_i)$, $\tilde{V}_i^k(x^k, \theta_i)$ becomes the sufficient statistics of $V_i^k(h^k, \theta_i)$. By replacing $\sigma_i^k(a_i^k | h^k, \theta_i)$ to $\tilde{\sigma}_i^k(a_i^k | x^k, \theta_i)$ and $V_i^k(h^k, \theta_i)$ to $\tilde{V}_i^k(x^k, \theta_i)$ in (5.7), we can obtain a new dynamic programming equation:

$$\begin{aligned} &\tilde{V}_i^{k-1}(x^{k-1}, \theta_i) = \sup_{\tilde{\sigma}_i^{k-1}} E_{\theta_{-i} \sim b_i^{k-1}, \tilde{\sigma}_i^{k-1}, \tilde{\sigma}_{-i}^{*,k-1}} \\ &[\tilde{V}_i^k(f^k(x^{k-1}, a_1^{k-1}, a_2^{k-1}), \theta_i) + \tilde{J}_i^{k-1}(x^{k-1}, a_1^{k-1}, a_2^{k-1}, \theta_1, \theta_2)]. \end{aligned} \tag{5.8}$$

5.4.2.7 PBNE Computation by Bi-linear Programming

To compute the PBNE, we need to solve a coupled system of the forward belief update in (5.5) that depends on the PBNE strategies plus a backward PBNE computation in (5.8) that can also be influenced by the type belief. If there are no additional structures to explore, we have to use a forward and backward iteration with the boundary condition of the initial belief $b_i^0(\theta_{-i})$ and final stage utility-to-go $\tilde{V}_i^{K+1}(x^{K+1}, \theta_i) = 0$. In particular, we first assign any feasible value to the type belief $b_i^k, k \in \{1, 2 \cdots, K\}$, then solve (5.8) from stage $k = K$ to $k = 0$ and use the resulted PBNE strategy pair to update (5.5). We iteratively compute (5.8) and (5.5) until both the K-stage belief and the PBNE strategy do not change, which provides a consistent pair of the PBNE and the belief. If the iteration process does

not converge, then the PBNE does not exist. Define l_{m_i} as the column vector of ones with a dimension of m_i, we propose a bi-linear program to solve the PBNE strategy for any given belief $b_i^k, k \in \{1, 2 \cdots, K\}$, which leads to Theorem 5.2. The type space can be either discrete or continuous. We refer reader to Section 4.4 in [12] for the proof of the theorem.

Theorem 5.2 *A strategy pair* $(\tilde{\sigma}_1^{*,k}, \tilde{\sigma}_2^{*,k})$ *with the feasible state* $x^k \in \mathscr{X}^k$ *and the consistent belief sequence* b_i^k *at stage* $k \in \{0, 1, \cdots, K\}$ *constitutes a mixed-strategy PBNE of the multi-stage Bayesian game in Definition 5.5, if, and only if, there exists a sequence of scalar function pair* $(s^{*,k}(\theta_1), w^{*,k}(\theta_2))$ *such that* $\tilde{\sigma}_1^{*,k}(\cdot|x^k, \theta_1), \tilde{\sigma}_2^{*,k}(\cdot|x^k, \theta_2)$, $s^{*,k}(\theta_1), w^{*,k}(\theta_2)$ *are the optimal solutions to the following bi-linear program for each* $k \in \{0, 1, \cdots, K\}$*:*

$$
\begin{aligned}
&\sup_{\tilde{\sigma}_1^k, \tilde{\sigma}_2^k, s, w} \sum_{\theta_1 \in \Theta_1} b_1^k(\theta_1) \sum_{\theta_2 \in \Theta_2} b_2^k(\theta_2) \sum_{a_1^k \in \mathscr{A}_1^k} \tilde{\sigma}_1^k(a_1^k|x^k, \theta_1) \sum_{a_2^k \in \mathscr{A}_2^k} \tilde{\sigma}_2^k(a_2^k|x^k, \theta_2) \\
&\qquad \sum_{i=1}^{2} [\tilde{J}_i^k(x^k, a_1^k, a_2^k, \theta_1, \theta_2) + \tilde{V}_i^{k+1}(f^k(x^k, a_1^k, a_2^k), \theta_i)] \\
&\qquad + \sum_{\theta_2 \in \Theta_2} b_2^k(\theta_2) w(\theta_2) + \sum_{\theta_1 \in \Theta_1} b_1^k(\theta_1) s(\theta_1) \\
s.t. \quad &(a). \sum_{\theta_1 \in \Theta_1} b_1^k(\theta_1) \sum_{a_1^k \in \mathscr{A}_1^k} \tilde{\sigma}_1^k(a_1^k|x^k, \theta_1) [\tilde{J}_2^k(x^k, a_1^k, a_2^k, \theta_1, \theta_2) \\
&\qquad + \tilde{V}_2^{k+1}(f^k(x^k, a_1^k, a_2^k), \theta_2)] \leq -w(\theta_2) l_{m_2}, \forall \theta_2 \in \Theta_2 \\
&(b). \sum_{\theta_2 \in \Theta_2} b_2^k(\theta_2) \sum_{a_2^k \in \mathscr{A}_2^k} \tilde{\sigma}_2^k(a_2^k|x^k, \theta_2) [\tilde{J}_1^k(x^k, a_1^k, a_2^k, \theta_1, \theta_2) \\
&\qquad + \tilde{V}_1^{k+1}(f^k(x^k, a_1^k, a_2^k), \theta_1)] \leq -s(\theta_1) l_{m_1}, \forall \theta_1 \in \Theta_1.
\end{aligned}
\tag{5.9}
$$

□

Note that the solution of (5.9) at stage $k+1$ provides the value of $\tilde{V}_i^{k+1}$ and $\tilde{V}_i^{K+1} = 0$ is a known value. Thus, we can solve (5.9) from $k = K$ to $k = 0$ for any given type belief.

5.4.2.8 An Illustrative Case Study

We adopt the same binary type space in Sect. 5.4.2.1 and consider the following three-stage ($K = 2$) transition. The proactive defensive actions listed in the case study should be combined with the reactive methods such as the firewall to defend attacks other than APTs.

Initial Stage

We consider the web phishing scenario for the initial entry. The state space $\mathscr{X}^0 := \{0, 1\}$ of the initial stage is binary. Let $x^0 = 0$ represent that the user sends the email from an external IP domain, while $x^0 = 1$ represents an email from the internal network domain. The attacker can also start from state $x^0 = 1$ due to the insider threats and the social engineering techniques.

To penalize the adversarial exploitation of the open-source intelligence (OSINT) data, the defender can create avatars (fake personal profiles) on the social network or the company website. The user P_2 at the initial stage can send emails to a regular employee $a_2^0 = 0$, a Chief Executive Officer (CEO) $a_2^0 = 1$, or the avatar $a_2^0 = 2$. The email can contain a legitimate shortening Uniform Resource Locator (URL). If the user is legitimate, the URL will lead to the right resources, yet if the user is malicious, the URL will redirect to a malicious site and then take control of the client's computer. As for the defender, suppose that P_1 proactively equips the computer with an anti-virus system that can run the email in the sandbox and apply penetration test. However, the limited budget can only support either the employees' computer or the CEO's computer. Thus, the defender also has three possible actions, i.e., equips the CEO' computer $a_1^0 = 2$, the employee's computer $a_1^0 = 1$, or does not equip the anti-virus system $a_1^0 = 0$ to avoid a deployment fee c_0^0. The defender of high-security awareness θ^H will deploy an advanced anti-virus system that costs higher installation fee than the regular anti-virus system, i.e., $c_2^0 > c_1^0$, yet also provides a higher penalty to the attacker, i.e., $r_4^0 > r_3^0$. Define $c_0^0 := c_1^0 \mathbf{1}_{\{\theta_1 = \theta^L\}} + c_2^0 \mathbf{1}_{\{\theta_1 = \theta^H\}}$ as the deployment fee for two types of the defender and $r_0^0 := r_3^0 \mathbf{1}_{\{\theta_1 = \theta^L\}} + r_4^1 \mathbf{1}_{\{\theta_1 = \theta^H\}}$ as the penalty for attackers. The attacker $\theta_2 = \theta^b$ will receive a faked reward $r_5^0 > 0$ when contacting the avatar, yet he then arrives at an unfavorable state, thus receives limited rewards in the future stages. The equivalent utility matrix $\tilde{J}_i^0(x^0, a_1^0, a_2^0, \theta_i, \theta_{-i})$ is shown in Table 5.4. Although the legitimate user can also take action $a_2^0 = 2$, he should assign zero probability to that action as the payoff is $-\infty$, i.e., a legitimate user should not contact a person who does not exist.

Table 5.4 The utility matrix $(\tilde{J}_1^0, \tilde{J}_2^0)$ for player $i = 1, 2$ under different types. Although the utility matrix is independent of the current state x^0, the action will affect the state transition f^0 and then the final state x^K where the utility is state-dependent

$\theta_2 = \theta^g$	Employee	CEO	Avatars
NOP	$(0, r_1^0)$	$(0, r_1^0)$	$(0, -\infty)$
Employee	$(-c_0^0, r_1^0)$	$(-c_0^0, r_1^0)$	$(-c_0^0, -\infty)$
CEO	$(-c_0^0, r_1^0)$	$(-c_0^0, r_1^0)$	$(-c_0^0, -\infty)$
$\theta_2 = \theta^b$	Employee	CEO	Avatars
NOP	$(-r_2^0, r_2^0)$	$(-r_2^0, r_2^0)$	$(0, r_5^0)$
Employee	$(-c_0^0, -r_0^0)$	$(-c_0^0, r_2^0)$	$(-c_0^0, r_5^0)$
CEO	$(-c_0^0, r_2^0)$	$(-c_0^0, -r_0^0)$	$(-c_1^0, r_5^0)$

Suppose that there are three possible states $\mathscr{X}^1 = \{0, 1, 2\}$ as the output of the initial state transition function f^0, i.e., user P_2 can reach the employee's computer

$x^1 = 1$, the CEO's computer $x^1 = 2$, or the honeypot $x^1 = 0$. Assume that the state transition from the initial state $x^0 = 1$ is determined only by the user's action, i.e., the defender's action does not affect the email delivery from the internal network. On the other hand, the state transition from the external domain $x^0 = 0$ is represented as follows. If defender chooses not to apply malware analysis system $a_1^0 = 0$, then user's action $a_2^0 = 0, 1, 2$ will lead the initial state $x^0 = 0$ to state $x^1 = 1, 2, 0$, respectively. If defender chooses a proactive deployment on the employee's computer $a_1^0 = 1$, then user's action $a_2^0 = 0, 2$ will drive the initial state $x^0 = 0$ to state $x^1 = 0$ and user's action $a_2^0 = 1$ will drive the initial state $x^0 = 0$ to state $x^1 = 2$. The mitigation of the attack is at the tradeoff of blocking some emails from the legitimate user. Likewise, if defender chooses a proactive deployment on the CEO's computer $a_1^0 = 2$, then user's action $a_2^0 = 1, 2$ will lead the initial state $x^0 = 0$ to state $x^1 = 0$ and user's action $a_2^0 = 0$ will lead the initial state $x^0 = 0$ to state $x^1 = 1$.

Intermediate Stage

Without loss of generality, we use the privilege escalation scenario in Table 5.3 as the intermediate stage $k = 1$. Although the utility matrix is independent of the current state x^1, the action will influence the long-term benefit by affecting the state transition f^1 as follows. The output state space $\mathscr{X}^K = \{0, 1, 2, 3\}$ represents four different levels of privilege from low to high. If the user is at the honeypot $x^1 = 0$, then he will end up at the honeypot with level-zero privilege $x^K = 0$ whatever actions he takes. For the user who has arrived at the employee's computer $x^1 = 1$, if the defender allows privilege escalation $a_1^1 = 0$, then if the user chooses NOP $a_2^1 = 0$, the user arrives at level-one privilege $x^K = 1$, else if the user requests escalation $a_2^1 = 1$, he arrives at level-two privilege $x^K = 2$. If the defender restricts the privilege escalation $a_1^1 = 1$, then P_2 arrives at state $x^K = 1$ regardless of his action. The user who arrive at the CEO's computer $x^1 = 2$ possesses a higher privilege level. Then, action pair $a_1^1 = 0, a_2^1 = 0$ leads to $x^K = 2$, and $a_1^1 = 0, a_2^1 = 1$ leads to $x^K = 3$, and $a_1^1 = 1, a_2^1 = 0/1$ leads to $x^K = 2$.

Final Stage

Table 5.5 Two players' utility when the user is either adversarial or legitimate. Define $r_0^K := r_2^K \mathbf{1}_{\{\theta_1 = \theta_1^L\}} + r_3^K \mathbf{1}_{\{\theta_1 = \theta_1^H\}}$ as the monitoring reward for two types of systems

$\theta_2 = \theta^b$	NOP	Access
NOP	$(0, 0)$	$(r_1^K, r_4^K - r_1^K)$
Monitor	$(-c^K, 0)$	$(r_0^K - c^K, -r_0^K)$

$\theta_2 = \theta^g$	NOP	Access
NOP	$(0, 0)$	(r_4^K, r_4^K)
Monitor	$(-c^K, 0)$	$(r_4^K - c^K, r_4^K)$

At the final stage $k = K$, we use the Tennessee Eastman (TE) Challenge Process [26] as an example to illustrate how attackers tend to compromise the sensors to cause physical damages (state deviation) of an industrial plant and monetary losses (Table 5.5). The user's action is to get access to the sensor controller $a_2^K = 1$ or not $a_2^K = 0$, yet a user at different levels of privilege x^K determines which sensors he can control in the TE process. If the attacker changes the sensor reading, the system states such as the pressure and the temperature may deviate from the desired value, which degrades the product quality and even causes the shutdown of the entire process if the deviation exceeds the safety threshold. Thus, the shutdown time, as well as the product quality, can be used as the operating reward measure. By simulating the TE process, we can determine the reward under the regular operation of the TE process $r_4^K(x^K)$ as well as the reward under the compromised sensor readings $r_1^K(x^K)$. Both r_4^K and r_1^K are a function of the state x^K. Assume that the attacker benefits from the reward reduction under the attacking operation $r_4^K(x^K) - r_1^K(x^K)$ and the system loss under attacks is higher than the monitoring cost $r_4^K(x^K) - r_1^K(x^K) > c^K > 0, \forall x^K \in \mathscr{X}^K$. On the other hand, the defender chooses to monitor the sensor controller $a_1^K = 1$ with a cost c^K or not to monitor $a_1^K = 0$. Also, we assume $r_3^K > r_2^K > c^K > 0$ because the high-type system can collect more information from the monitoring data and the benefit outweighs the monitor cost.

5.5 Conclusion and Future Works

The area of cybersecurity is an uneven battlefield. First, an attacker merely needs to exploit a few vulnerabilities to compromise a system while a defender has to eliminate all potential vulnerabilities. Second, the attacker has a plenty of time to study the targeted system yet it is hard for the defender to predict possible settings of attacks until they have happened. Third, the attacker can be strategic and deceptive and the defender has to adapt to variations and updates of the attacker. In this chapter, we aim to avoid the route of analyzing every attacks and taking costly countermeasures. However, we endeavor to tilt the unfavorable situation for the defender by applying a series of game theory models to capture the strategic interactions, the multi-stage persistence, as well as the adversarial and defensive cyber deceptions. Future directions include a combination of the theoretical models with data from the simulated or real system under attacks. The analysis of the game theory model provides a theoretic underpinning for our understandings of cybersecurity problems. We can further leverage the scientific and quantitative foundation to investigate mechanism design problems to construct a new battlefield that reverses the attacker's advantage and make the scenario in favor of the defender.

5.6 Exercise

5.6.1 Question 1: Equilibrium Computation

1.1. Write a bi-linear program to compute the PBNE of multi-stage game with one-sided incomplete information, i.e., only the user has a type $\theta_1 \in \Theta_1$, the defender does not have a type or P_1 knows her type. Represent it in a matrix form. (Hint: Readers can refer to Corollary 1 in [12].)

1.2. Compute the mixed-strategy BNE for the static Bayesian game in Table 5.2 with unbiased belief $b_1(\theta^b) = b_1(\theta^g) = 0.5$. You can program it in Matlab with the toolbox `Yalmip`[2] and a proper non-linear solver such as `fminicon`.[3] (Hint: PBNE degenerates to BNE when we take $K = 0$.)

5.6.2 Question 2: The Negative Information Gain in Game Theory

Consider a static Bayesian game with the binary type space $\Theta = \{\theta^1, \theta^2\}$ and initial type belief $b_1(\theta^1) = b_1(\theta^2) = 0.5$ as shown in Table 5.6. Player 1 is the row player and P_2 is the column player. Both players are rational and maximize their own payoffs.

Table 5.6 A static Bayesian game under two possible types θ^1 and θ^2

$\theta = \theta^1$	a	b
A	(10,10)	(18,4)
B	(7,19)	(17,17)

$\theta = \theta^2$	a	b
A	(10,10)	(18,18)
B	(14,18)	(20,20)

2.1. Compute the BNE strategy and the value of the game, i.e., each player's utility under the BNE strategy. (Hint: You should obtain a pure-strategy BNE (B, b) and the value is (18.5, 18.5).)

2.2. Suppose that the type value is known to both players, determine the NE under θ^1 and θ^2, respectively.

2.3. Compute the BNE with one-sided incomplete information, i.e., only P_1 knows the type value, which is *common knowledge*. The term *common knowledge* means that P_1 knows the type, P_2 knows that P_1 knows the type, and P_1 knows that P_2 knows that P_1 knows the type, etc.

[2] https://yalmip.github.io/.

[3] https://www.mathworks.com/help/optim/ug/fmincon.html.

2.4. Compare the results in question 1–3, does more information always benefit the player with extra information? If not, please give an explanation for the negative information gain in the game setting?

References

1. Aghassi, M., Bertsimas, D.: Robust game theory. Mathematical Programming **107**(1–2), 231–273 (2006)
2. Akerlof, G.A., Shiller, R.J.: Phishing for phools: The economics of manipulation and deception. Princeton University Press (2015)
3. Axelsson, S.: Intrusion detection systems: A survey and taxonomy. Tech. rep., Technical report (2000)
4. Chen, J., Zhu, Q.: Security investment under cognitive constraints: A gestalt Nash equilibrium approach. In: Information Sciences and Systems (CISS), 2018 52nd Annual Conference on, pp. 1–6. IEEE (2018)
5. Coppolino, L., D'Antonio, S., Romano, L., Spagnuolo, G.: An intrusion detection system for critical information infrastructures using wireless sensor network technologies. In: Critical Infrastructure (CRIS), 2010 5th International Conference on, pp. 1–8. IEEE (2010)
6. Corporation, S.: Advanced persistent threats: A symantec perspective. URL https://www.symantec.com/content/en/us/enterprise/white_papers/b-advanced_persistent_threats_WP_21215957.en-us.pdf
7. Farhang, S., Manshaei, M.H., Esfahani, M.N., Zhu, Q.: A dynamic Bayesian security game framework for strategic defense mechanism design. In: Decision and Game Theory for Security, pp. 319–328. Springer (2014)
8. Harsanyi, J.C.: Games with incomplete information played by "Bayesian" players, i–iii part i. the basic model. Management science **14**(3), 159–182 (1967)
9. Horák, K., Zhu, Q., Bošanskỳ, B.: Manipulating adversary's belief: A dynamic game approach to deception by design for proactive network security. In: International Conference on Decision and Game Theory for Security, pp. 273–294. Springer (2017)
10. Huang, L., Chen, J., Zhu, Q.: A large-scale Markov game approach to dynamic protection of interdependent infrastructure networks. In: International Conference on Decision and Game Theory for Security, pp. 357–376. Springer (2017)
11. Huang, L., Zhu, Q.: Adaptive strategic cyber defense for advanced persistent threats in critical infrastructure networks. In: ACM SIGMETRICS Performance Evaluation Review (2018)
12. Huang, L., Zhu, Q.: Analysis and computation of adaptive defense strategies against advanced persistent threats for cyber-physical systems. In: International Conference on Decision and Game Theory for Security (2018)
13. Jajodia, S., Ghosh, A.K., Swarup, V., Wang, C., Wang, X.S.: Moving target defense: creating asymmetric uncertainty for cyber threats, vol. 54. Springer Science & Business Media (2011)
14. Jajodia, S., Shakarian, P., Subrahmanian, V., Swarup, V., Wang, C.: Cyber warfare: building the scientific foundation, vol. 56. Springer (2015)
15. Lei, C., Ma, D.H., Zhang, H.Q.: Optimal strategy selection for moving target defense based on Markov game. IEEE Access **5**, 156–169 (2017)
16. Mahon, J.E.: The definition of lying and deception. In: E.N. Zalta (ed.) The Stanford Encyclopedia of Philosophy, winter 2016 edn. Metaphysics Research Lab, Stanford University (2016)
17. Maleki, H., Valizadeh, S., Koch, W., Bestavros, A., van Dijk, M.: Markov modeling of moving target defense games. In: Proceedings of the 2016 ACM Workshop on Moving Target Defense, pp. 81–92. ACM (2016)

18. Manshaei, M.H., Zhu, Q., Alpcan, T., Bacşar, T., Hubaux, J.P.: Game theory meets network security and privacy. ACM Computing Surveys (CSUR) **45**(3), 25 (2013)
19. Miao, F., Zhu, Q., Pajic, M., Pappas, G.J.: A hybrid stochastic game for secure control of cyber-physical systems. Automatica **93**, 55–63 (2018)
20. Pawlick, J., Colbert, E., Zhu, Q.: A game-theoretic taxonomy and survey of defensive deception for cybersecurity and privacy. arXiv preprint arXiv:1712.05441 (2017)
21. Pawlick, J., Colbert, E., Zhu, Q.: Modeling and analysis of leaky deception using signaling games with evidence. arXiv preprint arXiv:1804.06831 (2018)
22. Pawlick, J., Zhu, Q.: Deception by design: evidence-based signaling games for network defense. arXiv preprint arXiv:1503.05458 (2015)
23. Pawlick, J., Zhu, Q.: A Mean-Field Stackelberg Game Approach for Obfuscation Adoption in Empirical Risk Minimization. arXiv preprint arXiv:1706.02693 (2017). URL https://arxiv.org/abs/1706.02693
24. Pawlick, J., Zhu, Q.: Proactive defense against physical denial of service attacks using Poisson signaling games. In: International Conference on Decision and Game Theory for Security, pp. 336–356. Springer (2017)
25. Rass, S., Alshawish, A., Abid, M.A., Schauer, S., Zhu, Q., De Meer, H.: Physical intrusion games–optimizing surveillance by simulation and game theory. IEEE Access **5**, 8394–8407 (2017)
26. Ricker, N.L.: Tennessee Eastman Challenge Archive. http://depts.washington.edu/control/LARRY/TE/download.html (2013)
27. Xu, Z., Zhu, Q.: A Game-Theoretic Approach to Secure Control of Communication-Based Train Control Systems Under Jamming Attacks. In: Proceedings of the 1st International Workshop on Safe Control of Connected and Autonomous Vehicles, pp. 27–34. ACM (2017). URL http://dl.acm.org/citation.cfm?id=3055381
28. Zhang, T., Zhu, Q.: Strategic defense against deceptive civilian GPS spoofing of unmanned aerial vehicles. In: International Conference on Decision and Game Theory for Security, pp. 213–233. Springer (2017)
29. Zhu, Q., Başar, T.: Game-theoretic approach to feedback-driven multi-stage moving target defense. In: International Conference on Decision and Game Theory for Security, pp. 246–263. Springer (2013)
30. Zhu, Q., Clark, A., Poovendran, R., Basar, T.: Deployment and exploitation of deceptive honeybots in social networks. In: Decision and Control (CDC), 2013 IEEE 52nd Annual Conference on, pp. 212–219. IEEE (2013)
31. Zhu, Q., Rass, S.: On multi-phase and multi-stage game-theoretic modeling of advanced persistent threats. IEEE Access **6**, 13958–13971 (2018)
32. Zhuang, J., Bier, V.M., Alagoz, O.: Modeling secrecy and deception in a multiple-period attacker–defender signaling game. European Journal of Operational Research **203**(2), 409–418 (2010)

Part III
Network-Based Deception

Chapter 6
CONCEAL: A Strategy Composition for Resilient Cyber Deception: Framework, Metrics, and Deployment

Qi Duan, Ehab Al-Shaer, and Mazharul Islam

Abstract Cyber deception is a key proactive cyber resilience technique to reverse the current asymmetry that favors adversaries in cyber warfare by creating a significant confusion in discovering and targeting cyber assets. One of the key objectives for cyber deception is to hide the true identity of the cyber assets in order to effectively deflect adversaries away from critical targets, and detect their activities early in the kill chain.

Although many cyber deception techniques were proposed including using honeypots to represent fake targets and mutating IP addresses to frequently change the ground truth of the network configuration (Jafarian et al., IEEE Transactions on Information Forensics and Security 10(12):2562–2577 (2015)), none of these deception techniques is resilient enough to provide high confidence of concealing the identity of the network assets, particularly against sophisticated attackers. In fact, in this chapter our analytical and experimental work showed that highly resilient cyber deception is unlikely attainable using a single technique, but it requires an optimal composition of various concealment techniques to maximize the deception utility. We, therefore, present a new cyber deception framework, called CONCEAL, which is a composition of mutation, anonymity, and diversity to maximize key deception objectives, namely concealability, detectability, and deterrence, while constraining the overall deployment cost. We formally define the CONCEAL metrics for concealability, detectability, and deterrence to measure the effectiveness of CONCEAL. Finally, we present the deployment of CONCEAL as a service to achieve manageability and cost-effectiveness by automatically generating the optimal deception proxy configuration based on existing host/network

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

Q. Duan (✉) · E. Al-Shaer · M. Islam
Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC, USA
e-mail: qduan@uncc.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_6

configuration, risk constraints of network services, and budget constraints. Our evaluation experiments measure both the deception effectiveness based on the above metrics and the scalability of the CONCEAL framework.

6.1 Introduction and Motivation

Cyber deception is an act of intentional misrepresentation of facts in order to induce an incorrect perception of reality in adversaries' mind. As a result, an adversary can be misled about the true configuration of the system such that their belief will be different from the reality. This false reality can assist in deflecting adversaries to a desired state of knowledge for invalidating their effort, slowing down their progress, and/or learning about their goals and techniques, even when adversary's techniques are unknown.

We believe that an effective cyber deception technique requires at least these five criteria: (1) It should provide a resilient *concealment* of the identity of cyber assets (at least the critical ones) including the real and fake (e.g., honeypots services), even if they are at some point discovered, (2) It should significantly increase the potential mistakes of the attackers (*detectability*), (3) It should significantly increase attackers' effort to achieve the target (*deterrence*); (4) It should provide automated configuration management that is highly transparent to users; (5) It should scalable to a large number of services and hosts, and (6) It can be tuned to provide cost-effective configurations based on mission risk and deception cost. Most of the existing deception techniques fail to satisfy these criteria, thereby they provide very limited deception effectiveness. In specific, honeypots configuration is static and can be easily detected and blacklisted by skilled attackers. In addition, while IP mutations are effective in slowing down reconnaissance attackers proactively, they cannot fully hide the identity of hosts and service against fingerprinting attackers. Moreover, deploying a large number of honeypots is expensive and unmanageable.

Since every deception technique has its own benefit and cost, and every asset may have its own risk based on exploitability and impact, it is important to find a composition of multiple deception techniques to achieve superlinear effectiveness than applying them individually, while satisfying the budget, risk, and operation constraints. In this chapter, we present a new deception framework called CONCEAL that composes m-mutation for address anonymization, k-anonymity for fingerprint anonymization, and l-diversity for configuration diversification in order to satisfy the above objectives. Here m-mutation means that for every 1/m seconds, the addresses of the hosts are mutated, k-anonymity means that for each network host a group of $k-1$ hosts with identical fingerprints (called shadow services) are placed in the network, and l-diversity means that for every service type (e.g., web service), at least $l-1$ fake (operationally unused) services of the same type but with different vendors, versions, or patch levels are placed in the network (e.g., Apache, IIS, etc. for web services).

To develop CONCEAL as a cloud service, the clients' requests will be directed to the CONCEAL gateway which will then translated and redirected based on the site configuration to the appropriate real or proxy (shadow or diversity service) hosts. To minimize the number of CONCEAL proxies, the gateway may redirect requests destined to many IP addresses to a single physical proxy machine. Therefore, we model the problem of finding satisfiable composition of mutation, anonymization, and diversity as a constraint satisfaction problem using satisfiability modulo theories (SMT) [8] to find the appropriate values of m, k, l, and the gateway configurations that minimize the cost of physical proxies for shadow and diverse services.

Our implementation and evaluation validate the effectiveness and scalability of the CONCEAL framework. The main tool we use for CONCEAL constraints formalization is SMT. SMT is a powerful tool to solve constraint satisfaction problems arise in many diverse areas including software and hardware verification, type inference, extended static checking, test-case generation, scheduling, planning, graph problems, etc. [8]. An SMT instance is a formula in first-order logic, where some function and predicate symbols have additional interpretations.

6.2 Threat Model and Objectives

In this section, we first present threat model addressed by our approach, as well as generic defense objective against this threat model.

6.2.1 Threat Model

Reconnaissance is the primary and initial step of any advanced and persistent intrusion on enterprise networks [9]. At the network and host levels, reconnaissance refers to the process of (1) discovering and enumerating network hosts, (2) scanning these hosts at network- and transport-level (detecting OS and open ports), or application-level (identifying services names), and (3) discovering exploitable vulnerabilities for each host [15].

If attackers know the name of a host, they can obtain its IP address from DNS and using the IP to attack that host. However, adversaries usually avoid this to remain stealthy, as their frequent request to DNS can be detected. In addition, only public servers can exist in the public DNS; however, internal services will still need to be discovered through local reconnaissance. Due to these reasons, we believe that sophisticated attackers will only rely on stealthy host/service scanning to discover resources and propagate their attacks. Examples of network reconnaissance tools to scanning and fingerprinting include `Nmap` and `Nessus` [15].

As attackers need to probe hosts to scan the address space and compromise services, benign users usually reach their remote services after querying the server's IP address from the corresponding authoritative DNS. Some attackers may also

issue a reverse-DNS query to obtain the domain name from the host's address after discovering an active IP address. However, in both cases, this can be used for analyzing and detecting malicious behavior due to reconnaissance.

While random host IP mutation (RHM) [12] is effective for automated worms and naive adversaries, skilled human adversaries can still use any available information such as a host's fingerprint to re-identify that host even if its addresses have been mutated. This is because a host identity can still be recognized and traced by its fingerprint in RHM. This means a skilled attacker can fingerprint a host and use that fingerprint in future to distinguish that host. This is the major motivation to apply a more sophisticated deception approach in this chapter.

6.2.2 Defense Objectives

The defender has the following three ways to invalidate the attacker's knowledge gained from reconnaissance:

- **Address mutation** refers to the act of changing host addresses over time. With no *address mutation*, attacker can use a host IP address to identify it and avoid re-probing that host.
- **Fingerprint anonymization**: refers to the notion of hiding a host fingerprint in a pool of honeypots with identical fingerprints, thus preventing a skilled adversary to trace a host by its fingerprint. A skilled attacker can use the potentially unique fingerprint of a host to identify it later if host fingerprint anonymization is not applied.
- **Configuration diversification** refers to the act of diversifying fingerprints of network OS and services over time, thus making that any existing OSes or services will have a number of similar OSes and services in the same network.

6.3 Technical Approach

6.3.1 CONCEAL Framework Key Components

CONCEAL is a multi-strategy deception technique that combines three techniques, m-mutation, k-anonymization, and l-diversity.

Strategy 1: m-Mutation for address anonymization. For every $1/m$ seconds, CONCEAL mutates addresses of hosts. The goal of mutation is to anonymize host IP addresses over time. Note that mutating with rate m IP/Sec means an IP address is only active for $1/m$ seconds. In other words, IP addresses are anonymized every $1/m$ seconds. Mutation rate is a trade-off between benefit and cost (for address translation, updates, etc.). The limitation of mutation is that mutation is costly, and sometimes there is not enough available addresses. For a skilled attacker,

host fingerprint may serve as a quasi-identifier, which means mutation may not be enough for deception.

Strategy 2: k-Anonymity for fingerprint anonymization. For each network host, CONCEAL places a group of $k - 1$ hosts with identical fingerprints; the $k - 1$ addresses are redirected to one physical honeypot representing a group of shadow hosts.

In data privacy, quasi-identifiers are anonymized by the concept of k-anonymity.

In our domain, k-anonymity is achieved by shadow hosts. That is, for each real host, $k-1$ honeypots are included in address space with same fingerprints. A shadow host exhibits an identical fingerprint of a real host.

By satisfying k-anonymity, we anonymize identity of an individual host. However the adversary can still figure out what platforms and services are running in the real network, and the number of real hosts and unique fingerprints in the network. Thus, attackers need to make exploits for these services only.

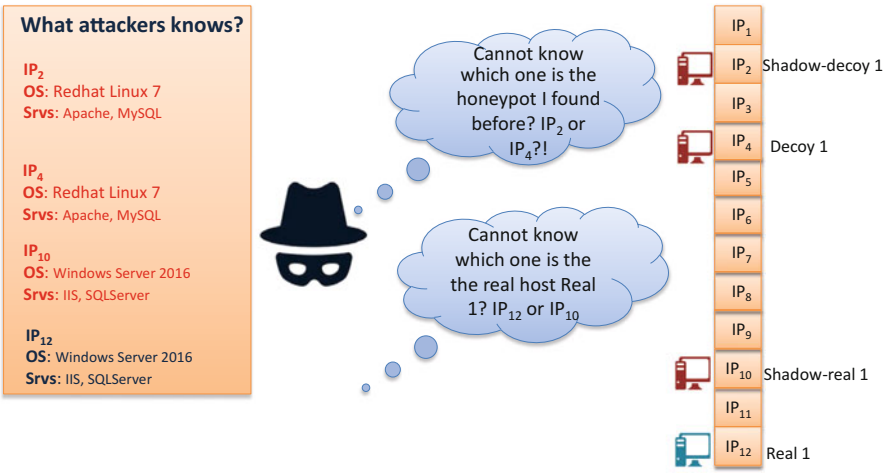

Fig. 6.1 An example of 2-anonymity

Figure 6.1 shows an example of 2-anonymity. In the system there is one real host and one proxy (decoy) host. For both of them there is one shadow host which has a different IP but identical configuration. The attacker cannot differentiate the shadow hosts from real or proxy host.

Strategy 3: l-Diversity for configuration anonymization. For every service type, CONCEAL places in the network at least $l - 1$ fake services of the same type but with different vendors, versions, or patch levels. Here service types may include OS, HTTP Server, FTP Server, etc. Services of type HTTP may include IIS 8.0, IIS 7.0, Apache, etc.

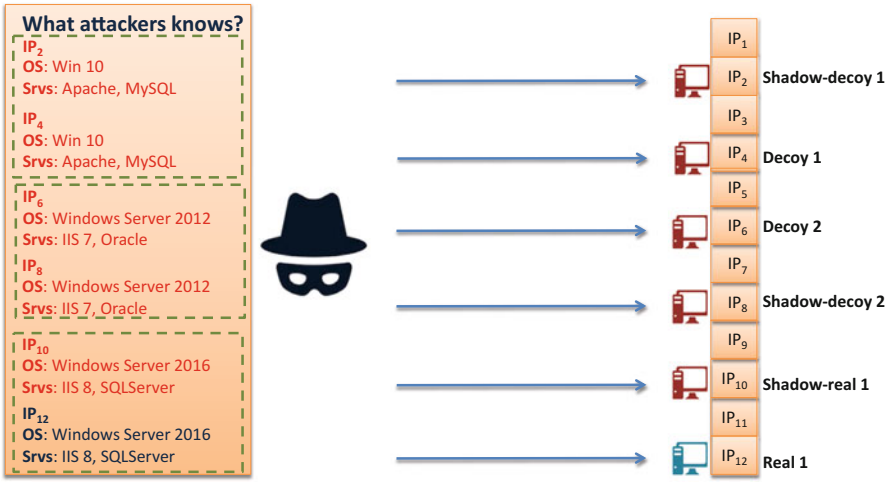

Fig. 6.2 An example of 2-anonymity, 3-diversity

Figure 6.2 shows an example of 2-anonymity, 3-diversity. In the example, the network has one real host and two proxy (decoy) hosts, while every real host and proxy host have an identical shadow host. Here we have 3-diversity for OS, HTTP service, and database service. 3-diversity for OS is achieved with Win Server 2016, Win Server 2012, and Win 10; for HTTP Services is achieved with IIS 7, IIS 8, and Apache; for database services is achieved with SQL Server, MySQL, and Oracle.

In order to find the satisfiable CONCEAL configuration, one needs to solve the following questions:

Question 1: What m, k, and l are good for a given network? How to quantify benefit and cost for a given (m, k, l) triple?

Question 2: Given m, k, and l, what are the optimal decoy services and configurations needed?

Note that these two questions are not independent. Both of them are related to budget and risk constraints. We will try to find the solution for both of them.

6.3.2 CONCEAL Effectiveness Metrics

We define the measurement for CONCEAL effectiveness (CE) as the combination of the following three metrics:

(1) Concealability measure (CM) that is the likelihood that the attacker will fail to identify the target despite his stealthiness.

(2) Detectability measure (DM) that is the likelihood that the attacker being detected despite his success.
(3) Deterrence measure (TM) that is the cost of deception on the adversary (e.g., number of probes).

CE is defined to be the combination of the three metrics:

$$CE = \alpha_1 \cdot CM + \alpha_2 \cdot DM + \alpha_3 \cdot TM \tag{6.1}$$

where α_1, α_2, and α_3 are appropriate coefficients which can be determined by different system security requirements.

As the risk of an asset increases, CE must also increase to offer cost-effective and scalable deception. In RHM, if the scanning speed (m_1) is the same as the mutation speed m, the probability that attackers fail to identify/compromise a host (CM) is the probability that the host is not identified by scanning due to mutation plus the probability that a proxy (not a real host) is hit:

$$CM = e^{-1} + (1 - e^{-1})(1 - \frac{1}{kl}) \tag{6.2}$$

The quantity e^{-1} results from invisible hosts due to mutation [11], and the quantity $(1 - \frac{1}{kl})$ results from proxy or shadow hosts due to k-anonymity and l-diversity.

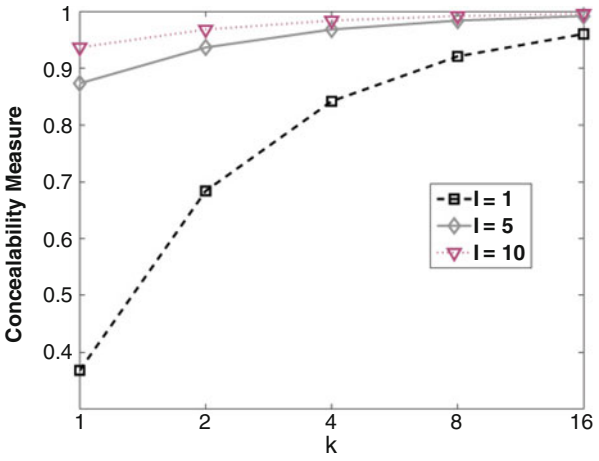

Fig. 6.3 Concealability measure

Figure 6.3 shows the concealability measure of CONCEAL with different k and l.

Deterrence (TM) is defined as the number of round of scan attempts to make the probability of hit more than the desired threshold. If one has only mutation, then the probability of hit (denoted as T) with n round of scans is known to be [11]:

$$T = 1 - (e^{-1})^n \tag{6.3}$$

For example, to guarantee $T > 0.9$ one needs to make $n > 2.3$ rounds of scanning.

In general T can be computed as:

$$T = 1 - (CM)^n \tag{6.4}$$

For example, when $k = 2$, $l = 4$, to guarantee $T > 0.9$, one needs to make $n > 22$ rounds of scanning.

Now we can calculate TM to be:

$$TM = log(1 - T_h)/log(CM) \tag{6.5}$$

where T_h is the required hitting threshold. Note that here we assume every round of scan is an independent event.

Figure 6.4 shows the T value of CONCEAL with different k, l, and T_h. We can see that for RHM the number of required scans (TM) is very small but even for CONCEAL with relative small k and l (such as 6) TM can be very high (more than 100). This means that the composition of mutation, diversity, and anonymity has much better effectiveness than applying only RHM.

Detectability (DM) is defined to be the probability that a compromise attempt is detected:

$$DM = (1 - \frac{1}{kl}) \cdot d_1 \cdot d_2 \tag{6.6}$$

Here $(1 - \frac{1}{kl})$ is probability of hitting a shadow or proxy host, d_1 is probability of detecting the attack attempt at a shadow or proxy host (IDS true positive), and d_2 is the quality/robustness of deception (which means $(1 - d_2)$ is the probability that the deception is detected by the attacker).

