GAME-THEORETIC INTRUSION RESPONSE AND RECOVERY

BY

SAMAN ALIARI ZONOUZ

DISSERTATION

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 2011

Urbana, Illinois

Doctoral Committee:

Professor William H. Sanders, Chair and Director of Research
Professor Roy H. Campbell
Professor Tarek F. Abdelzaher
Dr. Himanshu Khurana, Honeywell
Dr. Kaustubh R. Joshi, AT&T Research Labs

ABSTRACT

The severity and number of intrusions on computer networks are rapidly increasing. Preserving the availability and integrity of networked computing systems in the face of those fast-spreading intrusions requires advances not only in detection algorithms, but also in intrusion tolerance and automated response techniques. Additionally, the rapid size and complexity growth of computer networks, and their recently increasing integrations with physical systems signify the quest for systems that detect their own compromises and failures and automatically repair themselves. In particular, the ultimate goal of the intrusion tolerant system design is to adaptively react against malicious attacks in real-time, given offline knowledge about the network's topology, and online alerts and measurements from system-level sensors.

Addressing all the practical and theoretical difficulties in design and deployment of an intrusion response framework in practice is a challenging problem. In particular, at each time instant, the response system needs to accurately determine the current security state of the system, given online sensory information. Moreover, decision upon a proactive strategy against attackers requires the knowledge about possible future attacks, or equivalently, system vulnerabilities and how to monitor and detect exploitations of those vulnerabilities. Additionally, prioritizing a specific response strategy over all other possible strategies demands an algorithm to compare criticality levels of compromised system assets. Finally, an efficient mathematical decision-making framework is needed to select the optimal response strategy by taking into account the possible future exploitations and damages as well as the criticality level of potentially compromised systems assets.

This dissertation proposes a model-based solution to building a theoretically well-founded automated intrusion response and recovery framework in practice. In particular, we present an approach to address each of the abovementioned challenges. In particular, we introduce a security state estimation algorithm for cyber-physical networks that accounts for inherent uncertainties in

intrusion detection systems' alert notifications and sensor measurements. To update the knowledge about the possible future attacks, our proposed intrusion forensics algorithm gradually identifies previously unknown system vulnerabilities and updates the deployed set of monitors every time a zero-day exploitations happens. Additionally, we introduce a consequence-centric security metric to automatically determine criticality level of the compromised system assets in any system state. Finally, to choose online countermeasure actions, we propose an intrusion response and recovery solution which employs a game-theoretic response strategy against adversaries by modeling the interaction between the system and the attacker as a two-player sequential game. The engine chooses optimal response actions by solving a partially observable competitive Markov decision process model.

We validate the proposed solutions using real-world implementations, and showed that the proposed response system can efficiently recover the compromised computer networks automatically back to their normal operational mode.

To my parents, for their love and support.

ACKNOWLEDGMENTS

I feel fortunate to have had the opportunity to work with a talented and dedicated group of people in the PERFORM group. First and foremost, I would like to thank my advisor Prof. William H. Sanders for supporting my endeavors and giving me the freedom to pursue my interests. He has been a constant source of encouragement, ideas, and advice throughout my time with the PERFORM group. I would also like to thank Dr. Himanshu Khurana, Tim Yardley, Dr. Robin Berthier, Dr. Kaustubh Joshi, and Dr. Harigovind Ramasamy who co-advised me on my work. I thank them for their insight and advice, and for ensuring that I stayed focused in my research pursuits. This work would not have been possible without them.

I thank Jenny Applequist for her tremendous help that is not completely expressible, but will always be appreciated. Many thanks go to Dr. Shravan Gaonkar and Eric Rozier for our technical discussions we have had on cyber security, and also for being great people to share an office with. It has been a pleasure to have known them. Additionally, I would like to appreciate my new officemates, Michael Ford and Dr. Elizabeth LeMay, for making the office an enjoyable place to do research by countless motivating scientific discussions.

I also thank the members of my committee, Professor William H. Sanders, Professor Roy H. Campbell, Professor Tarek F. Abdelzaher, Dr. Himanshu Khurana, and Dr. Kaustubh R. Joshi for the time they spent in reviewing this thesis. Their constructive and technical feedback helped immensely in giving me direction, and strengthening both theoretical and systems-oriented aspects of my Ph.D. research.

I would also like to thank all the people with whom I enjoyed collaborating on different research projects and writing research papers. In particular, I am very grateful to Dr. Robin Berthier, Dr. Himanshu Khurana, Dr. Kaustubh Joshi, Timothy M. Yardley, Professor Peter W. Sauer, Kate Rogers, Dr. Rakesh Bobba, Professor Nikita Borisov, Amir Houmansadr, Dr. Parisa Haghani,

Aashish Sharma, Dr. Zbigniew T. Kalbarczyk, Dr. HariGovind V. Ramasamy, Dr. Birgit Pfitzmann, Dr. Kevin McAuliffe, Professor Ravishankar K. Iyer, Dr. Eric Cope, Sankalp Singh, Professor Thomas J. Overbye, and Professor Alejandro Dominguez-Garcia.

I would like to thank the funding agencies that have financially supported my research. This material is based upon work supported by the National Science Foundation, Department of Energy, and Department of Homeland Security under Grant Number CNS-0524695, as part of the Trustworthy Cyber Infrastructure for Power Center, and by the Department of Energy and Department of Homeland Security under Award Number DE-OE0000097 as part of the Trustworthy Cyber Infrastructure for Power Grid.

The members of the PERFORM group have always provided me with an atmosphere that was both stimulating and enjoyable. I thank Dr. Robin Berthier, Doug Eskins, Michael Ford, Dr. Ryan Lefever, Dr. Shravan Gaonkar, Eric Rozier, Sankalp Singh, Ken Keefe, Dr. Elizabeth (Van Ruitenbeek) LeMay, Dr. Kaustubh Joshi, and Dr. Shuyi Chen for their friendship, and for making my stay here enjoyable.

I thank my lovely parents, Ali and Monireh, my caring sister, Marina, and my precious brother, Keyvan for the encouragement they have given me over the years, and for their constant support. This thesis would not have been possible without their support.

Last but not least, thank you Parisa for your love, wonderful companionship, invaluable support, everlasting energy, stable perseverance, and most importantly, motivating happiness!

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1

INTRODUCTION

The severity and number of intrusions on computer networks are rapidly increasing. Incident-handling techniques [4] can be categorized into three broad classes. First, there are intrusion prevention methods that take actions to prevent occurrence of attacks, e.g., network flow encryption to prevent man-in-the-middle attacks. Second, there are intrusion detection systems (IDSes), such as Snort [5], which try to detect inappropriate, incorrect, or anomalous network activities, e.g., perceiving CrashIIS attacks by detecting malformed packet payloads. Finally, there are intrusion response techniques that take responsive actions based on received IDS alerts to stop attacks before they can cause significant damage and to ensure safety of the computing environment. So far, most research has focused on improving techniques for intrusion prevention and detection, while intrusion response usually remains a manual process performed by network administrators who respond to intrusions after receiving notifications via IDS alerts. This manual response process inevitably introduces some delay between notification and response, which could easily be exploited by the attacker to achieve his or her goal and significantly increase the damage a system [6]. Therefore, to minimize the severity of attack damage resulting from delayed response, an automated intrusion response is required that provides quick response to intrusion.

During the last five years, three types of techniques aimed at enhancing automation in intrusion response have been proposed. The majority of those techniques are based on lookup tables filled with predefined mappings, e.g., (response actions, intrusion alerts) [7, 8]. Such methods allow response systems to deal with intrusions quickly. However, they suffer from a lack of 1) flexibility, mainly because these systems completely ignore the intrusion cost factor, and 2) scalability, since it is infeasible to predict all the alert combinations from IDSes in a large-scale computer network. A second group of intrusion response systems (IRSes) employs a dynamic rule-based selection procedure [9] that selects response actions based on a certain attack metric, e.g., confidence or

severity of attack. Finally, there has been increasing interest in developing cost-sensitive models of response selection [10, 11]. The main objective in applying such a model is to compare intrusion damage and response cost to ensure system recovery with minimum cost without sacrificing the normal functionality of the system under attack.

In this thesis, we propose an automated online intrusion response system. The main responsibility of the response system is to receive alerts from distributed IDSes in a network, and to mitigate detected problems by selecting and carrying out relevant recovery actions based on those alerts. To design and deploy an automated intrusion response system that decides upon optimal response actions and recovers the system after it has been compromised, several challenges must be addressed.

1.1 Challenges

In this section, we discuss some of main challenges in designing a practical and theoretically sound intrusion tolerance solution for critical large-scale cyber-physical networks, in which quick responses are often needed when an attack occurs.

To decide upon how to react optimally against the attackers, the response system needs to know the current security state[1] of the system accurately, given the past measurements and triggered alerts from sensors and deployed security monitors. the security state estimation problem becomes even more challenging in cyber-physical networks, e.g., power grid infrastructures, in which accurate determination of the the system's current state requires fusion of information from different *types* of sensors, i.e., cyber-side intrusion detection monitors and power-side sensors. In particular, the algorithm needs to consider the process control network's topology and its access control policy configuration, as well as the underlying power system's model and the flow equations.