Figure 6.5 shows the detectability measure of CONCEAL with $d_1 = 1$, $l = 2$, and different d_2 and k.

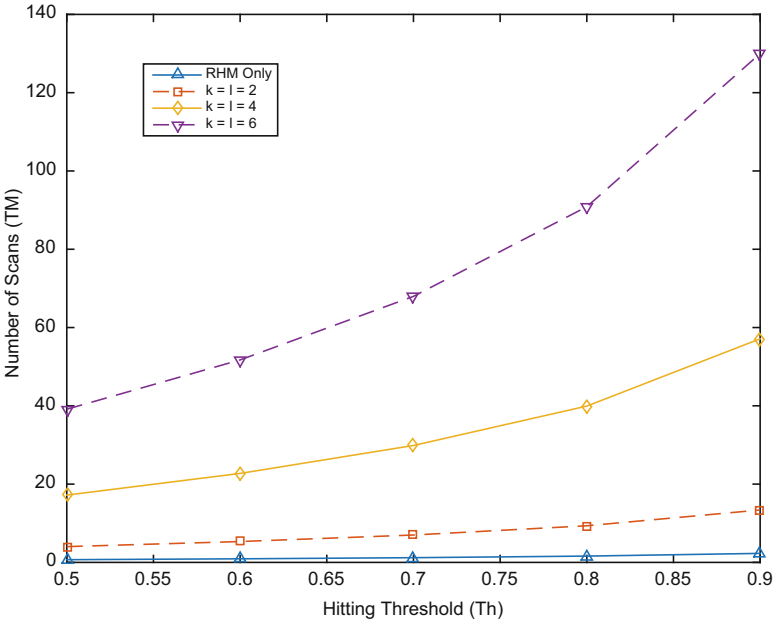

Fig. 6.4 Deterrence measure

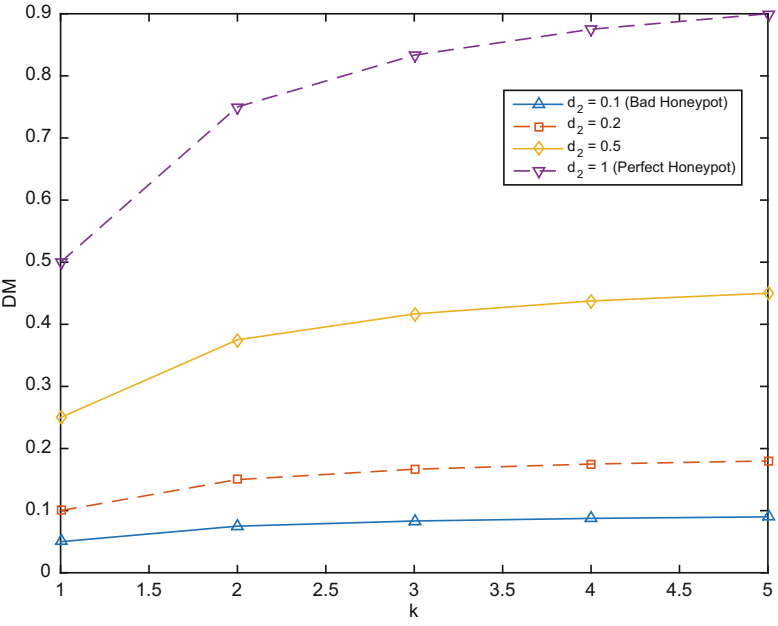

Fig. 6.5 Detectability measure

6.3.3 CONCEAL Architecture and Planner

Figure 6.6 shows the overall CONCEAL architecture. In CONCEAL architecture, clients communicate with the target network through the CONCEAL gateway of the target network, which is located in front of the honeypot cloud which includes real and proxy (shadow) hosts. The configuration of real and proxy (shadow) hosts is mutable and adjusted by the CONCEAL planner which determines the satisfiable configuration for mutation, anonymity, and diversity.

Figure 6.7 shows the framework of CONCEAL planner. Here the CONCEAL engine takes user input of available address space, existing host configuration (number of hosts, OS, service type, vulnerabilities), cost of proxies and shadows, budge limit, vulnerability impact, risk bound, and operational input, which includes feasible configuration, mission/business-related configuration (such as context and consistency constraints). These inputs are converted into CONCEAL constraints and sent to the SMT solver. If the problem is solvable, the results are returned to CONCEAL engine, which will generate the detailed host configurations. If the problem is not solvable, then one needs to relax some of the inputs such as increasing budget or decreasing risk bound.

The following is the formal definition of the CONCEAL planning problem. The input of the problem includes the configuration of n real hosts $C_1, \ldots, C_n$, where

$$C_i =< OS_i, s_{i,1}, \ldots, s_{i,k} > \tag{6.7}$$

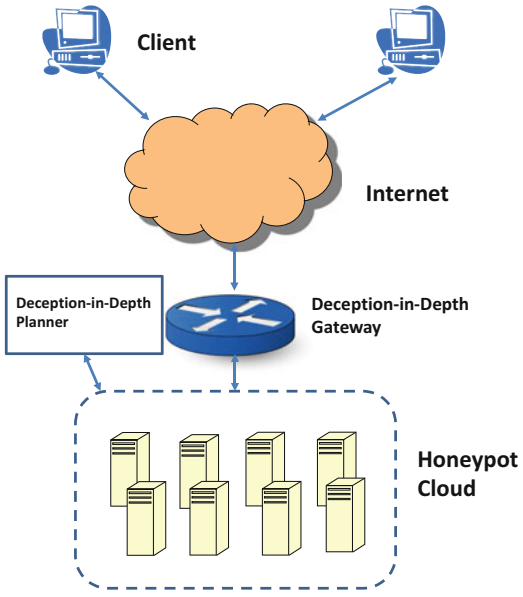

Fig. 6.6 CONCEAL architecture

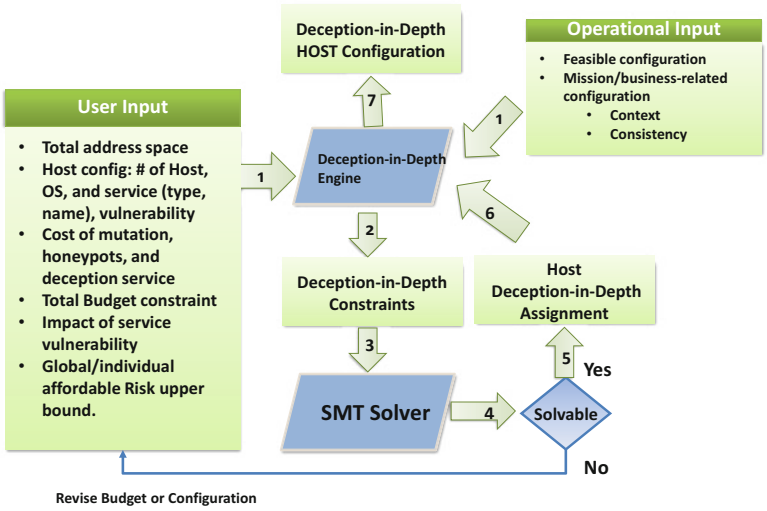

Fig. 6.7 CONCEAL planner framework

where $s_{i,j} \in T_j$, T_j is the service type, and set of vulnerabilities for every service, impact of vulnerabilities, cost of deploying mutation (C_m), proxy (C_p), diversity service (C_d), anonymity (shadow) service (C_a), budget (B), and risk bound (R). The output includes the appropriate m, k, l, number of required proxies (p), and configuration of the proxies.

6.3.4 *Formalization of Constraints*

We formalize the CONCEAL constraints as the input for SMT solver.

First, we have the *Budget constraint*

$$C_p \cdot p + C_d \cdot l + C_a \cdot k + C_m \cdot m \leq B \tag{6.8}$$

Next, we have the *Risk constraint*

$$\sum_i S_i \cdot (1 - CE) \cdot I_i \leq R \tag{6.9}$$

where S_i and I_i are the severity and impact of the vulnerability of service i, respectively.

Diversity constraint is formalized as

$$\forall i,\ dis(a_{ijk}) \geq l \tag{6.10}$$

where a_{ijk} is the binary variable that denotes that service j with type i is deployed in proxy k, and $dis(a_{ijk})$ denotes the number of distinct services of type i in all possible proxies.

We also have the *unique assignment constraint*

$$\forall i, k, \ \sum_{j} a_{ijk} \leq 1 \tag{6.11}$$

which means that every proxy can only contain one service of a fixed type.

The *context constraint* can be formalized as

$$\forall i, j, v, k, \ (b_{vk} = 1) \rightarrow (a_{ijk} \neq 1) \tag{6.12}$$

which means that OS type v should not contain service j of type i, where b_{vk} denotes that OS type v is deployed in proxy j. For example, OS Linux should not have any type of IIS services.

The *consistency constraint* can be formalized as

$$\forall k, \ (a_{i_1 j_1 k} + a_{i_2 j_2 k}) \leq 1 \tag{6.13}$$

where j_1 and j_2 are incompatible or inconsistent services to co-exist in the same proxy.

As an example, suppose we have four real hosts in the system as shown in Fig. 6.8. The cost values of C_p, C_d, C_a, C_m are 4K, 2K, 1K, 20K, respectively, and budget limit B is 28K, risk bound R is 8K, and $m_1 = 0.2$. We also have context constraint that any SQL server cannot be deployed in Linux OS.

Host	OS	Service/type	
H1	Win10	IIS/web	SQL Server/DB
H2	Win10	Azure/cloud	VM/platform
H3	Linux	Amazon/cloud	Oracle/DB
H4	Mac	Apache/web	Macincloud/platform

Fig. 6.8 Original host configuration

The SMT solver returns that $l = 3$, $k = 2$, $m = 0.2$. The solution has two proxy host H_5 with Linux OS, web service Nginx, and DB service MySQL, and H_6 with Win 10 OS, platform service Zen, and cloud service Onedrive to achieve 3-diversity (for simplicity, we only consider service diversity here). Additional shadow hosts H_7, H_8, H_9 ,H_{10}, H_{11}, H_{12} need to be created for hosts H_1, H_2, H_3, H_4, H_5, H_6 to achieve 2-anonymity. Total cost is

$$(4 * 2 + 4 * 2 + 6 * 1 + 0.2 * 20)K = 26K < B = 28K$$

Figure 6.9 shows details of the solution.

Host	OS	Service/type	
H1 (original)	Win10	IIS/web	SQL Server/DB
H2 (original)	Win10	Azure/cloud	VM/platform
H3 (original)	Linux	Amazon/cloud	Oracle/DB
H4 (original)	Mac	Apache/web	Macincloud/platform
H5 (diversity)	Linux	Nginx/web	MySQL/DB
H6 (diversity)	Win10	Zen/platform	Onedrive/cloud
H7 (anonymity)	Win10	IIS/web	SQL Server/DB
H8 (anonymity)	Win10	Azure/cloud	VM/platform
H9 (anonymity)	Linux	Amazon/cloud	Oracle/DB
H10 (anonymity)	Mac	Apache/web	Macincloud/platform
H11 (anonymity)	Linux	Nginx/web	MySQL/DB
H12 (anonymity)	Win10	Zen/platform	Onedrive/cloud

Fig. 6.9 SMT solution of CONCEAL hosts

6.4 Evaluation

We evaluate the effectiveness and scalability of the CONCEAL framework. The evaluation work is done in a machine of Intel Quad Core 3.4GHz machine with 16G memory. We use the Yices SMT solver [2] to find the solutions to the constraints.

CONCEAL Effectiveness As we discussed, CE is the metric to measure the effectiveness of CONCEAL. Figures 6.10, 6.11, and 6.12 show the concealability measurement (CM), detectability measurement (DM), and deterrence measurement (TM) values for different number of services and budget bound B, respectively. For the TM value, the threshold T_h is set to be 0.9. For the DM value, d_1 and d_2 are set to be 0.95. We can see that all the CM, DM, and TM values increase with high values of B and high number of services. However the increase of B has less significant impact when the number of services reaches some point.

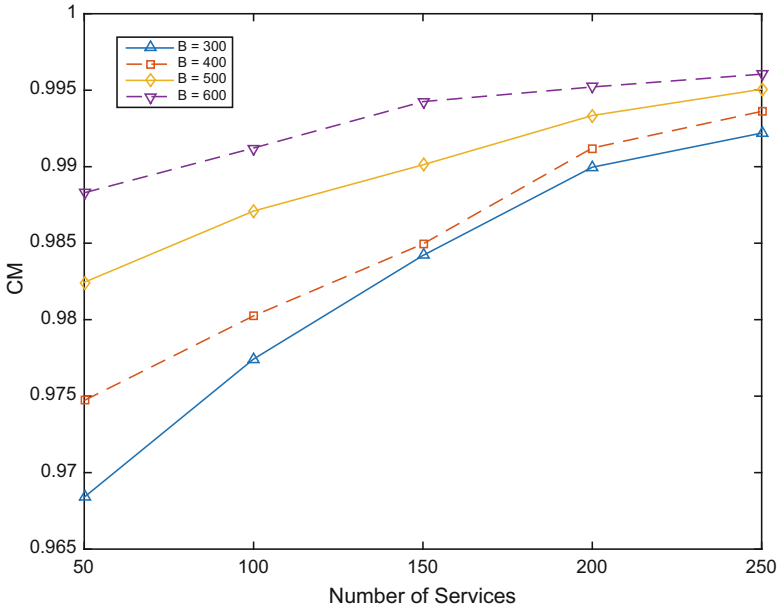

Fig. 6.10 Concealability measurement

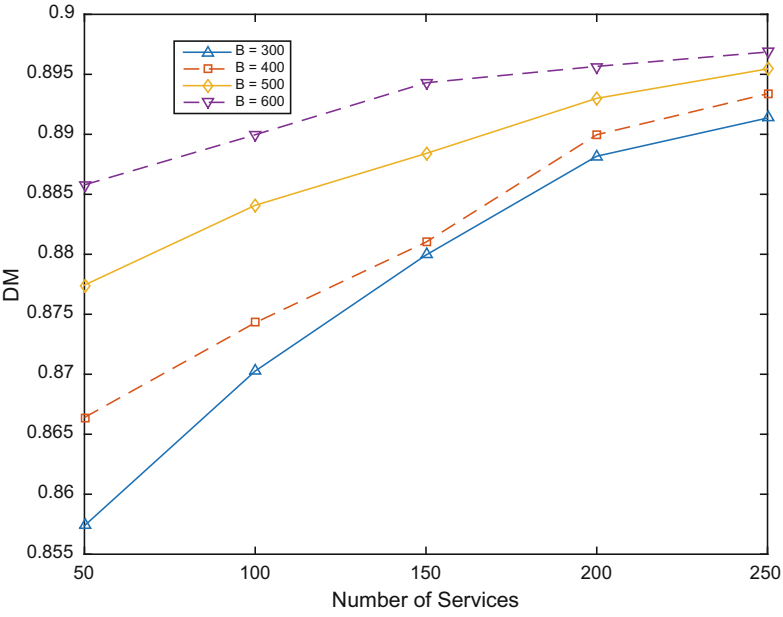

Fig. 6.11 Detectability measurement

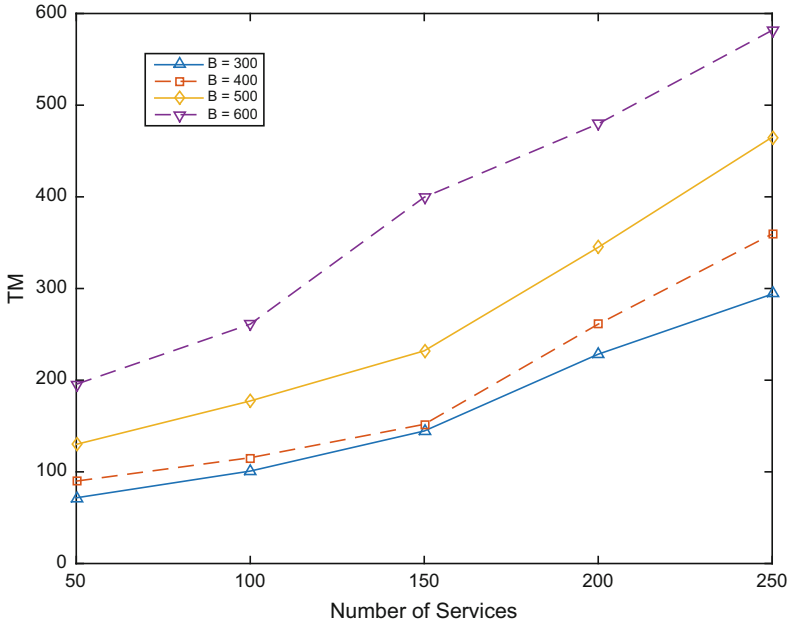

Fig. 6.12 Deterrence measurement

Saving of Proxies for Different *B* The CONCEAL framework can save the number of actual used proxies since multiple services can share the same proxy as long as they are compatible. Figure 6.13 shows the percentage of saved proxies with different B. We can see that as B decreases the percentage of saved proxies will increase. This is because smaller B will cause the SMT solver to find smaller value of p to reduce cost. However, smaller B may make the problem unsolvable and increase the solving time.

Saving of Proxies for Different Number of Correlation Figure 6.14 shows the percentage of saved proxies with different number of correlations. The X-axis *cor* in the figure is the percentage of consistent services where consistent means the services cannot exist in the same host. We can see that as *cor* increases the percentage of saved proxies will decrease. This is because higher *cor* will cause the SMT solver to use higher value of p to satisfy the constraints.

Solving Time for Different *B* Figure 6.15 shows the SMT solving time for different B. We can see that as B increases the solving time will decrease. This is because higher B means more resources and it will be easier for the SMT solver to find the solution to satisfy the constraints.

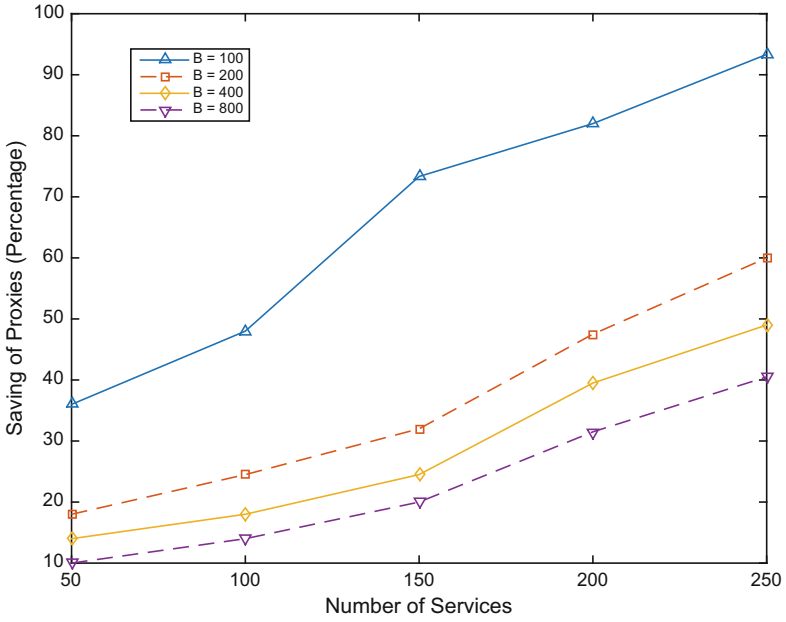

Fig. 6.13 Save of proxies with different B

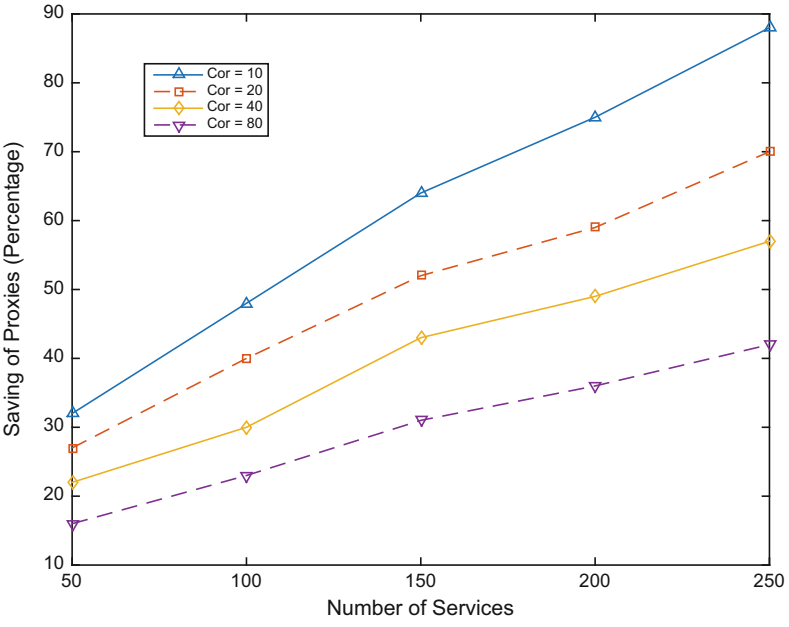

Fig. 6.14 Save of proxies with different correlation

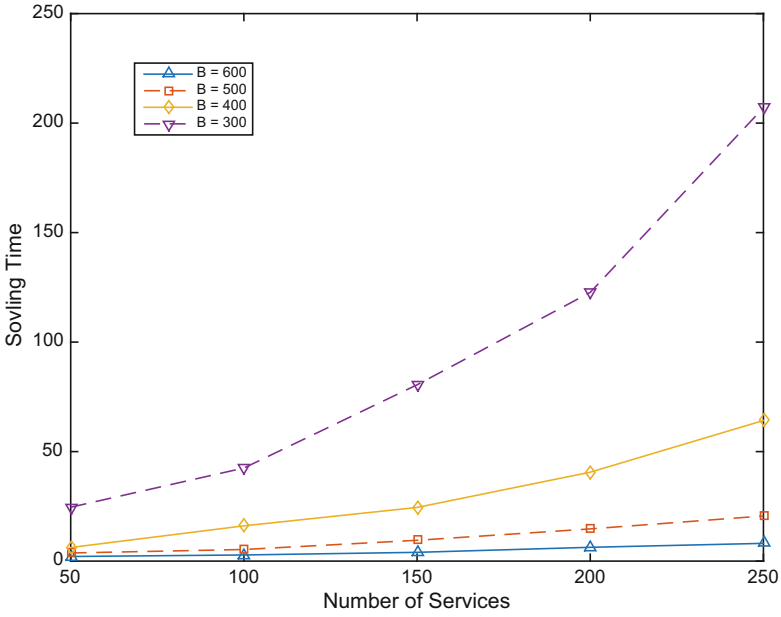

Fig. 6.15 Solving time with different B

Solving Time for Different Number of Correlation Figure 6.16 shows the SMT solving time for different number of correlations *cor*. We can see that as *cor* increases the solving time will increase. This is because higher *cor* means more constraints for the SMT solver to find the satisfiable solution.

6.5 Implementation

We implemented CONCEAL as a deception of service that can be accessible through a web interface. The CONCEAL implementation mainly consists of three different components: (1) The web interface, (2) SMT Solver, and (3) ActiveSDN; shown in Fig. 6.17. Each of these three services running in different process can communicate with each other through rest APIs. Although the API descriptions are out of the scope of this chapter, however, interested reader can look into our implementations [1].

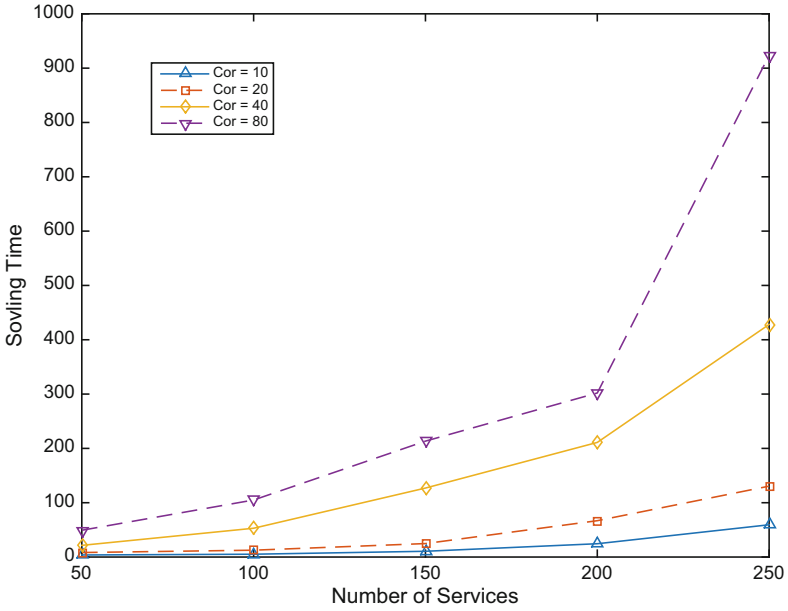

Fig. 6.16 Solving time with different correlation

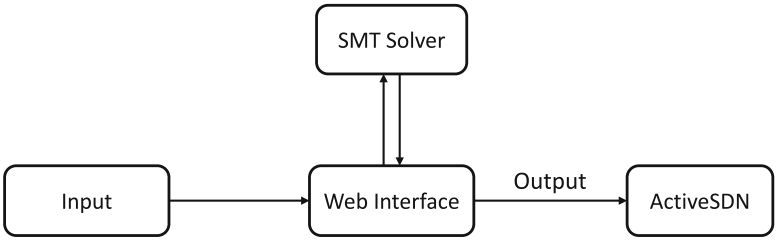

Fig. 6.17 CONCEAL implementation

6.5.1 Web Interface

The web interface is divided into three sections: (1) costs, (2) services, and (3) hosts. The user will provide all of the costs like mutation, anonymity, diversity, and proxy costs along with total risk and budget costs. For the *services*, user can create as many services as she wants and those services will be added in the *hosts* section. After submitting, the constraints will be solved by the SMT solver which is running in a different process and if there exist any solution it will be shown as output, else user need to relax the constrains and submit again. Figure 6.18 shows the web interface.

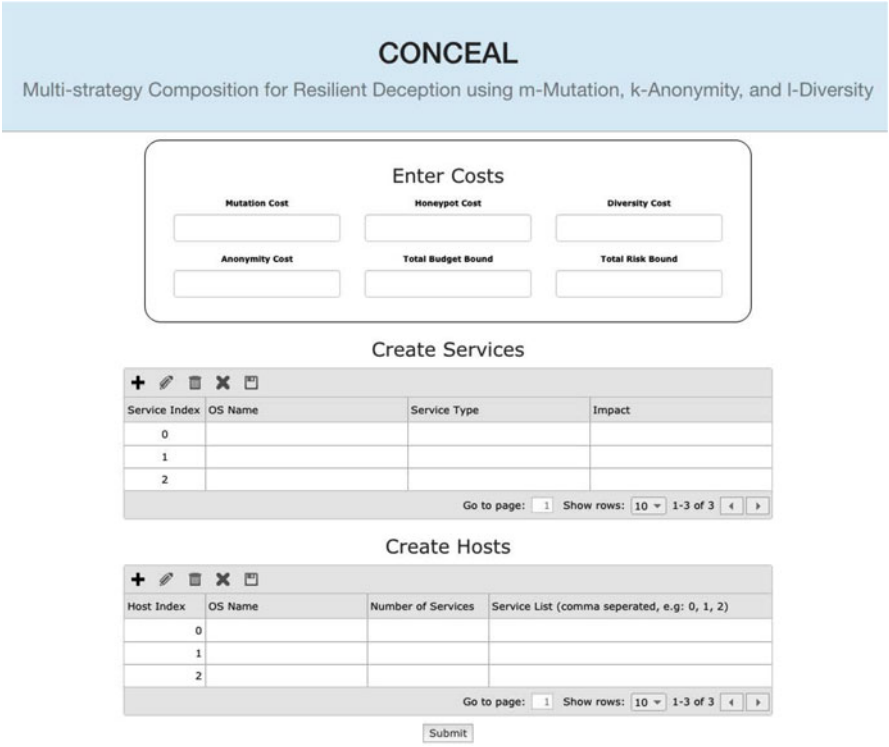

Fig. 6.18 CONCEAL web interface

6.5.2 SMT Solver

We implement our constraint satisfaction solver using Yices [2] in C++. All the constraints mentioned in the Sect. 6.3.3 are implemented in our solver. The solver is running in a different process that can be accessible through a web interface.

6.5.3 ActiveSDN

ActiveSDN is a decision-making OpenDaylight controller [16] where we implement all the services required for CONCEAL, like mutation, anonymization, and shadowing. After submitting the user inputs through web interface, the SMT solver will generate the output configuration, which will be used in ActiveSDN to deploy conceal in a *Mininet* network [21]. For conceal, we have three services implemented in ActiveSDN: *IP mutation*, *host anonymization*, *service diversity*.

Figure 6.19, 6.20, and 6.21 show the IP mutation, host anonymization, and service diversity functions in ActiveSDN, respectively. For IP mutation, *aIP* indicates the list of available IP addresses that will be mutated using *vIPs* mentioned in *how*. The mutation parameter *when* has an attribute *x* which indicates the mutation cycle (mutate *aIPs* after *x* seconds). The host anonymity function takes the IP address of a host as an argument *hostIP* and its corresponding *fingerprints*. For the *hostIP*, *k* new host with identical fingerprints will be created. The service diversity function architecture is similar to host anonymity function.

Defense Actions	Parameters	Descriptions
ipMutate()	<**aIP**>	List of available IPs, e.g. <192.168.10.20/32, 192.168.10.21/32, 192.168.55.99/32,...> etc.
	when (*m*)	**-1**: Deactivate IP mutation.
		0: One time mutation.
		x: Time based mutation. Mutate IPs after *x* seconds. Hence, x = 1/m.
		other-mutation: For future use.
	how	**specific vIP**: User defined virtual IP (vIPs) which can be selected form <aIP>, e.g. 192.168.10.20/32
		randomFunction(): A function which will provide random vIPs in a specific time window *x*.
Example	**ipMutate**(10.0.0.2/32, 5, uniform())	

Fig. 6.19 IP mutation function

Defense Actions	Parameters	Descriptions
hostAnonymity()	**hostIP**	IP address, e.g. 192.168.10.20/32
	fingerprints	**OS**: Operating system running in the **hostIP**
		<openPorts>: List of open ports on each host.
		<services>: List of services running on each host.
	k	The number of host will be created for anonymization.
Example	**hostAnonymity**(10.0.0.2/32, fingerprints(), 5)	

Fig. 6.20 Host anonymization function

Defense Actions	Parameters	Descriptions
serviceDiversity()	**srcIP**	IP address of the host which service will be diverse, e.g. 192.168.10.20/32.
	services	Services running on **srcIP**. For example HTTP: IIS 8.0, IIS 7.0, Apache server etc.
	l	The number of services will be created for diversity.
Example	**serviceDiversity**(10.0.0.3/32, services(), 6)	

Fig. 6.21 Service diversity function

6.6 Related Works

Proactive techniques for disrupting reconnaissance could be broadly divided into two categories: MTD-based and deception-based approaches. Moving target defense relies on randomizing static system parameters, to invalidate attacker's reconnaissance information.

In this direction, several approaches have been proposed [11–13] for mutation of IP addresses, especially over time. NASR [6] relies on DHCP to randomize IP addresses over time. RHM [12] and OF-RHM [11] propose using DNS for address randomization without changing the actual IP address of network hosts, in order to prevent TCP/UDP sessions from being broken. RHM [12] identifies additional need for mutating MAC addresses to defeat LAN-level reconnaissance, and mutating domain names to counter reverse-DNS queries.

The notion of mutable networks as a frequently randomized changing of network addresses and responses was initially proposed in [3]. The idea was later extended as part of the MUTE network which implemented the moving target through random address hopping and random fingerprinting [4].

Existing IP mutation techniques include dynamic network address translation (DYNAT) [14, 17, 18], random host IP mutation (RHM) [12], OpenFlow random host IP mutation (OF-RHM) [11], etc.

DYNAT is a technique developed to dynamically reassign IP addresses to confuse any would-be adversaries sniffing the network. They obfuscate the host identity information (IP and Port) in TCP/IP packet headers by translating the identity information with preestablished keys. Both BBN [14] and Sandia [17, 18] have done research work on DYNAT. BBN ran series of red-team tests to test the effectiveness of DYNAT, while Sandia's DYNAT report [17, 18] examines many of the practical issues for DYNAT deployment.

RHM [12] and OF-RHM [11] use DNS for address randomization without changing the actual IP address of network hosts and random mutate host IPs while satisfying mission, operational, and cost constraints.

While these works are effective against automated scanners and worms, in our evaluation we show that they have limited effectiveness against human attackers.

On the other hand, *deception-based technologies*, trying to attract attackers to decoy hosts such as honeypots, can mislead the attacker and prolong reconnaissance and attack progress. The work in [19] shows that honeypots are highly effective in slowing down worm breakouts, and decreasing amount of attacks on production hosts.

Dynamic honeypots were first proposed by Budiarto [7]. Dynamic honeypots discover production systems on a network and use this information to create honeypots that are similar to production hosts and mix in the surrounding network. *Shadow* honeypots were proposed by Anagnostakis et al. [5], which is an identical copy of production hosts, and used as an strategy to deploy honeypots in a production environment.

The work in [20] correctly identifies the need of strengthening address mutation with decoy services against sophisticated adversaries, but their approach neglects important theoretical aspects of anonymization such as mutation and anonymization of host/honeypot fingerprints. Moreover, efficacy of their proposed model is not tested against real human attackers.

None of the above approaches recognizes the need and synergy of combining the multi-dimensional proactive defense for defeating adversarial reconnaissance performed by skilled adversaries.

Our preliminary work of proactive multi-dimensional deception technique in [10] tries defeating reconnaissance by skilled attackers, through mutating configuration (name, addresses, fingerprint) of network hosts, while populating address space with moving honeypots with strategically designed but randomly changing configurations. However it does not define the clear metric for multi-dimensional deception effectiveness and provide no synthesis to find the satisfiable combination of the multi-dimensional techniques.

6.7 Conclusion

In this chapter we present a new deception as a service paradigm called CONCEAL that combines m-mutation for address anonymization, k-anonymity for fingerprint anonymization, and l-diversity for configuration diversification. We define three CONCEAL metrics such as concealability, deterrence, and detectability to measure the effectiveness of CONCEAL, and develop the framework to automatically generate the optimal CONCEAL configuration that satisfies related budget, risk, and operation constraints. We evaluated the effectiveness and scalability of the CONCEAL framework, and implemented CONCEAL as a deception of service based on ActiveSDN. Our implementation and evaluation validates the effectiveness and scalability of the CONCEAL framework. The CONCEAL framework can solve problem instances up to 250 services and the save of proxies can reach as high as 90%.

6.8 Exercise Problems

1. An SMT exercise—Download Z3 solver and create the appropriate SMT constraints to solve the following constraint satisfaction problem:

   Three connected chairs in a row (C1, C2, C3). We need to place aunt (180 lb), sister (120 lb), and father (200 lb) such that

   (a) Aunt doesn't want to sit near father,
   (b) Aunt doesn't want to sit in the left chair,
   (c) Sister doesn't want to sit to the right of father,
   (d) Any two adjacent chairs should not carry more than 350 lb.

2. Given CONCEAL parameters $(m, k, l) = (0.1, 3, 4)$, and $d_1 = d_2 = 0.8$, $T_h = 0.9$, $m_1 = 0.1$, calculate the following CONCEAL effectiveness metrics (1) CM; (2) DM; (3) TM.
3. In the example shown in Fig. 6.3, we can see that if k or l is greater than some threshold values, the CM value only increases marginally. Given $l = 5$, find the threshold for k such that doubling the value of k will only improve CM by less than 10%.
4. In the example shown in Fig. 6.8, is there any other satisfiable values of (m, k, l) for CONCEAL planning problem (Hint: Use SMT to encode CONCEAL planning constraints and add an additional constraint to exclude the obtained solution in the chapter)?
5. In the example shown in Fig. 6.8, we set $\alpha_1 = 1$, and $\alpha_2 = \alpha_3 = 0$, this means that we only consider CM for the effectiveness. If we set $\alpha_1 = 1$, $\alpha_2 = 0.01$, $\alpha_3 = 0$ and $T_h = 0.9$, and other values remain the same, is the problem be still solvable?

Acknowledgements This research was supported in part by United States Army Research Office under contract number W911NF1510361. Any opinions, findings, conclusions or recommendations stated in this material are those of the authors and do not necessarily reflect the views of the funding sources.

References

1. (2018) https://github.com/rakeb
2. (2018) Yices: An SMT solver. http://yices.csl.sri.com/
3. Al-Shaer E (2009) Mutable networks, National cyber leap year summit 2009 participants ideas report. Tech. rep., Networking and Information Technology Research and Development (NTIRD)
4. Al-Shaer E (2011) Toward network configuration randomization for moving target defense. In: Jajodia S, Ghosh AK, Swarup V, Wang C, Wang XS (eds) Moving Target Defense, Advances in Information Security, vol 54, Springer New York, pp 153–159
5. Anagnostakis KG, Sidiroglou S, Akritidis P, Xinidis K, Markatos EP, Keromytis AD (2005) Detecting targeted attacks using shadow honeypots. In: Usenix Security
6. Antonatos S, Akritidis P, Markatos EP, Anagnostakis KG (2007) Defending against hitlist worms using network address space randomization. Comput Netw 51(12):3471–3490, DOI http://dx.doi.org/10.1016/j.comnet.2007.02.006
7. Budiarto R, Samsudin A, Heong CW, Noori S (2004) Honeypots: why we need a dynamics honeypots? In: Information and Communication Technologies: From Theory to Applications, 2004. Proceedings. 2004 International Conference on, IEEE, pp 565–566
8. Davis M, Putnam H (1960) A computing procedure for quantification theory. J ACM 7:201–215, DOI http://doi.acm.org/10.1145/321033.321034, URL http://doi.acm.org/10.1145/321033.321034
9. Hutchins EM, Cloppert MJ, Amin RM (2011) Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains. Leading Issues in Information Warfare & Security Research 1:80

10. Jafarian H, Niakanlahiji A, Al-Shaer E, Duan Q (2016) Multi-dimensional host identity anonymization for defeating skilled attackers. In: Proceedings of the 2016 ACM Workshop on Moving Target Defense, ACM, New York, NY, USA, MTD '16, pp 47–58, DOI 10.1145/2995272.2995278, URL http://doi.acm.org/10.1145/2995272.2995278
11. Jafarian JH, Al-Shaer E, Duan Q (2012) Openflow random host mutation: transparent moving target defense using software defined networking. In: Proceedings of the first workshop on Hot topics in software defined networks, ACM, pp 127–132
12. Jafarian JH, Al-Shaer E, Duan Q (2015) An effective address mutation approach for disrupting reconnaissance attacks. IEEE Transactions on Information Forensics and Security 10(12):2562–2577
13. Jafarian JHH, Al-Shaer E, Duan Q (2014) Spatio-temporal address mutation for proactive cyber agility against sophisticated attackers. In: Proceedings of the First ACM Workshop on Moving Target Defense, ACM, MTD '14, pp 69–78
14. Kewley D, Fink R, Lowry J, Dean M (2001) Dynamic approaches to thwart adversary intelligence gathering. DARPA Information Survivability Conference and Exposition 1:0176, DI http://doi.ieeecomputersociety.org/10.1109/DISCEX.2001.932214
15. McClure S, Scambray J, Kurtz G, Kurtz (2005) Hacking exposed: network security secrets and solutions, vol 6. McGraw-Hill/Osborne New York
16. Medved J, Varga R, Tkacik A, Gray K (2014) Opendaylight: Towards a model-driven SDN controller architecture. In: World of Wireless, Mobile and Multimedia Networks (WoWMoM), 2014 IEEE 15th International Symposium on a, IEEE, pp 1–6
17. Michalski J, Price C, Stanton E, Lee E, Seah CK, TAN YH, Pheng C (2002) Final report for the network security mechanisms utilizing network address translation LDRD project. technical report sand2002-3613. Tech. rep., Sandia National Laboratories
18. Michalski JT (2006) Network security mechanisms utilising network address translation. International Journal of Critical Infrastructures 2(1):10–49
19. Rowe NC, Custy EJ, Duong BT (2007) Defending cyberspace with fake honeypots. Journal of Computers 2(2):25–36
20. Sun J, Sun K (2016) DESIR: Decoy-enhanced seamless IP randomization. In: INFOCOM 2016
21. Team M (2012) Mininet: An instant virtual network on your laptop (or other pc)

Chapter 7
NetShifter: A Comprehensive Multi-Dimensional Network Obfuscation and Deception Solution

Gahng-Seop Ahn, Kyung Joon Kwak, Alexey Bogaevskiy, Jason Li, Gregory Briskin, and Robert Vaeth

Abstract Adaptive defense is a cyber defense strategy in which a set of system configurations are dynamically changed to increase uncertainty and complexity for adversaries that try to discover and exploit vulnerabilities. To improve cyber agility of networks, the NetShifter performs multi-dimensional network-level adaptive defense in full scale beyond physical constraints of the networks by adopting the software-defined network (SDN).

Keywords Adaptive network defense · Software defined network · Network obfuscation

7.1 Introduction

Conventional cyber defense endeavors to protect systems that operate in relatively static configurations. Such static configurations bring great advantage for adversaries in that they have time to discover and exploit vulnerabilities. Attackers are continually changing their tactics and seeking out new vulnerabilities. Conventional cyber defense strategies that present security barrier for static systems are not effective enough. Network obfuscation and deception is a cyber defense strategy in which a set of system configurations is dynamically changing to increase uncertainty and complexity for adversaries seeking to discover and exploit vulnerabilities.

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

G.-S. Ahn (✉) · K. J. Kwak · A. Bogaevskiy · J. Li · G. Briskin
Intelligent Automation Inc., Rockville, MD, USA
e-mail: gahn@i-a-i.com

R. Vaeth
Air Force Research Laboratory, Rome, NY, USA

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_7

Although network obfuscation and deception is a promising and appealing idea, it is still in the early stages of maturation process. Specifically, existing network obfuscation and deception solutions for enterprise networks are not sufficient in thwarting adversary attempt to perform reconnaissance, launch an attack, and exfiltrate information. The shortcomings are mostly due to physical constraints of the network (i.e., effective within a given subnet/enclave). The set of configurations to be played are constrained by static physical wired connections and physical locations. Also, IP addresses of hosts and services are tied within an enclave unless additional measures are applied to route the packets. In addition, the time scale of dynamics in network obfuscation and deception is still an issue to investigate and explore the trade-off design space. In order to be a true game changer in enterprise network security solution, a comprehensive and highly dynamic obfuscation and deception techniques are needed.

NetShifter is a highly dynamic network obfuscation and deception solution that overcomes limitations of existing solutions. Figure 7.1 illustrates an operational view of NetShifter in a representative enterprise network infrastructure. NetShifter is a multi-dimensional network obfuscation and deception solution that mutates and randomizes multiple aspects of the network configurations simultaneously, each of which adds dimensions of complexity that cyberattacker must analyze. The solution employs comprehensive network and host level SDN-based network obfuscation and deception techniques and adds additional layers of network obfuscation by leveraging state-of-the-art virtualization techniques and DNS deception methods.

Specifically, NetShifter performs network obfuscation and deception techniques including: (1) packet header randomization, (2) host/server IP address mutation, (3) DNS/IP address binding mutation, (4) route/flow mutation, (5) topology mutation, and (6) service randomization.

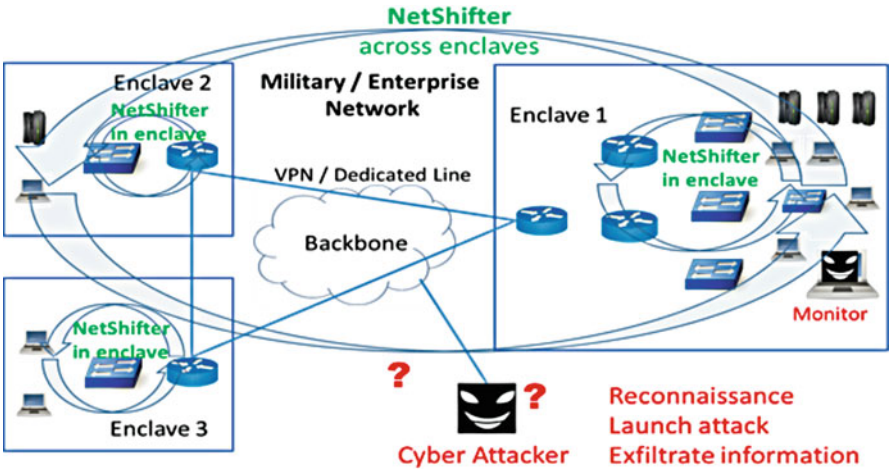

Fig. 7.1 Operational view of NetShifter solution

7.2 Summary of Existing Network Obfuscation and Deception Solutions

A summary of the existing network obfuscation and deception research efforts, which employ software-defined networking (SDN) to provide network obfuscation and deception capability, is presented in Sect. 7.2.1. SDN allows for abstraction of the data plane and control plane. Network operators manage network capabilities and services through the centralized control point. Network obfuscation and deception introduces a dynamic environment in order to delay, deter, or prevent attacks on a system. SDN can play a key role to provide attack surface obfuscation.

A summary of existing network obfuscation and deception solutions that are not leveraging the SDN is presented in Sect. 7.2.2. Some of the features in those solutions can be re-developed using SDN.

It is worth noting that the existing network obfuscation and deception solutions in the literature are limited by physical constraints of the network (i.e., effective within a given subnet/enclave) or have not been tested beyond the physical constraints. The set of configurations to be played in network obfuscation and deception are constrained by static physical wired connections and physical locations. For instance, the solutions presented in [1, 12, 13] dynamically change routes between source and destination within available routes in a static physical topology. Hence, attacker may be able to discover the static physical topology by combining all variances of routes. Furthermore, the number of disjoint paths that can be used for random route mutation is limited in wired network. IP address hopping solutions [2–6] also face the similar challenge due to limited IP address spaces assigned to a subnet or an enclave. The IP addresses of hosts and services are tied within an enclave unless additional measures are applied to route the packets. In addition, the time scale of dynamics in network obfuscation and deception is still an issue to investigate and explore the trade-off design space. For example, recent software defined network (SDN) technology-based network obfuscation and deception solutions [4] randomize IP addresses per flow. However, long-lived IP flows such as secure socket layer (SSL) connections will hinder network configuration from dynamically changing, which gives an attacker sufficient time to analyze network behavior and topology.

7.2.1 SDN-Based Solutions

The OpenFlow random host mutation (OF-RHM) [4] uses solver and equation to randomize source and destination IP addresses. It keeps real IP address of a host unchanged and uses virtual IP addresses for routing purpose. Even with a limited and fragmented unused address space, each host will mutate with its required rate such that no IP address is reused (assigned to any host more than once) for a reasonably long time. This prevents, deters, and mitigates scanning-based attacks.

The authors of the OF-RHM also proposed random route mutation [1] that enables changing randomly the route of the multiple flows in a network simultaneously to defend against reconnaissance, eavesdrop, and DoS attacks, while preserving end-to-end QoS properties. This solution can be deployed as add-on functionality to their SDN-based solution.

The SDN Shuffle [5] uses host-based address manipulation and only introduces the minimal time to perform NAT translations on packets after the connection was established. The computational overhead occurs at reasonable rate at the network layer. A malicious application running at the user-level on the client or server could learn the synthetic address of the communicating counterpart and attempt to establish a concurrent connection in order to launch an attack. The SDN Shuffle does not require mapping state in network switches or NAT devices thus eliminating potential scalability concerns.

The SDN-based solutions for moving target defense network protection [6] enforce network obfuscation in response to a TCP port scan. Steps can be implemented in SDN in order to enforce network obfuscation in response to a TCP port scan. The algorithm will reveal extra open and closed ports for the attacker but will also generate obfuscated port traffic (ACK, PUSH-ACK) against more advanced reconnaissance tests. SDN programmability can offload the packet generation effort to separate distributed or hosted programs without overloading the network infrastructure. Traffic to a destination that can be blocked according to a filtering policy can be silently dropped and SDN utilities can generate varying responses that will confuse the attacker.

7.2.2 Non-SDN-Based Solutions

Dynamic network address translation (DYNAT) [9] is a protocol obfuscation technique to protect the network traffic. DYNAT can be deployed to workstations, servers, routers, and gateways. DYNAT randomizes parts of a network packet header. This randomization can make it more difficult for cyberattackers to determine what is happening on a network, who is communicating with whom, what services are being used, and where the important systems are located depending on how the technique is deployed.

Randomized intrusion-tolerant asynchronous services (RITAS) [10] is a technique that builds a set of fault-tolerant consensus-based protocols on top of TCP and the IPSec protocol. TCP provides a reliable channel and IPSec provides integrity to the data being transmitted. This technique is to be used between a set of processes. The processes are assumed to be fully connected and each pair of processes shares a secret key. Using randomization, this technique implements a dynamic network that is capable of guaranteed delivery given limited number of malicious nodes.

Network address space randomization (NASR) [11] is a technique that involves changing the IP address of systems more frequently. The authors modified a

dynamic host configuration protocol (DHCP) server to have short IP address leases and to force an IP address change when a lease expires. The side effect of changing these IP addresses constantly is that persistent or active connections would be dropped during the address change. Domain name system (DNS) servers can be used for outside access to servers and services to mitigate the impact a constantly changing IP address would have on end users.

A mutable network (MUTE) [12] is a technique that involves changing IP addresses, port numbers, and routes to destinations inside of a network. This technique is proposed to be implemented as a sort of virtual overlay to the existing network so the original IP address and information on the systems never changes. The packets can be changed based on rules distributed among routing entities. It can change the source and destination IP address as well as source and destination ports.

Dynamic backbone (DynaBone) [13] is a technique that involves creating multiple inner virtual overlay networks inside of a larger outer virtual overlay network. Each of the inner networks can be using a different networking and routing protocol or hosting a different service to increase diversity among them. The entry points to these internal overlays have a collection of sensors that monitor performance and possible attack traffic. If an internal overlay is detected to be under attack or is suffering performance issues, traffic can be routed through different overlays (dynamic network aspect of DynaBone). This technique is built on top of XBone that is a dynamic network overlay technique that allows multiple simultaneous virtual overlays to co-exist.

Self-shielding dynamic network architecture (SDNA) [2] is a hypervisor-based technique that rewrites packets entering and exiting the OS to prevent the OS from observing real addresses in the network. By inserting a hypervisor within each network node, SDNA makes the network's appearance dynamic to observers while at the same time retaining the necessary semantics for transparency to legitimate users. SDNA is comprised of a combination of existing networking techniques, hypervisor technology, authentication, and IPv6.

7.3 NetShifter System Architecture

The NetShifter system architecture is illustrated in Fig. 7.2. NetShifter is developed mainly based upon software defined networks (SDN) architecture and OpenFlow standard implementation. NetShifter is a hybrid solution that uses SDN devices residing with legacy network devices with very minimal configuration changes. The hybrid solution ensures relatively easy acceptance to existing enterprise network operators and administrators. Each OpenFlow enabled routers/switches are controlled by an SDN controller using the OpenFlow standard. Distributed SDN controllers can also be used in the NetShifter architecture. In Fig. 7.2, we simplify the architecture to use a single SDN controller as an example. For large-scale enterprise networks, distributed SDN controllers can be used.

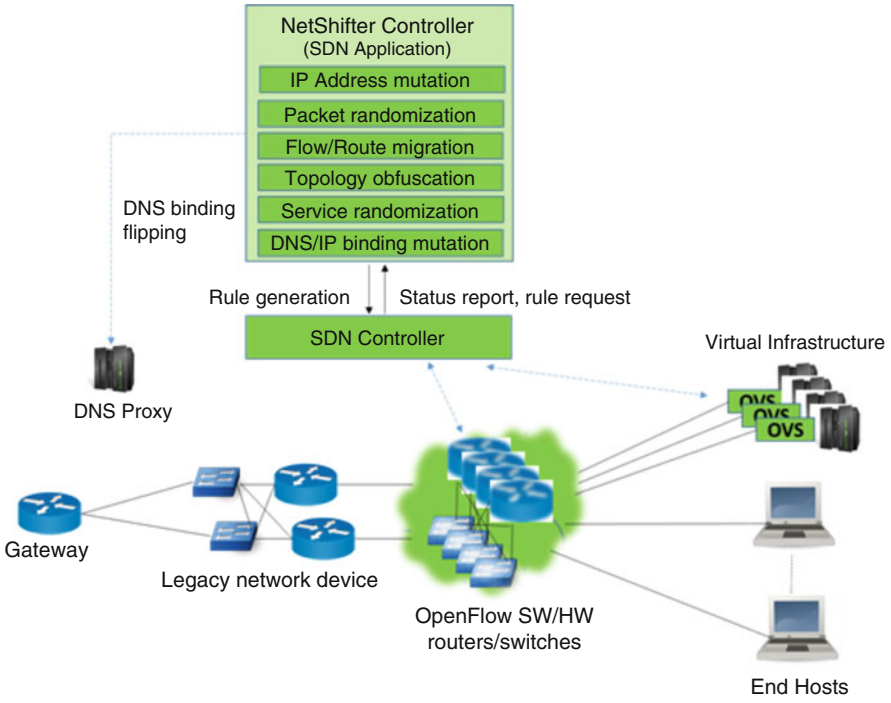

Fig. 7.2 NetShifter system architecture

The NetShifter controller is developed as a holistic and comprehensive SDN application that interacts with an SDN controller to perform the network obfuscation and deception techniques listed in the NetShifter controller box in Fig. 7.2. The NetShifter controller also dynamically reconfigures a DNS proxy to provide fast flux capability using well-defined APIs/standard protocol.

NetShifter can use both software and hardware OpenFlow implementations. We have tested NetShifter with hardware OpenFlow implementations such as HP switches and LinkSys routers as well as software OpenFlow implementation such as OpenVSwitch (OVS). OVS is a production quality, multilayer virtual switch. NetShifter can use OVS operating in virtual infrastructure that runs many enterprise network applications and services as shown in Fig. 7.2. Due to its open and vendor agnostic design approach, NetShifter is dynamically programmable and relatively transparent to legacy networks.

NetShifter leverages state-of-the-art virtualization solution to migrate an application or service from one physical location to another while the application or service is running (i.e., live application and service migration). While the application or service is migrating, underlying network devices will adapt to forward user traffic to right location, which is critical to be user transparent.

NetShifter augments a DNS proxy to create a DNS network obfuscation and deception cache. The cache is updated to change IP addresses dynamically for a

given service domain name and redirects service request to the designated server when it is migrated to a new location or its IP address is mutated.

7.4 NetShifter Network Obfuscation and Deception Techniques

NetShifter is a multi-dimensional network obfuscation and deception solution that applies multiple network obfuscation and deception techniques simultaneously. The network obfuscation and deception techniques that are performed in NetShifter solution are presented in this section.

7.4.1 IP Address Mutation Across Enclaves

IP address is a primary element of network obfuscation and deception space because static IP addresses assigned to a host/server offer great advantages for cyberattackers to discover an attack surface. Many IP address mutation (i.e., hopping or randomization) techniques [4–8] have been proposed in the literature. These techniques are limited, due to the physical constraints of military/enterprise network. DHCP-based [7] or NAT-based [8] network obfuscation and deception techniques are inevitably limited within a subnet. Other techniques [4–6] have not been tested to randomize IP address using a pool of IP addresses assigned for other enclaves nor provide any considerations or additional measures to enable it. Among these approaches, the technique presented in [4] that uses advanced SDN technology seems to be well-defined and practical, although it is still limited.

NetShifter also uses advanced SDN technology to mutate IP address of a host/server. However, NetShifter overcomes the limitation of existing IP address mutation solutions and uses the pool of IP addresses that is not tied to an enclave. The connectivity of the network with the mutated IP addresses will be maintained by using SDN technology and GRE/VPN tunneling.

In NetShifter solution, the IP address of a node (i.e., host or server) is not changed directly at the host/server. Instead, the IP address of packets to/from the node is mutated by an OpenFlow enabled switch (OF) that connects the node as shown in Fig. 7.3. OpenFlow enabled devices can dynamically update TCP/IP/ICMP header fields (e.g., source/destination IP, source/destination port, TOS, VLAN ID, etc.) based on specific matching rules and actions. Note that OpenFlow standard allows to manipulate more than 30 L2/L3 packet headers on the fly by using simple SDN flow rules. The NetShifter controller (NC) updates SDN flow rules on the OpenFlow enabled switch (OF) to enforce IP address mutation as shown in Fig. 7.3.

For example, the target node shown in Fig. 7.3 uses static IP address X0 at time t_0 and after time t_1. At time t_1, the NetShifter controller (NC) updates flow rules on the OpenFlow enabled switch (OF) that connects the target node to enforce the mutation of IP address from X0 to X1. According to the updated SDN flow rules,

the OpenFlow enabled switch (OF) mutates the source IP address in the header of packets from the node as X1. As a result, nodes that receive those packets will see the source IP address X1. Also, the OpenFlow enabled switch mutates the destination IP address in the header of packet to the node as X0. As a result, the node will receive those packets and see the destination IP address X0. Therefore, the IP address of the node is perceived as X1 after time t_1 from the network while the user of the node will not see any changes in IP address configuration. In this sense, NetShifter's IP address mutation is end-user transparent.

In NetShifter, the pool of mutable IP addresses of a node includes all available IP addresses of enclaves participating NetShifter solution. Hence, IP address of hosts and servers will be mutated randomly in a pool of all available IP addresses assigned for the organization across enclaves. NetShifter can also include private IP addresses, such as address ranges *192.168.0.0/16*, *172.16.0.0/12*, *10.0.0.0/8*, to the pool, thus making sure that the randomized IP addresses are chosen from a relatively large set of IP addresses.

The connectivity of the network with the mutated IP addresses will be maintained by using SDN technology and GRE (or VPN) tunneling. The method for delivering a packet destined to IP address assigned to a different enclave can be explained using Fig. 7.4. This method requires an OpenFlow (OF) enabled device connected

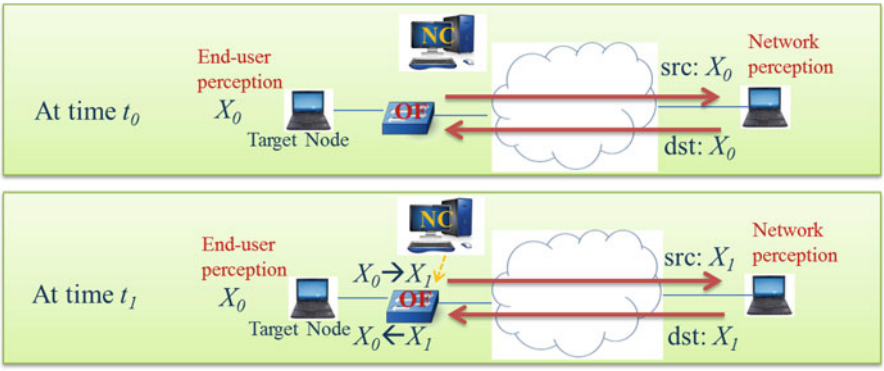

Fig. 7.3 End-user transparent IP address mutation

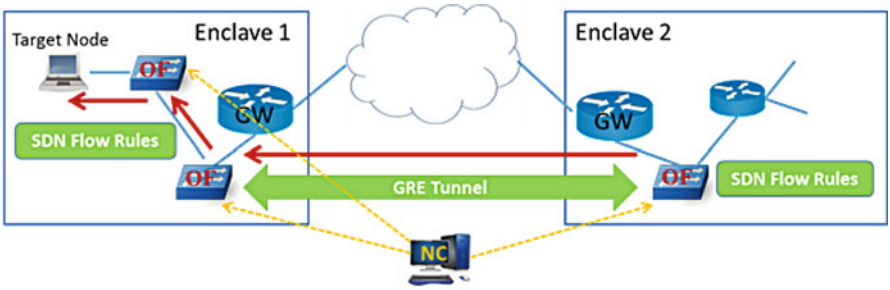

Fig. 7.4 Delivery of packets with mutated IP address across enclaves

to the gateway router (GW) in each enclave and GRE (or VPN) tunnels established between those OF devices. In enterprise networks, an inter-enclave GRE (or VPN) tunnel setups may be readily available, and thus can be re-used for NetShifter solution. When a packet destined to a mutated IP address reaches one of the GW routers, an OF device that is connected to the GW router forwards the packet using the SDN flow rules that is updated by the NetShifter controller. OpenFlow standard enables SDN flow rules to be enforced quickly on every OpenFlow enabled devices. The GW routers are legacy devices that do not support SDN flow rules. The OF devices use the inter-enclave GRE (or VPN) tunnel to pass the packet according to the SDN flow rules without conflicting with legacy routing rules of the GW routers.

For example, if the mutated IP address of the target node shown in Fig. 7.4 belongs to Enclave 2, the backbone network will forward the packet to the GW router of Enclave 2 based on legacy routing. The GW router will also forward the packet to the OF device in Enclave 2 based on legacy routing. Then, the OF device in Enclave 2 will forward the packet to the OF device in Enclave 1 through the GRE tunnel based on SDN flow rules. Also, the OF devices in Enclave 1 will forward the packet to the target node based on SDN flow rules. The NetShifter controller updates the SDN flow rules to make sure the packets are delivered successfully to the target node.

7.4.2 Flow Migration

A network obfuscation and deception solution to randomize a route between a given source and destination pair is necessary to defeat cyberattackers who try to launch an eavesdropping or denial of service attacks on a node or a link in the route. In [1], a route randomization using overlay networking is proposed. In [6], authors mentioned that route randomization can be implemented by using SDN and integrating software-defined custom policies. NetShifter leverages the SDN and software-defined policies to migrate and randomize the route.

In NetShifter solution, an OpenFlow (OF) enabled device randomizes the route of a flow according to SDN flow rules updated by the NetShifter controller (NC). For example, when a flow (i.e., a traffic session) is initiated from a source to a destination at time t0, the OF device forwards the traffic to a router R1 as shown in the left of Fig. 7.5. At time t1, the NetShifter updates SDN flow rules on the OF device. Then, the OF device forwards the traffic to another router R2 according to the updated SDN flow rules as shown in the right of Fig. 7.5. As a result, the flow has migrated from R1 to R2.

7.4.3 *Topology Mutation*

The topology of a network is critical information that can be used for planning an effective cyberattack. Hence, a network obfuscation and deception solution to mutate the topology is necessary. In wired network, the physical connections are static and re-wiring is difficult. To make the connections flexible, NetShifter solution takes a hybrid approach that uses SDN devices residing with legacy network devices with very minimal configuration changes. The topology mutation will be controlled by the NetShifter controller using OpenFlow standard.

For example, OpenFlow (OF) enabled devices are placed between legacy routers as shown in Fig. 7.6. The NetShifter controller (NC) updates these OF devices with the SDN flow ruleset #1 to enforce traffic forwarding as shown in the left of Fig. 7.6. As a result, the perceived topology is set to the virtual topology #1. If NC updates these OF devices with the SDN flow ruleset #2, the perceived topology will be set to the virtual topology #2.

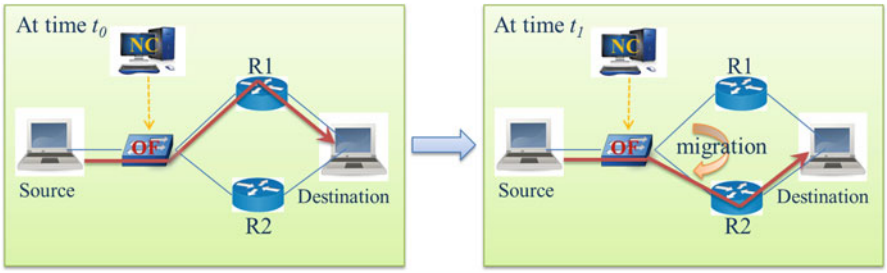

Fig. 7.5 Flow migration

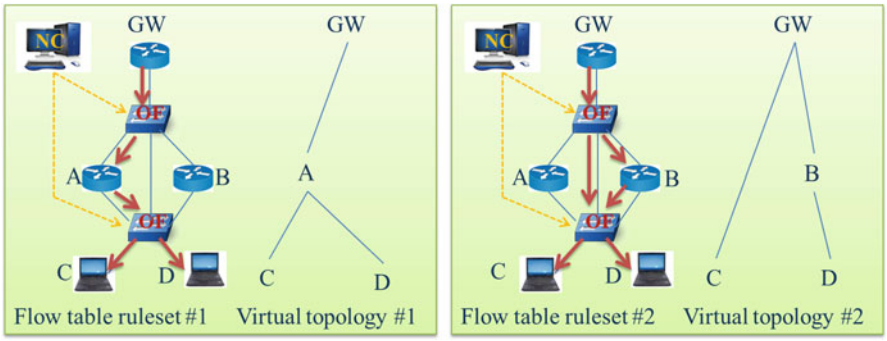

Fig. 7.6 Topology mutation

7.4.4 DNS/IP Binding Mutation

We developed DNS/IP binding mutation that dynamically mutates the binding between a domain name and an IP address exploiting fast flux DNS. The domain name system (DNS) is a hierarchical decentralized naming system for computers, services, or any resource connected to the Internet or a private network. It associates information with domain names assigned to each of the participating entities. Fast flux DNS is a technique that takes advantage of the way load balancing is built into the domain name system. DNS allows an administrator to register multiple IP addresses associated with a single fully qualified domain name. Typically, the IP addresses associated with a host domain do not change very often, if at all. However, it is possible to hide a host server's IP address to domain name mapping by using very short time-to-live (TTL) settings for the DNS resource records and swapping the records' associated IP addresses in and out with high frequency using sets of round-robin IP addresses. Website hostnames, for example, may be associated with a new set of IP addresses as often as every 3–6 min. NetShifter's DNS/IP binding mutation introduces a novel extension to this method, whereas NetShifter DNS proxy is augmented to create a DNS network obfuscation and deception cache. DNS resource records (RR) inside the cache are updated to rotate IP address numbers for a given service domain name. NetShifter network obfuscation and deception port redirectors are used as "blind" proxies-redirectors to forward requests to the real Internet server. These "blind" proxies-redirectors have very short life span, with the next proxy starting (using the next rotating IP address to be referenced in the DNS RR record) when the current redirector's lifetime is about to expire. When an attacker attempts DNS record harvesting, she gets a current "snapshot" of the entire DNS record cache from the NetShifter's DNS proxy. These domain name records contain the transient IP addresses of the current port redirectors, instead of IP addresses of the real servers. The snapshot will become obsolete in a matter of minutes, due to continuous IP address rotation. In the case of a legitimate user trying to connect to an Internet server using domain name, her request will be forwarded to a real Internet server through the currently active port redirector. This novel network obfuscation and deception technique aimed to conceal the network address-to-domain-name mappings, and it can be used for HTTP, SMTP, SIP, IMAP, POP, and other Internet services.

7.4.5 Service Randomization

Services in enterprise network are potential attack surfaces for cyberattacks. We developed two service randomization techniques that randomize the location of the server (i.e., service migration) or randomize the IP address of the server (i.e., service IP address mutation).

Using virtualization technology, a server running as a virtual machine (VM) can be migrated from one physical location to another physical location with minimal disruption without any loss of service states in the memory. For example, a server VM in enclave 1 can migrate from Enclave 1 to Enclave 2 as shown in Fig. 7.7. The NetShifter controller (NC) can control the VM migration using interfaces provided by virtualization products (e.g., VirtualBox, VMware, Xen, etc.). The NetShifter can also control the DNS server (or proxy) to update the DNS/IP address binding for the service. Hence, the legitimate users can still access the service in the new location.

Alternatively, NetShifter can control OF devices to perform IP address mutation (as described in Sect. 7.4.1) of the service as shown in Fig. 7.7. The IP address mutation can be performed across enclaves as we have explained in Sect. 7.4.1. Hence, the service IP address mutation can provide an equivalent result as the service migration.

7.5 Case Study

This section provides case study of how NetShifter can operate in enterprise network. We established an enterprise network testbed and demonstrated the operation of NetShifter in the following use cases:

- Case 1: IP address mutation within an enclave
- Case 2: Cross domain IP address mutation
- Case 3: Flow migration/topology mutation
- Case 4: Service migration

7.5.1 Enterprise Network Testbed

For the demonstration of the use cases, we established an enterprise network testbed using COTS devices including OpenFlow capable routers, conventional routers, hosts, and servers as shown in Fig. 7.8.

The testbed mimics realistic enterprise networks that are interconnected through a wide area network (WAN) or Internet. The testbed consists of three enclaves. Each enclave forms a small local area network (LAN) with COTS network devices (i.e., routers and switches).

We use LinkSys routers as OpenFlow enabled routers in the testbed. Each OpenFlow enabled routers/switches are controlled by the NetShifter controller which is implemented as an SDN application on an OpenDaylight (ODL) OpenFlow controller. The NetShifter controller updates SDN flow rules on OpenFlow enabled devices using the OpenFlow standard to perform NetShifter network obfuscation and deception techniques. In addition, the NetShifter controller dynamically

reconfigures DNS server to provide fast flux capability to add another layer of complexity in the NetShifter solution. A network accessible storage (NAS) maintains an image of application server (e.g., web server) to support service migration.

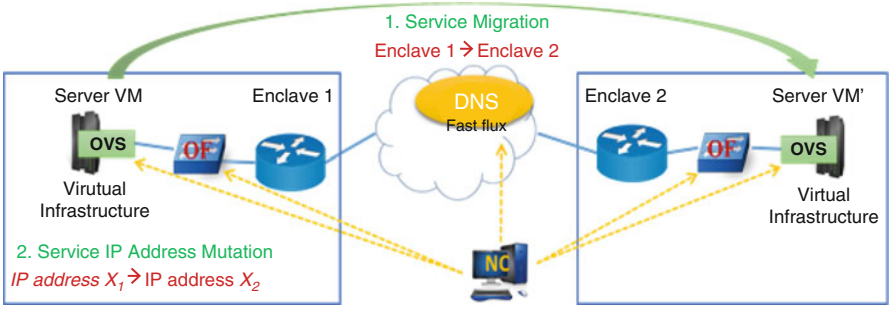

Fig. 7.7 Service randomization

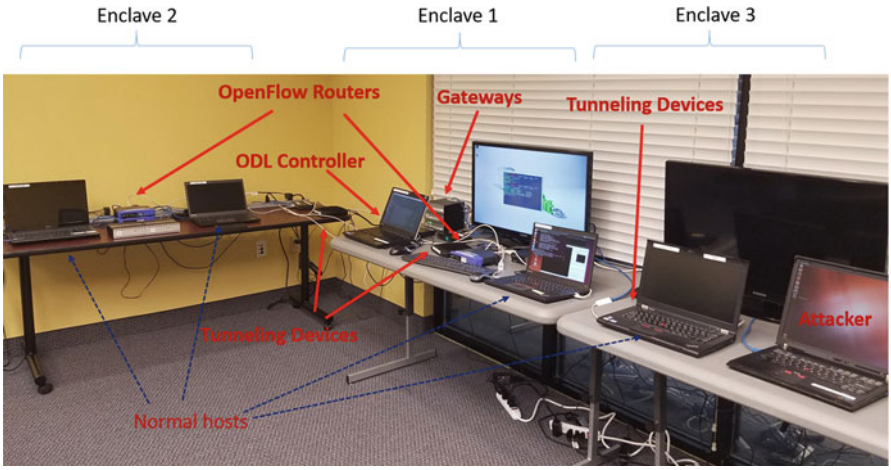

Fig. 7.8 Enterprise network testbed

We deployed a tunneling device between the gateway and the SDN routers in each enclave because the LinkSys routers do not support GRE tunneling. The tunneling devices are implemented using COTS Linux box devices. The tunneling devices establish a GRE tunnel with each tunneling devices in other enclaves. We can use other COTS OpenFlow enabled routers/switch that supports GRE tunneling. Then, we do not need to use those tunneling devices.

We emulate the reconnaissance from an attacker using zenmap tool, which is an open source network scanning tool that provides graphical user interface.

7.5.2 Case 1: IP Address Mutation Within an Enclave

This case shows user transparent IP address mutation of a web server within an enclave. Also, DNS/IP binding mutation is demonstrated to support users to access the web server using an updated IP address.

In Fig. 7.9, the IP address of Host 1 is *10.101.2.33* at time t_0. Also, the DNS server provides IP binding of www.target.i-a-i.com as *10.101.2.33* at time t_0. At time t_1, the NetShifter controller sends commands to the OpenFlow router and the DNS server to mutate IP address to *10.101.2.150*, which is an IP address in the same subnet/enclave as the original IP address. According to the commands, the OpenFlow router mutates the IP address in the header of packets being sent to/from Host 1. Also, the DNS server mutates the IP bind of www.target.i-a-i.com as *10.101.2.150*. Hence, Host 2 can access Host 1 (web server) anytime with the aid of the DNS server.

We emulated an attacker that performs reconnaissance from outside as shown in Fig. 7.9 using zenmap tool. The attacker scans an address space of *10.101.2.0/24* at time t_0 and after time t_1. The attacker finds the original IP address (*10.101.2.150*) of Host 1 (web server) at time t_0 as shown in the left of Fig. 7.10. After time t_1, the attacker cannot find the original IP address after t0 as a result of the IP address mutation. Instead, the attacker finds the mutated IP address (*10.101.2.150*) as shown in the right of Fig. 7.10.

7.5.3 Case 2: IP Address Mutation Across Enclaves

IP address mutation of a web server across enclaves is demonstrated in this section. Also, DNS/IP binding mutation and GRE tunneling is performed to support users to access the web server.

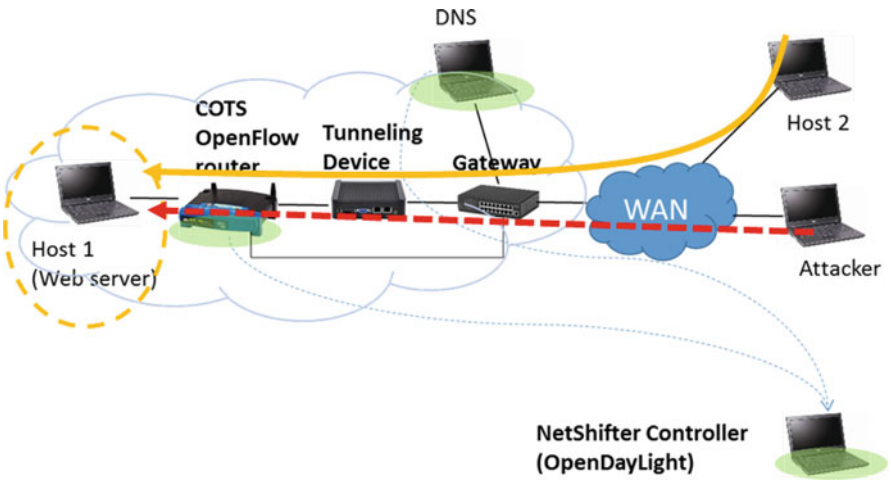

Fig. 7.9 Case 1

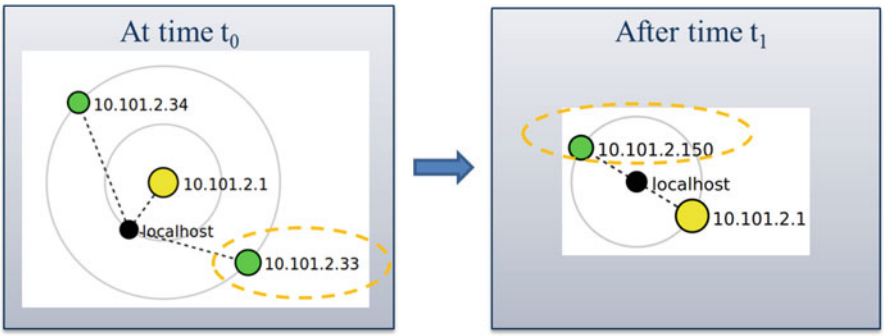

Fig. 7.10 Case 1 result on zenmap tool

In Fig. 7.11, the IP address of Host 4 is *10.102.3.15* at time t_0. Also, the DNS server provides IP binding of www.target.i-a-i.com as *10.102.3.15* at time t_0.

At time t_1, the NetShifter controller sends commands to the OpenFlow router and the DNS server to mutate IP address to *10.101.2.31*, which is an IP address in a different enclave as the original IP address (*10.102.3.15*) as shown in Fig. 7.12. Also, the DNS server mutates the IP bind of www.target.i-a-i.com as *10.101.2.31*.

The wide area network (WAN) and the gateway routers forward the packets with destination IP addresses in *10.101.0.0/16* to Enclave 1. However, Host 4 is actually located in Enclave 2. To avoid this conflict, a GRE tunnel is used. The packets with destination IP address *10.101.2.31* are forwarded to Enclave 2 using the GRE tunnel between the tunneling device in Enclave 1 and the tunneling device in Enclave 2 as shown in Fig. 7.12. Hence, Host 1 can access Host 4 (web server) anytime with the aid of the DNS server, tunneling devices, and OpenFlow routers.

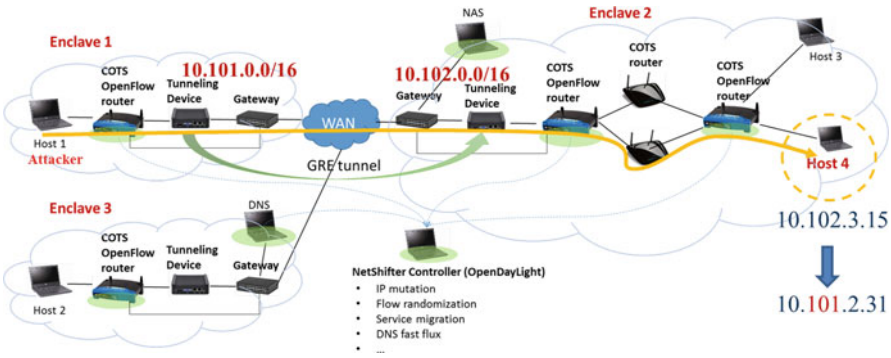

Fig. 7.11 Case 2 at time t_0

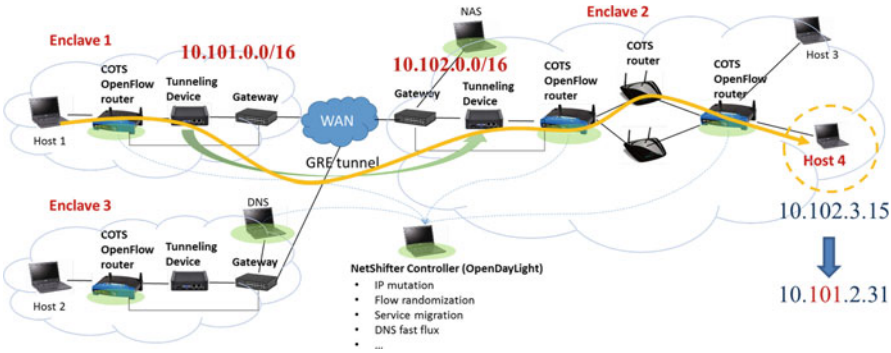

Fig. 7.12 Case 2 after time t_1

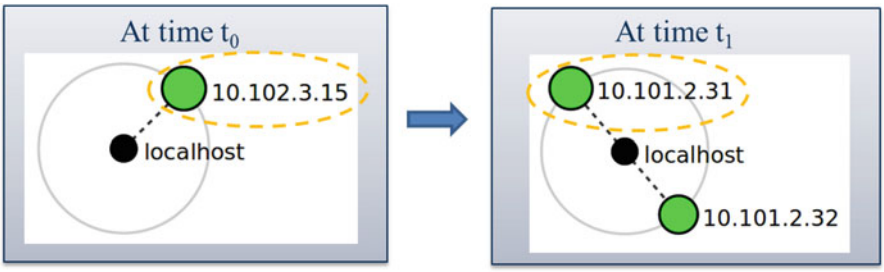

Fig. 7.13 Case 2 result on zenmap tool

The attacker scans an address space of *10.102.3.0/24* at time t_0 and after time t_1. The attacker finds the original IP address (*10.102.3.15*) of Host 4 (web server) at time t_0 as shown in the left of Fig. 7.13. After time t_1, the attacker cannot find the original IP address as a result of the IP address mutation. If the attacker scans the address space of 10.101.2.0/24, the attacker will find the mutated IP address (*10.101.2.31*) as shown in the right of Fig. 7.13. It will be much more difficult for attackers to link the original IP address with the mutated IP address compared to Case 1. The attacker may not even try to scan the address space of *10.101.2.0/24*. The attacker may scan entire address space of *10.0.0.0/4*, which will take a very long time and make the attacker easier to be detected by cyber defenders.

7.5.4 *Case 3: Flow Migration/Topology Mutation*

In this case, we demonstrate the flow migration and the topology mutation in NetShifter solution. At time t_0, the network traffic from Host 1 to Host 3 and the network traffic from Host 1 to Host 4 are forwarded through the route illustrated as yellow arrows in Fig. 7.14.