As continuous monitoring of all aspects of the system is often infeasible because of monitors' computational costs, a cost-optimal set of monitors and sensors should be selected, deployed, and used dynamically. Furthermore, to reason about the attacker's future malicious actions, and, more

[1]Throughout the thesis, we will use the following definition of the system's security state. At each time instant, the security state consists of two types of state variables: 1) the past consequences, e.g., a Web server process crash, caused by the attacker in that state; and 2) the attacker's privileges, e.g., root access on host A, in that state.

specifically, to determine the set of possible vulnerability exploitations from the current system state, system vulnerabilities must either be already known or be identified by the response system once they have been exploited by zero-day attacks. In general, there are two main challenges to effective attack detection and vulnerability identification: cost and coverage. For instance, in the case of detecting buffer overflows, there are relatively inexpensive techniques with imperfect coverage, e.g., address space randomization, and expensive methods with excellent coverage, e.g., strict bounds checking. However, it is difficult to find a technique that achieves both low cost and perfect coverage. The scope of a detection mechanism also impacts its coverage. Techniques that only monitor a single process (e.g., a web server) may not be able to detect attacks such as SQL injection, regardless of the cost, because the attacks do not produce symptoms in the monitored process. Finally, as one broadens the range of vulnerability types to be detected, multiple types of detection mechanisms are often required, and the costs associated with each individual detector become an even more important factor.

Additionally, to mark and take one of the possible actions from the system's current state as the optimal one, the response system must be aware of the state to which each action transitions the system, and needs a metric to quantify how secure that state is. To be practical for large-scale networks, determining the state transitions and the security assessment needs to minimize their reliance on human knowledge and involvement. As discussed later, complete automation of the security metric definition are not usually feasible, because criticality levels of different system assets heavily depend on the organizational business goals which are subjective and can not be automatically deduced accurately. Furthermore, to accurately assess the system security and determine whether the critical system assets are directly or indirectly affected as result of an attack, system assets' inter-dependencies should be taken into account. So, for example, an indirect modification of a sensitive system file by a legitimate process spawned by a malicious process is taken into consideration during the security assessment.

Finally, given all the required information, the response system needs a practically scalable and mathematically provable algorithm to choose and carry out the cost-optimal response action in the system that it is protecting. In particular, the response selection strategy must account for planned adversarial behavior in which attacks occur in stages in which adversaries execute well-planned strategies and address defense measures taken by system administrators along the way. Moreover,

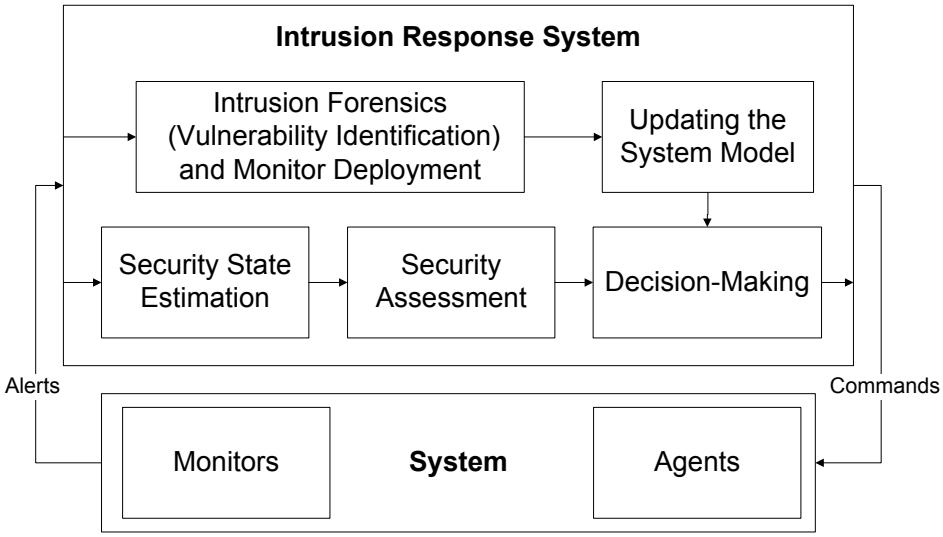


Figure 1.1: High-level Conceptual Architecture of an Intrusion Tolerance Solution

it has to account for inherent uncertainties in IDS alert notifications and sensor measurements. That is important because sensors and IDSes today and in the near future will be unable to generate alerts that match perfectly to successful intrusions, and response techniques must therefore allow for this imperfection in order to be practical.

Figure 1.1 shows a high-level overview of a hypothetical intrusion response system, and how its different components, representing the abovementioned challenges, are logically interconnected. In this thesis, we introduce and validate an approach to each of the challenges, and highlight benefits and limitations of each solution compared to the related literature.

1.2 Contributions

In this section, we briefly describe the contributions of this dissertation, and discuss which of the abovementioned challenges each contribution addresses. The overall contribution of this dissertation is the development of a theoretical model-based framework to perform automated intrusion recovery and adaptation, even when the system's state is not precisely known. Our thesis is that a game-theoretic-based automated intrusion response and recovery system can provide a practical and theoretically sound intrusion tolerance solution for cyber-physical critical infrastructures that often need quick responses once they are attacked. In brief, the contributions of this dissertation

are as follows:

- **Optimal Response Action Selection.** We present a mathematical decision-making and optimal action selection framework that models the security battle between the system and the attacker as a multi-step, sequential, hierarchical, non-zero-sum, two-player stochastic game.
- **A Consequence-centric Security Metric.** We introduce a consequence-based security metric to assess security of each possible state of the system with a minimum possible set of manual inputs from system administrators.
- **Managing Unknown Exploitations.** We propose an theoretically provable intrusion forensics technique that a response system can use to gradually identify previously unknown vulnerabilities once they have been exploited by zero-day attacks, and deploy a cost-optimal set of IDSes in the network efficiently and dynamically.
- **Security state Estimation.** We present an information fusion solution to accurately estimate the security state of a cyber-physical system using online information from cyber-side intrusion detection systems and the physical sensors.
- **Case Study Evaluation.** We evaluate the proposed response engine in an enterprise network environment in which there are business-level response actions, such as hiring of a red team, as well as IT-level actions, such as server reboot.

The contributions in this dissertation enables quick automated reactions against malicious targeted attacks that try to damage the system by taking a sequence destructive actions. In particular, we propose two types of solutions: offline and online.

For the offline phase (before the system is set up), we introduce two techniques to generate system models that are needed later during online operation of the response system. We propose a simple-to-design logical and graphical formalism for system administrators to manually design models for small-scale networks. The designed models are converted to a mathematical decision-making framework later used by the response system to tolerate security attacks. The proposed formalism facilitates the system model design process significantly, as it does not require any system instrumentations. However, manual model generation is not usually suitable for large-scale

networks, since it heavily relies on detailed human knowledge about the system configurations. Our automated model generation techniques minimizes the human involvement by automatically extracting system-level information and learning the system details.

During system's online operation, at each time instant, the current security state of the system is determined, given the generated models and online sensory information. Then, the generated models and knowledge of the current state are used to select and carry out the optimal response strategy. Additionally, our online forensics algorithm is used to gradually update system model based on recently identified system vulnerabilities.

1.3 Game-theoretic Intrusion Response and Recovery Overview

The rapid size and complexity growth of computer networks, and their recently increasing integrations with physical systems motivates the quest for systems that detect their own compromises and failures and automatically repair themselves. As discussed in Section 1.1, addressing all practical and theoretical difficulties in designing a mathematically provable intrusion response framework in practice is a challenging problem. In this section, we give a more detailed overview of our solutions to each challenge represented by a component in Figure 1.1.

1.3.1 Optimal Response Action Selection

The decision-making core of a response system is in charge of decision upon the optimal response strategy. To do so, we propose a mathematical framework, called *Response and Recovery Engine (RRE)*, that models the security battle between itself and the attacker as a multi-step, sequential, hierarchical, non-zero-sum, two-player stochastic game. In each step of the game, the response system leverages a new extended attack tree formalism, called *attack-response tree* (ART), and received IDS alerts to evaluate various security properties of the system. ARTs provide a formal way to describe system security based on possible intrusion and response scenarios for the attacker and response engine, respectively. More importantly, ARTs enable the response system to consider inherent uncertainties in alerts received from IDSes (i.e., false positive and false negative rates) when estimating the system's security and deciding on response actions. Then, the response sys-

tem automatically converts the attack-response trees into partially observable competitive Markov decision processes that are solved to find the optimal response action, in the sense that the maximum discounted accumulative damage that the attacker can cause later in the game is minimized. Using that game-theoretic approach, the response system adaptively adjusts its behavior according to the attacker's possible future reactions, thus preventing the attacker from causing significant damage to the system by taking an intelligently chosen sequence of actions. To deal with security issues with different granularities, the response system's two-layer architecture consists of local engines, which reside in individual host computers, and the global engine, which resides in the response and recovery server and decides on global response actions once the system is not recoverable by the local engines. This hierarchical architecture improves scalability, ease of design, and performance of the response system, so that it can protect computing assets against attackers in large-scale computer networks.

To validate our proposed decision-making algorithm empirically, we evaluated it in a simulated enterprise network environment in which there are some business-level cost functions and response actions, such as hiring of a red team, as well as IT-level actions, such as server reboot. We chose three main IT system-level attack classes from a pool of 150 security incidents observed in the National Center for Supercomputing Applications (NCSA) at the University of Illinois over a five-year period [12], and implemented an incident-replay engine to simulate their occurrence in the enterprise.