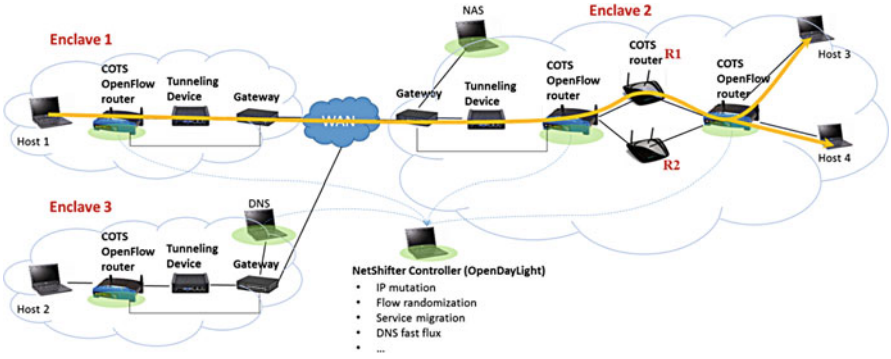

Fig. 7.14 Case 3 at time t_0

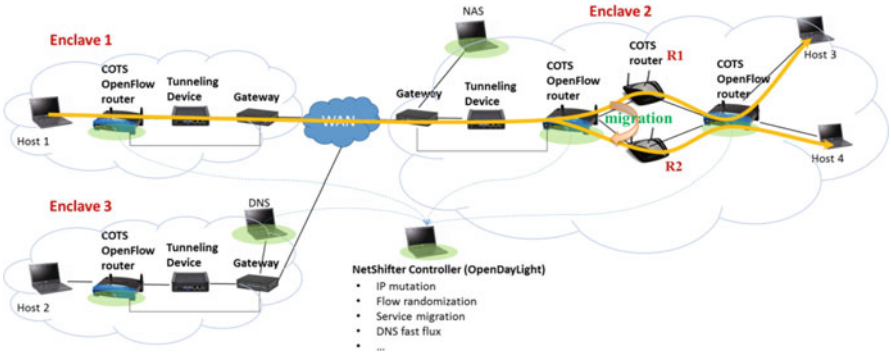

Fig. 7.15 Case 3 after time t_1

At time t_1, the NetShifter controller sends SDN flow rules to OpenFlow routers to forward the network traffic from Host 1 to Host 3 and the network traffic from Host 1 to Host 4 through the route illustrated as yellow arrows in Fig. 7.15. As a result, the traffic flow from Host 1 to Host 4 has migrated from the COTS router R1 to the COTS router R2. Also, as a collective result of the SDN flow rules for the network traffic from Host 1 to Host 3 and the network traffic from Host 1 to Host 4, the virtual topology is mutated at time t1.

When Host 1 in Enclave 1 scans Enclave 2 network and draw network diagram using zenmap tool at time t_0, Host 1 sees the topology of Enclave 2 network as shown in the left of Fig. 7.16. After time t_1, Host 1 sees the topology of Enclave 2 network as shown in the right of Fig. 7.16 which is different from the topology at time t_0.

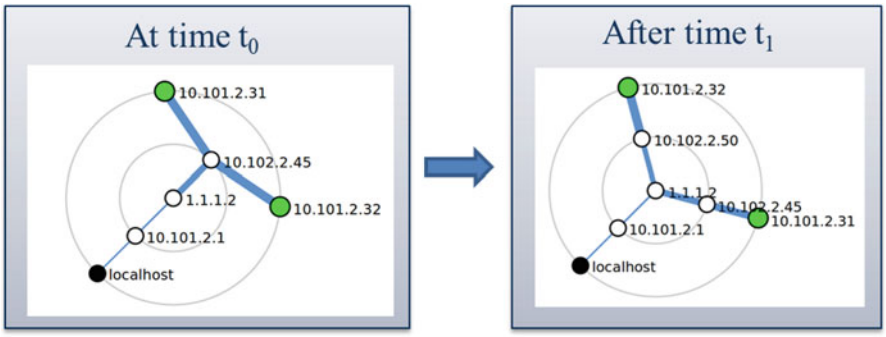

Fig. 7.16 Case 3 result on zenmap tool

7.5.5 Case 4: Service Migration

Using this case, the service migration in NetShifter solution is demonstrated using virtual machine (VM) migration.

In this case, Host 1 and Host 2 serve as virtual infrastructures on which the web server VM can be resided. At time t_0, the web server virtual machine (VM) is residing on Host 1 as shown in Fig. 7.17. The IP address of the web server VM is *10.101.2.33* at time t_0. Also, the DNS server provides IP binding of www.target.i-a-i.com as *10.101.2.33* at time t_0.

At time t_1, the NetShifter controller sends commands to Host 1 and Host 2. Then, Host 1 and Host 2 perform the migration of web server VM from Host 1 to Host 2. Also, the NetShifter controller sends commands to the DNS server to mutate the IP address of the web server VM. According to the commands, the DNS server mutates the IP bind of www.target.i-a-i.com as *10.103.2.11*. Hence, user laptop 1 can access Host 1 (web server) anytime with the aid of the DNS server.

The attacker scans an address space of *10.101.2.0/24* at time t_0 and after time t_1. The attacker finds the original IP address (*10.101.2.11*) of the web server VM at time t0 as shown in the left of Fig. 7.18. After time t_1, the attacker cannot find the original IP address as a result of the service migration. If the attacker scans the address space of *10.103.2.0/24*, the attacker will find the mutated IP address (*10.103.2.11*) as shown in the right of Fig. 7.18.

It will be much more difficult for attackers to link the original IP address with the mutated IP address compared to Case 1. Furthermore, it may be more difficult to attackers to figure out whether a network obfuscation and deception solution is being performed compared to Case 2. In Case 2, the attacker may notice some inconsistency of configurations because only the IP address is mutated to other Enclave and the actual location is not changed. In Case 3, the actual location is changed and the IP address is changed according to the location. Hence, the attacker cannot see any inconsistency of configurations.

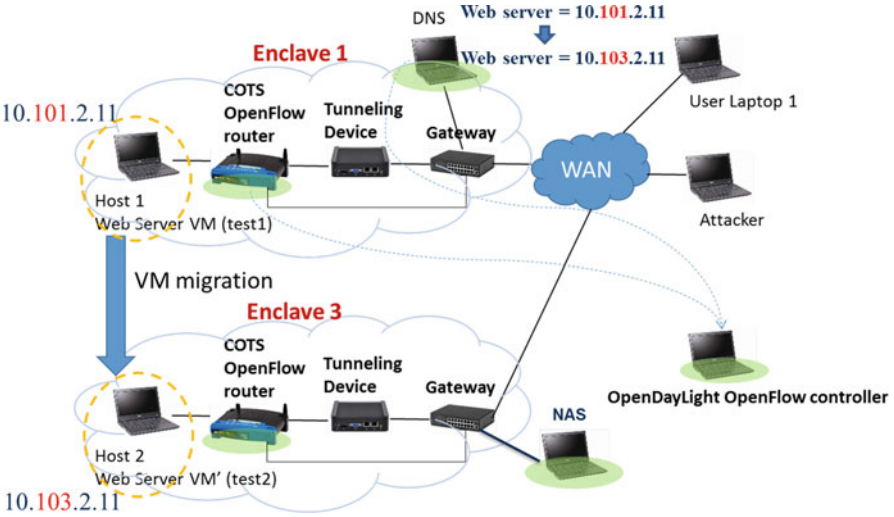

Fig. 7.17 Case 4

7.6 Summary

This chapter has presented an overview of NetShifter, a comprehensive multi-dimensional network obfuscation and deception solution. The key features of NetShifter include:

- **Multi-dimensional network obfuscation and deception**: NetShifter applies multiple network obfuscation and deception techniques simultaneously. It obfuscates and randomizes multiple aspects of the network configurations, each of which adds dimensions of complexity that cyberattacker must analyze. In particular, NetShifter employs the following list of obfuscation and deception techniques.

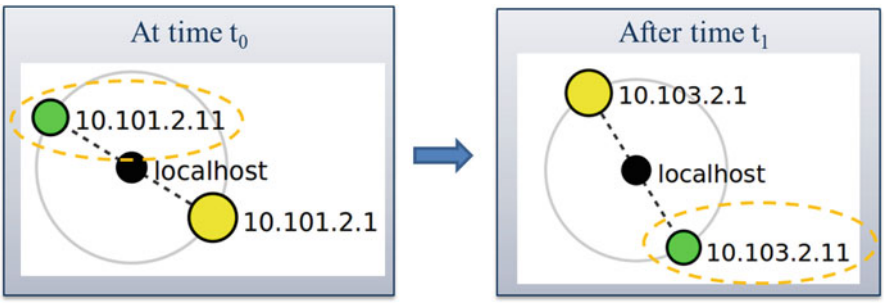

Fig. 7.18 Case 4 result on zenmap tool

- **Cross-enclave network obfuscation and deception**: With coordination between multiple NetShifter obfuscation and deception techniques, NetShifter assigns random/private IP that does not belong to current subnet/enclave to mislead both insider and outside attackers while traffic is forwarded to proper services and the destination.
- **Transparency to users/end hosts**: Configurations of the network will be controlled by using centralized or distributed controllers, in order to effectively manage large-scale network operation in which user does not need to be aware of any configuration updates or require any actions. The connectivity of hosts and servers will be maintained by NetShifter solution without user notification or actions.
- **Resiliency even to insider attacks**: By obfuscating network topology and traffic as well as anonymizing server and user activity, NetShifter provides high level of complexity for outsider attacker to infer network topology and user activity. The insider attacker will not see anything different from the outside attacker except for the change of IP address assigned to the host because of the separation of control plane and data plane in NetShifter.
- **Easy Deployment and transition**: NetShifter can be implemented in a different format, depending on user preferences and network administrative policy. NetShifter can be realized as an SW-based solution by installing an SW package on virtual infrastructure. NetShifter also can be implemented as a hardware appliance in the loop (e.g., dongle or SDN-enabled router/switch) that resides between legacy network devices. Such implementation flexibility makes NetShifter relatively easily deployed in diverse operational environment with unique requirements and constraints.

NetShifter is mainly based on SDN, virtualization techniques, and legacy network services, which makes NetShifter easily extended to cloud-based service and applications to secure virtual infrastructure and network traffic, while managing the service/application more effectively.

By performing innovative and comprehensive obfuscation and deception techniques, NetShifter will make it harder for the adversary to discover and analyze network topology and user activity.

7.7 Exercises

Problem 1: List 3 fields in IP packet header that can be modified to confuse an eavesdropper.

Problem 2: Define an OpenFlow rule to change packet destination IP address from 10.1.1.1 to 10.2.2.1.

Problem 3: List design considerations to properly route destination IP address 10.2.2.1 to the right host (i.e., destination IP address 10.1.1.1).

References

1. Qi Duan, et al, "Efficient Random Route Mutation Considering Flow and Network Constraints", In proceedings of Hot Topics in Software Defined Networking (HotSDN 12), 2012.
2. Justin Yackoski, et al, "A self-shielding dynamic network architecture", In proceedings of IEEE MILCOM, November 2011, pp. 1381–1386.
3. Active Repositioning in Cyberspace for Synchronized Evasion (ARCSYNE), 2012, Final Technical Report
4. J.H. Jafarian, et al, "OpenFlow Random Host Mutation: Transparent Moving Target Defense using Software Defined Networking", In Proceedings of ACM SIGCOMM HotSDN workshop, August 2011.
5. D.C. MacFarland et al, "The SDN Shuffle: Creating a Moving-Target Defense using Host-based Software-Defined Networking", In Proceedings of the Second ACM Workshop on Moving Target Defense, pp 37–41.
6. P. Kampanakis et al, "SDN-based Solutions for Moving Target Defense network Protection" In Proceedings of IEEE Symposium on a World of Wireless, Mobile and Multimedia Networks (WoWMoM), June 2014, pp. 1–6.
7. S. Antonatos, et al, "Defending against hitlist worms using network address space randomization", In proceedings of the 2005 ACM workshop on Rapid malcode (WORM '05), 2005.
8. D. Kewley, R. Fink, J. Lowry, and M. Dean, "Dynamic approaches to thwart adversary intelligence gathering", In proceedings of DARPA Information Survivability Conference Exposition II (DISCEX'01), 2001.
9. C.M. Price, E. Stanton, E.J. Lee, J.T. Michalski, K.S. Chua, Y.H. Wong, and C.P. Tan, "Network Security Mechanisms Utilizing Dynamic Network Address Translation LDRD Project," Sandia National Labs, 2002 Nov 01.
10. H. Moniz, N.F. Neves, M. Correia, and P. Verissimo, "Randomized Intrusion-Tolerant Asynchronous Services," International Conference on Dependable Systems and Networks, June 2006.
11. S. Antonatos, P. Akritidis, E.P. Markatos, and K.G. Anagnostakis, "Defending against hitlist worms using network address space randomization," Comput. Netw. 51, 12 (August 2007), 3471–3490.

12. E. Al-Shaer, "Toward Network Configuration Randomization for Moving Target Defense," Moving Target Defense: Creating Asymmetric Uncertainty for Cyber Threats, S. Jajodia, A.K. Ghosh, V. Swarup, C. Wang, X.S. Wang (eds.), New York: Springer, 2011, 153–159.
13. J.D Touch, G.G. Finn, Y.-S. Wang, and L. Eggert, "DynaBone: dynamic defense using multi-layer Internet overlays," In Proceedings of DARPA Information Survivability Conference and Exposition, vol. 2 (22–24 April 2003), pp. 271–276.

Chapter 8
Deception-Enhanced Threat Sensing for Resilient Intrusion Detection

Frederico Araujo, Gbadebo Ayoade, Kevin W. Hamlen, and Latifur Khan

Abstract Enhancing standard web services with deceptive responses to cyberattacks can be a powerful and practical strategy for improved intrusion detection. Such deceptions are particularly helpful for addressing and overcoming barriers to effective machine learning-based intrusion detection encountered in many practical deployments. For example, they can provide a rich source of training data when training data is scarce, they avoid imposing a labeling burden on operators in the context of (semi-)supervised learning, they can be deployed post-decryption on encrypted data streams, and they learn concept differences between honeypot attacks and attacks against genuine assets.

The approach presented in this chapter examines how deceptive web service responses can be realized as software security patches that double as feature extraction engines for a network-level intrusion detection system. The resulting system coordinates multiple levels of the software stack to achieve fast, automatic, and accurate labeling of live web data streams, and thereby detects attacks with higher accuracy and adaptability than comparable non-deceptive defenses.

8.1 Introduction

Detecting previously unseen cyberattacks before they reach unpatched, vulnerable web servers (or afterward, for recovery purposes) is an increasingly central component to multi-layered defense of modern computer networks. High-impact zero-day

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

F. Araujo
IBM T.J. Watson Research Center, Yorktown Heights, NY, USA

G. Ayoade · K. W. Hamlen (✉) · L. Khan
The University of Texas at Dallas, Richardson, TX, USA
e-mail: hamlen@utdallas.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_8

vulnerabilities now appear at a weekly or daily rate, and studies indicate that over 75% of websites have unpatched vulnerabilities [19]. The cost of data breaches resulting from software exploits was estimated at $2.1 trillion for 2019 [14].

Intrusion detection [10] is an important means of mitigating such threats. Rather than implementing vulnerability-specific mitigations (which is difficult when the vulnerability is unknown to defenders), intrusion detection systems more generally alert administrators when they detect deviations from a model of *normal* behavior in the observed data [20]. This capitalizes on the observation that the most damaging and pernicious attacks discovered in the wild often share similar traits, such as the steps intruders take to open back doors, execute files and commands, alter system configurations, and transmit gathered information from compromised machines. Starting with the initial infection, such malicious activities often leave telltale traces that can be identified even when the underlying exploited vulnerabilities are unknown to defenders. The challenge is therefore to capture and filter these attack trails from network traffic, connected devices, and target applications, and develop defense mechanisms that can effectively leverage such data to disrupt ongoing attacks and prevent future attempted exploits.

However, despite its great power, the deployment of machine learning approaches for web intrusion detection is often hindered by a scarcity of realistic, current cyberattack data with which to train the system, and by the difficulty of accurately and efficiently labeling such datasets, which are often prohibitively large and complex. This can frustrate comprehensive, timely training of intrusion detection systems (IDSes), causing the IDS to raise numerous false alarms and elevating its susceptibility to attacker evasion techniques [6, 9, 13, 16, 18].

To mitigate these dilemmas, this chapter presents a deception-based approach to enhance IDS web data streams for faster, more accurate, and more timely evolution of intrusion detection models to emerging attacks and attacker strategies.

8.2 Deceptive Collection of Attack Data

Deception has long been recognized as a key ingredient of effective cyber warfare (cf., [23]), but many realizations limit the potential power of deception by isolating and separating deceptive assets from the data stream in which intrusions must actually be detected. A typical example is the use of dedicated *honeypots* to collect attack-only data streams [21]. Such approaches unfortunately have limited training value in that they often mistrain IDSes to recognize only attacks against honeypots, or only attacks by unsophisticated adversaries unable to identify and avoid honeypots. For example, attacks that include substantial interactivity are typically missed, since the honeypot offers no legitimate services, and therefore collects no data characterizing attacks against legitimate services.

One way to overcome this limitation is to integrate deceptive attack response capabilities directly into live, production web server software via *honey-patching* [2–4]. Honey-patches are software security patches that are modified to avoid alerting adversaries when their exploit attempts fail. Instead of merely

blocking the attempted intrusion, the honey-patch transparently redirects the attacker's connection to a carefully isolated decoy environment running an unpatched version of the software. Adversaries attempting to exploit a honey-patched vulnerability therefore observe software responses that resemble unpatched software, even though the vulnerability is actually patched. This deception allows the system to observe subsequent actions by the attacker until the deception is eventually uncovered. Thus, honey-patches offer equivalent security to conventional patches, but can also enhance IDS web data streams by feeding them a semantically rich stream of pre-labeled (attack-only) data for training purposes. These *deception-enhanced* data streams thus provide IDSes with concept-relevant, current, feature-filled information with which to detect and prevent sophisticated, targeted attacks.

Honey-patches are often easy to implement via only a minor change to a vendor-released software patch. For example, buffer overflow vulnerabilities are typically patched by adding a bounds check that tests whether a dereferenced pointer or array index falls within the bounds of the buffer. Such patches can easily be reformulated into honey-patches by retaining the check, but changing what happens when the check fails. Instead of aborting the connection or reporting an error, the honey-patch redirects the connection to an unpatched decoy, where the buffer overflow is permitted to succeed.

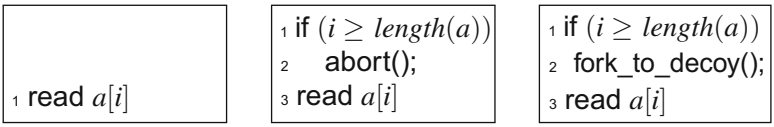

Fig. 8.1 Pseudo-code for a buffer overflow vulnerability (left), a patch (middle), and a honey-patch (right)

Figure 8.1 demonstrates the approach using pseudo-code for a buffer-overflow vulnerability, a conventional patch, and a honey-patch. The honey-patch retains the logic of the conventional patch's security check, but replaces its remediation with a deceptive fork to a decoy environment. The decoy contains no valuable data and offers no legitimate services; its sole purpose is to monitor attacker actions, such as shellcode or malware introduced by the attacker after abusing the buffer overflow to hijack the software. The infrastructure for redirecting attacker connections to decoys can remain relatively static, so that honey-patching each newly discovered vulnerability only entails replacing the few lines of code in each patch that respond to detected exploits.

Honey-patches constitute an integrated deception mechanism that offers some important advantages over conventional honeypots. Most significantly, they observe attacks against the defender's genuine assets, not merely those directed at fake assets that offer no legitimate services. They can therefore capture data from sophisticated attackers who monitor network traffic to identify service-providing assets before launching attacks, who customize their attacks to the particular activities

of targeted victims (differentiating genuine servers from dedicated honeypots), and who may have already successfully infiltrated the victim's network before their attacks become detected. The remainder of this chapter examines how the deception-enhanced data harvested by honey-patches can be of particular value to network-level defenses, such as firewalls equipped with machine learning-based intrusion detection.

8.3 Intrusion Detection Challenges

Despite the potential power of machine learning in intrusion detection applications, its success in operational environments can be hampered by specific challenges that arise in the cybersecurity domain. In this section we argue that cyber deception can be a highly effective strategy for avoiding or overcoming many of these challenges.

Fundamentally, machine learning algorithms perform better at identifying similarities than at discovering previously unseen outliers. Since normal, non-attack data is usually far more plentiful than realistic, current attack data, many classifiers must be trained almost solely from the former, necessitating an almost perfect model of normality for any reliable classification [18]. Deceptive defenses help to offset this imbalance by providing a continuous source of realistic attack data specialized to the defender's network and assets.

Feature extraction [7] is also unusually difficult in intrusion detection contexts because security-relevant features are often not known by defenders in advance. The task of selecting appropriate features to detect an intrusion (e.g., features that generate the most distinguishing intrusion patterns) can create a bottleneck in building effective models, since it demands empirical evaluation. Identification of attack traces among collected workload traces for constructing realistic, unbiased training sets is particularly challenging. Current approaches usually require manual analysis aided by expert knowledge [6, 9], which reduces the model's evolutionary and update capabilities, making it susceptible to attacker evasions. The approach presented in this chapter shows how including deceptions within software security patches can overcome this difficulty.

A third obstacle is analysis of encrypted data. Encryption is widely employed to prevent unauthorized users from accessing sensitive web data transmitted through network links or stored in file systems. However, since network-level detectors typically discard cyphered data, their efficacy is greatly reduced by the widespread use of encryption technologies [13]. In particular, attackers benefit from encrypting their malicious payloads, making it harder for standard classification strategies to distinguish attacks from normal activity. Deceptive defenses can often be placed after decryption within the software stack, evading this problem.

High false positive rates are another practical challenge for adoption of machine learning approaches [16]. Raising too many alarms renders intrusion detection meaningless in most cases, as actual attacks are often lost among the many alarms.

Studies have shown that effective intrusion detection therefore demands very low false alarm rates [5]. Deception-enhanced data streams can ameliorate this by improving the concept-relevance of the collected training data, improving attack detection accuracy.

8.4 Mining Deception-Enhanced Threat Data

To mitigate these challenges, this chapter introduces an approach to enhance intrusion detection with threat data sourced from honey-patched [4] applications. Figure 8.2 shows an overview of the approach. Unlike conventional approaches, our framework incrementally builds a model of *legitimate* and *malicious* behavior based on audit streams and attack traces collected from honey-patched web servers. This augments the classifier with security-relevant feature extraction capabilities not available to typical network intrusion detectors, effectively reducing the anomaly detection task to a *semi-supervised* learning process.

Such capabilities are transparently built into the framework, requiring no additional developer effort (apart from routine patching) to convert the target application into a potent feature extractor for anomaly detection. Since traces extracted from decoys are always contexts of *true* malicious activity, this results in an effortless labeling of the data and supports the generation of higher-accuracy detection models.

Honey-patches add a layer of deception to confound exploits of known (patchable) vulnerabilities. Previously unknown (i.e., zero-day) exploits can also be mitigated through IDS cooperation with the honey-patches. For example, a honey-patch that collects identifying information about a particular adversary seeking to exploit a known vulnerability can convey that collected information to train a classifier, which can then potentially identify the same adversary seeking to exploit a previously unknown vulnerability. This enables training intrusion detection models that *capture features of the attack payload*, and not just features of the actual exploitation of the vulnerability, thus more closely approximating the true invariant of an attack.

To facilitate such learning, our approach classifies *sessions* as malicious, not merely the individual packets, commands, or bytes within sessions that comprise each attack. For example, observing a two-phase attack consisting of (1) exploitation of a honey-patched vulnerability, followed by (2) injection of previously unseen shellcode might train a model to recognize the shellcode. Subsequent attacks that exploit an unpatched zero-day to inject the same (or similar) shellcode can then be recognized by the classifier even if the zero-day exploit is not immediately recognized as malicious. Conventional, non-deceptive patches often miss such learning opportunities by terminating the initial attack at the point of exploit, before the shellcode can be observed.

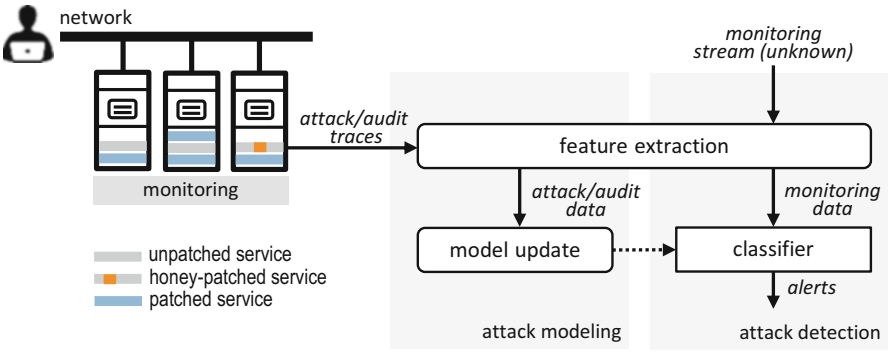

Fig. 8.2 System architecture overview

Our approach therefore essentially repurposes security patches in an IDS setting as automated, application-level feature extractors. The maintenance burden for these extractors is relatively low: most of the patch code is maintained by the collective expertise of the entire software development community, as they discover new vulnerabilities and release patches for them. Via honey-patching, defenders can reimagine those patches as highly accurate, rapidly co-evolving feature extraction modules for an IDS. The extractor detects previously unseen payloads that exploit known vulnerabilities at the application layer, which can be prohibitively difficult to detect by a strictly network-level IDS.

By living inside web servers that offer legitimate services, a deception-enhanced IDS can target attackers who use one payload for reconnaissance but reserve another for their final attacks. The facility of honey-patches to deceive such attackers into divulging the latter is useful for training the IDS to identify the final attack payload, which can reveal attacker strategies and goals not discernible from the reconnaissance payload alone. The defender's ability to thwart these and future attacks therefore derives from a synergy between the application-level feature extractor and the network-level intrusion detector to derive a more complete model of attacker behavior.

8.5 Use Case: Booby-Trapping Software for Intrusion Detection

8.5.1 Architectural Overview

The architecture depicted in Fig. 8.2 embodies this approach by leveraging application-level threat data gathered from attacker sessions misdirected to decoys. Within this framework, developers use honey-patches to misdirect attackers to

decoys that automatically collect and label monitored attack data. The intrusion detector consists of an *attack modeling* component that incrementally updates the anomaly model data generated by honey-patched servers, and an *attack detection* component that uses this model to flag anomalous activities in the monitored perimeter.

The decoys into which attacker sessions are forked can be managed as a pool of continuously monitored containers (e.g., LXC on Linux). Each container follows the following life cycle: Upon attack detection, the honey-patching mechanism *acquires* the first available container from the pool. The acquired container holds an attacker session until (1) the session is deliberately closed by the attacker, (2) the connection's *keep-alive* timeout expires, (3) the ephemeral container crashes, or (4) a session timeout is reached. The last two conditions are common outcomes of successful exploits. In any of these cases, the container is released back to the pool and undergoes a recycling process before becoming available again.

After decoy *release*, the *container monitoring component* extracts the session trace (delimited by the acquire and release timestamps), labels it, and stores the trace outside the decoy for subsequent feature extraction. Decoys only host attack sessions, so precisely collecting and labeling their traces (at both the network and OS level) is effortless.

Evaluating the framework requires distinguishing three separate input data streams: (1) the *audit stream*, collected at the target honey-patched server, (2) *attack traces*, collected at decoys, and (3) a *monitoring stream*, which consists of a actual test stream collected from regular servers. Each of these streams contains network packets and operating system events captured at each server environment. To minimize performance impact, a powerful and highly efficient software monitor is recommended. Recommended candidates include *sysdig* (to track system calls and modifications made to the file system) and *tcpdump* (to monitor ingress and egress of network packets). Specifically, monitored data is stored outside the decoy environments to avoid possible tampering with the collected data.

Using the continuous audit stream and incoming attack traces as labeled input data, the intrusion detector incrementally builds a machine learning model that captures legitimate and malicious behavior. The raw training set (composed of both audit stream and attack traces) is piped into a feature extraction component that selects relevant, non-redundant features (see Sect. 8.5.2) and outputs feature vectors—*audit data* and *attack data*—that are grouped and queued for subsequent model update. Since the initial data streams are labeled and have been preprocessed, feature extraction becomes very efficient and can be performed automatically. This process repeats periodically according to an administrator-specified policy. Finally, the *attack detection* module uses the most recently constructed attack model to detect malicious activity in the runtime *monitoring data*.

8.5.2 Detection Models

To assess our framework's ability to enhance intrusion detection data streams, we have designed and implemented two feature set models: (1) *Bi-Di* detects anomalies in security-relevant network streams, and (2) *N-Gram* finds anomalies in system call traces. The feature set models and classifier presented in this section serve as illustrative use case. Applications of the IDS framework should consider other machine learning models and contrast trade-offs and their effectiveness for attack detection.

8.5.2.1 Network Packet Analysis

Bi-Di (bi-directional) is a packet-level network behavior analysis approach that extracts features from sequences of packets and *bursts*—consecutive packets oriented to the same direction (*viz.*, uplinks from client to server, or downlinks from server to client). It uses distributions from individual burst sequences (*uni-bursts*) and sequences of two adjacent bursts (*bi-bursts*). To be robust against encrypted payloads, we limit feature extraction to packet headers.

Network packets flow between client (*Tx*) and server (*Rx*). Bi-Di constructs histograms using features extracted from packet lengths and directions. To overcome dimensionality issues associated with burst sizes, *bucketization* is applied to group bursts into correlation sets (e.g., based on frequency of occurrence). Table 8.1 summarizes the features used in our approach. It highlights new features proposed for uni- and bi-bursts as well as features proposed in the prior works [1, 12, 15, 22].

Uni-burst Features include burst *size*, *time*, and *count*—i.e., the sum of the sizes of all packets in the burst, the amount of time for the entire burst to be transmitted, and the number of packets it contains, respectively. Taking direction into consideration, one histogram for each is generated.

Bi-burst Features include time and size attributes of *Tx-Rx-bursts* and *Rx-Tx-bursts*. Each is comprised of a consecutive pair of downlink and uplink bursts. The size and time of each are the sum of the sizes of the constituent bursts, and the sum of the times of the constituent bursts, respectively.

Bi-bursts capture dependencies between consecutive packet flows in a TCP connection. Based on connection characteristics, such as network congestion, the TCP protocol applies flow control mechanisms (e.g., window size and scaling, acknowledgement, sequence numbers) to ensure a level of consistency between Tx and Rx. This influences the size and time of transmitted packets in each direction. Each packet flow (uplink and downlink) thereby affects the next flow or burst until communicating parties finalize the connection.

Table 8.1 Packet, uni-burst, and bi-burst features

Category	Features
Packet (Tx/Rx)	Packet length
Uni-burst (Tx/Rx)	Uni-burst size
	Uni-burst time
	Uni-burst count
Bi-burst (Tx-Rx/Rx-Tx)	Bi-burst size
	Bi-burst time

Algorithm 1: *Ens-SVM*

Data: training data: $TrainX$, testing data: $TestX$
Result: a predicted label $\mathscr{L}_{\mathscr{I}}$ for each testing instance $\mathscr{I}$

begin
  $\mathbb{B} \leftarrow$ *updateModel*(Bi-Di, *TrainX*)
  $\mathbb{N} \leftarrow$ *updateModel*(N-Gram, *TrainX*)
  for *each* $\mathscr{I} \in$ *TestX* **do**
    $\mathscr{L}_{\mathbb{B}} \leftarrow$ *label*($\mathbb{B}$, $\mathscr{I}$)
    $\mathscr{L}_{\mathbb{N}} \leftarrow$ *label*($\mathbb{N}$, $\mathscr{I}$)
    if $\mathscr{L}_{\mathbb{B}} == \mathscr{L}_{\mathbb{N}}$ **then**
      $\mathscr{L}_{\mathscr{I}} \leftarrow \mathscr{L}_{\mathbb{B}}$
    else
      $\mathscr{L}_{\mathscr{I}} \leftarrow label\left(\underset{c \in \{\mathbb{B},\mathbb{N}\}}{\arg\max}\, confidence(c, \mathscr{I}),\ \mathscr{I}\right)$
    end
  end
end

8.5.2.2 System Call Analysis

The monitored data also includes system streams comprising a collection of OS events, where each event contains multiple fields including event type (e.g., *open*, *read*, *select*), process name, and direction. Our prototype implementation was developed for Linux x86_64 systems, which exhibit about 314 distinct possible system call events. Our framework builds histograms from these system calls using N-Gram—a system-level approach that extracts features from contiguous sequences of system calls.

There are four feature types: *Uni-events* are system calls, and can be classified as enter or exit events. *Bi-events* are sequences of two consecutive events, where system calls in each bi-event constitute features. Similarly, *tri-* and *quad-events* are sequences of three and four consecutive events (respectively).

Bi-Di and N-Gram differ in feature granularity; the former uses coarser-grained bursting while the latter uses only individual system call co-occurrences.

8.5.3 Attack Classification

Bi-Di and N-Gram both use SVM for classification. Using a convex optimization approach and mapping non-linearly separated data to a higher dimensional linearly separated feature space, SVM separates positive (attack) and negative (benign) training instances by a hyperplane with the maximum gap possible. Prediction labels are assigned based on which side of the hyperplane each monitoring/testing instance belongs.

Ens-SVM Bi-Di and N-Gram can be combined to obtain a better predictive model. A naïve approach concatenates features extracted by Bi-Di and N-Gram into a single feature vector and uses it as input to the classification algorithm. However, this approach has the drawback of introducing normalization issues. Alternatively, *ensemble methods* combine multiple classifiers to obtain a better classification outcome via majority voting techniques. For our purposes, we use an ensemble, *Ens-SVM*, which classifies new input data by weighing the classification outcomes of Bi-Di and N-Gram based on their individual accuracy indexes.

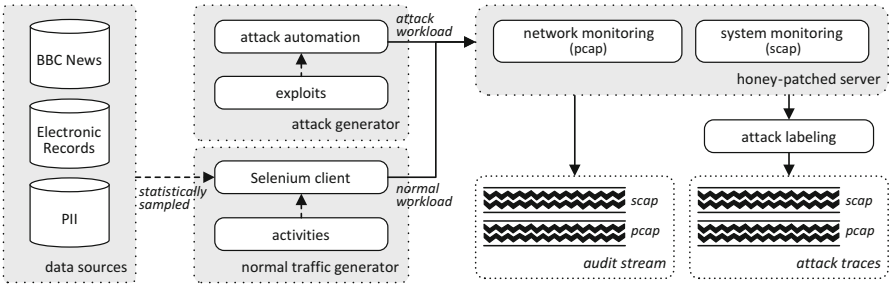

Fig. 8.3 Web traffic generation and testing harness

Algorithm 1 describes the voting approach for Ens-SVM. For each instance in the monitoring stream, if both Bi-Di and N-Gram agree on the predictive label (line 7), Ens-SVM takes the common classification as output (line 8). Otherwise, if the classifiers disagree, Ens-SVM takes the prediction with the highest SVM confidence (line 10). Confidence is rated using Platt scaling [17], which uses the following sigmoid-like function to compute the classification confidence:

$$P(y = 1|x) = \frac{1}{1 + \exp(Af(x) + B)} \tag{8.1}$$

where y is the label, x is the testing vector, $f(x)$ is the SVM output, and A and B are scalar parameters learned using maximum likelihood estimation (MLE). This yields a probability measure of how much a classifier is confident about assigning a label to a testing point.

8.6 Evaluation Testbed

Objective, scientific evaluation of cyber deceptions is often very difficult, because evaluations on live attackers tend to be subjective (there is usually no way to know whether an anonymous attacker was genuinely deceived or just "playing along"), anecdotal (samples of hundreds or thousands of provably distinct attackers are required to draw quantifiable conclusions), and impossible to replicate. Much of the prior work in this space has been criticized on those grounds. Our work therefore offers a more rigorous evaluation methodology, which demonstrates that objectively quantifiable success metrics for IDSes significantly improve when exposed to deception-enhanced data, and the experimental results are reliably reproducible at large sample sizes.

8.6.1 Realistic Web Traffic Generation

To demonstrate the practical advantages and feasibility of deception-enhanced intrusion detection, we built a web traffic generator and test harness. Figure 8.3 shows an overview of our evaluation testbed, inspired by prior work [8]. It streams realistic *encrypted* legitimate and malicious workloads onto a honey-patched web server, resulting in labeled audit streams and attack traces (collected at decoys) for training set generation.

Legitimate Data Normal traffic is created by automating complex user actions on a typical web application as shown in Table 8.2, leveraging *Selenium* to automate user interaction with a web browser (e.g., clicking buttons, filling out forms, navigating a web page). We generated web traffic for 12 different user activities (each repeated 200 times), including web page browsing, e-commerce website navigation, blog posting, and interacting with a social media web application. The setup included a CGI web application and a PHP-based Wordpress application hosted on a monitored Apache web server. To enrich the set of user activities, the Wordpress application was extended with *Buddypress* and *Woocommerce* plugins for social media and e-commerce web activities, respectively.

To create realistic interactions with the web applications, our framework feeds from online data sources, such as the BBC text corpus, online text generators for personally identifiable information (e.g., usernames, passwords), and product names to populate web forms. To ensure diversity, we statistically sampled the data sources to obtain user input values and dynamically generated web content. For example, blog title and body is statistically sampled from the BBC text corpus, while product names are picked from the product names data source.

Table 8.2 Summary of synthetic data generation

Normal workload summary		
Activity	Application	Description
Post	CGI web app	Posting blog on a guestbook CGI web application
Post	Wordpress	Posting blog on Wordpress
Post	Wordpress Buddypress plugin	Posting comment on social media web application
Registration	Wordpress Woocommerce plugin	Product registration and product description
E-commerce	Wordpress Woocommerce plugin	Ordering of a product and checkout
Browse	Wordpress	Browsing through a blog post
Browse	Wordpress Buddypress	Browsing through a social media page
Browse	Wordpress Woocommerce plugin	Browsing product catalog
Registration	Wordpress	User registration
Registration	Wordpress Woocommerce plugin	Coupon registration

Attack Data Attack traffic is generated based on real vulnerabilities as shown in Table 8.3. For this evaluation, we selected 16 exploits for eight well-advertised, high-severity vulnerabilities. These include CVE-2014-0160 (heartbleed), CVE-2014-6271 (shellshock), CVE-2012-1823 (improper handling of query strings by PHP in CGI mode), CVE-2011-3368 (improper URL validation), CVE-2014-0224 (change cipher specification attack), CVE-2010-0740 (malformed TLS record), CVE-2010-1452 (Apache mod_cache vulnerability), and CVE-2016-7054 (buffer overflow in OpenSSL with support for ChaCha20-Poly1305 cipher suite). In addition, nine attack variants exploiting CVE-2014-6271 (shellshock) were created to carry out different malicious activities (i.e., different attack payloads), such as leaking password files and invoking bash shells on the remote web server. These vulnerabilities are important as attack vectors because they range from sensitive data exfiltration to complete control and remote code execution. To emulate realistic attack traffic, we interleaved attacks and normal traffic following the strategy of Wind Tunnel [8].

Dataset The traffic generator is deployed on a separate host to avoid interference with the testbed server. To account for operational and environmental differences, our framework simulates different workload profiles (according to time of day), against various target configurations (including different background processes and server workloads), and network settings, such as TCP congestion controls. In total, we generated 12 GB of (uncompressed) network packets and system events over a period of three weeks. After feature extraction, the training data comprised 1200 normal instances and 1600 attack instances. Monitoring or testing data consisted of 2800 normal and attack instances gathered at unpatched web servers, where the distribution of normal and attack instances varies per experiment.

8.6.2 Experimental Results

Using this dataset, we trained the classifiers presented in Sect. 8.5.2 and assessed their individual performance against test streams containing both normal and attack workloads. In the experiments, we measured the true positive rate (*tpr*) where true positive represents the number of actual attack instances that are classified as attacks, false positive rate (*fpr*) where false positive represents the number of actual benign instances classified as attacks, accuracy (*acc*), and F_2 score of the classifier, where the F_2 score is interpreted as the weighted average of the precision and recall, reaching its best value at 1 and worst at 0. An RBF kernel with $Cost = 1.3 \times 10^5$ and $\gamma = 1.9 \times 10^{-6}$ was used for SVM [15].

Detection Accuracy To evaluate the accuracy of intrusion detection, we tested each classifier after incrementally training it with increasing numbers of attack classes. Each class consists of 100 distinct variants of a single exploit, as described in Sect. 8.6.1, and an n-class model is one trained with up to n attack classes. For example, a 3-class model is trained with 300 instances from 3 different attack classes. In each run, the classifier is trained with 1200 normal instances and $100 * n$ attack instances where $n \in [1, 16]$ attack classes. In addition, in each run, we execute ten experiments where the attacks are shuffled in a cross-validation-like fashion and the average is reported. This ensures training is not biased towards any specific attacks.

Table 8.3 Summary of attack workload

#	Attack type	Description	Software
1	CVE-2014-0160	Information leak	OpenSSL
2	CVE-2012-1823	System remote hijack	PHP
3	CVE-2011-3368	Port scanning	Apache
4–10	CVE-2014-6271	System hijack (7 variants)	Bash
11	CVE-2014-6271	Remote password file read	Bash
12	CVE-2014-6271	Remote root directory read	Bash
13	CVE-2014-0224	Session hijack and information leak	OpenSSL
14	CVE-2010-0740	DoS via NULL pointer dereference	OpenSSL
15	CVE-2010-1452	DoS via request that lacks a path	Apache
16	CVE-2016-7054	DoS via heap buffer overflow	OpenSSL

Testing on Decoy Data The first experiment measures the accuracy of each classifier against a test set composed of 1200 normal instances and 1600 uniformly distributed attack instances gathered at decoys. Figure 8.4a–b presents the results, which serve as a preliminary check that the classifiers can accurately detect attack instances resembling the ones comprised in their initial training set.

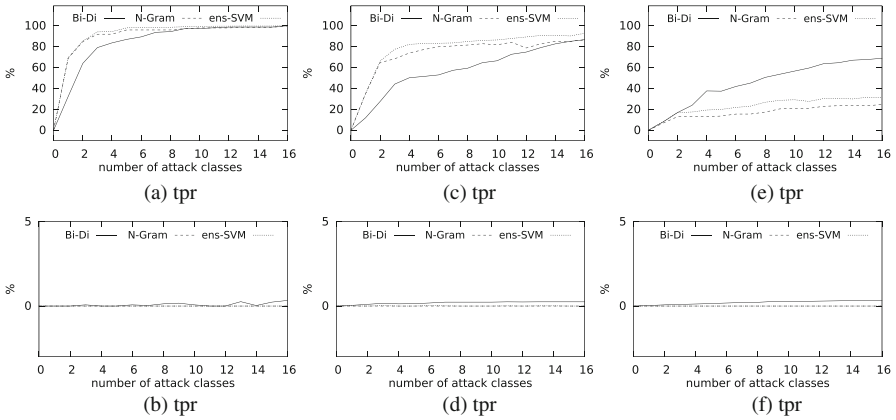

Fig. 8.4 Classification accuracy of Bi-Di, N-Gram, and Ens-SVM for 0–16 attack classes for (**a**)–(**b**) training and testing on decoy data, (**c**)–(**d**) training on decoy data and testing on unpatched server data, and (**e**)–(**f**) training on regular-patched server data and testing on unpatched server data

Testing on Unpatched Server Data The second experiment also measures each classifier's accuracy, but this time the test set was derived from monitoring streams collected at regular, *unpatched* servers, and having a uniform distribution of attacks. Figure 8.4c–d shows the results, which indicate that the detection models of each classifier generalize beyond data collected in decoys. This is critical because it demonstrates the classifier's ability to detect previously unseen attack variants. Our framework thus enables administrators to add an additional level of protection to their entire network, including hosts that cannot be promptly patched, via the adoption of a honey-patching methodology.

The results also show that as the number of training attack classes increases—which are proportional to the number of vulnerabilities honey-patched—a steep improvement in the true positive rate of both classifiers is observed, reaching an average *tpr* of above 92% for the compounded Ens-SVM, while average false positive rate in all experiments remained below 0.01%. This demonstrates the positive impact of the *feature-enhancing* capabilities of deceptive application-level attack responses like honey-patching.

Training on Regular-Patched Server Data To compare our approach against analogous, standard IDSes that do not employ deception, we trained each classifier on data collected from non-deceptive, regular-patched servers, and tested them on the unpatched server data, using the same set of attacks. Figure 8.4e–f shows the results, which outline the inherent challenges of traditional intrusion detection models on obfuscated, unlabeled attack traces. Unlike honey-patches, which capture and label traces containing patterns of successful attacks, conventional security patches yield traces of failed attack attempts, making them unfit to reveal patterns of attacks against unpatched systems.

Table 8.4 Base detection rates for approximate targeted attack scenario ($P_A \approx 1\%$) [11]

Classifier	*tpr*	*fpr*	*acc*	F_2	*bdr*
OneSVM-Bi-Di	55.56	13.17	68.96	59.69	4.09
OneSVM-N-Gram	84.77	0.52	91.07	87.09	62.22
Bi-Di	86.69	0.25	92.29	89.02	77.79
N-Gram	86.52	0.01	92.30	88.89	98.98
Ens-SVM	**92.76**	**0.01**	**95.86**	**94.12**	**99.05**

Baseline Evaluation This experiment compares the accuracy of our detection approach to the accuracy of an unsupervised outlier detection strategy, which is commonly employed in typical intrusion detection scenarios [9], where labeling attack data is not feasible or prohibitively expensive. For this purpose, we implemented two *one-class SVM* classifiers, *OneSVM-Bi-Di* with a polynomial kernel and $\nu = 0.1$ and *OneSVM-N-Gram* with a linear kernel and $\nu = 0.001$, using Bi-Di and N-Gram models for feature extraction, respectively. We fine-tuned the one-class SVM parameters and performed a systematic grid search for the kernel and ν to get the best results.

One-class SVM uses an unsupervised approach, where the classifier trains on one class and predicts whether a test instance belongs to that class, thereby detecting *outliers*—test instances outside the class. To perform this experiment, we incrementally trained each classifier with an increasing number of *normal* instances, and tested the classifiers after each iteration against the same unpatched server test set used in the previous experiments. The results presented in Table 8.4 highlight critical limitations of conventional outlier intrusion detection systems: reduced predictive power, lower tolerance to noise in the training set, and higher false positive rates.

In contrast, our supervised approach overcomes such disadvantages by automatically streaming onto the classifiers labeled security-relevant features, without any human intervention. This is possible because honey-patches identify security-relevant events at the point where such events are created, and not as a separate, *post-mortem* manual analysis of traces.

8.6.3 Discussion

Methodology Our experiments show that just a few strategically chosen honey-patched vulnerabilities accompanied by an equally small number of honey-patched applications provide a machine learning-based IDS sufficient data to perform substantially more accurate intrusion detection, thereby enhancing the security of the entire network. Thus, we arrive at one of the first demonstrable measures of value for deception in the context of cybersecurity: its utility for enhancing IDS data streams.

Supervised Learning Our approach facilitates supervised learning, whose widespread use in the domain of intrusion detection has been impeded by many challenges involving the manual labeling of attacks and the extraction of security-relevant features [9]. Our results demonstrate that the language-based, active response capabilities provided via application-level honey-patches significantly ameliorates both of these challenges. The facility of deception for improving other machine learning-based security systems should therefore be investigated.

Intrusion Detection Datasets One of the major challenges in evaluating intrusion detection systems is the dearth of publicly available datasets, which is often aggravated by privacy and intellectual property considerations. To mitigate this problem, security researchers often resort to synthetic dataset generation, which affords the opportunity to design test sets that validate a wide range of requirements. Nonetheless, a well-recognized challenge in custom dataset generation is how to capture the multitude of variations and features manifested in real-world scenarios [6]. Our evaluation approach builds on recent breakthroughs in dataset generation for IDS evaluation [8] to create statistically representative workloads that resemble realistic web traffic, thereby affording the ability to perform a meaningful evaluation of IDS frameworks.

8.7 Conclusion

This chapter outlined the implementation and evaluation of a new approach for enhancing web intrusion detection systems with threat data sourced from deceptive, application-layer, software traps. Unlike conventional machine learning-based detection approaches, our framework incrementally builds models of legitimate and malicious behavior based on audit streams and traces collected from these traps. This augments the IDS with inexpensive and automatic security-relevant feature extraction capabilities. These capabilities require no additional developer effort apart from routine patching activities. This results in an effortless labeling of the data and supports a new generation of higher-accuracy detection models.

8.8 Exercises

8.8.1 Software Engineering Exercises

∗ 1. Give an example of a high-profile software exploit cyberattack whose impact has been reported recently in the news, and for which the cyber-deceptive software techniques described in this chapter might have proved beneficial, if deployed. Based on any technical details available, advise how such a defense might have helped in that scenario, and discuss potential implementation issues or risks involved.

** 2. For each of the following vulnerability types, find an example patch for one such vulnerability (e.g., from MITRE CWE), and then write code that reformulates the patch into a honey-patch. In your honey-patch, use the function call *fork_to_decoy()* to indicate where your code would fork the attacker's connection to a decoy environment. Remember, a good honey-patch implementation should not impact legitimate users!

   a. buffer underflow/overflow (overwrite, overread, underwrite, or underread)
   b. C format string vulnerability
   c. TOCTOU (time-of-check / time-of-use) vulnerability
   d. SQL injection
   e. XSS (cross-site scripting)

*** 3. Install older (non-fully patched) versions of OpenSSL and Apache, and identify from a CVE list some of the unpatched vulnerabilities. Implement a honey-patch for any of the CVEs. Invite classmates to operate as a red team to penetrate your server. Were they able to distinguish the decoy environment from any successful compromise? Would any data collected from detected attacks potentially help your server resist subsequent exploit attempts?

8.8.2 Machine Learning Exercises

* 1. Given a set of data traces with packet data, what type of features can be extracted from packets?
* 2. Similarly, given a set of data traces with system calls, what type of features can be extracted to train a machine learning classifier?
* 3. Given the confusion matrix in Table 8.5, and defining *positives* to be alarms raised by the defense, calculate the following metrics: accuracy, FPR, and TPR.

Table 8.5 Confusion matrix

Total no. of instances: 160		Actual classes	
		Attack	Benign
Predicted classes	Attack	20	30
	Benign	10	100

** 4. Why is false positive rate (FPR) important in evaluating machine learning based intrusion detection systems?
** 5. Implement an IDS using support vector machine that leverages packet data traces to classify and detect attack in collected data traces. For this exercise, you can follow the following steps:

   - *Extract packet information:* Use the `dpkt` python toolkit to extract packet information, such as length, count, packet direction, and packet time.

- Build a histogram of the packet length for each trace. Each trace will generate an instance to train your classifier.
- After generating your dataset, use the `Scklearn` python machine learning module to build an SVM classifier.

*** 6. Implement an ensemble classifier using support vector machine to leverage both packet data and system call data to classify attack traces. You can follow the steps described in previous question to complete this exercise.

** 7. Calculate the following metrics with the classifier you implemented in exercises 5 and 6: accuracy and FPR. How do you explain the significance of the FPR compared to the accuracy?

*** 8. Run your algorithm on data collected from a honey-patched system (see software engineering exercises 2–3) and compare the performance to the data collected on a system with no honey-patch.

*** 9. Based on software engineering exercise 3, implement your own data collection mechanism that captures packet and system call level data. Apply your machine learning implementation from exercise 5 on the data traces collected. Compare your performance with the supplied data. To complete this exercise, you can use tcpdump (already installed on Linux systems) to collect packet trace data and sysdig[1] to collect system call data. To reduce noise in your data collection, run each attack independently and collect the associated traces. Remember to run each attack and trace collection multiple times to account for variations in system operation.

References

1. K. Alnaami, G. Ayoade, A. Siddiqui, N. Ruozzi, L. Khan, and B. Thuraisingham. P2V: Effective website fingerprinting using vector space representations. In *Proceedings of the IEEE Symposium on Computational Intelligence*, pages 59–66, 2015.
2. F. Araujo and K. W. Hamlen. Compiler-instrumented, dynamic secret-redaction of legacy processes for attacker deception. In *Proceedings of the USENIX Security Symposium*, 2015.
3. F. Araujo and K. W. Hamlen. Embedded honeypotting. In S. Jajodia, V. Subrahmanian, V. Swarup, and C. Wang, editors, *Cyber Deception: Building the Scientific Foundation*, chapter 10, pages 195–225. Springer, 2016.
4. F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 942–953, 2014.
5. S. Axelsson. The base-rate fallacy and its implications for the difficulty of intrusion detection. In *Proceedings of the ACM Conference on Computer and Communications Security*, pages 1–7, 1999.
6. M. H. Bhuyan, D. K. Bhattacharyya, and J. K. Kalita. Network anomaly detection: Methods, systems and tools. *IEEE Communications Surveys & Tutorials*, 16(1):303–336, 2014.

[1] https://sysdig.com/opensource/sysdig/install.

7. A. L. Blum and P. Langley. Selection of relevant features and examples in machine learning. *Artificial Intelligence*, 97(1):245–271, 1997.
8. N. Boggs, H. Zhao, S. Du, and S. J. Stolfo. Synthetic data generation and defense in depth measurement of web applications. In *Proceedings of the International Symposium on Recent Advances in Intrusion Detection*, pages 234–254, 2014.
9. V. Chandola, A. Banerjee, and V. Kumar. Anomaly detection: A survey. *ACM Computing Surveys*, 41(3):15, 2009.
10. D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering*, 13(2):222–232, 1987.
11. D. Dudorov, D. Stupples, and M. Newby. Probability analysis of cyber attack paths against business and commercial enterprise systems. In *Proceedings of the IEEE European Intelligence and Security Informatics Conference*, pages 38–44, 2013.
12. K. P. Dyer, S. E. Coull, T. Ristenpart, and T. Shrimpton. Peek-a-boo, I still see you: Why efficient traffic analysis countermeasures fail. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 332–346, 2012.
13. P. Garcia-Teodoro, J. Diaz-Verdejo, G. Maciá-Fernández, and E. Vázquez. Anomaly-based network intrusion detection: Techniques, systems and challenges. *Computers & Security*, 28(1):18–28, 2009.
14. Juniper Research. The future of cybercrime and security: Financial and corporate threats and mitigation, 2015.
15. A. Panchenko, L. Niessen, A. Zinnen, and T. Engel. Website fingerprinting in onion routing based anonymization networks. In *Proceedings of the Annual ACM Workshop on Privacy in the Electronic Society*, pages 103–114, 2011.
16. A. Patcha and J.-M. Park. An overview of anomaly detection techniques: Existing solutions and latest technological trends. *Computer Networks*, 51(12):3448–3470, 2007.
17. J. C. Platt. Probabilistic outputs for support vector machines and comparisons to regularized likelihood methods. In *Advances in Large Margin Classifiers*, pages 61–74. MIT Press, 1999.
18. R. Sommer and V. Paxson. Outside the closed world: On using machine learning for network intrusion detection. In *Proceedings of the IEEE Symposium on Security & Privacy*, pages 305–316, 2010.
19. Symantec. Internet security threat report, vol. 21, 2016.
20. C.-F. Tsai, Y.-F. Hsu, C.-Y. Lin, and W.-Y. Lin. Intrusion detection by machine learning: A review. *Expert Systems with Applications*, 36(10):11994–12000, 2009.
21. E. Vasilomanolakis, S. Karuppayah, M. Mühlhäuser, and M. Fischer. Taxonomy and survey of collaborative intrusion detection. *ACM Computing Surveys*, 47(4), 2015.
22. T. Wang, X. Cai, R. Nithyanand, R. Johnson, and I. Goldberg. Effective attacks and provable defenses for website fingerprinting. In *Proceedings of the USENIX Security Symposium*, 2014.
23. J. Yuill, D. Denning, and F. Feer. Using deception to hide things from hackers: Processes, principles, and techniques. *Journal of Information Warfare*, 5(3):26–40, 2006.

Chapter 9
HONEYSCOPE: IoT Device Protection with Deceptive Network Views

Reham Mohamed, Terrence O'Connor, Markus Miettinen, William Enck, and Ahmad-Reza Sadeghi

Abstract The emergence of IoT has brought many new device manufacturers to the market providing novel products with network connectivity. Unfortunately, many of these new entrants to the market lack security engineering experience and focus heavily on time-to-market. As a result, many home and office networks contain IoT devices with security flaws and no clear path for security updates, making them attractive targets for attacks, e.g., recent IoT-centric malware such as Mirai. In this chapter, we discuss a network centric approach to protecting vulnerable IoT devices. We describe a system called HoneyScope, which seeks to achieve two goals. First, each IoT device has a different view of its local network, which limits the damage when a device is compromised. Second, virtual IoT devices are created to confuse and deceive attacker with sophisticated motivations (e.g., fake WiFi connected cameras). To achieve these goals, HoneyScope uses an SDN-based security gateway to create virtualized views of the network and nodes therein providing fine-grained control over the communications that individual devices may have.

9.1 Introduction

One of the big challenges facing IoT networks in homes and small offices—in comparison to traditional networks—is their relatively high susceptibility to security threats. Numerous heterogeneous IoT devices are being deployed in small office, home office (SOHO) networks, broadening the potential attack surface for

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

R. Mohamed · M. Miettinen · A.-R. Sadeghi
Technische Universität Darmstadt, Darmstadt, Germany

T. O'Connor · W. Enck (✉)
North Carolina State University, Raleigh, NC, USA
e-mail: whenck@ncsu.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_9

adversaries, as many new IoT devices are affected by inherent security vulnerabilities. This is due to hundreds of new IoT manufacturers entering a market that is largely untapped and considered unsaturated, providing players that are first-to-market opportunities to gain considerable market share. As many manufacturers' focus is therefore on quickly shipping their products, this leaves little time and resources to focus on proper security design, implementation, and testing of new device models. Moreover, manufacturers are often producing low-cost devices (such as connected light bulbs or smart plugs) with hardly any or no budget at all for security. This results in many IoT device vendors shipping products that contain security vulnerabilities that are relatively easy to exploit by knowledgeable attackers.

Due to the newness of IoT, there is a lack of regulations and laws governing development and production of IoT devices. There exist also no dominant security standards that all device vendors would adhere to.

In most cases, the responsibility of securing devices is therefore left to end-users. Access to devices is typically controlled using default easy-to-guess credentials, but vendors do not force users to update them during the device on-boarding procedure. Many users will, however, not understand the risks (among others) associated with such default passwords, and many of them will not even care, leaving devices at the mercy of potential attackers. Other devices, on the other hand, are designed to work in a plug and play mode by default, leaving no room for users to modify security settings, even if they would like to.

Another major security threat is given by benign but intrusive functionality of devices that can possibly breach the privacy of users by, e.g., recording private conversations, taking photos, or recording videos and automatically uploading such information to the cloud without the user's consent. Already now smart voice assistants like Amazon Alexa and Google Assistant have been, intentionally and accidentally alike, triggered by viral ads like Burger King's ad that forced Google Assistant to recite the definition of the Whopper from Wikipedia [8], or other incidents like one affecting Amazon's Alexa, which—as explained by Amazon—misheard the wake word during the conversation of a wife with her husband before sending a recording of it to the wife's colleague [1].

9.1.1 Principle of "Need to See"

At the root of our approach to deception in SOHO networks is the principle of *need to see*, which is a variant of the traditional principle of *need to know* used in environments with sensitive information. A key observation is that while future SOHO networks may be filled with tens of IoT devices, most devices do not interact with one another. First, many devices communicate exclusively with the Internet. Second, for devices that communicate within the network, interaction is often with a small set of *controller* devices, such as smartphones and smart speakers (e.g., Alexa, Google Home). Therefore, there is no need for most IoT devices to *see* one another on the network.

The simplest security policy in any setting is that of strict isolation. It is simple to express and enforce. WiFi isolation is built into all commodity routers and access points. In fact, most hotels use WiFi isolation to ensure that guests can access the Internet, but not interact with one another directly. However, WiFi isolation is not suitable for SOHO networks. First, SOHO networks contain traditional devices where isolation impedes functionality, e.g., desktops, laptops, printers, and network attached storage (NAS) devices. Second, some intra-network communication is needed for controller devices to coordinate actions with IoT devices. Third, some IoT devices may be designed to work directly with one another, such as those from the same manufacturer (e.g., WeMo, D-Link) or using the same standardized protocols (e.g., HomeKit). Therefore, the *need to see* in SOHO environments is more complex than strict isolation.

9.1.2 Deception Through Network Views

WiFi isolation is a degenerative type of network view. That is, each device can only see itself and the network gateway. Consider the more general model, where there exists a policy for each network device that defines which other network devices it can see. More formally, let N be the set of devices on the network. Each $n \in N$ has a policy $P_n \in \mathscr{P}(N)$, where $\mathscr{P}(N)$ is the power set of N.[1] The policy P_n defines a specific network view for n.

The network view presented to an adversary, whether it be a compromised or misbehaving device, influences its perspective of the attack surface of the network. There is also no requirement that each $n \in N$ is a physical device. For example, N can include a virtual WiFi baby monitor camera that simply plays a feed on a loop. Such virtual deception devices can be used in several ways. Consider the scenario where one of the users' IoT devices is compromised and is used as an attack pivot, which proceeds to scan for other vulnerable network devices. First, the virtual deception device can act as a honeypot. Under normal scenarios, the virtual deception device should not receive any network connections from real devices. Second, the virtual deception device may make the adversary believe she has control of a real device, e.g., watching a live feed of a baby. Gaining access to the baby camera may be the adversary's goal, and the simulated feed may keep the adversary from burglarizing the home.

Network views provide usability in addition to deception. For example, the network view policy can exclude the virtual deception devices from the view of controller devices (e.g., smartphones). In this way, users will not be confused by the potentially many virtual deception devices.

[1] The power set of S is the set of all subsets of S.

9.1.3 HONEYSCOPE

In the remainder of this chapter, we present HONEYSCOPE, a security framework for small office and home office (SOHO) networks built on top of the concept of network views. HONEYSCOPE is a protection layer built on top of the local network and provides a fine-grained control over the communications of individual IoT devices in the network. HONEYSCOPE uses software defined networking (SDN) technologies [9] to realize *device-* and *device-group*-specific views of the network that *reduce the attack surface* against vulnerable devices in the network, *contain effects of device infections* in case of successful device compromise, and enforce effective measures for blocking unwanted release of contextual data from within the network to the outside. At the same time HONEYSCOPE acts as a deceptive obfuscation layer that decouples the network appearance of devices from their actual physical interfaces, providing the network owner fine-grained control over how devices and the network topology are presented to other devices and to the outside.

9.2 Design of HONEYSCOPE

The core idea on which HONEYSCOPE builds is to provide *device-group-specific* views on the local IoT network in order to be able to control the exposure of devices and contain potential security incidents, without adversely affecting the benign functionality of devices. HONEYSCOPE seeks to realize this through following design principles.

Grouping of Devices According to Their Vendor Due to the lack of proper interoperability standards for IoT devices, many device manufacturers offer vendor-specific interoperability solutions, often supported by vendor-specific cloud-based back-end services. Due to this approach, many IoT devices seldom—if ever—have the need to communicate with devices that do not fall within their vendor-specific device category. HONEYSCOPE uses this property to compartmentalize the local IoT network by placing individual IoT devices into vendor-specific groups. By limiting communication to happen only within the vendor group (and potentially the vendor cloud service), the attack surface of IoT devices inside the group can be effectively reduced and security attacks across vendor groups effectively mitigated.

Grouping of Devices According to Their Functionality The differentiation between device groups can also happen based on properties other than the device's manufacturer. For a number of specific applications like smart lighting there already exist to some degree protocols that enable interoperability between devices of different vendors (e.g., ZigBee Light Link). To support such (future) functionality, HONEYSCOPE supports orthogonally also groupings that are based on the declared functionality of devices, if this functionality requires interoperability across vendor-specific groups. As mentioned, this grouping is orthogonal to the vendor-specific groups, meaning that devices can be in parallel member of a vendor-specific group and one or more functionality-specific groups.

Vulnerable Device Isolation HONEYSCOPE isolates devices that are known to have vulnerabilities in a specific group with a very constrained view on other devices in the network. The vulnerable device group shall not have access to any other devices in the local IoT network, nor shall external devices be able to communicate with devices in this group. The only exception is access to vendor cloud services that are necessary to maintain the benign functionality of the device. This has twofold goals: (1) to protect vulnerable devices from being compromised by malware or active attacks originating from outside the network, and (2) to protect the rest of the IoT network's devices in case an adversary manages to compromise a vulnerable device.

Deceptive Views of the Network HONEYSCOPE also provides the possibility to create various deceptive views of the real, physical network. Each physical device in the IoT network has a virtual representation in the HONEYSCOPE virtualization layer. This allows the network owner to define how devices are perceived by other devices and the outside network. The deceptive views may also include virtual representations of non-existent devices to provide *honey views* of the network that allow to completely obfuscate the true topology of the actual physical network setup.

9.2.1 HONEYSCOPE *Implementation Approaches*

There are multiple options for HONEYSCOPE to implement network views.

Option 1: Multiple (V)LANs One way to realize the group separation of HONEYSCOPE would be to use a number of LANs or VLANs to represent the different groups of the local network. However, this would be incompatible with existing discovery protocols, as devices on separate LANs would be unable to discover each other. For example, mDNS, which is used by HomeKit, assumes that all devices are on the same LAN.

Option 2: Multiple SSIDs A second option is to use separate SSIDs of a WiFi access point. The main drawback of using this approach lies in the increased complexity of the bootstrapping process, as each device would need to be provisioned on the correct SSID. This would in general be a too complex task to be handled correctly by regular users. For example, many IoT devices have a bootstrap process that includes a smartphone app that automatically copies the SSID and WPA2 password from the smartphone. Therefore, the user would need to navigate multiple SSIDs when setting up devices.

Option 3: Use of SDN Technologies for Group Separation Software defined network (SDN) technology such as OpenFlow provides a unified network abstraction in which all devices in the network are controlled by the SDN controller, allowing fine-grained control over network connectivity of individual nodes. SDN provides the most versatility to programmatically implement network views. It also allows the architecture to extend to an arbitrary number of access points, which is increasingly

common as newer WiFi protocols such as 802.11ac provide faster performance when devices are close to the access point.

HONEYSCOPE uses SDN technology to control the topology of the IoT network in order to provide maximal flexibility to manage nodes in the local network and to realize a virtualization layer implementing the security design of HONEYSCOPE and enforce its security policies. In effect, the local WiFi access point acts as a security gateway controlled by a HONEYSCOPE SDN controller. The task of the security gateway is to realize the HONEYSCOPE virtualization layer by enforcing network connectivity policies defining the device group topology and create required virtual representations of nodes belonging to the honey views comprising the deception aspect of the HONEYSCOPE framework.

Erickson et al. [10] have developed a mechanism for identifying vulnerable devices and blocking them from accessing and attacking other devices in the network. Using this mechanism, they eliminate Man-In-the-Middle attacks at the link and service discovery layers. To realize this mechanism, they use a different SSID and WPA password for each device. However, this approach does not work properly with IoT networks as the IoT devices get the network information from their companion app on the mobile device of the user. HONEYSCOPE would overcome this drawback by using one SSID, and one LAN for all devices in the network using the SDN technology.

9.2.2 HONEYSCOPE *Network Structure*

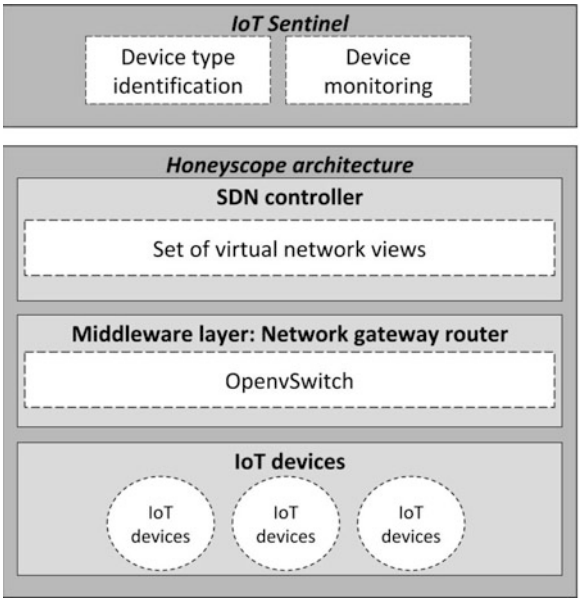

Fig. 9.1 HONEYSCOPE layered architecture

The HONEYSCOPE architecture consists of three main layers as shown in Fig. 9.1. The top level is the HONEYSCOPE controller: an SDN controller that controls and manages the HONEYSCOPE router controlling the local IoT network. This Open vSwitch-based router uses the OpenFlow protocol to communicate with the HONEYSCOPE SDN controller to manage the IoT devices in the local network.

The SDN controller is responsible for creating the network views for each IoT device connected to the local LAN. This is done by grouping devices into vendor- and function-specific groups as discussed above in Sect. 9.2 and applying a specified *network view* on each group in the network. The created virtual network views are unidirectional views. For example, if group x can view and send information to group y, this does not necessarily mean that group y can access and view group x.

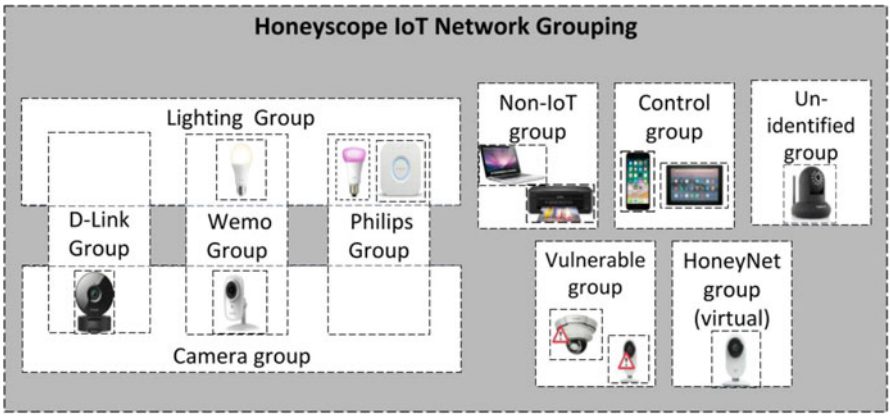

Fig. 9.2 HONEYSCOPE grouped IoT network

9.2.2.1 HONEYSCOPE Groups

There are five default groups that are created by the HONEYSCOPE controller in addition to the device vendor-specific groups and the function-specific groups as shown in Fig. 9.2:

- **Control group:** This group consists of controlling devices such as smartphones or tablets that should be able to control and therefore have access to any of the IoT devices in the network. Devices in this group can view and have access to any device in the IoT network. It is the only group in which devices have a virtual network view that looks like the real, physical network. All devices from any other groups can communicate with any device in this group as well, except for devices in the vulnerable devices group. Also, devices in this group don't see the deceptive virtual devices in the HoneyNet group. Typically, the smartphone of the home owner is the main device mapped to this group.

- **Vulnerable devices:** This group contains any detected vulnerable IoT devices in the local network. Devices in this group cannot view and do not have access to any devices in other groups in the network except the HoneyNet group. They are thus isolated from other IoT devices. The virtual network view for this group only contains a unidirectional connection to it from the control group and a bidirectional communication possibility with the HoneyNet group. Only the control group can send updates, messages, or information to this group.
- **Unidentified devices:** Devices that cannot be identified by the controller, such as devices that have just been released onto the market and are therefore yet unknown to the system providing device identification, are added to this group.
- **HoneyNet group:** This group's main task is to create virtual devices to deceive the intruder, like the virtual baby camera discussed in Sect. 9.1.2 providing a fake instance of a device to confuse the adversary in case it is able to intrude into the network.
- **Vendor-specific groups:** When a device is joining the network, the SDN controller identifies its type based on its communication behavior, and adds it to the corresponding manufacturer group. Devices in each group can communicate with each other but don't have access to devices from other groups they are not members of. An example of a vendor specific could be, e.g., the smart home automation gadgets provided by D-Link. These include smart power plugs, motion sensors, water sensors, IP cameras, door and window sensors, and alarm sirens. All of these devices can be configured to work together using rules defined in a cloud-assisted vendor-specific smartphone app. It makes therefore sense to place all such devices from this vendor to the same group.
- **Function-specific groups:** Devices from different manufacturers may need to be placed in function-specific groups when functional interoperability is needed. For example, devices providing smart lighting (e.g., smart light bulbs, switches) would be placed in a lighting-specific group in order to enable the light switch of one manufacturer to control smart light bulbs of another vendor.
- **Non-IoT devices group:** In the local network there are also other non-IoT devices like desktops, laptops, NAS devices, etc. that should be treated separately from IoT devices, as their functionality is much richer than that of typical IoT devices. Such devices are therefore added to a dedicated group that contains non-IoT and legacy devices. Devices in this group have a flexible virtual network view configurable by the user allowing the devices to view and interact with other IoT or non-IoT devices. The view for each device is defined according to the need of this device to communicate with other devices (e.g. interacting with printers, or specific IoT devices in the network) and can be derived, e.g., based on the set of applications installed on the device. For example, a non-IoT device like a laptop should be able to view other laptops in the network, in addition to printers, VoIP phones, cameras, smartphones, and tablets. Smartphones and tablets are located in the control group, while printers and VoIP phones are located in the non-IoT devices group. The laptop may also need access to IP cameras located in the function-specific "camera" group. This way, the laptop will have a virtual

network view that allows it to have access to the non-IoT group, the control group, and the camera group.

As shown in Fig. 9.2, there are intersection points between vendor-specific and function-specific groups. For example, in this network, there is a Philips Hue smart device in both the lighting group (function-specific) and the Philips group (vendor-specific). The function-specific groups contain devices from different vendors but they share the same function and usage, so they need to communicate together in a separate group.

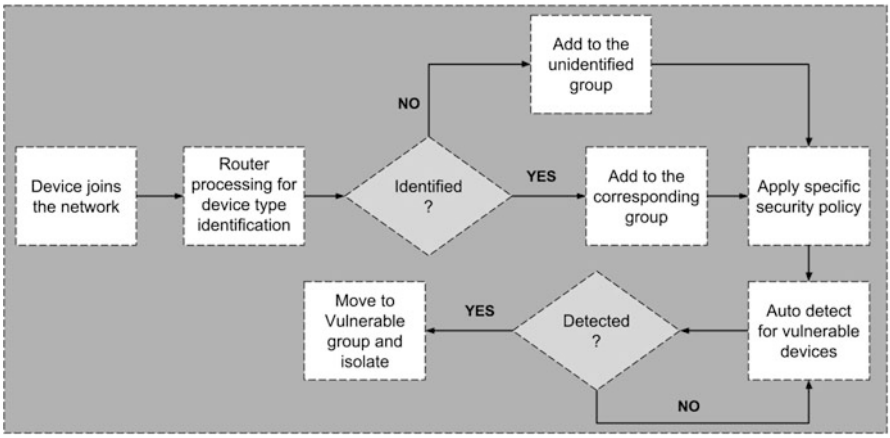

Fig. 9.3 Grouping and isolation processes

9.2.3 Device Type Identification

Most devices joining the network do not explicitly advertise what type of device they are. They need therefore to be identified before assigning them to HONEYSCOPE groups. When a device joins the local IoT network, the SDN controller identifies its type by using the developed IoT sentinel model by Miettinen et al. [12] for unknown device type identification. IoT sentinel is based on identifying the device type by profiling the communication behavior of the device and using machine learning classifiers to identify its device type. It uses a database that contains pre-captured behaviors for most of the current IoT devices in the market, and from this database, the device type is determined and defined. The main idea of IoT sentinel is to create device fingerprints by monitoring the communication behavior of the new added device during the setup phase. From this generated fingerprint, IoT sentinel is able to map the device to its corresponding device type using machine learning-based classifiers.

If the device is identified successfully, it is added to its corresponding group in the local network: a vendor-specific group and possibly one or more function-specific groups.

For example, if the device was detected to be a Philips Hue light, it will be added to the Philips Hue group as well as the lighting group. When the device is added to the corresponding group, the SDN controller creates a virtual network view that matches the groups the device is added to. If IoT sentinel fails to successfully identify the joining device, the device will be added to the unidentified group which contains all unidentified devices in the network. The joining device will be able to view the control group, the HoneyNet group, and any other devices in the unidentified group.

The HONEYSCOPE IoT network will be defined by different customized network views for each device added to it according to this device type identification process.

IoT sentinel will notify the SDN controller if any vulnerabilities become known for any of the IoT devices registered with the SDN controller. For obtaining this information, IoT sentinel utilizes an IoT security service, which aggregates network-wide information about known vulnerabilities associated with particular IoT device types. If a vulnerability is detected that affects a particular device, it will be moved from its group, regardless whatever this group is, to the vulnerable group. This way, the device will be isolated from other devices in the network not to affect any of them until the device vulnerability has been removed, e.g., by applying an appropriate firmware patch. The device will not have any access to any devices in the local network. However, devices in the control group will have a unidirectional connection to it, e.g., in order to check device status or send updates and other messages.

9.3 HONEYSCOPE Components

9.3.1 HONEYSCOPE *Controller*

There are many SDN controllers that can be used with HONEYSCOPE. However, we selected RYU controller [6] which is based on Python, because of its support for the higher and newest versions of the OpenFlow protocol [11], it supports the 1.5 OpenFlow standard. In addition to that, it has the Nicira extensions for OpenFlow matching [5]. Although Nicira is a vendor specific implementation, it is luckily implemented in OpenvSwitch. It provides some additional criteria to match on. We have developed a Ryu application to implement the management and control of the whole network.

9.3.2 HONEYSCOPE *Security Gateway*

For the gateway router, our prototype uses a Linksys WRT1900AC [2] router running a modified OpenWRT firmware. OpenWRT gives developers flexibility in

creating the network design and flashing the desired operating system image to it [4]. The Linksys router is considered as one of the most stable hardware that can be used with OpenWRT. However, one of the main drawbacks of using it is its limited memory. OpenvSwitch [3] is installed on the Linksys router to enable using SDN and OpenFlow. Instead of the normal bridge in the router, an OVS bridge is added to work properly with SDN. OpenvSwitch is a multiplayer virtual switch that provides more automation and programmability. SDN needs such automation to be able to perform the separation between the control plane and the data plane smoothly.

As mentioned in the previous section, one of the main goals of HONEYSCOPE is to divide the local IoT network to groups in which devices are visible to each other while they are invisible to any other devices out of this specific group's range. Figure 9.3 describes the process of grouping and isolating the new added devices. When a device joins the local IoT network, the SDN controller examines the packets coming from and to this device and processes the device's behaviors to be able to identify the device type accordingly.

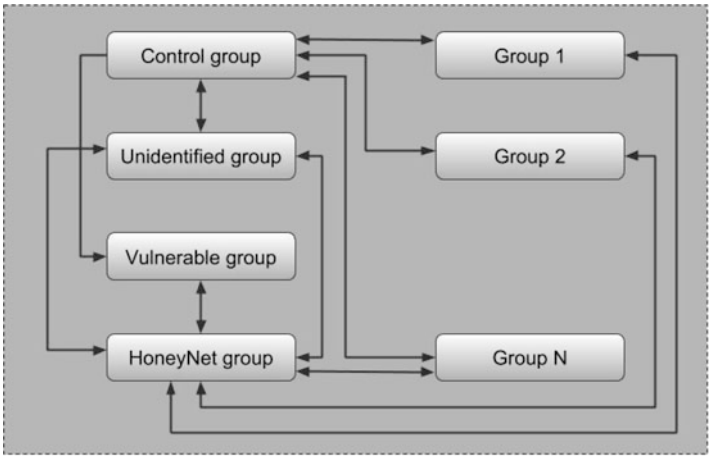

Fig. 9.4 Communication between network groups

9.3.3 Communication Between Network Groups

As shown in Fig. 9.4, the direction of the arrows represents how the communication goes between any two groups, for example, the communication between the control group and the vulnerable group is unidirectional from the control to the vulnerable group, as communication in the other direction is denied. As shown, there is no communication between different vendor-specific or function-specific groups, nor between any of these groups and the vulnerable group. Each of these groups cannot

view or access other groups. There is no need for these devices to communicate and by isolating them we reduce the attack surface of devices in the local IoT network.

The communication between the control group and the vulnerable group requires special arrangements. As mentioned, the control group has access to the vulnerable group through unidirectional communication. However, the vulnerable group doesn't have access to the control group and can't send any packets to devices in it. However, in some use cases like when devices in the vulnerable group need to be configured or software updates need to be installed on them, there needs to be a way for the controlling devices to discover devices in the vulnerable group.

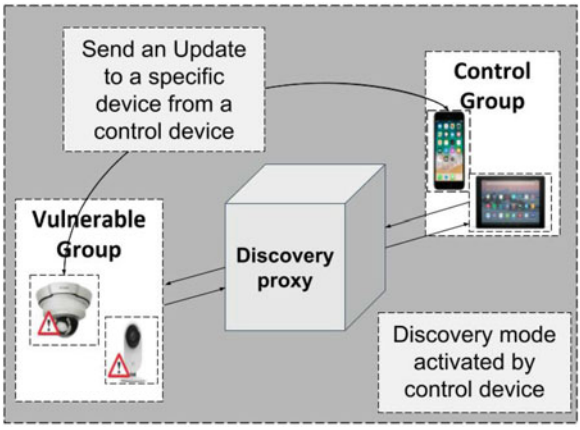

Fig. 9.5 Communication between control and vulnerable groups

Software updates for most IoT devices are facilitated using a specific smartphone application to get updates from the manufacturer servers and install them onto the device. The mobile application downloads the firmware update and uploads it to the corresponding IoT device. Note that in some cases the IoT device connects directly to the manufacturer's servers without the help of its mobile application. In such cases access to the control group may not be required.

To be able to perform software updates with the help of the controlling device, IoT devices in the vulnerable group use multicast and broadcast-based discovery protocols to allow the controlling device (e.g., smartphone, or tablet of the user) to be aware of them.