1.3.2 Consequence-centric Security Metric

The main role of the security assessment component in a response system is to score security of the current system state at each time instant. The proposed approach, called *Seclius*, makes use of information flows among system assets, such files and processes, to enable response systems to assess system security before deciding upon actions. The only input that the security assessment algorithm needs from the administrators is a list of major mission-critical assets in the organization. Indeed, this input is subjective and thus impossible to generate automatically via observation of the system. In particular, the list of critical assets is provided in a tree structure called the *consequence tree (CT)*. Even in very large organizations, the CT usually contains very few assets, e.g., a Web

server; the administrators do not need detailed system-level expertise to list all other assets on which the mission-critical assets depend, since the dependencies, as system-level characteristics, are learned during an offline learning phase. In particular, during the learning phase, the assessment engine captures the normal communication patterns of individual assets in the system. It does so by tracing the information flow, at a time when there is no attack, and then automatically developing a dependency graph, i.e., a Bayesian network, that represents the system characteristics when there is no attack. Later, while the system is operating, given the learned dependency graph and the mission-critical assets, the assessment engine evaluates the security score for the current state of the system, based on the triggered IDS alerts. In particular, to evaluate the security of the system, the assessment engine employs a taint-tracking approach by running a Bayesian belief propagation algorithm, i.e., Monte Carlo Gibbs sampling [13], on the learned dependency graph to calculate the probability that the critical assets are still secure.

The security assessment engine separates security requirements, i.e, a subjective definition of the mission-critical assets of the organization, from system characteristics, i.e., an identification of the low-level relationships among system components. In fact, the separation enables the assessment algorithm to minimize required manual inputs. Furthermore, given the low-level state estimate, the assessment engine evaluates the high-level security of the whole system according to a two-layer[2], consequence-centric metric function that calculates how much the mission-critical assets are affected in that state. In addition to providing security assessment and situational awareness, the engine is useful for ranking IDS alerts, and for highlighting the most critical affected organizational assets that the response system should handle first. We note that the ranking is possible because the assessment engine captures and considers dependencies between the assets, thus providing a precise understanding of the scope and security impact of IDS alerts.

1.3.3 Managing Unknown Exploitations

The on-line intrusion forensics and on-demand detector selection component in a response system enables the response system to gradually identify previously unknown system vulnerabilities and

[2]The top part of the metric function is the tree structure (the security requirements) whose leaf nodes represent vertices in the bottom part, which is the learned Bayesian network (system characteristics).

deploy the right intrusion detectors dynamically in a cost-effective manner when the system is threatened by an exploit. The proposed forensics solution, called *FloGuard*, relies on often easy-to-detect symptoms of attacks, e.g., participation in a botnet or DDoS, and works backwards by iteratively deploying off-the-shelf detectors closer to the initial attack vector. The forensics engine takes a system-wide, cost-sensitive approach to attack detection and tracing. It uses the observation that although it may be difficult to notice the immediate effects of an attack inexpensively and accurately, the ultimate consequences of attacks are often easier to detect. For example, inexpensive in-network scanning techniques can detect participation in a DDoS attack, botnet, or infection by a worm. Malware scanners can detect the artifacts produced by previously known payloads (even if the attack vector is unknown), and in-host anomaly detectors can detect deviations in a system's performance. Once an attack has been detected in this manner, the forensics engine rolls the system back to a clean checkpoint[3], employs a forensics algorithm to determine possible paths the attack could have taken to reach its detection point, and deploys additional security detectors to catch and detect the attack at an earlier stage the next time that it is attempted. By iteratively repeating this process, it deploys detectors successively closer to the initial attack vector until such a time that the attack can be stopped by automatic prevention techniques, such as input signature generation or quarantine.

To determine which set of detectors to enable and disable in each iteration, the proposed forensics solution uses an *Attack-Graph-Template* (AGT), which is an extended attack graph that enumerates all the *possible* attack and privilege escalation paths in the system. AGTs include any possible paths, not just ones known to be in the particular implementation being protected. For example, an AGT for a server written in C can include privilege escalation using a buffer-overflow exploit, even though there may be no such known vulnerability. Therefore, it is possible to construct AGTs automatically by using system call monitoring to track *all* information and control flow paths among a system's privilege levels via processes, sockets, and files. During the forensics phase, the engine determines which detectors to deploy for the next round by solving an optimization problem based on the outputs of the deployed detectors, the cost and coverage of the unused

[3]We define the clean snapshot to be the last system snapshot taken before the introduction of any potential attack sources (entry points), e.g., a malicious socket connection that originates an intrusion. The set of potential attack sources is determined according to a reverse data flow analysis, using the logged syscalls going backward from the detection point.

detectors, and the paths encoded by the AGT.

1.3.4 Security State Estimation

Using the security state estimation component, a response system determines the current security state of the system. In complex large-scale systems, determining the current security state of the system from a set of online sensory information is not always a simple task. As a case in point, current state identification in cyber-physical systems, which feature a tight combination of computational and physical elements, is usually a challenging problem, since sensory information is coming both from cyber IDSes and from the underlying physical components' sensors. In particular, our focus is on power grid critical infrastructures in which the underlying power generation, transmission, and distribution infrastructures are monitored and controlled by a complex cyber network to guarantee that power is correctly delivered to end users.

The cyber-physical security state estimation component makes use of sensory information from both power-side sensors and cyber-side intrusion detectors to identify cyber attacks and potential compromises of power system measurement data. First, in an offline preprocess phase, the framework, called *Cyber-Physical Intrusion Detection System (CPIDS)*, automatically generates an attack graph of the network based on enforced access control policies. The attack graph is later enhanced to update system state according to real-time information from sensors. During operational mode, the engine monitors the physical power and the communication network, detects and analyzes attacks based on the attack graph, and then probabilistically determines a set of computer systems and power system measurements that are likely to have been maliciously compromised. The engine then uses that probabilistic information as a basis for ignoring suspicious measurements to protect the power system state estimator from the potentially malicious data. The IDS reports and the updated power system state estimator outputs enable the proposed framework, at each time instant, to provide power grid situational awareness to the system operators by continuously presenting them with a clear and complete report of the cyber-physical security state of the power grid.

1.4 Thesis Organization

The remainder of this dissertation is structured as follows. Chapter 2 introduces the game-theoretic response selection algorithm to choose the optimal response action using the attack-response tree formalism (Section 2.3.1). Chapter 3 gives a detailed explanation of the security assessment tool that measures the probability of critical assets being affected using a learned system profile (Section 3.3). Chapter 4 introduces the attack graph template model (Section 4.3) and discusses how it is used to identify previously unknown system vulnerabilities. Chapter 5 shows how information from different cyber and physical sensors are fused to estimate a unified state of the system (Section 5.3). Chapter 6 presents a case study evaluation of the proposed intrusion response solution on an enterprise network environment (Section 6.2). Finally, Chapter 7 presents concluding remarks and discusses some possibilities for future expansion of the work described in this dissertation.

CHAPTER 2

RESPONSE AND RECOVERY ENGINE

Keeping systems and networks secure is a continuous arms race against attackers. The growing number of security incidents [12] indicates that current security design solutions fail to sufficiently contain the increasing variety and sophistication of threats and block the attacks before the systems are compromised. Therefore, organizations have a desire to detect malicious activities while they occur throughout the system, so, efficient intrusion detection systems (IDSes) [14] are deployed to monitor the system and identify misbehaviors. Such intrusion detection systems usually report all potentially malicious traffic, without regard to the actual network configuration, vulnerabilities, and mission impact. Given large volumes of network traffic, IDSes with even small error rates can overwhelm operators with false alarms. Even when genuine intrusions are detected, the actual mission threat is often unclear, and operators are unsure as to what actions they should take. In fact, to prioritize response and recovery actions in order to respond effectively to system compromises, security administrators need to get precisely and continuously updated estimate summaries regarding the security status of their mission-critical assets based on the already fired IDS alerts. However, in practical large-scale networks, manual monitoring and response are not always feasible and/or may not be fast enough to handle rapidly spreading attacks.

Preservation of the availability and integrity of networked computing systems in the face of fast-spreading intrusions requires advances not only in detection algorithms, but also in automated response techniques. In this chapter, we propose the mathematical framework of the *Response and Recovery Engine (RRE)* Our engine employs a game-theoretic response strategy against adversaries modeled as opponents in a two-player Stackelberg stochastic game [15]. RRE applies attack-response trees to analyze undesired security events and their countermeasures using Boolean logic to combine lower-level attack consequences. In addition, RRE accounts for uncertainties in intrusion detection alert notifications. RRE then chooses optimal response actions by solving a par-

tially observable competitive Markov decision process that is automatically derived from attack-response trees.

2.1 Problem Formulation

We formulate the optimal response selection as a decision-making problem in which the goal is to choose the cost-optimal response action at each time instant. The optimal action m is picked out of the set of all possible response actions $m \in \mathcal{M}$, including the No-OPeration (NOP) action. For example, an intrusion response system can respond to SQL's buffer overflow exploitation by closing its TCP connection. The optimization problem is solved in the response system, given the following inputs:

$\mathcal{W}$: a set of the computing assets $w \in \mathcal{W}$, e.g., a SQL server, that are to be protected by the response engine.

O: a set of IDS alerts $o \in O$ that specifically indicate an adversarial attempt to exploit the existing specific vulnerabilities of the assets, e.g., alerts from Snort [5] warning about a packet transferring the Slammer worm [16] that exploits a buffer overflow vulnerability in a SQL server.

$\mathcal{G}$: a set of ART graphs $g \in \mathcal{G}$ that systematically define how intrusive (responsive) scenarios about the attacker (response engine) affect system security (see Section 2.3.1).

The following sections are devoted to a solution to the response selection problem; in other words, we will focus on how the RRE finds the optimal response action based on given input arguments.

2.2 RRE's High-level Architecture

Before giving theoretical design and implementation details, we provide a high-level architecture of RRE, as illustrated in Figure 2.1. It has two types of decision-making engines at two different layers, i.e., local and global. This hierarchical structure of RRE's architecture, as discussed later, makes it capable of handling very frequent IDS alerts, and choosing optimal response actions. Moreover, the two-layer architecture improves its scalability for large-scale computer networks, in which RRE is supposed to protect a large number of host computers against malicious attackers. Finally, separation of high- and low-level security issues significantly simplifies the accurate design

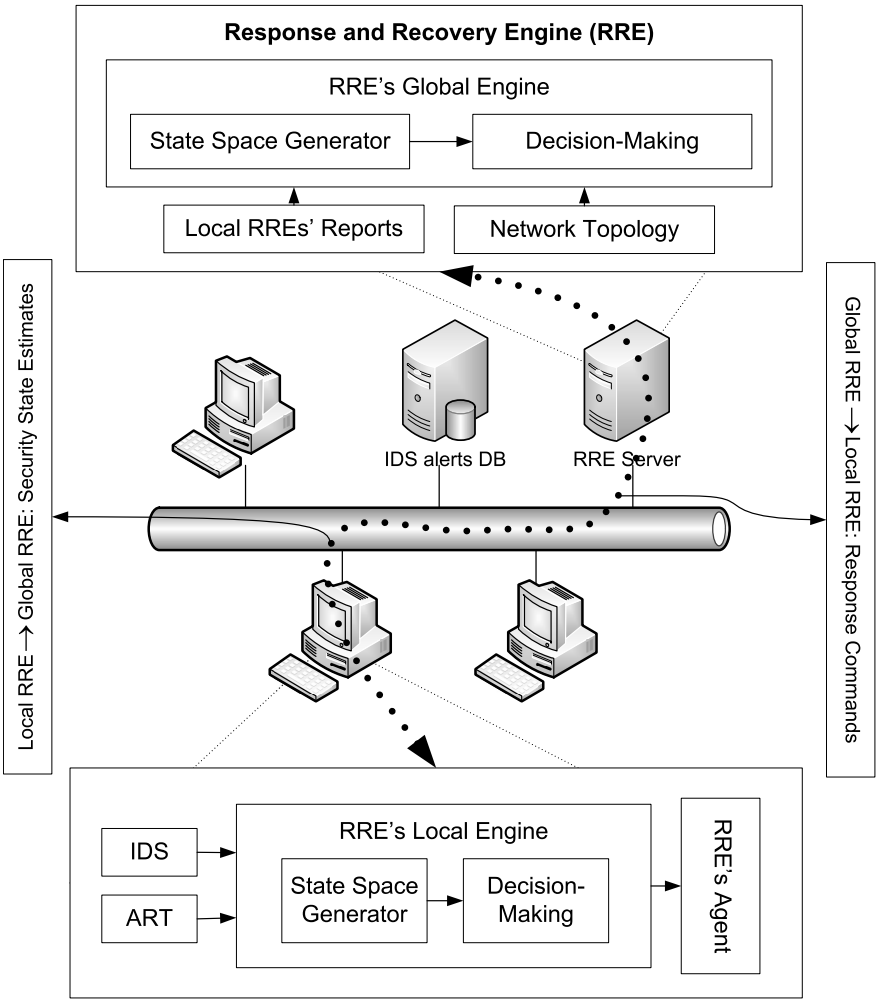


Figure 2.1: High-level Architecture of the RRE

of response engines.

At the first layer, RRE's local engines are distributed in host computers. Their main inputs consist of IDS alerts and attack-response trees (described in Section 2.3.1). All IDS alerts are sent to and stored in the alert database (Figure 2.1) to which each local engine subscribes to be notified when any of the alerts related to its host computer is received. Using the mentioned local information, local engines compute local response actions and send them to RRE agents that are in charge of enforcing received commands and reporting back the accomplishment status, i.e., whether the command was successfully carried out. The internal architecture of engines includes two major components: the state space generator, and the decision engine. Once inputs have been received, all possible cyber security states in which the host computer could be are generated. As discussed

later in Section 2.6, the state space might be intractably large; therefore, RRE partially generates the state space so that the decision-making unit can quickly decide on the optimal response action. The decision-making unit employs a game-theoretic algorithm that models attacker-RRE interaction as a two-player game in which each player tries to maximize his or her overall benefit. This implies that, once a system is under attack, immediate greedy response decisions are not necessarily the best choices, since they may not guarantee the minimum total accumulative cost involved in complete recovery from the attack.

Although individual local engines attempt to protect their corresponding host computers, they may become malicious themselves if they get compromised. Furthermore, it could become very complicated, even impossible, for local engines to choose and take a global network-level response action, due to their limited local knowledge. To deal with these problems, RRE's global engine, as its second layer, obtains high-level information from all host computers in the network, decides on optimal global response actions to take, and coordinates RRE agents to accomplish the actions by sending them relevant response commands. In addition to local security estimates from host computers, network topology is also fed into the global engine in the form of an ART graph, which shows 1) what combinations of compromised host computers will change the security status of the whole network, and 2) what global response actions are available to terminate attacks. The ART graphs, in the global engine, depend on the network's topology; therefore, they should be specifically designed by experts for each network. In contrast, the ART graphs for local assets, once designed, are simply re-used.

2.3 Local Response and Recovery

Having given a high-level overview of how hierarchically structured components in RRE interact with each other, we present the design of these components in detail. Starting with the lowest-level modules in RRE, we explain how local engines, residing in host computers, protect local computing assets using security-related information, i.e., IDS alerts, about them.

Table 2.1: Response and Recovery Action Classification [1]

Action class	Description	Examples
Rollback	bring the component back to a saved secure state	freeze/restore SW (CryoPID [17])
Rollforward	find a new state, from which the component can operate	journal/restore/update process [18]
Isolation	perform physical/logical exclusion of the faulty components	block attacker's IP
Reconfiguration	switch in spare component or reassign tasks to others	switch to 2nd Apache server
Reinitialization	check/record new configuration and update system tables	restart a TCP connection

2.3.1 Attack-Response Tree

To protect a local computing asset, its corresponding local engine first tries to figure out what security properties of the asset have been violated as result of an attack, given a received set of alerts. Attack trees [19] offer a convenient way to systematically categorize the different ways in which an asset can be attacked. Local engines make use of a new extended attack tree structure, called an attack-response tree (ART), that makes it possible 1) to incorporate possible countermeasure (response) actions against attacks, and 2) to consider intrusion detection uncertainties due to false positives and negatives in detecting successful intrusions, while estimating the current security state of the system. The attack-response trees are designed offline by experts for each computing asset, e.g., a SQL server, residing in a host computer. It is important to note that, unlike the attack tree that is designed according to all possible *attack scenarios*, the ART model is built based on the *attack consequences*, e.g., a SQL crash; thus the designer does not have to consider all possible attack scenarios that might cause those consequences.

The purpose of an attack-response tree $g_w \in \mathcal{G}$ for an asset $w \in \mathcal{W}$ is to define and analyze possible combinations of attack consequences that lead to violation of some security property of the asset. This security property, e.g., integrity, is assigned to the root node of the tree that is also called the *top-event* node. In the current implementation of RRE's local engines, there are at most three ART graphs $\mathcal{G}_w = \{g_w^c, g_w^i, g_w^a\}$ for each asset w, which are typically concerned with confidentiality, integrity, and availability of assets; $\mathcal{G}_w \subset \mathcal{G}$ can be expanded to include other security properties. An attack-response tree's structure is expressed in the node hierarchy, allowing one to decompose an abstract attack goal (consequence) into a number of more concrete consequences called sub-consequences. A node decomposition scheme could be based on either 1) an `AND` gate, where all of the sub-consequences must happen for the abstract consequence to take place, or 2)

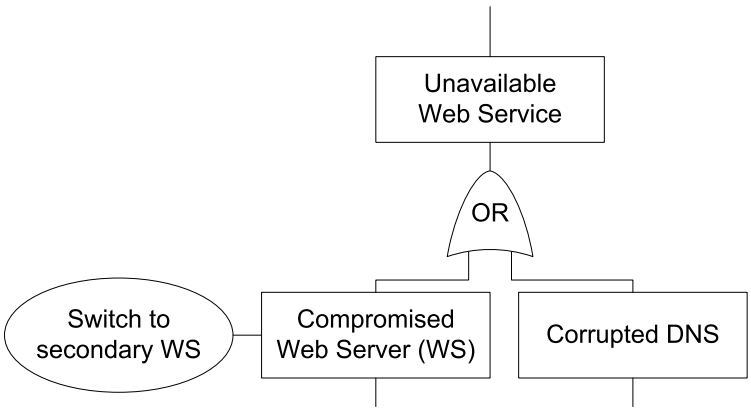


Figure 2.2: Node Decomposition in ART

an OR gate, where occurrence of any one of the sub-consequences will result in the abstract consequence. For a gate, the underlying sub-consequence(s) and the resulting abstract consequence are called *input(s)* and *output*, respectively. Being at the lowest level of abstraction in the attack-response tree structure, every leaf node consequence $l \in \mathcal{L}$ is mapped to (reported by) its related subset of IDS alerts $O_l \subseteq O$, each of which represents a specific vulnerability exploitation attempt by the attacker.

Some of the consequence nodes in an ART graph are tagged by response boxes that represent countermeasure (response) actions $m \in \mathcal{M}$ against the consequences to which they are connected. Table 2.1 illustrates different classes of response actions that might be applicable to a consequence, depending on the type of consequence and the involved assets. Figure 2.2 illustrates how a sample abstract consequence node (output), i.e., an unavailable web service, is decomposed into two sub-consequences (inputs) using an OR gate; this means that the web service becomes unavailable if either the web server is compromised or the domain name server is corrupted. Furthermore, if a web service is unavailable due to the compromised web server, the response engine can switch to the secondary web server. Figure 2.3 shows how a typical ART would finally look.