To enable the controlling device to discover vulnerable devices, a virtual discovery proxy is used for facilitating limited communication for this discovery purpose from the vulnerable group towards the control group as shown in Fig. 9.5, demonstrating how the communication between these groups is handled. The discovery proxy is activated by the controlling device (e.g., by activating a special configuration/update mode of the system). When activated, the proxy will forward broadcast and multicast messages from devices in the vulnerable group to the specific controlling device that activated the discovery mode. Thus, the controlling device can discover the presence of IoT devices while limiting their communication

to any other devices in the local IoT network. We envisage that the user can activate the discovery mode of HONEYSCOPE with the help of a smartphone application on the controlling device, thereby activating the discovery proxy. The communication between both groups can be summarized as follows:

- Communication from any control device to the vulnerable devices: this is granted by direct communication from the control device to the vulnerable group.
- Communication from vulnerable devices group to the control group: This communication is denied. It can only temporarily enabled through the discovery proxy when the controlling device initiates a special discovery mode. It acts like a "virtual VPN" between the controlling device and vulnerable group, strictly limiting the delivery of multicast and broadcast packets of devices in the vulnerable group to the controlling group only.

9.3.4 Case Study

Here we will discuss an exemplary scenario to demonstrate how HONEYSCOPE adds a level of security to the local IoT network. Assume there is a D-Link IP camera in the local IoT network that is susceptible to be infected by an IoT malware like Mirai [7]. At first, when this IP camera is joining the network, it will be identified and added to the D-Link and camera groups. Only D-Link devices, in addition to all cameras in the network, can view and have access to this D-Link camera.

Once the IoT security service notifies HONEYSCOPE that the camera is vulnerable to, e.g., Mirai, the SDN controller will move the camera to the vulnerable devices group and remove it from both the D-Link and camera groups.

When it is moved to the vulnerable group, the infected camera will have a new limited virtual network view in which it can't view any devices in the local network except the virtual devices in the HoneyNet group, and can be viewed by the controlling devices as well. This way, other devices in the D-Link and camera groups will be protected from the vulnerable device should it be infected by the IoT malware.

The communication between the controlling devices and the vulnerable devices is unidirectional: this is to enable the controlling device, e.g., to send software updates for fixing the vulnerability of the camera. After that, the camera can again be added back to the D-Link and camera groups.

9.4 Conclusion

This chapter has provided an overview of the architecture, design, and the deception function of HONEYSCOPE. HONEYSCOPE is based on creating virtual deceptive network views for each IoT device in the local IoT network. SDN technology is used to manage the network by using an SDN controller that is responsible for creating these deceptive network views for each device according to its identified

device type. For device type identification, the IoT sentinel system is used. Through the use of deceptive network views, HONEYSCOPE is able to provide a higher layer of security to the local IoT network against both external and internal attacks.

9.5 Hands-on Exercises

1. **(Intermediate)** In this exercise, you will use the Mininet SDN emulation environment (http://mininet.org/) to implement the core network views concept behind HONEYSCOPE. The best way to become familiar with Mininet is to go through the tutorial. Once you are familiar with Mininet, construct the topology shown in the left half of Fig. 9.6. The goal of this exercise is to construct the network views policy shown in the right half of Fig. 9.6. To accomplish this, you may wish to modify Mininet's "Learning Switch" tutorial to enforce access control based on hard-coded MAC addresses. Use the OpenFlow protocol to program a switch to perform custom packet forwarding. Implement a software defined network controller that can read frame and packet source and destination fields in order to implement the HONEYSCOPE network views policy. Test network visibility using the `ping` command. Note that for this exercise, you only need to worry about unicast traffic. Finally, you may choose to use the Pox controller, which is the default with Mininet, or choose another controller such as Ryu, ONOS, or OpenDaylight. Mininet can be configured to use a *remote controller* (e.g., a controller outside the control of Mininet).

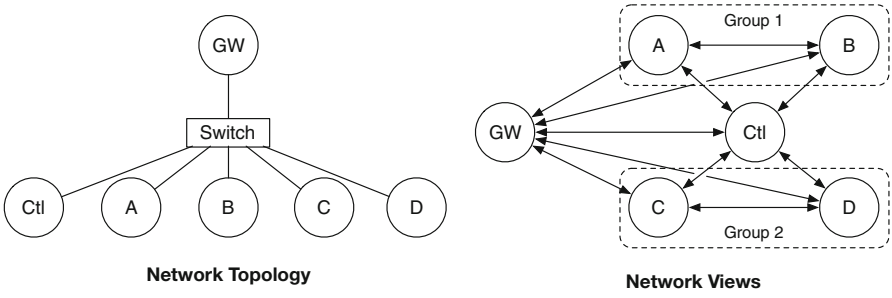

Fig. 9.6 Network topology and views for exercises. GW is the gateway and Ctl is a controlling device such as a smartphone. Arrowheads indicate network visibility. For example, GW and Ctl can see A, B, C, and D; however, A and B cannot see C and D

2. **(Advanced)** In this exercise, you will extend Exercise 1 to real hardware. To complete this exercise, you will need a router/access point capable of running OpenWRT/LEDE (https://openwrt.org/) and a Raspberry Pi. Start by setting up OpenWRT with OpenVSwitch (OVS) and your controller running on the Raspberry Pi. The WiFi SDN project at Helsinki is a good starting place (https://wiki.helsinki.fi/display/WiFiSDN/Software-Defined+Wi-Fi+Networks+with+Wireless+Isolation). Once the data plane and control plane is setup, port your

solution to Exercise 1 to this environment. Note the controller will only see one port: `wlan0`. However, by enabling WiFi isolation mode, all of the network traffic will be forced through the soft-switch, allowing the controller to define flow-mod rules that restrict which devices can receive packets from one another. Again, for this exercise, only worry about unicast traffic.

3. **(Advanced)** In the previous two exercises, you only considered unicast traffic. However, many IoT devices depend on multicast protocols (e.g., HomeKit uses mDNS), which may leak information between network groups. Further, IoT discovery protocols (e.g., SSDP) rely on multicast protocols for advertisement and discovery of network services, providing information about IoT device applications and services outside the scope of HONEYSCOPE policies. Extend Exercise 2 to also mediate multicast traffic. To test your solution, explore the use of Avahi (https://www.avahi.org/) and nss-mdns (https://github.com/lathiat/nss-mdns) from Linux. Alternatively, macOS devices advertise services via mDNS (aka Bonjour). Hint: consider making copies of multicast packets and sending the copies to hosts allowed by the policy.

References

1. Amazon's Alexa recorded private conversation and sent it to random contact. https://www.theguardian.com/technology/2018/may/24/amazon-alexa-recorded-conversation. Accessed: 2018-06-20.
2. Linksys WRT 1900AC. https://www.linksys.com/us/p/P-WRT1900AC/. Accessed: 2018-06-03.
3. OpenvSwitch. https://www.openvswitch.org/. Accessed: 2018-06-03.
4. OpenWRT. https://openwrt.org/. Accessed: 2018-06-03.
5. RYU Nicira extensions. http://ryu.readthedocs.io/en/latest/nicira_ext_ref.html. Accessed: 2018-06-03.
6. RYU SDN controller. https://osrg.github.io/ryu/. Accessed: 2018-06-03.
7. Manos Antonakakis, Tim April, Michael Bailey, Matt Bernhard, Elie Bursztein, Jaime Cochran, Zakir Durumeric, J Alex Halderman, Luca Invernizzi, Michalis Kallitsis, et al. Understanding the Mirai botnet. In *USENIX Security Symposium*, 2017.
8. H. Chung, M. Iorga, J. Voas, and S. Lee. Alexa, can I trust you? *Computer*, 50(9):100–104, 2017.
9. ONF Market Education Committee et al. Software-defined networking: The new norm for networks. *ONF White Paper*, 2012.
10. Jeremy Erickson, Qi Alfred Chen, Xiaochen Yu, Erinjen Lin, Robert Levy, and Z. Morley Mao. No one in the middle: Enabling network access control via transparent attribution. In *Proceedings of the 2018 on Asia Conference on Computer and Communications Security*, ASIACCS '18, pages 651–658, New York, NY, USA, 2018. ACM.
11. Nick McKeown, Tom Anderson, Hari Balakrishnan, Guru Parulkar, Larry Peterson, Jennifer Rexford, Scott Shenker, and Jonathan Turner. OpenFlow: Enabling innovation in campus networks. *SIGCOMM Comput. Commun. Rev.*, 38(2):69–74, March 2008.
12. Markus Miettinen, Samuel Marchal, Ibbad Hafeez, N. Asokan, Ahmad-Reza Sadeghi, and Sasu Tarkoma. IoT Sentinel: Automated device-type identification for security enforcement in IoT. In *Proc. 37th IEEE International Conference on Distributed Computing Systems (ICDCS 2017)*, June 2017.

Part IV
Malware Deception

Chapter 10
gExtractor: Automated Extraction of Malware Deception Parameters for Autonomous Cyber Deception

Mohammed Noraden Alsaleh, Jinpeng Wei, Ehab Al-Shaer, and Mohiuddin Ahmed

Abstract The lack of agility in cyber defense gives adversaries a significant advantage for discovering cyber targets and planning their attacks in stealthy and undetectable manner. While it is very hard to detect or predict attacks, adversaries can always scan the network, learn about countermeasures, and develop new evasion techniques. Active Cyber Deception (ACD) has emerged as effective means to reverse this asymmetry in cyber warfare by dynamically orchestrating the cyber deception environment to mislead attackers and corrupting their decision-making process. However, developing an efficient active deception environment usually requires human intelligence and analysis to characterize the attackers' behaviors (e.g., malware actions). This manual process significantly limits the capability of cyber deception to actively respond to new attacks (malware) and in a timely manner.

In this chapter, we present a new analytic framework and an implemented tool, called *gExtractor*, to analyze the malware behavior and automatically extract the deception parameters using symbolic execution in order to enable the automated creation of cyber deception plans. The deception parameters are environmental variables on which attackers depend to discover the target system and reach their goals; yet, they can be reconfigured and/or misrepresented by the defender in the cyber environment. Our *gExtractor* approach contributes to the scientific and system foundations of reasoning about autonomous cyber deception. Our prototype was developed based on customizing symbolic execution engine for analyzing Microsoft

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

M. N. Alsaleh
Eastern Michigan University, Ypsilanti, MI, USA

J. Wei (✉) · E. Al-Shaer · M. Ahmed
Software and Information Systems, University of North Carolina at Charlotte, Charlotte, NC, USA
e-mail: jwei8@uncc.edu

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_10

Windows malware. Our analysis of over fifty of recent malware instances shows that *gExtractor* has successfully identified various critical parameters that are effective for cyber deception.

10.1 Introduction

Malware attacks have evolved to be highly evasive against prevention and detection techniques. It has been reported that at least 360, 000 new malicious files were detected every day and one ransomware attack was reported every 40 seconds in 2017 [27]. This reveals severe limitations in the prevention and detection technologies, such as anti-virus, perimeter firewalls, and intrusion detection systems. Active Cyber Deception (ACD) has emerged as an effective defense for cyber resilience [13] that can corrupt and steer adversaries' decisions to: (1) *deflect* them to false targets, (2) *distort* their perception about the environment, (3) *deplete* their resources, and (4) *discover* their motives, tactics, and techniques [1, 14].

Advanced cyber threats often start with intensive reconnaissance by interacting with cyber to learn the true values of its parameters, such as keyboard layout, geolocation, hardware ID, IP address, service type, OS/platform type, and registry keys to discover vulnerable targets and achieve their goals. We call such parameters "Critical Parameters." ACD can be particularly effective during this phase by providing false perceptions about the configuration of the cyber environment to achieve the deception goals [1]. There are two key mechanisms to accomplish this: (1) parameter *mutation* to frequently change the ground truth (i.e., the real value of the system parameter) of cyber configuration such as IP address [16] or route [12], or (2) parameter *misrepresentation* to change only the value returned to the attacker (i.e., the ground truth is intact). We call such critical parameters that can be feasibly and cost-effectively mutated or misrepresented the "Deception Parameters." Figure 10.1b shows the two deception mechanisms with respect to the environment parameter p. It shows that the adversary knowledge about p was falsified by either changing p to a new value (mutation) or lying about its true

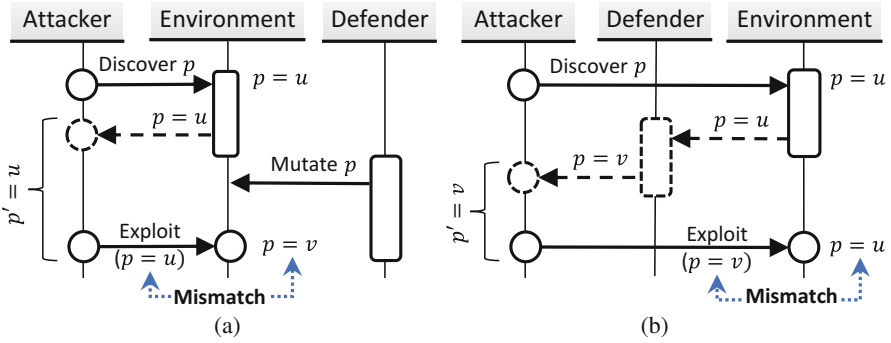

Fig. 10.1 Attacker's dependency on system parameters. (**a**) Mutation. (**b**) Misrepresentation

value (misrepresentation). Both mechanisms are needed in cyber deception because mutation can be infeasible or too expensive, and misrepresentation can sometimes be uncovered.

An effective planning of cyber deception requires a sequence of mutations and/or misrepresentations of deception parameters in order to steer the adversary to the desired deception goals. The key challenge that we address in this research is to identify the most appropriate *deception parameters* against any arbitrary malware by symbolically executing and analyzing the malware binary. In this chapter, we present a systematic approach and automated tool to analyze malware binary code and identify (1) *"what"* deception parameters are the most appropriate to accomplish the deception goals and (2) *"how"* to effectively mutate or misrepresent their values. This requires deception-oriented analysis of malware behaviors that goes beyond existing dynamic analysis that is usually tailored towards attack detection. Thus, we extended the existing dynamic analysis and symbolic execution frameworks to track the execution of malware symbolically, and analyze system and library API calls that particularly entail interactions with the cyber environment. We then identify the deception parameters that can impact the malware decision-making. Since these parameters can be interdependent and they might exhibit varying deception accuracy and cost, our analysis guarantees the selection of consistent sets of parameters that can obtain resilient and cost-effective deception plans. We summarize our contributions as follows:

- We present *gExtractor*, a deception-oriented malware symbolic execution analysis that intercepts and tracks the malware interactions with the environment, and maps them to specific deception parameters.
- We developed formal constraints to extract deception parameters that constitute consistent, resilient, and cost-effective deception.
- We implemented *gExtractor* and evaluated it using various types of malware codes. Our evaluation demonstrates the ability of *gExtractor* to extract effective deception parameters, as manually verified by experts.

While some previous work, such as Moving Target Defense (MTD) [12, 18, 36, 38, 41, 44, 45] and decoy technologies [3, 25, 37] attempt to invalidate attacker's perception, the deception parameters and schemes were engineered manually, which significantly limits the ability for creating deception actions automatically against novel malware. The ultimate goal of this research is to automate active cyber deception against novel malware attacks. Thus, unlike IPS/IDS, our objective is to detect and deceive, rather than detect and block, by enabling the malware to execute in a real or virtual deception environment configured based on the extracted deception parameters. To the best of our knowledge, this is the first work that uses automated reasoning to infer deception parameters based on malware analysis.

We implemented *gExtractor* on top of the Selective Symbolic Execution engine (S^2E) [7] with the assist of our custom plugins to execute malware in a real controlled environment, intercept system and library API calls, mark the relevant symbolic information, and collect contextual logs. This facilitates the construction of a comprehensive malware behavior model that covers all possible execution paths. The constructed model is further processed to: (1) prune out execution paths

that are not relevant to the deception goals, and (2) eliminate the don't-care symbolic variables that have no impact on the deception goals.

To demonstrate the value of our approach, we used *gExtractor* to analyze over fifty recent malware instances. We present in the evaluation section representatives of three major malware families: Cryptocurrency-mining malware, ransomware, and Credential-stealing malware. For each representative, we modeled its behavior, extracted candidate deception parameters, and show how they can be used to design different deception schemes for different goals. Our case studies presented in this chapter show that our approach can discover effective deception parameters. For example, the bitcoin miner case study (Sect. 10.4.1) reveals multiple parameters including Windows Script Host engine, win32_processor WMI (Windows Management Instrumentation) class that can be used to deflect the malware by misinforming false platform type, and the bitcoin hashing results that can be used to corrupt the results in mining pool and depleting the adversary resources (i.e., score).

The gExtractor program can be incorporated in the production systems to automatically analyze, extract, and deceive malware without human intervention. Although there are various techniques to trigger the detection of malware such as signature analysis [10], behavior analysis [28], decoy software [37], and decoy bugs [3], the focus of this research is on extracting the deception parameters to automate creating deception actions and schemes.

The rest of this chapter is organized as follows: In Sect. 10.2, we present the process of constructing the malware behavior model by executing the malware symbolically. Then, we present our approach to refine the malware behavior models and extract candidate deception parameters in Sect. 10.3. Real malware case studies are presented in Sect. 10.4. Finally, we discuss the related works and conclude in Sects. 10.5 and 10.6, respectively.

10.2 Modeling Attack Behavior Using Binary Symbolic Execution

To extract the complete behavior of a cyberattack, we execute its binaries (i.e., malware) symbolically and build a model that represents its behavior with respect to selected system parameters. Given that the correct set of system parameters is selected, symbolic execution can cover all relevant execution paths. Before going through the technical steps of the symbolic malware analysis, we present the attack behavior model.

10.2.1 Attack Behavior Model

The *attack behavior model* describes how the attack behaves based on the results of its interaction with the environment. The malware interacts with its environment through system and user library APIs characterized by their input and output

arguments. Some of these arguments may be attacker-specific variables and cannot be controlled by the environment, while other parameters can be reconfigured or misrepresented. We assume that a mapping between the selected system or library APIs' arguments and the corresponding parameters in the environment, such as *files*, *registry entries*, *system time*, *processes*, *keyboard layout*, *geolocations*, *hardware ID*, *C&C*, *Internet connection*, *IP address* or *host name*, and *communication protocols*, is given. For example, the *from* argument of the *recvfrom* API can be mapped to a system parameter that represents the IP address of the sender machine.

We define the attack behavior model as a graph of *Points of Interaction (PoI)* nodes and *Fork* nodes. The PoIs refer to the points in the malware control flow at which the malware interacts with the environment by invoking system or library APIs. The fork nodes represent the points in the control flow at which the malware makes a control decision based on the results of its interactions with the environment.

To formally model the attack behavior, let Γ be the set of selected System and Library APIs, where each $\gamma \in \Gamma$ takes a fixed number of input augments ($I_\gamma = \{i_1, \ldots, i_n\}$) and returns a fixed number of output arguments ($O_\gamma = \{o_1, \ldots, o_m\}$). We model the attack behavior as the directed graph $G = (P, \mu, E, \nu)$, where:

- P is a set of nodes that represent the PoI and Fork nodes. The type function $\mu : P \rightarrow \{\boldsymbol{PoI}, \boldsymbol{Fork}\} \times (\Gamma \cup \emptyset)$ associates nodes with their types. If the node represents a PoI, μ further maps it to the appropriate system and library API from the set Γ.
- $E \subseteq P \times P$ is the set of edges that represents the dependency between the nodes in P. A directed edge $e = (p_i, p_j)$ is added from node p_i to node p_j if there is a control or data dependency between them. The dependency function $\nu : E \rightarrow \mathscr{L}_O$ associates each edge to a constraint expressed as a logic formula in the logic $\mathscr{L}_O$ with support for quantifier-free integer, real, and bit-vector linear arithmetic. Expressions in $\mathscr{L}_O$ are defined over the set of output arguments $O = \bigcup_{\gamma \in \Gamma} O_\gamma$.

In Fig. 10.2, we show an example of attack behavior model that represents a portion of the Blaster worm that delivers a copy of the worm to an exploited victim. Round nodes represent PoIs and square nodes represent fork points. The solid edges represent control dependency, while dashed ones represent data dependency. In this model, the worm first sends an instruction to a remote command shell process running on the exploited victim through the *send* library API, then it waits for a download request through the *recvfrom* API call. The attack code checks if these operations are executed successfully and terminates otherwise as depicted through the conditions shown on the outbound edges from the fork nodes 2 and 5. At node 7, the worm starts reading its executable file from the disk into a memory buffer, through *fread*, and sending the content of the buffer to the remote victim, through the *sendto* API. There is a data dependence between the third argument of the *sendto* call, which represents the number of bytes to transmit, and the return value of the *fread* call, which represents the number of bytes read from the worm file.

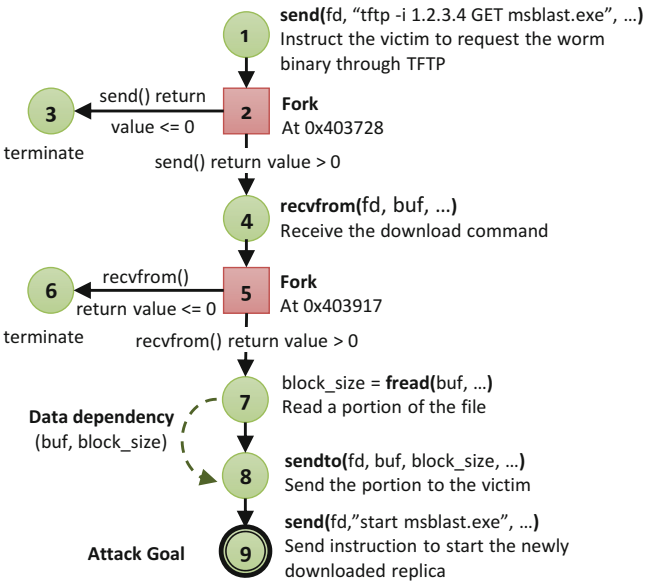

Fig. 10.2 Example of attack behavior model

10.2.2 Malware Symbolic Execution

We utilize the S^2E engine to symbolically execute malware binaries. The path coverage and the progress of the executed program depends on the correct marking of symbolic variables. Since we are interested in the interactions of the malware with its environment through selected system and library APIs, we intercept these calls and mark their output arguments as symbolic. This allows us to capture the malware decisions based on those arguments and track the corresponding execution paths. In the current version of our implementation, we select about 130 APIs that cover activities related to networking, file system and registry manipulation, system information and configuration, system services control, and UI operations.

Marking Symbolic Variables To mark the appropriate symbolic variables, we take advantage of the *Annotation* plugin provided by S^2E, which combines monitoring and instrumentation capabilities and executes user-supplied scripts, written in LUA language, at run time when a specific annotated instruction or function call is encountered. We define an annotation entry for each API. The annotation entry consists of the module name, the address of the API within the module, and the annotation function. We identified the module names and addresses using static/dynamic code analysis tools, such as IDA and Ollydbg. The annotation function is executed at the exit of the intercepted call. It reads the addresses of the return and output arguments of the call and marks the appropriate memory

locations and registers as symbolic. Note that output arguments may have different sizes and structures. Hence, we need custom scripts to mark each individual output argument of the intercepted APIs. The return values of APIs are typically held in the *EAX* register and we use special method provided by S^2E to mark its value as symbolic. It should be noted that system calls and user library APIs are invoked by all applications in the environment, not only the malware process. Therefore, our annotation functions check the name of the process that invokes them and ignore calls from irrelevant processes.

Building the Attack Behavior Model After preparing the appropriate annotation entries, we execute the malware using S^2E to collect the execution traces. We configured the annotation functions to record the arguments, the call stack, and other meta-data, such as the time-stamp and the execution path number for each intercepted system and library call. By design, S^2E intercepts branch statements whose conditions are based on symbolic variables and forks new states of the program for each possible branch. We collect the traces and branching conditions of all execution paths and build the attack behavior model as follows:

- We create a PoI node for each system or library API call logged by our annotation functions. Similarly, the traces contain special log entries for state forking operations. Those are used to create the *Fork* nodes in our model.
- For each node in the model, we add a control dependency edge from the node preceding it in the execution path. If the preceding node is a *Fork* node, the edge will be associated with a branching condition in terms of the symbolic variables.
- To capture the data dependency, we check the values of all the input arguments upon the entry of each API call. If the value is a symbolic expression, this implies that it is a transformation of previously created symbolic variables. Hence, we add a data dependency edge from the PoI nodes in which the symbols of the expression were created.

10.3 Deception Parameters Extraction

Given the attack behavior model generated through symbolic execution, we extract a set of system parameters that help in designing effective deception schemes to meet the deception goals. Recall that the attack behavior model describes the complete behavior of a malware with respect to selected system parameters. However, that does not mean that every parameter in the attack behavior model is a feasible candidate for deception. That is, mutating or misrepresenting its value may not be sufficient to successfully deceive the attacker. We analyze the attack behavior model to select the appropriate set(s) of deception parameters that can help in designing deception schemes without dictating particular ones.

We present the following four criteria that must be considered to decide on which parameters are appropriate for effective deception and which are not:

1. Goal Dependency. The selected deception parameters can directly or indirectly affect the outcomes of the attack in terms of whether the attacker can reach her goal. Hence, parameters that are used only in execution paths that do not lead to particular goals might be excluded.
2. Resilience. In cases where multiple attack paths lead to particular goals, selected parameters must provide deception in all the paths, not only one.
3. Consistency. The selected deception parameters must preserve the integrity of the environment from the attacker's point of view. As system parameters may be interdependent, deception schemes must take this into consideration, such that misrepresenting one parameter without misrepresenting its dependents accordingly does not disclose the deception.
4. Cost-Effectiveness. Although multiple parameters may exist in the execution paths leading to particular goals, mutating or misrepresenting different parameters may require different costs and provide different benefits from the defender's point of view. Defenders must select the most cost-effective set of parameters for deception.

The complete attack behavior model contains many execution paths that may not be relevant to our deception analysis. To extract the parameters that satisfy the deception parameters criteria, we (1) identify the set of execution paths that are relevant to deception, (2) eliminate the don't-care symbolic variables, and (3) select a set of the remaining parameters based on their cost.

10.3.1 Identifying Relevant Paths

Recall that deception is not about blocking attacks, rather, it is about misleading and forcing them to follow particular paths that serve the desired deception goals. Hence, the selection of relevant execution paths from the attack behavior model depends on the deception goal. Following our definitions of the four goals of deception, the paths relevant to distortion keep the malware misinformed about the environment to slow it down or force it to make more environment checks. This is reflected in the paths that exhibit aggressive interactions and queries with and about the environment. On the other hand, the relevant paths for depletion and discovery are those that lead the malware to interact with the remote master or adversaries, while paths in which the malware loses interest and abandons the system are relevant to the deflection goal.

Definition 10.1 (Relevant Paths) A relevant path with respect to a particular deception goal is an execution path that exhibits particular patterns of interactions with the environment that can be leveraged by the defender to achieve the deception goal.

Regardless of which deception goal is desired, it can be represented as a single call or a sequence of calls to system and library APIs leveraging existing tools that

identify specific behaviors through patterns of call sequences, such as [8, 33, 35]. Then, the PoI nodes in our attack behavior model will be used to identify the execution paths that exhibit that particular sequence of calls. By pruning out all other paths that do not exhibit the desired sequence, we end up with a portion of the original behavior model that contains only the paths relevant to the deception goal. In Fig. 10.3a, we show a simple example of an attack behavior model that has two paths, one leads to the desired goal and the other leads to attack termination. In this case, the left path is considered irrelevant and it will be pruned out. For a concrete real-world example, in order to deceive the FTP Credential-Stealer malware in Sect. 10.4.2 with honey FTP passwords, the environment must not run OllyDbg because otherwise the malware would follow an execution path irrelevant to the deception goal.

10.3.2 Eliminating Don't-Care Variables

To clarify this step, we need first to define the execution path constraints. A path constraint is a logical expression that captures the conditions on the selected symbolic variables that need to be met in order for the execution to follow that particular path. Recall that we associate a set of symbolic variables to each PoI node p in the attack behavior model ($p \in P, \mu(p) = PoI$), which correspond to its output arguments. Later in the execution, an expression will be generated for each branch at the following forking nodes in terms of the symbolic variables causing the fork. Those expressions are captured in the resulting edges of the fork nodes and mapped through the dependency function $\nu(.)$.

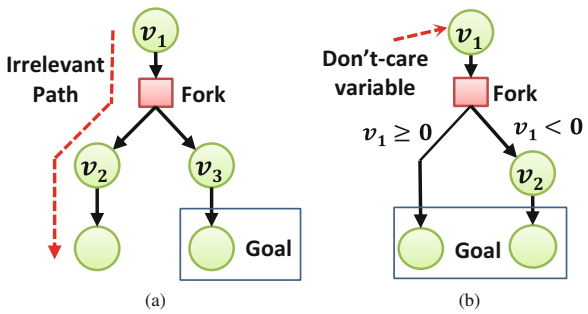

Fig. 10.3 Attack behavior model refinement. (**a**) Relevant path selection. (**b**) Don't-care elimination

The constraint of an entire path in our attack behavior model is simply the conjunction of the logical expressions associated with all the edges that belong to the path. Formally, let $\mathscr{P} = \{p_1, p_2, \ldots, p_n\}$, where $(\forall_{i \in [1,n]} : p_i \in P)$ represents a node path in the attack behavior model. Further, let $e_{i,j} \in E$ denote the edge between the nodes $p_i \in \mathscr{P}$ and $p_j \in \mathscr{P}$. The path constraint of the execution

path represented by $\mathscr{P}$ can be computed as $\bigwedge_{i\in[1,n-1]} \mu(e_{i,i+1})$, where $\mu(e_{i,i+1})$ is the expression of the edge $e_{i,i+1}$. We define the don't-care symbolic variables as follows.

Definition 10.2 (Don't-Care Variables) A don't-care variable with respect to particular deception goal is a symbolic variable that is part of one or multiple execution path constraints and its value is irrelevant to the desired deception goals.

As Fig. 10.3b illustrates, although there is a decision taken based on the symbolic variable v_1, the desired goal will be reached regardless of the variable's value. This makes v_1 a don't-care variable with respect to the desired goal and it can be excluded from further deception analysis.

After eliminating the irrelevant paths and the don't-care variables, we end up with refined path constraints for the relevant paths. The parameters extracted based on this refined model complies are guaranteed to satisfy the goal dependency. We further analyze the parameters and consider the cost of using each of them in deception and select the set of parameters that can achieve the desired deception goal using the minimum cost.

10.4 Evaluation

We analyzed over 50 recent malware variants using *gExtractor* and extracted candidate deception parameters for each of them. The variants we analyzed represent most common types of malware, including cryptocurrency-mining malware, ransomware, worms, spyware, and credential-stealing malware. To demonstrate that our systematic approach can indeed extract effective deception parameters, we selected two of the most prevalent malware, namely *Bitcoin Miner* and *FTP Credential-Stealer*. We discuss in detail the process of building the attack behavior model, extracting deception parameters, and we suggest deception schemes utilizing them.

10.4.1 Case Study I: Bitcoin Miner

We analyzed a recent bitcoin mining malware (MD5: efd1326e5289a935919 5120fd6c55290) that works in several stages. First, it drops and runs a Visual Basic (VB) script. Second, the script queries the Windows Management Instrumentation (WMI) service for the processor's information, such as the availability of GPU and the system architecture (32-bit or 64-bit), to download the right executable file for the target system from an external distribution website, *winxcheats.tk*. Third, the downloaded executable (*csrs.exe*) downloads yet another executable (*AudioHD.exe*) from *getsoed9.beget.tech*. The last program (*AudioHD.exe*) interacts with a bitcoin

mining pool server at *xmr.pool.minergate.com* to perform the mining on behalf of an account, which is hard-coded in the executable.

Malware Behavior Using *gExtractor*, we construct the behavior model of this malware (see the simplified version in Fig. 10.4), which covers the malware execution stages. We use common patterns of API calls to recognize significant malware activity. For example, the use of APIs that create new processes (e.g., *ShellExecuteExA* and *WshShell.Run*) indicates the beginnings of consecutive malware stages. Moreover, interacting with a well-known bitcoin mining pool server through networking and HTTP APIs reveals that one goal of this malware is to use the victim machine to perform bitcoin mining on behalf of the attacker. Therefore, we refine the malware behavior model by recognizing the relevant paths that lead to that goal and design deception schemes around it. After mapping the symbolic variables of the relevant paths' constraints to the system parameters, our analysis reveals the following necessary conditions for successful mining:

1. The file *C:\Windows\system32\wscript.exe* exists.
2. Windows Script Host (*WSH*) engine is enabled to run Visual Basic scripts [29].
3. *WMI* service and *Microsoft Win32 WMI* provider are running.
4. *win32_processor* WMI class reports the correct processor information.
5. The distribution website (http://winxcheats.tk) is available and hosts the executable file (under */miners/3/csrs.exe*).

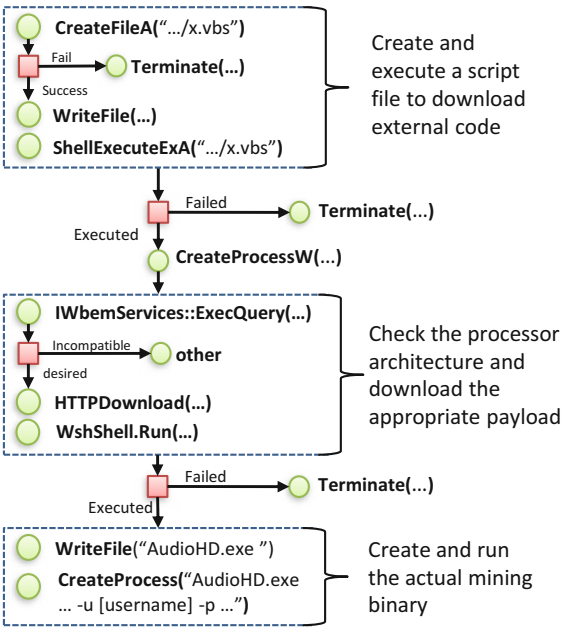

Fig. 10.4 Simplified behavior model of the bitcoin miner

6. The second distribution site (*getsoed9.beget.tech*) is available and hosts the second executable file (*AudioHD.exe*).
7. The bitcoin mining pool server (*xmr.pool.minergate.com*) is still running correctly.
8. The hard-coded account (*iden1930@mail.ru*) is authenticated successfully at the mining pool server.
9. The target system can run the file *AudioHD.exe* successfully.

To clarify how *gExtractor* facilitates the detection of such conditions, let us take condition 2 as an example. We mark the output parameter *"Buffer"* of the *RegQueryValueExW* API call, which is required to successfully complete the second stage of the malware, as symbolic. The API's input parameter, *hKey*, refers to the registry key *"HKLM\SOFTWARE\Microsoft\Windows Script Host\Settings\"* and the other input parameter *"ValueName"* is set to *"Enabled"*. Then, we observe that *"Buffer"* is used in a conditional jump, and in one path the message *"Windows Script Host access is disabled on this machine"* is displayed before the process terminates, while in another path we do not see this message. Alternatively, we see multiple queries to the WMI service. The first path will be regarded as irrelevant and pruned out by *gExtractor* and we will only consider the latter. Similarly, *gExtractor* can detect the dependence of this malware on the remaining conditions by tracking the decisions taken based on the associated symbolic variables and refining the behavior model.

Deception Parameters We analyzed the refined bitcoin miner behavior model with respect to different deception goals: deflection, distortion, depletion, and discovery. We identified the major deception parameter that satisfies our criteria defined in Sect. 10.3 and can be utilized to achieve each goal. In the following, we discuss a number of recommended deception schemes based on these parameters and we provide a summary with estimated deception costs in Table 10.1.

Table 10.1 Deception schemes against the bitcoin mining malware

Parameter	Goal	Deception action	Estimated cost
wscript.exe	Deflection	Replace it with a version that rewrites the input VB script for better protection	No CC; high OC; high DC
WSH engine	Discovery	Enable its capability to run VB scripts	Low CC; no OC; no DC
WMI class	Distortion	Change the way that it handles requests (e.g., returning misinformation about processors)	No CC; low OC if used on a honeypot; medium DC
The resulting hash	Depletion	Corrupt the resulting hash	No CC; no OC; high DC

(CC: configuration cost, OC: operation cost, DC: development cost)

Deflection Schemes For this purpose, we can enhance the designated script host *C:\Windows\system32\wscript.exe*. If the malicious VB script initiates a connection to a critical server, the enhanced *wscript.exe* can rewrite the VB script statement so that it connects to a honey server instead. This scheme could have high development cost because it requires a change to a Windows system utility, for which we do not have a source code. In terms of operation cost, this scheme can have high cost because it can confuse benign applications that need to run VB scripts, even if this is on a honeypot. However, it has little configuration cost because the current Windows OS does not have a configuration option to replace *wscript.exe* with an alternative version.

Discovery Schemes We can use the *Windows Script Host (WSH) engine* to construct a discovery strategy against malware that needs to run VB scripts. The WSH enables applications to run VB scripts and JScripts, and it provides a configuration option (via Windows registry) to enable/disable the VB script support. By enabling it, we can observe malware behavior through its VB scripts and have a better understanding of the malware. This strategy incurs only a low configuration cost.

Distortion Schemes Through the *"win32_processor WMI class"* parameter, we can construct a distortion scheme that returns misinformation about the system's architecture in order to confuse the malware (or the attacker behind the malware) who queries the *win32_processor class* interface. This strategy requires a change to the implementation of the *win32_processor class* interface, so there can be some development cost, and it can have a low operation cost if it is used on a honeypot.

Depletion Schemes The last parameter in Table 10.1 is the resulting hash, which the malware sends reliably to the mining pool server. We can create a depletion strategy by corrupting the results so that they become invalid. Deceiving the malware to send excessive invalid results that damages the attacker's reputation or causes financial losses (e.g., the mining pool server bans her account, freezes her mining wallet, or applies a penalty to her account). This scheme requires writing code to carry out the scheme, so it has some development costs. It has no operation cost because it modifies only the data of the malware.

We have experimentally confirmed the feasibility of depleting the attacker by corrupting the resulting hashes. In order to profit from mining on a victim machine, the attacker communicates with the mining pool server under her mining username in order to receive the credit. Therefore, at mining time, the attacker username must be present on the victim machine. We leveraged this fundamental "vulnerability" of this malware (i.e., revealing the mining username) for an effective deception. Based on our study of multiple mining pools, they establish various penalty policies for participants who submit invalid hashes. In Table 10.2, we summarize the negative impact of submitting invalid shares to several public mining pools. We can see that misbehaving users are often banned to some extent and their wallets can even be locked.

To prove the effectiveness of this depletion scheme, we built a tool that deliberately sends invalid hashes on behalf of a particular user. Different mining pool servers may implement different protocols to authorize jobs and submit resulting

Table 10.2 Negative impact of submitting invalid hashes

Mining pool	Banned	Payouts locked	Balance reduced
moneroocean.stream	For 1–10 min	No	No
xmrpool.net	Yes	No	No
supportxmr.com	Yes	Yes	No
www.viaxmr.com	Temporary	No	No
minergate.com	No	No	Yes
slushpool.com	Yes	No	No
moriaxmr.com	For 10 min	No	No
ratchetmining.com	For 10 min	No	No

hashes. However, most of them use a protocol called *STRATUM* over *HTTP* [39] and they define their own methods that can be used by the users to log into the server, get new jobs, and submit resulting hashes. We obtain the names and the required parameters of these methods along with other communication settings by analyzing the mining malware. Then, we submit a login request to the pool server. In response, the pool server returns a job and an Id that corresponds to the username. At that point, a legitimate miner will use the job data to generate a hash and send it back to the server. However, our tool will generate and send a random result instead, which will most likely be recognized as an invalid hash by the pool server.

To confirm that the mining pool servers penalize users who send invalid hashes, we created a user account at *Minergate* and sent a large number of random hashes on behalf of our new user. After submitting around 40, 000 invalid hashes, the account balance decreases from 0.00002398 to 0.00001973 which complies with the policy of *Minergate*. Figures 10.5 and 10.6 show the change in the account state before and after we sent the random hashes. Note that we have no means to verify whether a real attacker will be penalized if her username is used, because we do not have access to her account balance. However, when we perform the same actions using the attacker's username (iden1930@mail.ru) extracted from the malware analysis, the response from the mining pool server indicates that the submitted shares are detected as invalid. Since our account is penalized in compliance with the policy, we believe that the attacker's account should be punished as well.