For every ART graph, a major goal is to probabilistically verify whether the security property specified by ART's root node has been violated, given the sequence of 1) the received alerts, and 2) the successfully taken response actions. Boolean values are assigned to all nodes in the attack-response tree. Each leaf node consequence $l \in \mathcal{L}$ is initially 0, and is set to 1 once any alert o_l from its corresponding alert set $O_l \subseteq O$ (defined earlier) is received from the IDS. These values for other

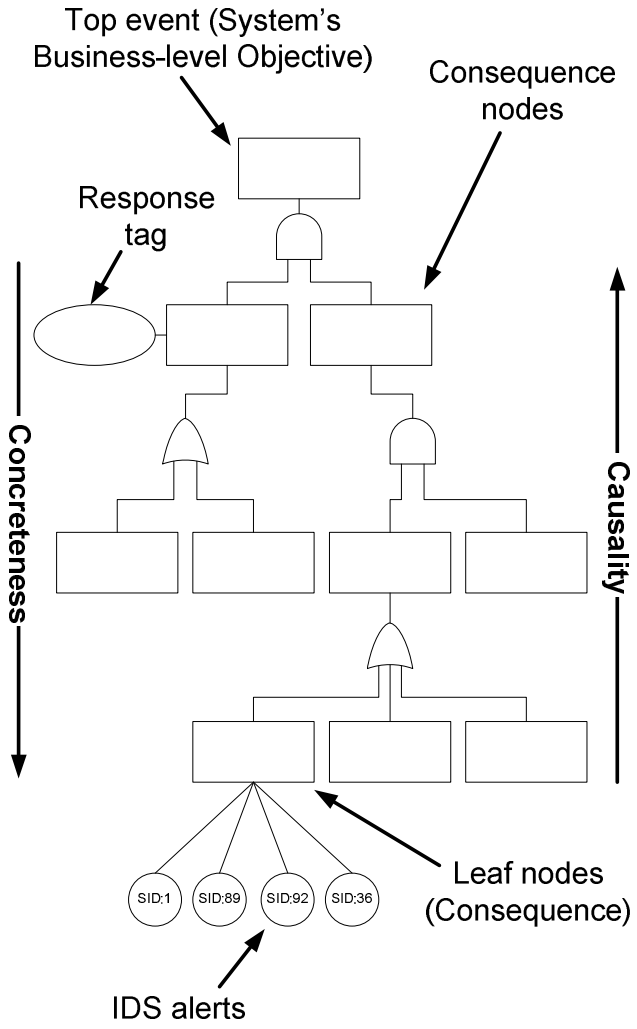


Figure 2.3: Attack Response Tree

consequence nodes, including the root node, are simply determined bottom-up according to leaf nodes' values in the subtree whose root is the consequence node under consideration. Response boxes are triggered once they are successfully taken by the response engine; as a result, all nodes in their subtree are reset to zero, and the corresponding received alerts are cleared. As a case in point, if the response box, that is connected to ART's root node is triggered, all nodes in the ART graph are reset to zero.

Dealing with uncertainties. In reality, determining Boolean values of the leaf node consequences in ART is more complicated, due to the uncertainty about whether 1) the received alerts actually represent some consequence occurrence, and 2) no consequence has happened if no alert has been received. Taking such uncertainties into account, RRE makes use of a *naive Bayes binary classifier*, similar to eBayes [20], that uses Bernouli variables, i.e., alerts, to determine the value of each leaf consequence node l, given the set of its related received alerts $O_l^r \subseteq O_l$:

$$\delta(l \mid O_l^r) = \frac{P(l) \cdot \Pi_{o_l \in O_l^r} P(o_l \mid l)}{P(l) \cdot \Pi_{o_l \in O_l^r} P(o_l \mid l) + P(\bar{l}) \cdot \Pi_{o_l \in O_l^r} P(o_l \mid \bar{l})} \tag{2.1}$$

where $P(l)$, the so-called class prior, is the probability of consequence l's occurrence, and $P(\bar{l})$ is simply its complement, i.e., $P(\bar{l}) = 1 - P(l)$. Furthermore, $P(o_l \mid l)$ denotes the probability that alert o_l was already received, given that consequence l has actually happened. These probability measures are calculated based on historical information about the system. One possible technique to obtain those measures is periodic *alert verification* [21], which is an automatic or manual, possibly time-consuming, process to periodically check whether the attack consequence l has occurred using the visible and checkable traces that a certain attack leaves at a host or on the network, e.g., a temporary file or an outgoing network connection. Consequently, $P(l)$ is calculated as a proportion of the past periodic checks that verified the occurrence of consequence l, and using Bayes' theorem:

$$P(o_l \mid l) = \frac{P(o_l, l)}{P(l)} \tag{2.2}$$

where $P(o_l, l)$ denotes the fraction of checks which verified that consequence l had occurred, and alert o_l had been received.

Given the satisfaction probabilities of leaf nodes, the output probability of gate q with inputs I

is simply calculated as follows:

$$\delta(q) = \begin{cases} \Pi_{i \in I} \delta(i) & \text{if q is an AND gate,} \\ 1 - \Pi_{i \in I}(1 - \delta(i)) & \text{otherwise} \end{cases} \tag{2.3}$$

where there is an implicit assumption that gate inputs are independent; otherwise, δ_q is computed using joint probability distributions of inputs.

Starting from the root node and recursively using (2.3), it is simple to obtain δ_g, i.e., that is the probability that the security property of the root node in ART graph g has been compromised. This value, as a local security estimate, is reported by the local engine to the RRE server, where optimal *global* response actions are decided upon according to received local estimates (see Section 2.4). Next, we will explain how ART graphs and their nodes' satisfaction probabilities are used in a game-theoretic algorithm to decide on the optimal response action.

2.3.2 Stackelberg Game: RRE vs. Attacker

Reciprocal interaction between the adversary and response engine in a computer system is a game in which each player tries to maximize his or her own benefit. The response selection process in RRE is modeled as a sequential Stackelberg stochastic game [15] in which RRE acts as the *leader* while the attacker is the *follower*; however, in our infinite-horizon game model, their roles may change without affecting the final solution to the problem.

Specifically, the game is a finite set of *security states S* that cover all possible security conditions that the system could be in. The system is in one of the security states s at each time instant. RRE, the leader, chooses and takes a response action $m_s \in \mathcal{M}$ admissible in s, which leads to a probabilistic security state transition to s'. The attacker, which is the follower, observes the action selected by the leader, and then chooses and takes an adversary action $o_{s'} \in O$ admissible in s', resulting in a probabilistic state transition to s''. At each transition stage, players may receive some reward according to a reward function for each player. The reward function for an attacker is usually not known to RRE, because an attacker's reward depends on his final malicious goal, which is also not known; therefore, assuming that the attacker takes the worst possible adversary action, RRE chooses its response actions based on the security strategy, i.e., *maximin*, as discussed later.

It is also important to note here that although S is a finite set, it is possible for the game to revert back to some previous state; therefore, the RRE-adversary game can theoretically continue forever. This stochastic game is essentially an antagonistic multicontroller Markov decision process, called a *competitive Markov decision process (CMDP)* [22].

A discrete competitive Markovian decision process Γ is defined as a tuple (S, A, r, P, γ) where S is the security state space, assumed to be an arbitrary non-empty set endowed with the discrete topology. A is set of actions, which itself is partitioned into response actions and adversary actions depending on the player. For every $s \in S$, $A(s) \subset A$ is the set of admissible actions at state s. The measurable function $r : K \to \Re$ is the reward where $K := \{(s,a,s') | a \in A(s); s, s' \in S\}$, and P is the transition probability function; that is, if the present state of the system is $s \in S$ and an action $a \in A(s)$ is taken, resulting in state transition to state s' with probability $P(s'|s,a)$, an immediate reward $r(s,a,s')$ is obtained by the player taking the action. γ is the discount factor, i.e., $0 < \gamma < 1$.

2.3.3 Automatic Conversion: ART-to-MDP

Using the ART graphs, RRE's local engines automatically construct response decision process models, where security states are defined as a binary vector whose variables are actually the set of satisfied/unsatisfied (1/0) leaf consequence nodes in the ART graph under consideration. In other words, as a binary string, each security state vector represents the leaf node consequences that have already been set to 1 according to the received alerts from IDS systems. For instance, the security state space for an ART graph with n leaf nodes consists of 2^n n-bit state vectors. For ART graphs with a large number of leaf nodes, this exponential growth of the security state space usually results in the state space explosion problem, which RRE deals with by making use of approximation techniques, as discussed in Section 2.6.

Once local engines have generated the security state space, the next step in the decision process model generation is to construct state transitions for each state s, i.e., $A(s)$. As mentioned above, in a current security state s, there can be either of two types of actions, responsive $A_r(s)$ and adversarial $A_a(s)$, depending on the player making the decision. First, in state s, a response action $m \in \mathcal{M}$ yields a transition to state s', in which bits are all similar to those of s except for those bits that reflect leaf nodes of the ART subtree, whose root is m (explained in Section 2.3.1), which are 0 in s'. For example, assume that the system is in state $s \in S$, and RRE decides to enforce

response action m_r, which is connected to the root node in the ART graph. The system's next state will be s' in which all bits are 0, i.e., the most secure state. Although it is tempting to always take m_r whenever any leaf nodes take 1, a cost-benefit evaluation, as discussed later, may result in the choice of another, less expensive response action, or in taking no action at all.

The second type of state-action-state transitions in CMDPs are those due to adversarial actions. During automatic ART-to-CMDP conversion in RRE, each leaf consequence node l in the ART graph is mapped to an adversarial action that causes that l to be set to 1. In other words, suppose that the system is in a security state s of CMDPs. For every leaf node l whose bit in s is 0, a transition is built to state s', where all bits are the same as in s, but the bit related to l is 1.

The probabilities of state transition arcs $k \in K$ in CMDP are assigned based on previous actions' success rates computed using reports from local agents (see Section 2.3.5); moreover, reward functions $r : K \rightarrow \Re$ must also be calculated. Indeed, $r(s,a,s')$ is payoff gained by the player, who is successfully taking action a in state s and causing a transition to state s':

$$r(s,a,s') = (\delta_g(s) - \delta_g(s'))^{\tau_1} C(a)^{\tau_2}, \tag{2.4}$$

where $0 \leq \tau_1 \leq 1$ and $\tau_2 \leq 0$ are two fixed parameters. $\delta_g(s)$ denotes the root node compromisation probability of the ART graph g whose leaf nodes' Boolean variables are set according to bits in s. This probability is simply computed using equations (2.1) and (2.3). Obviously, $\delta_g(s') \leq \delta_g(s)$, since a is a response action. Furthermore, $C(a)$ is the positive cost function for action a regardless of the source and destination states. This cost function should be defined specifically depending on the application for which it is used; for instance, one reasonable option would be the *mean-time-to-accomplish* measure.

Having automatically generated CMDP using the ART graph, RRE can now solve the decision process to find the optimal response action. As discussed earlier, due to a lack of knowledge about the attacker's cost function, given a system's current state, RRE uses the *maximin* approach to find the security strategy for the game. To do so, it must know the exact current state of the system. At every time instant, a reasonable choice (according to leaf nodes) is state s, where bits are 1 if their corresponding leaf nodes are set, and zero otherwise. Thus, in local engines, where leaf nodes of ART graphs are mapped to subsets of IDS alerts, the system's current state s consists of 1s for

bits which represent satisfied leaf nodes according to received alerts, and 0s for other bits in s. As mentioned earlier, the fact that leaf nodes have been set does not necessarily mean that they are truly positive, as in (2.1); hence, uncertainty in received information prevents RRE from precisely figuring out the current security state of the system. However, the probability of being in each state is calculated as follows:

$$b(s) = \prod_{l \in \mathcal{L}} (1_{[s_l=1]} \cdot \delta(l) + 1_{[s_l=0]} \cdot (1 - \delta(l))) \tag{2.5}$$

where $\mathcal{L}$ is the set of leaf nodes in the ART graph; $\delta(l)$ is computed as in (2.1); s_l is the bit in state s that corresponds to leaf node l; and $1_{[\text{expr}]}$ is the indicator function, and is 1 if expr is true, and 0 otherwise. It is worth noting that (2.5) is based on the implicit assumption of independence among leaf nodes.

Therefore, to consider uncertainty, instead of determining the exact current state of the system, we obtain a probability distribution $b(.)$ on state space $s \in S$ (called the *belief state*) using (2.5). The axioms of probability require that $0 \leq b(s) \leq 1$ for all $s \in S$ and that $\sum_{s \in S} b(s) = 1$. Uncertainty in updating inputs, i.e. IDS alerts, converts our Markovian decision process into a higher-level model, called a *partially observable competitive Markov decision process (POCMDP)*, which is similar to the model described in [23] with the subtle difference that [23] studies simultaneous games, whereas the game here is sequential. Indeed, states $b \in \mathcal{B}$, in this higher-level model, are probability distributions over a set of states S in the underlying Markovian decision process model.

2.3.4 Optimal Response Strategy

As the last step in the decision-making process in local engines, RRE solves the POCMDP to find an optimal response action from its action space, and sends an action command to its agents that are in charge of enforcing received commands. Action optimization in RRE is accomplished by trying to maximize the accumulative long-run reward measure received while taking sequential response actions. To accumulate sequential achieved rewards, here, we use the *infinite-horizon discounted cost* technique [24], which gives more weight to nearer future rewards. In other words, in each step, the game value is computed by recursively adding up the immediate reward after both players take their next actions and the discounted expected game value from then on.

$$\Psi(V,b,a) = \sum_{o\in O} P(o|b,a)\cdot\{\rho(b,a,b'_{b,a,o}) + \\ + \sqrt{\gamma}\cdot[\min_{a_a\in A_a(b'_{b,a,o})} \sum_{o'\in O} P(o'|b'_{b,a,o},a_a)\cdot\big(\rho(b'_{b,a,o},a_a,b''_{b',a_a,o'}) + \sqrt{\gamma}\cdot V(b''_{b',a_a,o'})\big)]\}$$

To formulize the explanation just given, the solution of a POCMDP consists in computing an optimal policy, which is a function π^* that associates with any belief state $b \in \mathcal{B}$ an optimal action $\pi^*(b)$, which is an action that maximizes the expected accumulative reward on the remaining temporal horizon of the game. As discussed above, this accumulative reward is defined as the discounted sum of the local rewards r that are associated with the actual action transitions. The Markovian decision process theory assigns to every policy π a value function V_π, which associates every belief state $b \in \mathcal{B}$ with an expected global reward $V_\pi(b)$ obtained by applying π in b. For finite-horizon POMDPs, the optimal value function is piecewise-linear and convex [25], and it can be represented as a finite set of vectors. In the infinite-horizon formulation, a finite vector set can closely approximate the optimal value function V^*, whose shape remains convex. Bellman's optimality equations (2.6) characterize in a compact way the unique optimal value function V^*, from which an optimal policy π^*, which is discussed later, can be easily derived:

$$V^*(b) = \max_{a_r\in A_r(b)} \Psi(V^*,b,a_r) \tag{2.6}$$

where $A(b) = \cup_{s\in S:b(s)\neq 0} A(s)$. $A(.)$ is partitioned into $A_r(.)$ and $A_a(.)$ for response and adversary actions, respectively. Ψ is defined in (2.7), in which ρ is the POCMDP reward function. ρ is computed using reward function r in the inherent CMDP:

$$\rho(b,a,b') = \sum_{s,s'\in S} b(s)b'(s')r(s,a,s'). \tag{2.7}$$

Here, $b'_{b,a,o}$ is the updated next belief state if the current state is b, action a is taken, and obser-

vation o is received from sensors:

$$\begin{aligned} b'_{b,a,o}(s') &= P(s'|b,a,o) \\ &= \frac{P(o|s')\sum_{s\in S} P(s'|s,a)b(s)}{\sum_{s''\in S} P(o|s'')\sum_{s\in S} P(s''|s,a)b(s)}, \end{aligned} \tag{2.8}$$

where, due to the independence assumption among the ART graph's leaf nodes, we have

$$P(o|s) = \prod_{l\in\mathcal{L}} (1_{[s_l=1]} \cdot P(o|l) + 1_{[s_l=0]} \cdot P(\bar{o}|l)), \tag{2.9}$$

where $P(\bar{o}|l) = 1 - P(o|l)$, and $P(o|l)$ is simply obtained using (2.2).

Once the partially observable decision process is formulized, the optimal response action is chosen based on the optimal value function. There are different techniques to obtain the optimal value function. The decision-making unit in RRE uses a value iteration technique [26]:

$$V_t(b) = \max_{a_r\in A_r(b)} \Psi(V_{t-1}, b, a_r), \tag{2.10}$$

that applies dynamic programming updates to gradually improve on the value until it converges to the ε-optimal value function, i.e. $\mid V_t(b) - V_{t-1}(b) \mid < \varepsilon$. Through improvement of the value, the policy is implicitly improved as well. Finally, optimal policy π^* maps the system's current belief state b to a response action:

$$\pi^*(b) = \arg\max_{a_r\in A_r(b)} \Psi(V^*, b, a_r), \tag{2.11}$$

which, in local engines, is sent to RRE agents that are in charge of carrying out the received response action commands. Agents then send status messages to the decision-making unit, indicating whether the received action command has been accomplished successfully. If it has, the decision-making unit updates the leaf nodes and variables in the corresponding ART graph.

So far, we have discussed how RRE's local engine estimates local security state and decides upon and takes local response actions following alerts received from the IDS. Next, we will address how RRE's server makes use of local information received from local engines to estimate the security status of the whole network, and then decide what global response actions to take. The information that is sent by local engines to RRE's server consists of root probabilities δ_g, as

computed in (2.3), of local ART graphs. In the current implementation of RRE, these include three root node probabilities of three ART trees reflecting confidentiality, integrity, and availability of local host systems.

2.3.5 Agents

In the abovementioned security battle between RRE and the adversary, agents play a key role in accomplishing each step of the game. They are in charge of taking response actions decided on by RRE engines. Actually, having received commands from engines, agents try to carry them out successfully and report the result, whether they were successful or not, back to the commander, i.e., the engine. If the agent's report indicates that some response action has been taken successfully, the engines update their ART trees' corresponding variables, which are leaf node values in the subtree for the successfully taken response action node. Consequently, as explained above, leaf node variables in ART trees are updated by two types of messages: IDS alerts and agents' reports.

2.4 Global Response and Recovery

Although host-based intrusion response is taken into account by RRE's local engines using local ART graphs and the IDS rule-set for computing assets, e.g., the SQL server, maintenance of global network-level security requires information about underlying network topology and profound understanding about what different combinations of secure assets are necessary to guarantee network security maintenance. In RRE, global network intrusion response is resolved in the central server, where, just as in local engines, ART graphs are used for correlating received information, and then maximin theory is applied to choose the optimal global response action. Such a choice is not possible in local engines due to either their limited local information or their inability to manage cooperation among distributed RRE agents.

In contrast to ART graphs in local engines for computing assets that demand one-time design effort for each asset (as in IDS rule-sets), global ART graphs in RRE's server for network security should be designed specifically for each individual network in which RRE is deployed, since these higher-level ART graph structures depend on network topology, which implicitly affects a network's global security state. In our current implementation, there is only one global, i.e.,

network-level, ART graph in RRE that must be designed by an expert.