Fig. 10.5 Account status before mining for Aeon coin

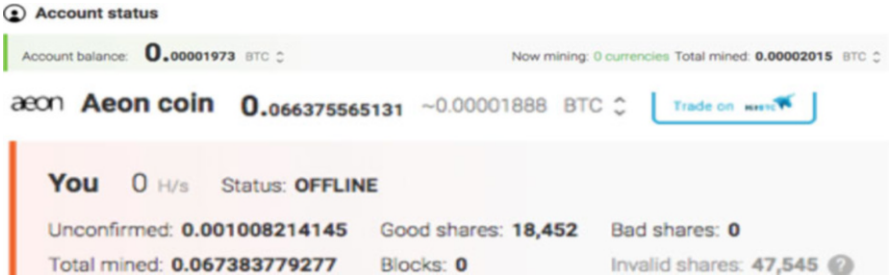

Fig. 10.6 Account status after submission of 40,000 invalid shares for Aeon coin

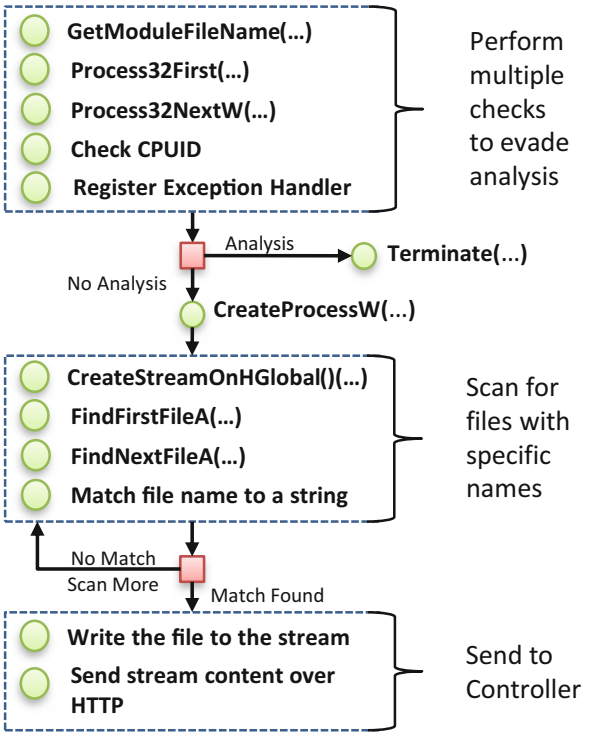

Fig. 10.7 Simplified model of the FTP credential-stealer

10.4.2 Case Study II: FTP Credential-Stealer

In this case study, we analyze a recent malware (MD5: 7572fb188134d141eac175 1b19b79a70) that scans the victim system for sensitive information, such as FTP login passwords and then sends the stolen information to a remote server.

Malware Behavior This malware consists of two processes. The first process employs multiple methods to check whether the malware is being analyzed, then terminates immediately if the checking result is positive. If no signs of analysis are detected, the first process drops and launches another piece of malware, which collects sensitive information from the victim system and sends it to a remote server under the adversary's control. A simplified version of the behavior model of this malware, generated by *gExtractor*, is shown in Fig. 10.7.

The first malware process is heavily obfuscated and employs multiple tricks to evade analysis and ensures a safe execution environment: (1) it tests whether the executable file's name contains any of the strings *"sandbox," "malware," "virus," or "self"*; (2) it scans the list of running processes for known dynamic analysis tools, such as *procmon.exe, procmon64.exe, procexp.exe, ollydbg.exe, and windbg.exe*; (3) it checks the *BeingDebugged* flag in its PEB (Process Environment Block) [11] at multiple places of its code section; (4) it checks whether it is running inside a virtual machine by matching the result of the *CPUID* instruction with *"KVMKVM," "XenVMM," "Microsoft Hv," and "pri hyperv"*; (5) it extracts the second malware binary from its resource section, decrypts it, and then uses process injection to launch it in a second process. If any sign of malware analysis is detected, the malware immediately terminates.

In the second process, the malware collects sensitive information from the Windows registry and the local file system and sends it to a remote site as follows. First, it searches certain Windows registry keys, which correspond to a specific list of FTP clients, for saved login credentials. For example, to steal information related to WinSCP, it searches for the key "Software\Martin Prikryl." If the key is found, it recursively enumerates the subkeys with the names *"HostName," "UserName," "Password," "RemoteDirectory,"* and *"PortNumber,"* reads their values, and stores them in a stream object for later exfiltration. Strings such as *"Software\Martin Prikryl"* and *"HostName"* are hard-coded in the malware. Second, it looks up files whose path contains particular patterns *(e.g., "WS_FTP," "LastSessionFile," "FTPRush," "Quick.dat," and "History.dat")*, and if any such file exists, it stores the file's path and content in the stream object. To optimize the search, it focuses on known folders, identified by their Constant Special Item ID List (CSIDL) values, such as the users' public documents, desktop, and local settings. It also searches the folders of installed applications discovered by their *"UninstallString"* registry values under the registry key *HKLM\SOFTWARE\Microsoft\Windows\CurrentVersion\Uninstall*.

After collecting the targeted information, the malware extracts the data from the stream object (via the API call sequence {*GetHGlobalFromStream*, *GlobalLock*}), then it constructs and sends an HTTP POST message to "http://www.luxzar.com/drake/november/omg/hot/gate.php". The HTTP communication is conducted by the API call sequence {*InternetCrackUrlA, ObtainUserAgentString, socket, connect, setsocketopt, send, closesocket*}.

Table 10.3 Defense strategies against FTP credential-stealing malware

Parameter	Goal	Deception action	Cost
Malware file name	Discovery	Avoid naming the malware "sandbox.exe," "malware.exe," "virus.exe," or "self.exe"	Low CC; no OC
Dynamic Analysis tools	Discovery	Rename the dynamic analysis tools	Low CC; no OC
CPUID result	Discovery	Deny that the true result is one of "KVMKVM," "XenVMM," "Microsoft Hv," and "pri hyperv"	Low CC; high OC; high DC
WinSCP registry entries	Depletion	Plant encrypted and invalid FTP passwords to waste the energy of the attacker who tries to decrypt and use those invalid passwords	Low CC; low OC; no DC
WinSCP registry entries	Discovery	Plant honey FTP passwords to entice the attacker to login to a honey FTP server	Med CC; low OC; no DC
Registry entries of applications that maintain login credentials	Depletion	Plant encrypted and invalid login credentials	Low CC; high OC; high DC
Registry entries of applications that maintain login credentials	Discovery	Plant honey login credentials	Med CC; low OC; no DC
Files that contain sensitive information	Depletion	Plant Honeyfiles with seemingly sensitive information to waste the energy of the attacker who tries to act upon the content of the files	Low CC; low OC; no DC
Files that contain sensitive information	Deflection Distortion Discovery	Depending on the meaning of the file content, plant crafted content that can help the defender achieve Deflection / Distortion / Discovery goals	Var CC; low OC; no DC

(CC: configuration cost, OC: operation cost, DC: development cost)

Deception Parameters We employ the methods discussed in Sect. 10.3 to the behavior model we obtain from the above analysis. We recognize a number of deception parameters that enable different deception schemes, as summarized in Table 10.3.

Discovery Schemes Since the malware applies many checks to evade analysis, these checks can be used to inspire effective discovery schemes that encourage the malware to run normally. Specifically, we can rename common analysis tools if we must run them and modify the behavior of the *CPUID* instruction so that it gives an impression that the environment is not a virtual machine, which is commonly the case for malware analysis. The cost of renaming common dynamic analysis tools is low. However, the cost of manipulating the result of the *CPUID* instruction can be high: it is cheap if the environment has a way to intercept *CPUID* instructions in software (e.g., on top of QEMU), but it is infeasible otherwise. Alternatively, the registry entries of the FTP clients, such as WinSCP, can be leveraged to lure the attacker to honeypots so we can learn more about its capabilities and intents. We can create honey FTP accounts, save the honey login credentials in WinSCP, and run the malware so it delivers the honey login credentials to the attacker. The configuration cost of this kind of scheme is medium because it is necessary to set up the honey FTP server and deploy monitoring tools.

Depletion Schemes The registry entries of the FTP clients can also be leveraged to feed the attacker fake login credential and deplete her resources and effort. For example, we can install WinSCP in the environment and save many sessions with fake values for the information targeted by the attacker (e.g., username and password) decreasing the likelihood of her landing on legitimate victims. An even better scheme is to create an encrypted version of an invalid password and save it in the Windows registry entry for WinSCP, which will give the attacker an additional burden to decrypt the password, thus further depleting the attacker's resources.

Deception Schemes Using the File System Similar to registry entries, files that contain sensitive information are useful parameters for multiple goals: depletion, deflection, distortion, and discovery. For example, we can plant honeyfiles with seemingly sensitive but useless information to waste the energy of the attacker who tries to act upon the content of the honeyfiles. Although the general idea is well known, the specific details as to which files should be planted can be greatly informed by analyzing the malware decisions. The cost of carrying out these kinds of strategies can vary depending on the purposes of the files: it may require simple editing of a file on one hand, or development of tools to create the files on the other; the operation cost may also vary depending on the purpose of the files: if they are used only by attackers, the cost is low, but if they are used by benign users, the cost can be quite steep, since the honey content can confuse benign users.

10.4.3 Challenges and Future Work

Through our case studies, we recognized a few technical challenges with respect to our approach. First, it is non-trivial to build a general deception parameter extraction technique due to inherent limitations of symbolic execution. For example, to avoid following back edges in a loop we have to supply the exact addresses of the source and destination instructions, which is unfortunately malware specific. One solution would be to automatically recognize back edges. Second, a naive use of symbolic execution cannot effectively discover interesting malware dependency on the environment because the execution can slip into paths leading to other than the desired goals, such as getting stuck in loops. We plan to develop new plugins that would guide the symbolic execution engine towards more meaningful paths leveraging existing approaches that were previously proposed to address similar challenges in dynamic taint analysis and mixed concrete and symbolic execution [20, 34].

10.5 Related Work

Randomization and moving target defense are well-investigated techniques towards agile cyber that can proactively disrupt advanced attacks. Randomization techniques, such as instruction set randomization [32], compiler-generated software diversity [15], and address space layout randomization [38], introduce unpredictability to confuse the adversary and invalidate her assumptions about the system. Moving target defense techniques, such as [12, 17, 19, 36, 44–46], mutate specific static system parameters proactively over time. For example, NASR [2] randomizes IP addresses based on DHCP over time. Similarly, the authors in [44] propose to periodically migrate VMs to make it harder for adversaries to locate targeted VMs. In another direction, deception techniques, such as honeynets and honeypots [3, 21, 25, 37], divert attackers away from their targets to consume their resources and protract their reconnaissance. Although these techniques and many other similar ones have been successful, within acceptable performance overheads, in deterring and deceiving the targeted attacks, they were designed in an ad hoc manner to counteract specific attacks. Our proposed analytic framework makes this process systematic and decreases the need for manual intervention and the reliance on human intelligence to design effective active cyber deception schemes.

Analyzing and exposing behaviors of malware is another research topic that has been extensively discussed in the literature [4, 6, 22, 26, 42, 43]. Forced execution [40] and X-Force [31] were designed for brute-force exhausting path space without providing semantics information for each path's trigger condition. To discover the trigger conditions, Brumley et al. [5] applied taint analysis and symbolic execution to derive the condition of malware's hidden behavior. Moser [30] introduced a snapshot-based approach that could be applied to expose

malware's environment-sensitive behaviors. Hasten [23] proposed as an automatic tool to identify and skip malware's stalling code. In [24], Kolbitsch et al. proposed a multipath execution scheme for Java-script-based malware. Other research [9, 40] proposed techniques to force the execution of different malware functionalities. While our work needs to analyze malware, we have a different goal: to automatically discover system parameters that can be mutated or misrepresented to deceive, rather than detect, malware. We can benefit from all existing malware analysis techniques, and in this chapter we choose symbolic execution in particular.

10.6 Conclusion

We present the first analytic framework towards automated creation of deception schemes based on rigorous malware binary code execution and automated reasoning of attack behaviors and decision-making process. We have implemented a tool that models the complete behavior of given malware in terms of its interactions with and dependence on the environment. We further analyze the malware behavior beyond traditional dynamic and symbolic malware analysis to track the malware decisions with respect to system parameters and identify those relevant to deception. Moreover, since multiple competing parameters may be identified, we employ optimization module theories based on Z3 to select the optimal set of parameters that can deliver consistent, resilient, and cost-effective deception. We analyzed over 50 recent malware and demonstrated through three detailed case studies how our deception-oriented analysis can lead to effective deception schemes against major malware types: cryptocurrency mining malware, credential-stealing malware, and ransomware. In addition, we have experimentally verified the deception schemes against bitcoin mining malware and ransomware.

10.7 Exercises

1) Download and install the Cuckoo sandbox and use it to analyze the Cerber ransomware (SHA256:e5a24badeecd951
40bddff4bb668aca96f33c9b5fc870cdbbd3a9092e809a4ea). Based on Cuckoo analysis result, answer the following:
   a. The malware will search directories for files using the *"FindFirstFileExW"* API. Report the first directory the malware searches.
   b. The malware tries to connect to a pool of IP addresses over TCP. Report the port number used in the frequent *sendto* API calls.
2) Study the documentation of the API *recvfrom* in Microsoft Developer Network (MSDN) and identify the system parameters that can influence its output arguments.

3) Download S^2E [7] and follow the documentation to create an analysis environment for Windows XP.
   a. Activate the Annotations plugin in S^2E and write a function annotation for the *"GetFileAttributes"* API in *Kernel32.dll* at address $0x7c80b7dc$.
   b. Modify your annotation function to write the value of the EAX register as symbolic at the function return. Name the symbolic variable as "gfAttrib-ret."
   c. After executing the malware in S^2E mode, open the messages log and locate the forking operations with respect to the variables "gfAttrib-ret" and report forking constraints.
4) Consider the following two execution paths for a ransomware. In *Path 1*, the malware scans the victim's machine, encrypts the files, sends the statistics to the C&C server, and displays a hard-coded instructions to the victim. In Path 2, the malware encrypts the files, sends the statistics to the C&C server, and receives a message from the C&C to display to the victim. Which path is more appropriate for depletion and how can that be implemented?

References

1. E. Al-Shaer and M. A. Rahman. Attribution, temptation, and expectation: A formal framework for defense-by-deception in cyberwarfare. In *Cyber Warfare*, pages 57–80. Springer, 2015.
2. S. Antonatos, P. Akritidis, E. P. Markatos, and K. G. Anagnostakis. Defending against hitlist worms using network address space randomization. *Computer Networks*, 51(12):3471–3490, 2007.
3. F. Araujo, K. W. Hamlen, S. Biedermann, and S. Katzenbeisser. From patches to honey-patches: Lightweight attacker misdirection, deception, and disinformation. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, CCS '14, pages 942–953, New York, NY, USA, 2014. ACM.
4. D. Balzarotti, M. Cova, C. Karlberger, C. Kruegel, E. Kirda, and G. Vigna. Efficient detection of split personalities in malware. In *Proc of NDSS'10*, 2010.
5. D. Brumley, C. Hartwig, Z. Liang, J. Newsome, P. Poosankam, D. Song, and H. Yin. Automatically identifying trigger-based behavior in malware. In W. Lee, C. Wang, and D. Dagon, editors, *Botnet Analysis and Defense*, volume 36, pages 65–88. Springer, 2008.
6. P. R. C. Song and W. Lee. Impeding automated malware analysis with environment-sensitive malware. In *Proc. of HotSec'12*, 2012.
7. V. Chipounov, V. Kuznetsov, and G. Candea. The s2e platform: Design, implementation, and applications. *ACM Transactions on Computer Systems (TOCS)*, 30(1):2, 2012.
8. M. Christodorescu, S. Jha, and C. Kruegel. Mining specifications of malicious behavior. In *Proceedings of the 6th Joint Meeting of the European Software Engineering Conference and the ACM SIGSOFT Symposium on The Foundations of Software Engineering*, ESEC-FSE '07, pages 5–14, New York, NY, USA, 2007. ACM.
9. P. M. Comparetti, G. Salvaneschi, E. Kirda, C. Kolbitsch, C. Krugel, and S. Zanero. Identifying dormant functionality in malware programs. In *Proc. of S&P'10*, 2010.
10. O. E. David and N. S. Netanyahu. Deepsign: Deep learning for automatic malware signature generation and classification. In *Neural Networks (IJCNN), 2015 International Joint Conference on*, pages 1–8. IEEE, 2015.
11. N. Falliere. Windows anti-debug reference. https://www.symantec.com/connect/articles/windows-anti-debug-reference. [Online; accessed 04-February-2018].

12. S. F. H. Gillani, E. Al-Shaer, S. Lo, Q. Duan, M. Ammar, and Ellen Zegura. Agile virtualized infrastructure to proactively defend against cyber attacks. In *Infocom*, 2015.
13. H. Goldman, R. McQuaid, and J. Picciotto. Cyber resilience for mission assurance. In *Technologies for Homeland Security (HST), 2011 IEEE International Conference on*, pages 236–241. IEEE, 2011.
14. K. E. Heckman, F. J. Stech, R. K. Thomas, B. Schmoker, and A. W. Tsow. *Cyber denial, deception and counter deception*. Springer, 2015.
15. T. Jackson, B. Salamat, A. Homescu, K. Manivannan, G. Wagner, A. Gal, S. Brunthaler, C. Wimmer, and M. Franz. Compiler-generated software diversity. In *Moving Target Defense*, pages 77–98. Springer, 2011.
16. H. Jafarian, Q. Duan, and E. Al-Shaer. Effective address mutation approach for disrupting reconnaissance attacks. *To appear in IEEE Transactions on Information Forensics and Security*, 2016.
17. J. H. Jafarian, E. Al-Shaer, and Q. Duan. Formal approach for route agility against persistent attackers. In *Computer Security - ESORICS 2013 - 18th European Symposium on Research in Computer Security, Egham, UK, September 9–13, 2013. Proceedings*, pages 237–254, 2013.
18. J. H. Jafarian, E. Al-Shaer, and Q. Duan. An effective address mutation approach for disrupting reconnaissance attacks. *IEEE Transactions on Information Forensics and Security*, 10(12):2562–2577, Dec 2015.
19. S. Jajodia, A. K. Ghosh, V. Subrahmanian, V. Swarup, C. Wang, and X. S. Wang, editors. *Moving Target Defense II - Application of Game Theory and Adversarial Modeling*, volume 100 of *Advances in Information Security*. Springer, 2013.
20. M. G. Kang, S. McCamant, P. Poosankam, and D. Song. Dta++: dynamic taint analysis with targeted control-flow propagation. In *NDSS*, 2011.
21. A. Kaur. Dynamic honeypot construction. 2013.
22. C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security'09*, 2009.
23. C. Kolbitsch, E. Kirda, and C. Kruegel. The power of procrastination: Detection and mitigation of execution-stalling malicious code. In *Proc. of CCS'11*, 2011.
24. C. Kolbitsch, B. Livshits, B. Zorn, and C. Seifert. Rozzle: De-cloaking internet malware. In *Proc. of S&P'12*, 2012.
25. S. Kyung, W. Han, N. Tiwari, V. H. Dixit, L. Srinivas, Z. Zhao, A. Doupé, and G.-J. Ahn. Honeyproxy: Design and implementation of next-generation honeynet via SDN. In *IEEE Conference on Communications and Network Security (CNS)*, 2017.
26. M. L, C. K., and M.Paolo. Detecting Environment-Sensitive Malware. In *Proc. of RAID'11*, 2011.
27. K. Lab. Kaspersky security bulletin. overall statistics for 2017. https://securelist.com/ksb-overall-statistics-2017/83453/, 2017.
28. H. D. Macedo and T. Touili. Mining malware specifications through static reachability analysis. In *Computer Security–ESORICS 2013*, pages 517–535. Springer, 2013.
29. W. Maples. Disable windows scripting host (WSH).
30. A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. of S&P'07*, 2007.
31. F. Peng, Z. Deng, X. Zhang, D. Xu, Z. Lin, and Z. Su. X-force: Force-executing binary programs for security applications. In *Proceedings of the 2014 USENIX Security Symposium*, San Diego, CA, August 2014.
32. G. Portokalidis and A. D. Keromytis. Global ISR: Toward a comprehensive defense against unauthorized code execution. In *Moving Target Defense*, pages 49–76. Springer, 2011.
33. Y. Qiao, Y. Yang, J. He, C. Tang, and Z. Liu. *CBM: Free, Automatic Malware Analysis Framework Using API Call Sequences*, pages 225–236. Springer Berlin Heidelberg, Berlin, Heidelberg, 2014.
34. P. Saxena, P. Poosankam, S. McCamant, and D. Song. Loop-extended symbolic execution on binary programs. In *Proceedings of the eighteenth international symposium on Software testing and analysis*, pages 225–236. ACM, 2009.

35. M. K. Shankarapani, S. Ramamoorthy, R. S. Movva, and S. Mukkamala. Malware detection using assembly and API call sequences. *J. Comput. Virol.*, 7(2):107–119, May 2011.
36. N. Soule, B. Simidchieva, F. Yaman, R. Watro, J. Loyall, M. Atighetchi, M. Carvalho, D. Last, D. Myers, and C. B. Flatley. Quantifying & minimizing attack surfaces containing moving target defenses.
37. J. Sun, K. Sun, and Q. Li. Cybermoat: Camouflaging critical server infrastructures with large scale decoy farms. In *Communications and Network Security (CNS), 2017 IEEE Conference on*, pages 1–9. IEEE, 2017.
38. P. Team. PaX address space layout randomization (ASLR). http://pax.grsecurity.net/docs/aslr.txt, 2015. [Online; accessed 10-Feburary-2017].
39. S. P. Team. Stratum mining protocol official documentation. https://slushpool.com/help/manual/stratum-protocol/, 2017.
40. J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of RAID'07*, 2007.
41. J. Xu, Z. Kalbarczyk, and R. K. Iyer. Transparent runtime randomization for security. In *22nd International Symposium on Reliable Distributed Systems, 2003. Proceedings.*, pages 260–269, Oct 2003.
42. Z. Xu, L. Chen, G. Gu, and C. Kruegel. PeerPress: Utilizing enemies' p2p strength against them. In *Proc. of CCS'12*, 2012.
43. Z. Xu, J. Zhang, G. Gu, and Z. Lin. Goldeneye: Efficiently and effectively unveiling malware's targeted environment. In *Proceedings of the 17th International Symposium on Research in Attacks, Intrusions and Defenses (RAID'14)*, September 2014.
44. Y. Zhang, M. Li, K. Bai, M. Yu, and W. Zang. Incentive compatible moving target defense against VM-colocation attacks in clouds. In D. Gritzalis, S. Furnell, and M. Theoharidou, editors, *Information Security and Privacy Research*, volume 376 of *IFIP Advances in Information and Communication Technology*, pages 388–399. Springer Berlin Heidelberg, 2012.
45. Q. Zhu and T. Başar. Game-theoretic approach to feedback-driven multi-stage moving target defense. In *Decision and Game Theory for Security*, pages 246–263. Springer, 2013.
46. Q. Zhu, A. Clark, R. Poovendran, and T. Basar. Deceptive routing games. In *Decision and Control (CDC), 2012 IEEE 51st Annual Conference on*, pages 2704–2711. IEEE, 2012.

Chapter 11
Malware Deception with Automatic Analysis and Generation of HoneyResource

Zhaoyan Xu, Jialong Zhang, Zhiqiang Lin, and Guofei Gu

Abstract Malware often contains many system-resource-sensitive condition checks to avoid any duplicate infection, make sure to obtain required resources, or try to infect only targeted computers, etc. If we are able to extract the system resource constraints from malware binary code, and manipulate the environment state as *HoneyResource*, we would then be able to deceive malware for defense purpose, e.g., immunize a computer from infections, or trick malware into believing something. Towards this end, this chapter introduces our preliminary systematic study and a prototype system, AUTOVAC, for automatically extracting the system resource constraints from malware code and generating HoneyResource (e.g., malware vaccines) based on the system resource conditions.

Keywords Malware analysis · Malware immunization · Malware deception

11.1 Introduction

Malware is a severe threat to our computer systems. To combat malware, the state-of-the-art defense at end hosts mainly focuses on detection techniques, which often fall into two categories: signature-based detection and behavior-based detection. A signature-based approach typically attempts to extract some unique string patterns from malware binaries. Unfortunately, the signature generation and update speed usually cannot keep up with the quickly increasing malware samples each day

The original version of this chapter was revised: Chapter authors have been added. The correction to this chapter is available at https://doi.org/10.1007/978-3-030-02110-8_12

Z. Xu · J. Zhang · G. Gu (✉)
Texas A&M University, College Station, TX, USA
e-mail: guofei@cse.tamu.edu

Z. Lin
The Ohio State University, Columbus, OH, USA

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_11

in the wild due to the wide use of polymorphisms/packers in malware. While a behavior-based approach could be relatively more stable in terms of detecting the same set of malware and their variants, it is typically very expensive and may cause a noticeable performance overhead on end hosts.

Therefore, the need of new lightweight and complementary techniques for effective malware defense is still pressing. Interestingly, we find malware infection works similarly to pandemic diseases. If we were able to deceive the malware that it has infected the protected host, we would have been able to prevent it from infecting a wider range of machines (considering the case of botnets). Fortunately, we find malware that often contains system-resource-sensitive condition checks or constraints to avoid any duplicate infection, make sure to obtain required resources, or try to infect only targeted computers, etc. For instance, many fast-spreading malware programs (e.g., Conficker [26]) will clearly mark an infected machine as *infected* such that they can avoid wasting time and effort in re-infecting the machine. As such, such resource manipulating scheme can be considered as a more effective and safer way for malware deception. In this context, such resource is one kind of HoneyResource which tricks malware into believing the existence/non-existence of itself.

In general, any system resource/environment variables that are directly or indirectly used in path conditions (such as registry, mutex), or those that lead to the failure of certain system calls, can all be considered as HoneyResource, because these external environment states can impact the behavior of the malware. While it might lead to an over-approximation by considering all these state variables, we can run tests to eliminate the mistakenly classified environment variables.

Based on the above observation, we propose AUTOVAC, a new technique to automatically generate HoneyResource for effective and efficient malware deception. While theoretically manipulating any variable that leads to a conditional check of malware execution could potentially be used as a HoneyResource, we would like to focus on the variables whose states can be controlled by the external environment such as registry, certain file names, etc. As such, the environment resources accessed by malware are of our interest. Specifically, we design a program analysis technique to determine whether the manipulation of these resources can successfully prevent malware's infection/execution. We treat such resources as our malware HoneyResource and derive concrete information needed for generating HoneyResource. After we generate the HoneyResource, we then inject them into end hosts. For example, HoneyResource is able to serve as a kind of *vaccine* for malware. To the best of our knowledge, AUTOVAC is the first systematic work of using program analysis to automatically generate HoneyResource for real-world malware deception.

In summary, we make the following contributions:

- We conduct the first systematic study of malware HoneyResource. We discuss all possible mutable resources of our interest and present a taxonomy of malware HoneyResource.

- We design and implement AUTOVAC, which can automatically track the malware path constraints as well as their propagation, associate them with the external environment resources, and automatically generate HoneyResource.
- We evaluate our system with a large set of real-world malware samples. Experimental results show that it is truly possible to generate working HoneyResource for many real-world malware families, such as Conficker, Sality, and Zeus, and use HoneyResource as a complementary approach in practice.

11.2 Problem Statement and Approach Overview

11.2.1 Malware HoneyResource Background

11.2.1.1 Definition of Malware HoneyResource

From our viewpoint, a malware HoneyResource is a computational preparation that deceives a particular malware program, e.g., trick it into believing something, or prevent its infection. Essentially, malware, like any generic program, usually conducts a series of operations on system resources and outputs the computation result.

Thus, we define a malware HoneyResource as a specific system resource (or a collection of them) that is created or used by malware in order for its normal infection and execution. Such malware HoneyResource typically has two kinds of behavior:

- It simulates the existence of certain computer organism (system environment/resource) such that malware will perform certain activities, e.g., exit upon the awareness of such existence (because it does not want to re-infect the victim again, or the victim does not have a targeted environment, etc.).
- It prevents malware from creating/accessing certain critical computer organism such that malware cannot obtain its essential resources to fulfill the functions.

Fig. 11.1 System architecture

11.2.1.2 A Taxonomy of Malware HoneyResource

Besides the aforementioned categories of malware HoneyResource, we can further define different types from different perspectives.

First, from the perspective of identification, the HoneyResource identifier is defined as a combination of *resource type* and *name* of malware-targeted resources. To avoid unwanted side effect to benign software running on end host, the HoneyResource identifier should be as *unique* and *deterministic* as possible. Thus, in our taxonomy, an identifier can be categorized as: static (e.g., constant value), partial static (e.g., it conforms to a specific regular expression), or algorithm-deterministic (e.g., it is calculated with customized algorithms).

To deceive different malware families, the effectiveness of a malware HoneyResource can vary. Based on the effectiveness, we can classify malware HoneyResource into two types: full deception that can completely cease the malware execution (e.g., negating the first few condition checks to prevent any malicious behavior execution), and partial deception that significantly affects the execution of some major functions in malware (e.g., malware is not able to keep persistent in the system if rebooted, or malware is not able to perform key network communication such as C&C, and self-updating).

In terms of HoneyResource delivery and deployment, there could be two categories: direct injection and creation of HoneyResource daemon. Direct injection is very lightweight, e.g., a specific `mutex` name or file name, and the HoneyResource can be simply injected into the target computer once and it will be effective afterwards. HoneyResource daemon requires running a service program (i.e., a daemon) on the targeted machine, and such daemon can prevent the creation (or other access types) of certain specific files, registries, libraries, system services, windows, and processes to further prevent malware from obtaining critical resources or information to fulfill its functionalities (such as for partial deception). More details are presented in Sect. 11.5.

It is worth noting that an ideal malware HoneyResource is those with full deception and one-time direct injection. However, other types of HoneyResource are also useful, as discussed later and shown in our evaluation (Sect. 11.6).

11.2.1.3 Use Case of HoneyResource

As a complementary technique to existing malware defense, HoneyResource may not be used to protect machines from all malware attacks. However, they can be used for current, high-profile, large-scale malware propagation and infections, which may last for a period of time, e.g., several days, weeks, or months. If we can capture the binary at the initial infection stage, we can quickly generate HoneyResource and protect our uninfected machines from the attacks, until a better detection or prevention solutions (e.g., a system/software patch to fix the vulnerability) are available and fully deployed.

11.2.1.4 Target and Assumptions

Not all malware can have HoneyResource. Our target is those malware that has specific system-resource-sensitive behavior, illustrated in the following scenarios:

- Some malware can work only in the scenario in which none of the same malware instances is present in the host. Thus, they have to uniquely *mark* their infected systems through creating and checking certain deterministic identifiers such as mutex, file, as shown in the Conficker example. Our HoneyResource can hence appear to be the malware vaccine to fool the malware and stop its infection.
- Some malware has issues in handling the failure of certain system resource access. Our HoneyResource can try to enforce such failures to make the malware run into their undesired status (e.g., process termination, or important functions being disabled).
- Some targeted malware is designed to work in a specific system environment. Our HoneyResource can attempt to make each protected system different from malware targeted environment, so as to protect machines from the infection.

It is true that some malware may not use system resource checks to make their infection decision. That is, AUTOVAC does have limitations and we discuss in great detail on the possible evasions in Sect. 11.7. We note that while evasions are possible, most of these scenarios are not within the scope and assumptions of our approach. The intention of AUTOVAC is not to replace existing defense approaches, but to complement them from a new perspective. As we show later, once we can successfully extract interested system resource constraints and generate HoneyResource, we can effectively and efficiently deceive malware.

11.2.2 Approach Overview

An overview of AUTOVAC is illustrated in Fig. 11.1. At a high level, it consists of three phases: *Candidate Selection*, *HoneyResource Generation*, and *HoneyResource Delivery/Deployment*.

In **Phase-I** (Sect. 11.3), we will first filter out malware samples that are unlikely to contain HoneyResource. In this step, we profile the normal execution of the malware to obtain an overview of the malware's accessed system resources, including the types of resources and the names of the corresponding resource-identifiers, the operations (e.g., create, and read/write) on the resources, and the corresponding results (e.g., succeed, or fail).

During our profiling, we will also apply a variant of dynamic taint analysis [7] to determine whether the malware's execution will be affected by certain resources it has accessed. The implication is that malware has to be *sensitive* to its resource access result. Otherwise, malware's behavior is deterministic regardless of its resource environment and no HoneyResource willexist for it. Hence, if we find

no program branches that depend on any system resource, we filter this malware because it does not contain HoneyResource that we can extract. At the end of this phase, we obtain a list of candidate resources that can affect the control flow of the malware execution.

In **Phase-II** (Sect. 11.4), our task is to generate HoneyResource by testing their exclusiveness and impact on malware execution. It contains three sub-steps.

- **Step-I: Exclusiveness Analysis** In general, system resources are also being used by benign programs. In this step, we would like to *filter the resource identifiers that are not exclusive to malware itself (e.g., some benign programs also use them), in order to avoid false positives*.
- **Step-II: Impact Analysis** The goal of this step is to *measure the potential impact of a certain system resource, i.e., whether it can affect the execution of some interested malware functions*. We start a second-round execution monitoring by manipulating the result of the specific malware's resource operation, which will generate a manipulated trace. We apply program alignment techniques [8] to compare the execution differences between the manipulated trace and the normal trace and determine if the system resource can (significantly) impact the malware functions, e.g., cause malware to stop the execution. At the end of this step, we generate a list of resources that can effectively stop the malware's infection (full deception), or significantly affect the malware's certain functions (partial deception).
- **Step-III: Determinism Analysis** We also have to measure the determinism of the specific system resource identifier, e.g., *filename* or `mutex` name. An effective malware HoneyResource should be *deterministic, such that it can be accurately reproduced/predicted to affect the targeted malware*. A deterministic value could be a fixed/static value, or a value that is generated from a deterministic algorithm (from deterministic resources) or even partial static if certain part is deterministic. To decide if a specific resource identifier is deterministic, we perform *backward taint analysis and program slicing* to fully understand the identifier generation logic and the parameters it depends on. Based on that, we further analyze the root-cause of the identifier generation and generate a program slice responsible for the identifier generation logic.

In **Phase-III**, we deploy the malware HoneyResource at an end host. There are also two situations: direct injection and HoneyResource daemon. We will present their details in Sect. 11.5.

11.3 Phase-I: Candidate Selection

Given a malware sample, AUTOVAC will first determine whether it is possible to generate a HoneyResource, and at the same time collect the behavior information to facilitate the next step analysis. Since our HoneyResourceis essentially composed

of system resources that have a direct or indirect (through propagation) impact on the malware execution, we adopt a variant of dynamic taint analysis [7] to achieve this.

11.3.1 Taint Sources

Taint sources define the origins of the tainted data. Our current focus is on those system-resource-related data that can possibly impact the malware behavior. However, there is a wide range of system resources and certainly some of them cannot be used such as system-assigned random objects. As such, we have to systematically study these resources and identify our taint source. In particular, we use the following criteria to decide whether a system resource should be tainted.

- **Unique Presence** Our focused system resources should be commonly used by malware, and these resources should be *uniquely identified*. Thus, those *transient* system resources, e.g., events, signals, and critical sections, are out of our interest.
- **Less Impact to Benign Software** Our targeted resources should have *little or minor impact* to benign programs. This requirement would exclude many system-wide objects and information, such as timers, performance counters, input/output devices, and removable devices, because they are commonly accessed by benign programs
- **Easier Deployment** Our targeted resources should be lightly deployed onto end hosts. To this end, injecting some specific files or `mutex` into the end host would be viable options. Therefore, files, `mutex`, or registry will be our main targeted resources.

11.3.1.1 API Labeling

After applying the above criteria, eventually `mutex`, static files, and registry items are of our particular interest. Meanwhile, the propagation use of these resources such as process, library, GUI window, and services are also of our interest because these resources depend on some deterministic resource identifiers. However, at the instruction level, these *resource-identifiers* often get accessed through system APIs. Thus, we have to examine each Windows API to define our taint sources.

More specifically, all the system resource access APIs (e.g., `NtQueryObject`) are of our interest. AUTOVAC will taint the return values as well as the affected arguments of these functions. In our design, we examined over 800 windows APIs and we classified them into the following two categories:

- **Tainting the return value** Most APIs only affect the return values (always stored in `EAX`), such as `OpenMutex`, and `NtSaveKey`. For them, we just taint the return value.

- **Tainting the argument** Some APIs store the affected values in the arguments. For instance, `NtOpenKey` and `NtOpenFile` store the return handler in their first parameters.

Besides tainting the return values or arguments, we also need to record the concrete values of the arguments to these APIs because eventually our HoneyResource work by affecting the system environments which are their arguments. Meanwhile, not all the arguments are of our interest, and only those *resource-identifiers*. This is also a tedious procedure to identify these *resource-identifiers*. Table 11.1 shows an example on how we label the two Windows APIs.

Table 11.1 Labeling examples for OpenMutex/ReadFile

	`OpenMutex`	`ReadFile`
Resource type	Mutex	File
Resource-identifier	3rd parameter: *lpName*	1st parameter: *hFile for Handle Map*
Success	`EAX`: Valid handle value	`EAX`: TRUE
Failure	`EAX`: NULL, `GetLastError: 0x02`	`EAX`: FALSE `GetLastError: 0x1E`

11.3.2 Taint Propagation

AUTOVAC has to propagate taint labels for data operations. That is, for any instruction whose source operand has been associated with the tainted labels, we taint the destination operand with the same label. Then, whenever we find a comparison (i.e., predicate) instruction whose operands have been tainted (e.g., `test,cmp`), we will flag this malware most likely having a HoneyResource and pass it to our next phase analysis.

11.3.2.1 Output from Phase-I

As our Phase-I runs the malware in normal settings, it provides a great opportunity to collect the normal malware behavior. To this end, we log all the executed APIs as well as their parameters, along with the precise calling context information including the call stack and the caller-PC (program counter). In addition, our log file also contains the list of the system-resource-sensitive APIs that have been executed, and their propagated taint record that is used in the predicate.

11.4 Phase-II: HoneyResource Generation

Once a malware sample has been flagged to "possibly have a HoneyResource" in **Phase-I**, it will be fed to our **Phase-II** to perform a deeper analysis, including exclusiveness analysis (Sect. 11.4.1), impact analysis (Sect. 11.4.2), and determinism analysis (Sect. 11.4.3). In this section, we present these analyses in greater detail.

11.4.1 Exclusiveness Analysis

The goal of our exclusiveness analysis is to exclude the resources that have been used in benign software. For instance, some resources such as library names `uxtheme.dll`, and `mscrt.dll` could be used in benign programs. We must exclude them otherwise our HoneyResource will have false positives.

In **Phase-I**, AUTOVAC has logged all the *resource-identifiers*, and next we would like to query whether or not each *identifier* is unique to the malware. Our basic idea is inspired by a Googling approach used in the previous studies [27]. Essentially, we use Google query APIs to search *resource-identifiers*. Based on the return results and their context, we infer whether these resources are already associated with benign software. We refer the readers to [27] for more details. In short, from our search query, if the *resource-identifiers* does not conflict with benign software or there is no any matching search result, then we proceed with further analysis.

11.4.2 Impact Analysis

Given a list of the system resources that can (in)directly affect the malware execution and the corresponding APIs provided in **Phase-I**, AUTOVAC will run the malware again in a controlled environment such that we can mutate the return value or involved arguments and test whether malware will exhibit different behavior or not. Our current design is to mutate each involved API one at a time and compare the behavior with our normal execution captured in **Phase-I**.

11.4.2.1 Trace Differential Analysis

Then, the next question is how we compare the malware behavior in two traces: one is a normal execution, and the other is a resource mutated execution.

Finding the differences in two traces has been discussed in the previous literature (e.g., [8, 25]). It is essentially a *program alignment* problem [8]. The basic idea is to align two execution points that are equivalent to each other and then compute

the differences only between the *unaligned* instructions. In our scenario, we try to obtain the high-level information such as whether the malware will terminate rather than the minor instruction-level execution differences. Thus, in our design, we use the API call sequences (as we have already logged all the executed APIs and their calling context information) and present an API sequence alignment algorithm as shown in Algorithm 2.

In particular, we adopted an alignment algorithm from Zeller [8], which uses the *execution context* for each instruction for the comparison. If the instruction and its execution context are equivalent (line 4), they are aligned together. However, we do not need to compare instruction by instruction, but rather at the granularity of APIs. Thus, we define a calling execution context as a *triple*:

Algorithm 2: Differential Analysis on the API-Call Traces

$\prod_m$: Manipulated Call Trace, $\prod_n$: Natural Call Trace
Δ_m: Unaligned Call Trace in $\prod_m$, Δ_n: Unaligned Call Trace in $\prod_n$,
$f_{\prod}$: ⟨ name, caller eip, parameter list⟩, f_{Δ}: ⟨ name, parameter list⟩

```
1  Δm ← ∅, Δn ← ∅
2  for call f_{∏m} in ∏m do
3      for call f_{∏n} in ∏n do
4          if isAligned(f_{∏m}, f_{∏n}) then
5              GOTO FIND_ALIGNED
6      Δm = Δm ⋃ f_{Δm}
7  Δn = ∏n
8  FIND_ALIGNED:
9      Δn = ∏m[0, index(f_{∏n})]
10     {f_{Δi}} = Diff(Δm, Δn)
11     return {f_{Δi}}
```

<API-name, Caller-PC, Parameter list>

For the *parameter list*, we only compare the *static* parameters that are *identical* across different executions. Note that all these information has been logged either in **Phase-I** for the normal execution, or logged in **Phase-II** for the mutated execution. Also, the reason we have to log the *Caller-PC* is for the preciseness.