Generally, global ART graphs in RRE's server have the same structure discussed in Section 2.3.1, though some clarifying explanations are needed regarding their root and leaf nodes. As discussed in Section 2.3.4, local engines send their local security estimates, i.e., root node probabilities δ_g of their ART graphs, to RRE's server. Indeed, local security estimates contribute to leaf nodes in the global ART graph in RRE's server. Furthermore, the top-event node of the ART graph in the global engine is labeled *"network security violation,"* and is defined and formulated according to the underlying nodes. In other words, network security is defined by the global ART graph in RRE's server using its leaf nodes, which are themselves root nodes of local ART graphs in RRE's local engines.

Global ART is employed for quantitative evaluation of a network's security state. The correlation and response selection calculations are the same as in local ART graphs (Section 2.3), except that for global ART, the basic leaf node l probability measures are computed as $\delta(l) = \delta_{g(l)}$, where $g(l)$ denotes the local ART graph corresponding to leaf node l of the global ART in RRE's server.

2.5 Case Study: A SCADA Network

In this section, we describe the response selection process for a case study process control network for a power grid. We have chosen a supervisory control and data acquisition (SCADA) network as our case study for two reasons. First, since they control physical assets, they need timely response in the presence of attacks. Second, in contrast to general IT computer networks, they consist of computing assets with predefined specific responsibilities and communication patterns; this simplifies the design process for comprehensive ART graphs and IDS with thorough alert set.

Figure 2.4 shows a sample 3-bus power grid and its SCADA network, which is responsible for monitoring and controlling the underlying power system. There are a total of three generators, any one of which is able to provide the power required by customers, i.e., load. To monitor the power system, each bus is attached to a sensor, i.e., a phasor measurement unit (PMU), which sends to SCADA voltage phasors, i.e., magnitudes and phase angles, of the bus and current phasors of transmission lines connected to that particular bus. Moreover, to control power generation, having received sensory data, SCADA computes optimal generation set points for individual gen-

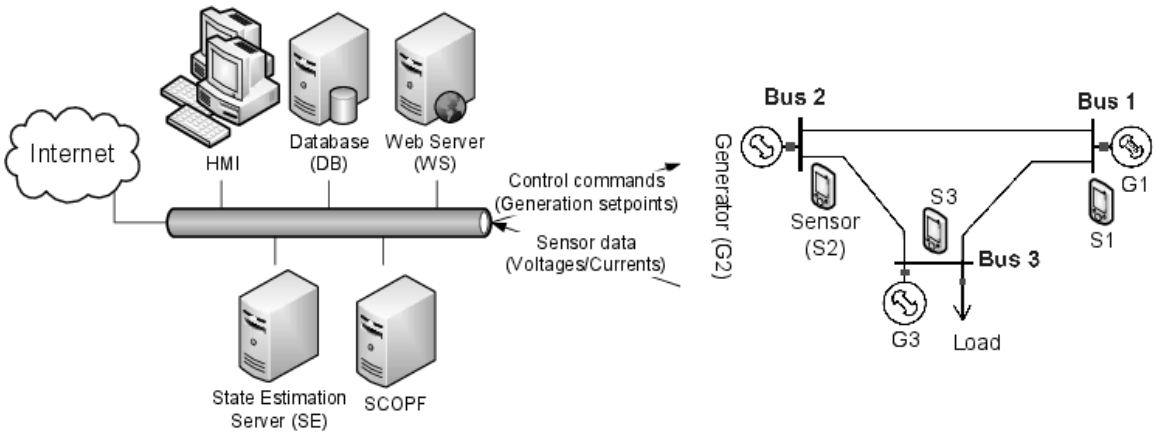


Figure 2.4: Case Study: SCADA Networks

erators. As shown in Figure 2.4, SCADA consists of different components, among which there are constrained communications. First of all, given noisy sensory data, the state estimation server is responsible for estimating the state of the whole power system. A database stores these states and other information that might be used later by administrators or customers through the web server. The human machine interface (HMI) and security constrained optimal power flow (SCOPF) compute control commands using those estimated states. As demonstrated, a hot spare HMI is also active and connected as part of the network.

Figure 2.5 illustrates a sample brief network-level attack response tree for the process control network described above. The top event is chosen to be "SCADA is compromised," and its children denote deficiencies in providing loads and report generation, which are two main goals of the supervisory network. For simplicity, leaf nodes here denote compromise of individual host systems that are received from local engines. As a case in point, G1, if set to 1, indicates that the controller device for the generator on bus 3 is compromised, and the upper response node "restart" shows that a countermeasure for the compromised controller is to reinstall the control software and restart the device. Details such as action costs, rates, and probabilities are not shown in this figure.

The cyber-security state space definition for the above attack-response tree is shown in Figure 2.6 as a binary vector, where each individual bit is set based on reports from local engines. For instance, the sample state vector in the figure indicates that HMI and G1 are compromised, according to reports from their corresponding local engines, while other hosts are in their normal

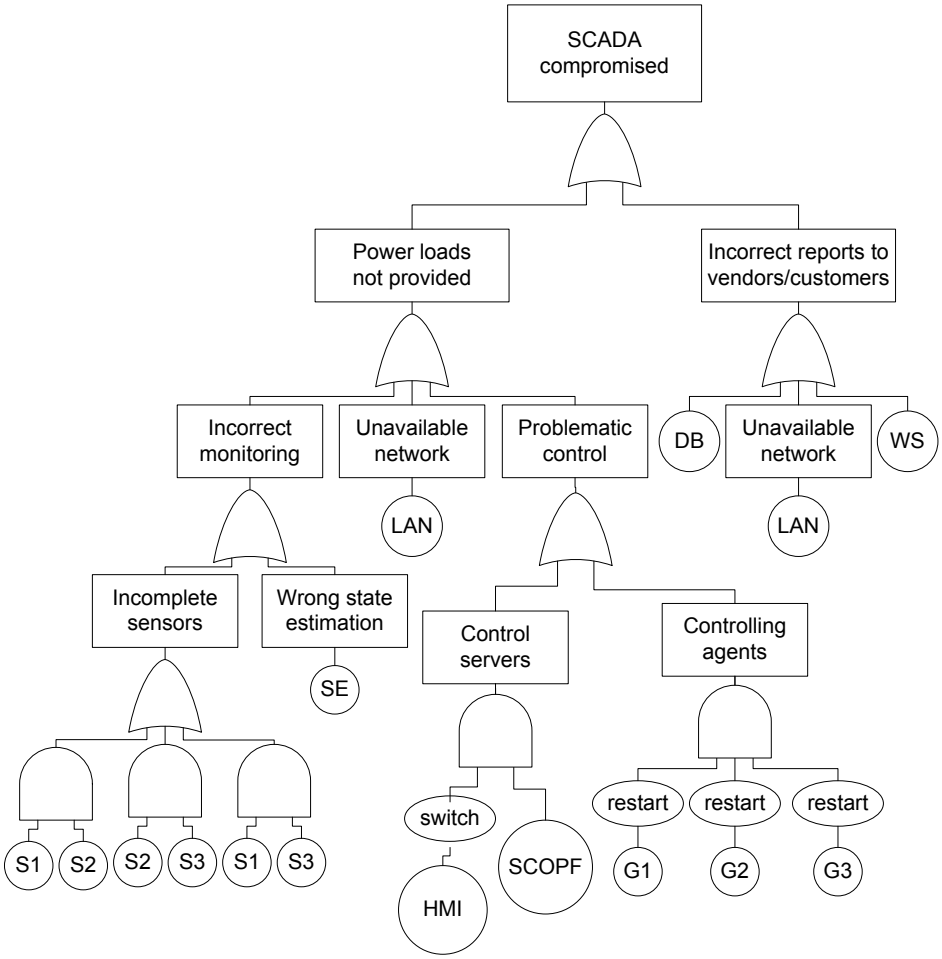


Figure 2.5: Attack-Response Tree

operational mode.

Given the attack-response tree and reports from local engines, RRE starts online construction of its accompanying partially observable competitive Markov decision process. Starting from the current state, i.e., $s = (000100000010)$ (Figure 2.6), there are a total of 13 possible transitions, partitioned into two subsets: 1) response actions $A_r(s) = \{restart(G_1), switch(HMI), NOP\}$, and 2) adversarial actions $A_a(s)$ regarding leaf consequence nodes. Here, we assume that responsive actions $A_r(s)$ require some manual assistance by a SCADA operator; hence, they cannot be accomplished simultaneously due to limited human resources. The solution for this model by RRE is *switch(HMI)* as the optimal response action, since if $restart(G_1)$ or NOP was chosen, the attacker could cause a huge amount of damage to the system afterwards by compromising the SCOPF server (Figure 2.5), leading to complete failure of the control subsystem that would consequently

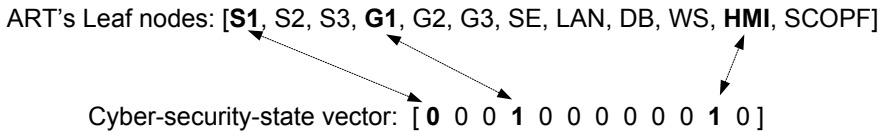


Figure 2.6: A Sample Cyber-Security State

affect how power loads were provided and finally result in the top event "SCADA compromised." In other words, as explained earlier, the engine chooses the response action that minimizes the maximum damage that the attacker can cause later.