As illustrated in Algorithm 2, our analysis begins from the start of the trace, then proceeds with a linear searching for each system/library call in the mutated trace and examines whether it could be aligned with some call in the normal run trace (line $2-8$). If we find an anchor point, we generate two difference sets Δ_m and Δ_n.

Next, we examine the two Δ sets to evaluate the further differences and classify the HoneyResource type. Specifically, we define three kinds of deception effects.

11.4.2.2 Full Deception

If we find APIs such as `ExitThread`, `TerminateProcess`, and `TerminateThread` in Δ, then certainly the mutated system resources can be served as a full deception HoneyResource, because the malware has killed itself.

11.4.2.3 Partial Deception

Some HoneyResource may significantly weaken certain important functions of malware. We consider them as partial deception. More specifically, we currently focus on the following four types of partial deception:

- **Type-I: Disable Kernel Injection** An important malicious function of malware is to raise its privilege. The common way they use is to inject a kernel driver into an end host. There are several system calls (mainly undocumented), such as `OpenSCManager` have been used for this. Furthermore, some malware commonly copies itself as a new file with its name ending with `.sys`, which implies that some kernel driver is created by the malware.
- **Type-II: Disable Massive Network Behavior** If we find that the normal execution is full of network-related functions, while the manipulated execution is clean from such calls, we consider such deception as **Type-II** Partial Deception.
- **Type-III: Disable Malware Persistence** Malware typically modifies specific registry entries such as `Run` subkeys in multiple register paths. Other autostart approaches include: (a) file operations on `startup` folder or `system.ini` files, (b) creation of new service entries, and (c) access of `winlogon` binary. Through differential analysis, we can tell if these operations are lost in the mutated execution while present in the normal execution.
- **Type-IV: Disable Benign Process Injection** To be more evasive, malware often inject themselves into some benign processes. Processes such as `explorer.exe` and `svchost.exe` are common targets. If we find such a clear pattern in the differential analysis, we consider these HoneyResource as **Type-IV** partial deception.

11.4.2.4 No Deception

If none of the above APIs are in the Δ, then we classify this HoneyResource with no effect to stop or affect malware behavior.

11.4.3 Determinism Analysis

We next need to verify the determinism of the extracted resource-identifiers.

11.4.3.1 Backward Taint Tracking and Program Slicing

Given a resource-identifier, we need to identify whether it is deterministic or entirely random. We choose to trace the root-cause for the generation of the resource-identifier.

To back track the procedure of how malware generates an identifier, we perform a backward taint tracking. The basic idea is to include all the instructions that have contributed to the creation of the resource-identifier, which is the argument of the API of our interest. To this end, starting from data-use of the argument, we back track each executed instruction to check whether or not their operands have been involved to define the data. If so, we taint the source operand as the same symbol and continue the backward propagation. We perform the analysis offline on logged traces.

The termination of our backward tracking is the point to identify the root-cause that generates the identifier's name. We continue backward propagation until tainted source is either from read-only regions (e.g., static strings), or constant values, or the return value of the system APIs. Based on these different sources, we decide whether the generation of the identifier is deterministic or not.

An identifier has a *non-deterministic* type if and only if *all* elements of its composition are resulted from some random functions (e.g., `GetPerformanceCounter` and `GetTempFileName`). As illustrated in the left part of Fig 11.2, if the termination data point is from a read-only segment such as `.rdata`, or constant values, we can easily mark it as *static*. Similarly, if an identifier is constructed using some non-deterministic value combined with some constant value, we can mark it as *partial static*, and such an identifier will be deployed using a slightly different strategy compared to the scenario of purely static identifier.

An identifier could be *algorithm-deterministic*, namely its identifier is generated through certain computation. Some appear-to-be random name can be generated from some invariable seed, such as computer name or hardware serial number. Algorithm-deterministic names will be backward propagated to some semantic-known APIs. We use these APIs to decide the root-cause type when generating the name. One example is shown in the middle part of Fig. 11.2. We use the `GetComputerName` to infer that the input should be a computer name.

For such algorithm-deterministic identifier, we also need to find the generation logic because we need to replay and compute it for each end host. We apply the existing backward program slicing[18] techniques to extract an independent, executable program slice for that. At the end of this step, we delete all the entirely random (non-deterministic) identifiers.

11.4.4 Malware Clinic Test

To further reduce the possible false positives, we design a *Malware Clinic Test* at the end of this phase. Malware Clinic Test aims to inject our HoneyResource into

real environments and test whether it will affect the normal use of a computer system. This test environment is automatically configured by running multiple benign software and services. Even though the scheme of clinic test is simple, it is essential to ensure the quality of our generated HoneyResource. If it affects the normal usage, it will be discarded.

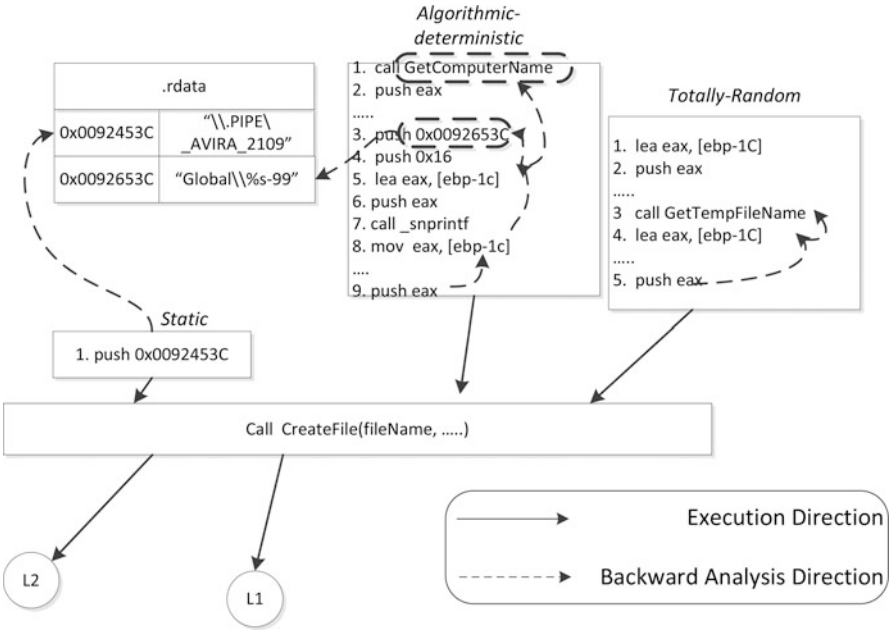

Fig. 11.2 Sample Malware code and the traced behavior

11.5 Phase-III: HoneyResource Delivery and Deployment

After we generate the HoneyResource, we next describe how to deliver and deploy the HoneyResource to an end-user computer.

11.5.1 Direct Injection

Direct injection works for static identifiers. If a HoneyResource stops malware execution by frustrating the presence checking of static type of resources, we inject it by creating or deleting the resources. For instance, if the malware needs to open

certain static file (or registry) before proceeding the malicious functionality, then we remove the static file (or registry), or vice versa. Moreover, we accordingly adjust the injected file's access privilege to disallow certain operation such as read and write. In these cases, when a low-privilege malware program attempts to access a resource, which is a common case at the initial infection stage, static HoneyResource efficiently stop further malicious behavior.

11.5.2 HoneyResource Daemon

Daemon works for algorithm-deterministic identifier and partially static identifier. For an algorithm-deterministic identifier, we have extracted a program slice of the resource-identifier generation logic with knowledge about its input, such as a computer name or an IP address. To generate the HoneyResource, we collect these information ahead and run the captured program slice. Such procedure works very similar to Inspector Gadget [18]. Our daemon process runs periodically to check whether the input has been changed and the HoneyResource needs to be re-generated.

Daemon is also designed for identifying resource name represented using regular expressions (i.e., distinguishable partial static HoneyResource). Specifically, at the end host, we dynamically intercept the APIs and resolve their resource-identifiers. If the daemon monitors that a resource identifier matches with our partial static HoneyResource, it will return the predefined result to stop the malware execution.

11.6 Evaluation

We have implemented AUTOVAC. While our online dynamic analysis can be implemented using virtual machine monitors such as TEMU [4], we use DynamoRIO [2] to implement due to its simplicity and flexibility in binary instrumentation. Our differential analysis module is implemented using offline parsing of the execution logs. Also, to perform tainted analysis, we translate the X86 instructions into an intermediate language *BIL* [10], and then we develop our own parser code to identify the resource-sensitive branches and perform differential analysis. Our exclusiveness analysis involves a search engine query component, for which we implement using the API provided by Google. In this section, we present our evaluation results.

11.6.1 Experiment Dataset

Our test dataset consists of 1,716 malware samples, which are collected from multiple online malware repositories (e.g., [1, 3]) with mostly from Anubis [1].

We also leverage an online malware classification tool, *VirusTotal* [5], to obtain the classification information for these malware. We summarize classification results in Table 11.2. We can see that these malware samples fall into 6 categories such as Backdoor (722 samples), Downloader (574 samples), and Trojan (184 samples).

Table 11.2 Malware's classification from VirusTotal

Category	# Malware	Percentage
Trojan	184	10.72%
Backdoor	722	42.07%
Downloader	574	33.44%
Adware	73	4.25%
Worm	104	6.06%
Virus	59	3.43%
Total	1,716	100%

11.6.2 Evaluation Result on Candidate Selection

In the first step (**Phase-I**), we monitor malware's access to system resources. We conduct this experiment by running these 1,716 malware samples in our analysis environment and each sample runs for 1 min (we tend to believe that the resource checks usually happen in the early stage of the malware execution and we thus choose this 1-min threshold). We hook 89 system/library calls as tainted sources that are related to resource operations. The resources in our evaluation include *file*, *mutex*, *registry*, *window*, *process*, *library*, and *service*. We measure the basic operations for these resources such as read/write for file and registry, and open/create for other resources. Meanwhile, for each execution instance of the hooked function, we examine their callers' `PC` and make sure that it does not belong to the system library's address space. Thus, we do not count the functions that are called inside the system/library calls.

For 1,716 malware samples, we successfully tracked 460,323 occurrences of these API calls. Through our taint analysis in this phase, we identified that 371,015(80.3%) occurrences of the calls will possibly deviate the execution of the malware samples. This result confirms that real-world malware is indeed resource sensitive.

Among these 371,015 occurrences, we further made a statistic study based on the resource type and its corresponding operations. The result is shown in Fig. 11.3. From the figure, we can see that around 37.39% of the resource accesses account for file operation. Mutex (7.07%) and registry (20.08%) are also commonly accessed by malware. We consider these three types of resources that can be efficiently delivered using the injection scheme. Meanwhile, malware's logic is also commonly sensitive to other types of resources such as windows (13.14%), process (8.02%), library (6.6%), and service (3.4%).

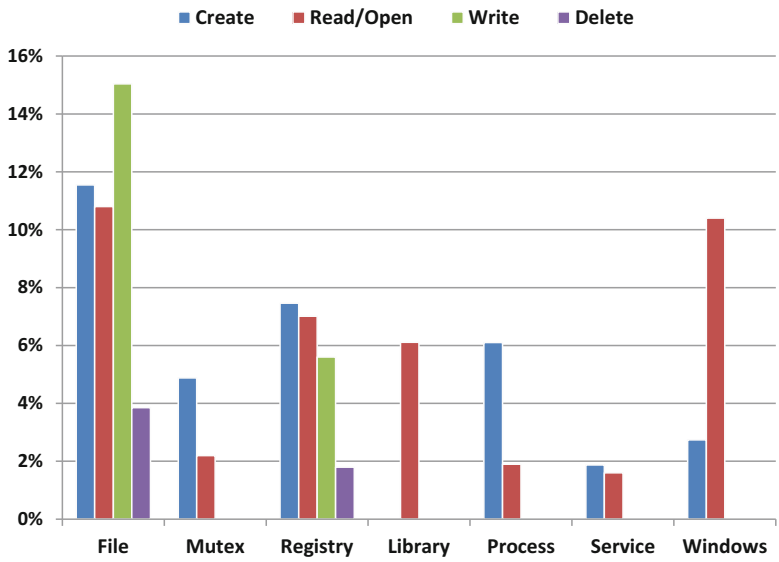

Fig. 11.3 Statistics on Malware's resource-sensitive behaviors

11.6.3 Evaluation on HoneyResource Generation

Table 11.3 HoneyResource samples (Operation type symbols—check Existence (E), Create (C), Read (R), and Write (W), Impact symbol—Termination (T), Process Hijacking (H), Persistence (P), Kernel Injection (K), and Network Massive Attack (N))

Seq	Type	OperType	Impact	Identifier	Malicious sample Md5
1	Mutex	E	T	!VoqA.I4	df1df624c5da833d3882d22a2e2456c9
2	File	C,R,W	P,H	%system32% \twinrsdi.exe	1b6fb589f36654af0ef44aa92f94324a
3	File	C,E,R,	P,H,N	%system32% \dwdsregt.exe	24784256bbbb936dc1e0999c307883c8
4	File	C,E,R,W	K,P	%system32% \driver\qatpcks.sys	27d18e20e253391112d50b2b49440aea
5	Mutex	E	T	GTSKISNAUOI	ee5878eab962b032c78c1d6eec7ec917
6	Mutex	E	P,H	fx221	af48ecfcc1812d6f814a26792107b80e
7	Mutex	C,E	T	)ryt-24qtqq26sn]9c	b534b75da5fc3b9b178c60bf10b1feca
8	Mutex	C,E,R	P,H	_AVIRA_2109	04a93b1f08a1675c67c9975a7024c3d6
9	File	C,E,R,W	P,H	%system32% \ shlmon.exe	af48ecfcc1812d6f814a26792107b80e
10	File	C,E,R,W	T,P	%system32%\ sdra64.exe	04a93b1f08a1675c67c9975a7024c3d6

In the evaluation, we analyzed all 1, 716 malware in a controlled environment. In total, we generated 536 HoneyResource that belong to 210 malware samples. The result is presented in Table 11.4. For each column, we classify the HoneyResource

as full deception or partial deception (Type-I to Type-IV). We also list the statistics on the HoneyResource distribution among different resource types in Table 11.4. Among all HoneyResource, we find that 373 HoneyResource have static identifiers, and 163 samples have *algorithm-deterministic* or *partial static* identifiers.

Table 11.4 Evaluation on HoneyResource generation

Resource	Full	Type-I	Type-II	Type-III	Type-IV	All
File	31	19	17	110	61	238
Registry	10	11	3	72	19	115
Mutex	5	3	3	16	3	30
Process	2	5	2	18	5	32
Windows	0	4	3	8	3	18
Library	19	5	1	10	19	54
Service	7	4	0	17	21	49
Total	74	51	29	251	131	536

Table 11.5 HoneyResource statistics on different Malware families

Type	Backdoor	Trojan	Worm	Adware	Downloader	Virus
File	33%	27%	24%	30%	45%	81%
Registry	15%	29%	21%	13%	20%	19%
Windows	3%	14%	0%	47%	11%	0%
Mutex	8%	12%	29%	0%	2%	0%
Process	8%	7%	14%	0%	10%	0%
Library	26%	9%	4%	0%	7%	0%
Service	7%	2%	8%	10%	5%	0%
Deployment						
Direct	67%	79%	63%	69%	69%	84%
Daemon	33%	21%	37%	31%	31%	16%

To zoom-in the details of these HoneyResource, we select 10 representative samples and describe them in Table 11.3. We can see that most of these HoneyResource stop several logic of malware's infections. In some cases, different operations on the resources can even cause different effects on malware's logic. For example, for the last malware in Table 11.3, we find that the failure of *creating* a file will stop malware's process hijacking logic, and the failure of *writing* a file will crash the malware process (Table 11.4).

For the generated 536 HoneyResource, we also combined their types with the 210 malware's classification information to see what is the common HoneyResource type for different kinds of malware. The result is shown in Table 11.5. From this table, we can see that the file resources are the common HoneyResource

for many malware families. Meanwhile, the windows resource HoneyResource is better suitable for adware because the windows resource HoneyResource is attempting to prevent adware from creating their malicious windows. If such operations fail, adware will possibly stop their further action. Last but not least, mutex HoneyResource works better for worm and backdoor malware. This is also reasonable, because these malware highly depend on the mutex to prevent duplicate infection.

We also report the statistics of our delivery for these 536 HoneyResource. As shown in Table 11.5, direct injection is the most common way to deploy HoneyResource on end hosts. Also, only about 20%–30% HoneyResource need a daemon for the deployment.

11.6.4 Case Studies

Next, we present two representative case studies to illustrate in greater detail on how each of our resource access-based HoneyResource can be used for malware infection immunization. In such case, our HoneyResource can work as malware HoneyResource to stop malware infection.

11.6.4.1 File-Based HoneyResource

One HoneyResource for Zeus/Zbot [6] family is a static file named `sdra64.exe` which is stored in the `system32` directory. We observe that if Zeus successfully creates this file, it will continue writing malicious bytes into that file using bytes in its resource and start a new process using this file.

Delivery: We deliver a HoneyResource by deliberately creating `sdra64.exe` at an end host. This file is owned by a super user and does not allow any creation operation by others. In this way, our HoneyResource prevents Zeus's attempt to start the malicious process.

11.6.4.2 Mutex-Based HoneyResource

One mutex HoneyResource is for Conficker, which is an algorithm-deterministic HoneyResource. This mutex HoneyResource can efficiently stop Conficker's infection at its initialization stage.

Several other mutex examples include `_AVIRA_21099`, `_AVIRA_2109`, `_AVIRA_2108`, which belong to Zeus/Zbot[6] malware. This set of HoneyResource can stop multiple malware logic such as kernel injection, process hijacking, and network communication.

Delivery: Direct injection is an efficient approach to deliver mutex HoneyResource. We simply create a deterministic `_AVIRA_` mutex in the system to prevent

Zbot's injection. For Conficker, we run the HoneyResource slice once at the end host and generate the mutex name for each computer.

11.6.5 HoneyResource Effect Analysis

In this test, we evaluate the effect of our HoneyResource on the malware samples. As reported in Sect. 11.6.3, our HoneyResource can stop or weaken 210 samples' malicious behaviors. In this test, we run these 210 samples in both deployed environment and the normal infection environment for 5 min. Then, we compare the differences of their native system calls (all the NT native calls) in these two environments. We define a metric Behavior Decreasing Ratio, $BDR = \frac{N_n - N_d}{N_n}$, where N_n is the number native system calls in the normal environment, while N_d is that number in the deployed environment. The larger BDR is, the more reduction of functions by the HoneyResource. In Fig. 11.4, we report the distribution of BDR according to different effectiveness type.

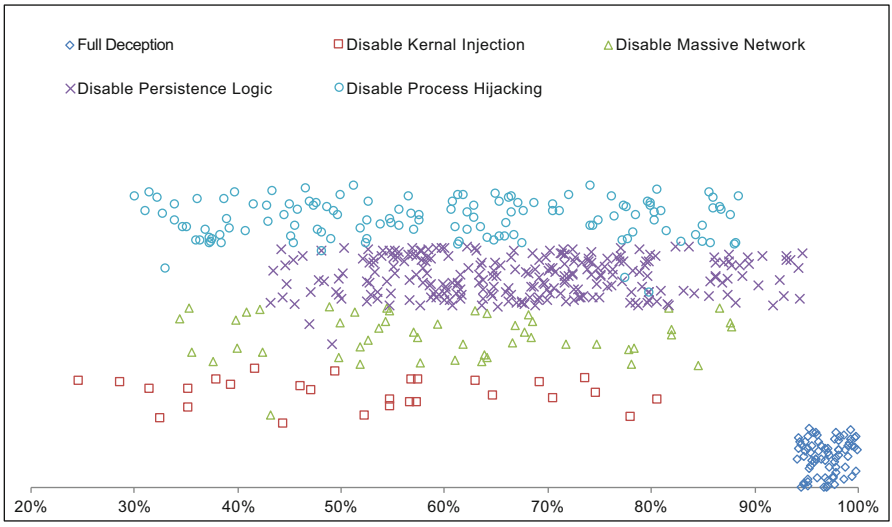

Fig. 11.4 Distribution of BDR

From this figure, we can see that the full deception HoneyResource are obviously the most effective ones and they all terminate the execution of malware (the reason why their BDR is not 100% is simply because of their initial executions before exit that also have some native system calls). Our partial deception HoneyResource all effectively achieve their goals by disabling key functions in the malware (through a careful manual examination, we confirm that all unwanted malicious logic has been disabled). One such example for Zeus is shown in Table 11.6. Even in the *worst* case

Table 11.6 Example of a high-profile Malware HoneyResource

Malware	HoneyResource	Type	Impact description
Zeus/Zbot	_AVIRA_2109	Mutex	Stop process hijacking

in terms of BDR, our partial deception HoneyResource can still reduce at least 24% malware's *important* system call activities. Note that BDR will certainly increase if we keep running the malware sample in a longer time period.

To further verify that our HoneyResource are effective for different variants in the same malware family, we choose 6 high-profile malware samples and perform another test. These samples are high-profile malware such as Conficker, Zeus/Zbot, and Sality, and for these 6 samples we have extracted a total of 17 different HoneyResource in our previous test. We then further collect 5 variants (binaries are different from what we have collected in the original dataset) belonging to each family (thus 30 new variants in total). Then, we run the 30 newly collected variants in both normal and deployed environments, similar to the previous experiment. We carefully analyze the execution differences and manually verify that whether the injected HoneyResource have achieved the goal or not. The result is showed in Table 11.7. Note that the 4th column indicates the number of malicious functions that can be stopped if ideally these HoneyResource work for all variants, the 5th column indicates the actual number from our test, and the 6th column shows the percentage of success.

From the result, we can see that overall our HoneyResource can take effect in almost all variants. However, we do find that some HoneyResource can work for some variants but fail on others. One example is the file HoneyResource `sdra64.exe` which we did not find its use in 2 other Zbot variants. Fortunately, for each malware, we have extracted more than one HoneyResource. Thus, even some may not be effective for all variants, the combination of these HoneyResource can still achieve satisfiable results. We believe that this test also highlights the importance of using an automatic tool (such as our AUTOVAC) to analyze malware samples to extract as many HoneyResource as possible, a goal otherwise very hard to achieve through manual analysis.

Table 11.7 Effectiveness evaluation on Malware variants

Malware	HoneyResource#	Type	Ideal case	Verified	Ratio
Zeus/Zbot	6	Mutex, file	30	23	77%
Conficker	2	Mutex	10	10	100%
Qakbot	2	Registry	10	10	100%
IBank	1	File	5	5	100%
Sality	3	Mutex, file	15	12	80%
PosionIvy	3	Mutex, file	15	10	67%
Total	17		85	70	82%

11.6.5.1 False Positive Test

Our next test is on the false positive evaluation, i.e., whether our generated HoneyResource will affect the normal program executions. We design a simple malware clinic test as mentioned in Sect. 11.4.4.

First, we install 5 different virtual machines running over 40 benign software (which includes the most common software typically seen on normal users' computers such as all kinds of browsers, programming environments, multimedia applications, Office toolkits, IM and social networking tools, anti-virus tools, and P2P programs). Then, we equally inject our HoneyResource into each test machine and monitor their system logs over a period of a week. The result shows that our HoneyResource did not cause any problem to our running environments.

One could argue that this automatic test may underestimate users' interaction. Hence, we conduct another test to install 200 HoneyResource on 4 lab machines. All these four machines are for normal everyday use. The result also shows that our generated HoneyResource did not cause any trouble for the operation of existing benign programs. While our clinic test could have a limited scope, we believe that a well-designed clinic test is still helpful to refine our automatically generated HoneyResource in a real-world scenario.

11.6.6 Performance Overhead

11.6.6.1 HoneyResource Generation Overhead

First, we measured the overhead of the automatic extraction. We run our test on machines with Intel Core i5 CPU and 6GB memory.

- **Generating the HoneyResource** In our test, we measure the time spent on analyzing the function traces, extracting the identifiers and filtering out common identifiers using search engine and pre-built whitelist. For each sample, it took 789 s to fulfill all these tasks on average. For backward slicing, we find that it took 214 s on average for each identifier. Meanwhile, the longest case is 530 seconds and the shortest case is 30 s.
- **Impact Analysis** We measure the overhead of our offline parsing part to handle two execution traces with 1-min malware running time. The overhead for 500 cases is around 24 h. It means that for each case, it takes around $2 - 3$ min to verify its impact.

We note that the generation is a one-time effort in the analysis environment. The more important overhead that users care about is the one on their end hosts.

11.6.6.2 Deployment Overhead

We now report the deployment overhead on each end host.

For *static and algorithm-deterministic HoneyResource*, the overhead is negligible (almost zero) because in most of the time we only need to install some system resource or replay the resource-identifier-generation slice *for one time*. In our experiment, it takes only 34s to install all the 373 static HoneyResource onto one end-host machine. It includes copying/activating the resources and correctly setting up their privileges. For 44 algorithm-deterministic HoneyResource, we need to run program slices on the machine. It takes 1,131s (25.70s for each HoneyResource on average) to deploy all the HoneyResource. Note these HoneyResource are packed with installation scripts and there are no user interactions involved.

For *partial static* HoneyResource, it adds a little more overhead to the end host. The overhead mainly comes from the identifier comparison after we intercept the call. In our test, the highest extra overhead is below 4.5% for injecting 119 partial static HoneyResource. Among 4.5% overhead, around 3.9% comes from the function hooking, which is relatively stable even the HoneyResource number increases. Hence, it could be expected that even the number of partial static HoneyResource has been expanded by 10 times, we could still efficiently control the overhead under 12% for each host. More importantly, in most cases, we do not need to inject all the HoneyResource at the same time (to be discussed in Sect. 11.7).

11.7 Limitations and Future Work

Our system is not perfect. In this section, we discuss its limitations and outline our future efforts.

11.7.1 Evasions from Malware

It is possible to evade our HoneyResource if malware authors are aware that we are using certain resource as the HoneyResource. They can drop the specific resource checking logic or change the resource name in the new version. However, the former will possibly lead to re-infection and thus may be not desired. While the latter approach is possible, if we consider the wide and random propagation of worm or botnet malware, our HoneyResource still makes the malware harder to decide whether the system has actually been infected or not. Hence, if the malware binary cannot run when over two instances on the same machine, our HoneyResource can bring the malware into a dilemma that the target system may have actually been infected before or it has installed our HoneyResource system. Even though malware

can run with multiple instances, periodically changing the identifiers may finally result in multiple instances running in one machine. It also creates extra risks of being detected.

Certainly, malware authors could obfuscate the malware code to frustrate our HoneyResource generation such as using control dependence to propagate data [24]. In fact, in some cases, there is actually no propagation chain and the conditional check is directly operated with the resource values. While future malware could deliberately introduce additional data propagation and obfuscate through control dependence, to address such problem will be one of our future efforts.

11.7.2 Limitation on Dynamic Analysis

In AUTOVAC, we intensively apply multiple data flow tracking techniques such as taint analysis and program slicing. Therefore, AUTOVAC unavoidably suffers from the problems brought by these dynamic analysis techniques [13]. For instance, in our candidate selection/analysis, our taint analysis could cause *overtainting* [7] thus resulting in more candidate resources to analyze. Fortunately, due to our impact analysis and exclusive analysis, we can still easily filter out those unsuitable HoneyResource.

In addition, some imprecise interpretation of differential function calls may cause the underestimation of the actual impact of certain resources. Some previous work [22] has discussed several approaches to gain a better understanding of malware's high-level behaviors. We could leverage these techniques to refine our result in future work.

11.7.3 Potential False Positive

Some of our automated analysis techniques (e.g., the use of search engine) may also return incomplete/inaccurate results. Meanwhile, our exclusiveness analysis and clinic test may not cover all benign programs such that it is possible to have some resource collision between our HoneyResource and some benign programs. Improving these issues is our another venue of future work.

11.7.4 Deployment Issues

One concern for the HoneyResource deployment is that injecting a large number of HoneyResource into end hosts may annoy the user. Note that most generated

HoneyResource in practice are just some files, mutexes, and registry entries, whose sizes are tiny or even with 0 byte. This is pretty lightweight compared with the case that AV tools typically store millions of signatures on an end host. In addition, as mentioned before, as a complementary technique to existing solutions, our prevention scheme can be mainly used for some high-profile, large-scale, and severe malware infections, instead of for all malware.

11.7.5 Deception Goals

In this work, we mostly focus on how to use HoneyResource to stop or impact the proper infection/execution of malware for the defense purpose. While these are important deception goals, many times we are also interested in deceiving malware into continuing the normal execution so that we can observe more activities, understand the intention from the malicious operators behind malware, or even mislead them for the defense purpose. Our future work will investigate more on this direction.

11.8 Related Work

11.8.1 Immunization-Based Defense

In [14], Manuel et al. proposed an end-to-end approach to make end hosts immune from fast-propagating worms through collaborative worm detection and self-certifying alerts. Packet Vaccine [32] followed this direction and derived the network signatures of malicious packets to be used at the network level to filter unwanted packets. Different from these previous work, AUTOVAC does not investigate the exploits nor vulnerabilities that malware targets, and instead it analyzes the system resource constraints of malware and attempts to extract effective HoneyResource to immunize a clean system from future malware infection.

In a concurrent study, Andre et al. [28] proposed the idea of using infection markers to prevent malware infection. While both are inspired by the biological vaccine concept, we systematically explore this problem and our HoneyResource are more general and broader than simple infection markers. Employed techniques are also substantially different; instead of treating the malware as a black box, AUTOVAC conducts more fine-grained binary analysis on malware internals, performs more analysis (e.g., exclusiveness, impact) in the automatic HoneyResource generation, and has more delivery/deployment options.

11.8.2 Dynamic Malware Analysis

Due to the severe threat of malware, tons of research has been carried out on analyzing malware behavior (e.g., [9, 12, 16, 20, 22]) and classifying malware (e.g., [15, 19, 30]). Certainly, AUTOVAC complements these techniques by exploring a new direction to stop malware infections.

In AUTOVAC, we design several dynamic binary analysis techniques to automate the production of malware HoneyResource. There has been a significant amount of work [11, 16–18, 21, 29] on dynamic binary analysis. In particular, prior research [23, 29] has explored the enforced execution and reverting to trigger malware's dormant functions [23, 29]. Our enforced execution applies similar techniques introduced in the forced execution [29] but we focus on these *environment/system resource-sensitive* branches.

We also leverage taint analysis and program alignment techniques. Different from full taint analysis in the previous work [17, 18] and block-level program alignment [25], our proposed solution avoids the overhead caused by full execution tracking with a particular focus on the targeted malware behavior in our problem domain.

11.9 Conclusion

In this chapter, we present AUTOVAC, a new complementary malware defense scheme that aims to automatically extract malware HoneyResource from given malware samples. Our evaluation shows that it is an appealing approach that works on many real-world malware families. In particular, the HoneyResource can be used to deceive malware for stopping its infection. To demonstrate the real-world practicability, we have implemented our prototype system using several dynamic program analysis techniques, and conducted empirical evaluations on a large set of real-world malware samples. Our experimental results show that we can successfully extract working HoneyResource for many malware families including Conficker, Sality, and Zeus.

11.10 Exercise

Ex. 1 Discuss the reasons why malware authors want to avoid duplicate infection? What is the effect of duplicate infection?

Ex. 2 Analyze the following assembly code. Is there any memory/register that will be tainted by AUTOVAC after the execution? Why?

Listing 1: Code Example

```
msg     byte    "mutex_test", 10
handle  dword   ?

        section .text
go:
        push    msg
        push    dword 0
        push    dword 0
        call    _OpenMutex
        xor      eax, eax
        push    -11
        call    _GetStdHandle
        mov     handle, eax
```

Ex. 3 Review the reference paper [7] and Sect. 11.7, and discuss how the limitation of tainted analysis may affect the effectiveness of AUTOVAC.

Ex. 4 Similar to biological HoneyResource which commonly has some side effect for patients, *malware HoneyResource* could also have some side effect on a user's system. Discuss possible side effects and how to prevent them.

Acknowledgements An early version of this chapter appeared in ICDCS'13 [31] . This research is partially supported by NSF (Grant No. CNS-0954096), AFOSR (Grant No. FA9550- 13-1-0077), and DARPA (Grant No. 12011593). All opinions, findings, and conclusions or recommendations expressed herein are those of the authors and do not necessarily reflect the views of NSF, AFOSR, or DARPA.

References

1. Anubis: Analyzing Unknown Binaries. https://seclab.cs.ucsb.edu/academic/projects/projects/anubis/.
2. DynamoRIO . http://dynamorio.org/.
3. malc0de. http://malc0de.com/database/.
4. Temu . http://bitblaze.cs.berkeley.edu/temu.html.
5. Virustotal. https://www.virustotal.com/.
6. Zeus Trojan horse. http://en.wikipedia.org/wiki/Zeus_(Trojan_horse).
7. T. Avgerinos, E. Schwartz, and D. Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Proc. of IEEE S&P 2010*.
8. A.Zeller. Isolating cause-effect chains from computer programs. In *Proc. of the 10th ACM SIGSOFT symposium on Foundations of Software Engineering*, 2002.

9. U. Bayer, P. Milani, C. Hlauschek, C. Kruegel, and E. Kirda. Scalable, behavior-based malware clustering. In *Proc. of NDSS'09*, 2009.
10. D. Brumley, I. Jager, T. Avgerinos, and E. J. Schwartz. BAP: A binary analysis platform. In *Proceedings of Computer Aided Verification (CAV)*, July 2011.
11. J. Caballero, P. Poosankam, C. Kreibich, and D. Song. Dispatcher: Enabling active botnet infiltration using automatic protocol reverse-engineering. In *Proc. of ACM CCS'09*, 2009.
12. Davide Canali, Andrea Lanzi, Davide Balzarotti, Christopher Kruegel, Mihai Christodorescu, and Engin Kirda. A quantitative study of accuracy in system call-based malware detection. In *Proc. of International Symposium on Software Testing and Analysis*, 2012.
13. L. Cavallaro, P. Saxena, and R. Sekar. On the limits of information flow techniques for malware analysis and containment. In *DIMVA 2008*.
14. M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham. Vigilante: End-to-end containment of internet worms. In *Proc. of SOSP'05*, pages 133–147, Brighton, United Kingdom, 2005.
15. M. Fredrikson, J. Somesh, M. Christodorescu, R. Sailer, and X. Yan. Synthesizing near-optimal malware specifications from suspicious behaviors. In *Proc. of the 2010 IEEE Symposium on Security and Privacy*, 2010.
16. S. T. King and P. M. Chen. Backtracking intrusions. In *Proceedings of ACM Symposium on Operating Systems Principles*, October 2003.
17. C. Kolbitsch, P. Milani Comparetti, C. Kruegel, E. Kirda, X. Zhou, and X. Wang. Effective and efficient malware detection at the end host. In *Proc. of USENIX Security'09*, 2009.
18. C. Kolbitsch, T. Holz, C. Kruegel, and E. Kirda. Inspector gadget: Automated extraction of proprietary gadgets from malware binaries. In *Proc. S&P'10*, 2010.
19. J. Zico Kolter and Marcus A. Maloof. Learning to detect and classify malicious executables in the wild. *J. Mach. Learn. Res.*, 7:2721–2744, December 2006.
20. A. Lanzi, D. Balzarotti, C. Kruegel, M. Christodorescu, and E. Kirda. Accessminer: using system-centric models for malware protection. In *Proc. of the 17th ACM CCS*, 2010.
21. Z. Lin, X. Zhang, and D. Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 17th Annual Network and Distributed System Security Symposium (NDSS'10)*, San Diego, CA, February 2010.
22. L. Martignoni, E. Stinsony, M. Fredrikson, S. Jhaz, and J. C.Mithchelly. A layered architecture for detecting malicious behaviors. In *RAID 2008*.
23. A. Moser, C. Kruegel, and E. Kirda. Exploring Multiple Execution Paths for Malware Analysis. In *Proc. S&P'07*, 2007.
24. M.Sharif, A. Lanzi, J. Giffin, and W. Lee. Impeding malware analysis using conditional code obfuscation. In *Proc. NDSS'08*, 2008.
25. N.Johnson, J.Caballero, Z.Chen, S.McCamant, P.Poosankam, D.Reynaud, and D.Song. Differential slicing: Identifying causal execution differences for security applications. In *Proceedings of the 2011 IEEE Symposium on Security and Privacy*, 2011.
26. P. Porras, H. Saidi, and V. Yegneswaran. An Analysis of Conficker's Logic and Rendezvous Points. http://mtc.sri.com/Conficker/, 2009.
27. I. Trestian, S. Ranjan, A. Kuzmanovic, and A. Nucci. Unconstrained Endpoint Profiling (Googling the Internet). In *ACM SIGCOMM'08*.
28. A. Wichmann and E. Gerhards-Padilla. Using infection markers as a vaccine against malware attacks. In *Proc. of the 2nd workshop on Security of Systems and Software resiLiency*, 2012.
29. J. Wilhelm and T. Chiueh. A forced sampled execution approach to kernel rootkit identification. In *Proc. of RAID'07*, 2007.
30. H. Xin, C. Tzi-cker, and S. Kang G. Large-scale malware indexing using function-call graphs. In *Proc CCS '09*, 2009.
31. Z. Xu, J. Zhang, G. Gu, and Z. Lin. Autovac: Towards automatically extracting system resource constraints and generating vaccines for malware immunization. In *Proceedings of the 33rd International Conference on Distributed Computing Systems (ICDCS'13)*, Philadelphia, July 2013.
32. X.Wang, Z.Li, J.Xu, M.Reiter, C.Kil, and J.Choi. Packet vaccine: black-box exploit detection and signature generation. In *Proc CCS'06*, 2006.

Correction to: Autonomous Cyber Deception

Correction to:
E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8

This book was inadvertently published as an authored work with the chapter authors mentioned in the footnotes of the chapter opening pages. This has now been updated and the chapter authors have been mentioned in the respective chapter opening pages as mentioned below:

Chapter 1 Using Deep Learning to Generate Relational HoneyData
Nazmiye Ceren Abay, Cuneyt Gurcan Akcora, Yan Zhou, Murat Kantarcioglu, and Bhavani Thuraisingham

Chapter 2 Towards Intelligent Cyber Deception Systems
Fabio De Gaspari, Sushil Jajodia, Luigi V. Mancini, and Giulio Pagnotta

The updated online versions of the chapters can be found at
https://doi.org/10.1007/978-3-030-02110-8_1
https://doi.org/10.1007/978-3-030-02110-8_2
https://doi.org/10.1007/978-3-030-02110-8_3
https://doi.org/10.1007/978-3-030-02110-8_4
https://doi.org/10.1007/978-3-030-02110-8_5
https://doi.org/10.1007/978-3-030-02110-8_6
https://doi.org/10.1007/978-3-030-02110-8_7
https://doi.org/10.1007/978-3-030-02110-8_8
https://doi.org/10.1007/978-3-030-02110-8_9
https://doi.org/10.1007/978-3-030-02110-8_10
https://doi.org/10.1007/978-3-030-02110-8_11
https://doi.org/10.1007/978-3-030-02110-8

E. Al-Shaer et al. (eds.), *Autonomous Cyber Deception*,
https://doi.org/10.1007/978-3-030-02110-8_12

Chapter 3 Honeypot Deception Tactics
Neil C. Rowe

Chapter 4 Modeling and Analysis of Deception Games Based on Hypergame Theory
Jin-Hee Cho, Mu Zhu, and Munindar Singh

Chapter 5 Dynamic Bayesian Games for Adversarial and Defensive Cyber Deception
Linan Huang and Quanyan Zhu

Chapter 6 CONCEAL: A Strategy Composition for Resilient Cyber Deception: Framework, Metrics, and Deployment
Qi Duan, Ehab Al-Shaer, and Mazharul Islam

Chapter 7 NetShifter: A Comprehensive Multi-Dimensional Network Obfuscation and Deception Solution
Gahng-Seop Ahn, Kyung Joon Kwak, Alexey Bogaevskiy, Jason Li, Gregory Briskin, and Robert Vaeth

Chapter 8 Deception-Enhanced Threat Sensing for Resilient Intrusion Detection
Frederico Araujo, Gbadebo Ayoade, Kevin W. Hamlen, and Latifur Khan

Chapter 9 HONEYSCOPE: IoT Device Protection with Deceptive Network Views
Reham Mohamed, Terrence O'Connor, Markus Miettinen, William Enck, and Ahmad-Reza Sadeghi

Chapter 10 gExtractor: Automated Extraction of Malware Deception Parameters for Autonomous Cyber Deception
Mohammed Noraden Alsaleh, Jinpeng Wei, Ehab Al-Shaer, and Mohiuddin Ahmed

Chapter 11 Malware Deception with Automatic Analysis and Generation of HoneyResource
Zhaoyan Xu, Jialong Zhang, Zhiqiang Lin, and Guofei Gu

Cover corrected to reflect editor information.