2.6 Computational Efficiency

Although the value iteration algorithm performs well in MDPs with several thousand states, RRE (like most state-based modeling techniques) faces the state space explosion problem when a large network that includes a large number of assets is to be protected using numerous alerts sent by distributed IDS systems. This exponential growth of the state space makes it infeasible to compute an optimal solution, i.e., response action, in large-scale applications. This becomes even worse when POCMDP is employed to find an optimal solution. Therefore, RRE uses two state-compaction techniques to deal with this problem.

First, the most likely state (MLS) approximation technique [27] is used to convert POCMDP to MDP, which is more tractable for real-time response decision-making. To do so, we compute the most likely state using:

$$s^* = \arg\max_s b(s), \tag{2.12}$$

and define policy as

$$\pi(s) = \pi^{MDP}(s^*), \tag{2.13}$$

which is computed using Bellman's optimality equations, as follows:

$$V^*(s) = \max_{a_r \in A_r(s)} \Upsilon(V^*, s, a_r) \tag{2.14}$$

$$\pi^{MDP}(s) = \arg\max_{a_r \in A_r(s)} \Upsilon(V^*, s, a_r) \tag{2.15}$$

$$\Upsilon(V,s,a) = \sum_{s' \in S} P(s'|s,a) \cdot \{r(s,a,s') + \sqrt{\gamma} \cdot [\min_{a_a \in A_a(s')} \sum_{s'' \in S} P(s''|s',a_a) \cdot \big(r(s',a_a,s'') + \sqrt{\gamma} \cdot V(s'')\big)]\} \tag{2.16}$$

$\Upsilon(.)$ is defined in (2.16). The value iteration algorithm [26] is employed to solve this MDP:

$$V(s) \leftarrow \max_{a \in A(s)} \Upsilon(V,s,a) \qquad \forall s \in S \tag{2.17}$$

Using MLS in RRE is quite reasonable, since the probability of the most likely state is far greater than the probability of other states; however, although it uses MLS, MDP in (2.13) is not small enough to deal with in real-time. Furthermore, due to its large state space, even off-line solution techniques are not usable, since most of them, e.g., value iteration, perform iterative updates over the entire state space, which is intractable in (2.13). To focus computations on only relevant states, an online *anytime algorithm*[1] called *envelope* [28] is employed, making RRE capable of deciding real-time responses even in large-scale computer networks. In brief, the *envelope* algorithm performs a finite look-ahead search on a subset of states reachable from a given current state, i.e., s^* in (2.12). This subset, called "envelope $\mathcal{E}_\pi$", initially contains only the current state and is progressively expanded. An approximate value function $\tilde{V}$ is used to evaluate the fringe states, i.e. the set of states that are not in the envelope but may be reached in one step of policy execution from some state in the envelope:

$$\mathcal{F}_\pi = \{s \in S - \mathcal{E}_\pi | \exists s' \in \mathcal{E}_\pi, P(s', \pi(s'), s) > 0\}. \tag{2.18}$$

The general scheme of the *envelope* is as follows:

1. *Initialization:* Generate an initial envelope $\mathcal{E}$.
2. While ($\mathcal{E} \neq S$) and (not deadline) do
   (a) *Fringe expansion:* Extend the envelope $\mathcal{E}$. Some fringe state s is chosen, and its value is updated (2.17).

[1] An algorithm is called *anytime* if it can be interrupted at any point during execution to return an answer whose value, at least in certain classes of stochastic processes, improves in expectation as a function of the computation time.

(b) *Ancestors update phase:* Generate an optimal policy π for the envelope. The values of the ancestors of the newly updated states are updated using (2.14).

3. Return π.

Having used the *envelope* algorithm, RRE can solve larger MDPs efficiently by producing partial policy, defined only on the envelope, without evaluating the entire state space.

2.7 Experimental Evaluation

In this section, we investigate how the proposed response and recovery engine performs in reality. We have implemented RRE on top of Snort 2.7 [5], which is an open-source signature-based IDS. The experiments were run on a computer system with a 2.2 GHz AMD Athlon 64 Processor 3700+ with 1 MB of cache, 2 GB of memory, and the Ubuntu (Linux 2.6.24-21) operating system.

2.7.1 Scalability

To evaluate how RRE handles complex networks consisting of a large number of host systems, we measured the time required by RRE to compute the optimal response action vs. various metrics. Figure 2.7[2] shows the average time-to-response over ten runs vs. the attack-response tree order, i.e., the maximum number of children for each node. For each tree order d, a balanced tree, in which each node has d children, is generated; gates are assigned to be `AND` or `OR` with equal probability, i.e., 0.5. The ε-optimality termination criterion in Bellman's equation and discounting factor were set to $\varepsilon = 0.1$ and $\gamma = 0.99$, respectively. Then, a decision process is constructed and solved, and the total time spent is recorded (see Figure 2.7). As expected, the figure shows that increasing the ART order leads to rapid growth of the required time-to-response by the response engine.

In another scalability evaluation experiment, we measured time-to-response vs. the number of nodes in balanced ART trees of order 2. Figure 2.8 shows average results on ten runs for three scenarios. First, given IDS alerts and the ART tree, the complete decision model consisting of all states in the state space was constructed. As shown in Figure 2.8(a), the response engine can

[2]The graphs, in this section, are not completely smooth as the results are averaged over only three runs.

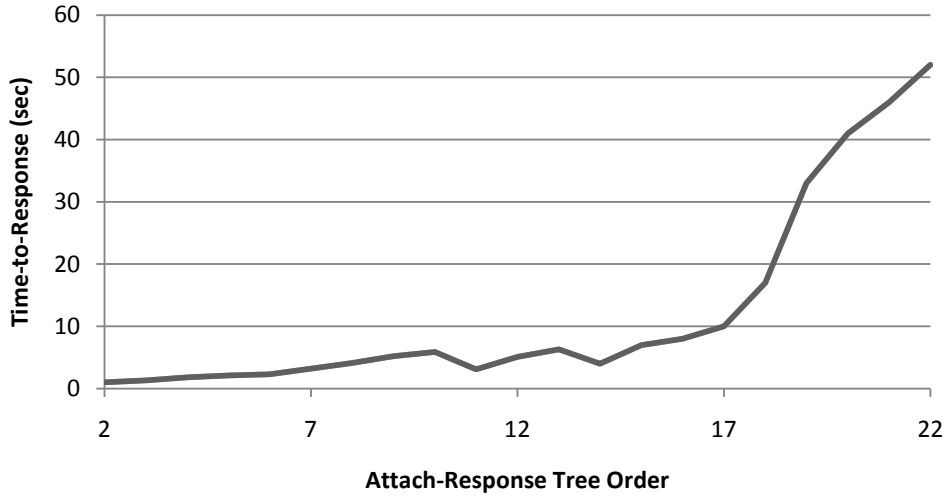
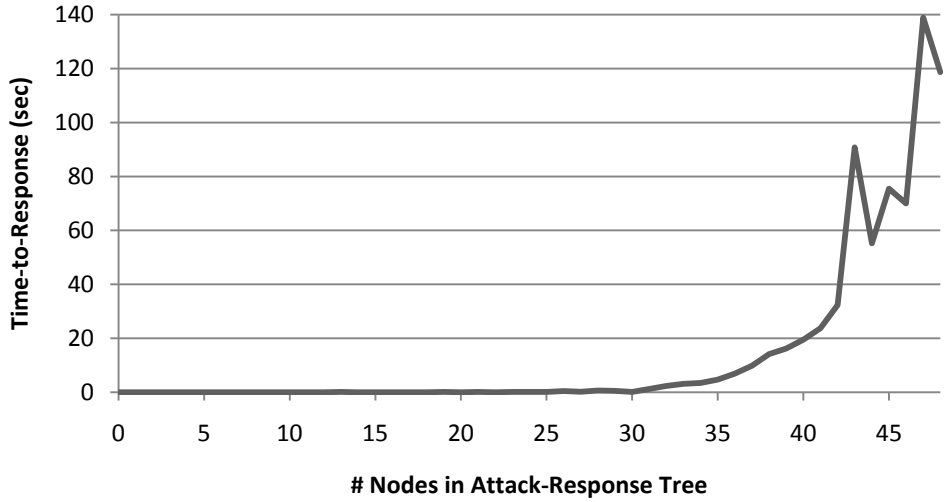


Figure 2.7: Tree Order vs. Time-to-Response

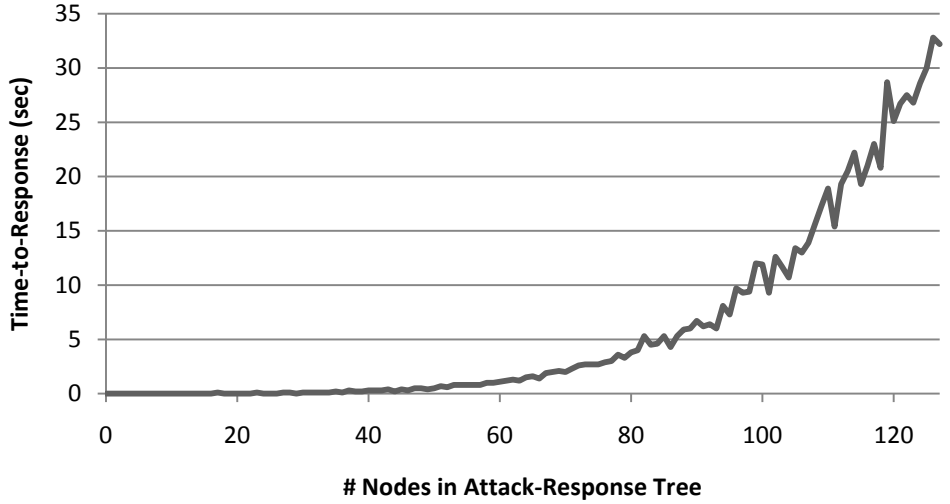
solve for optimal response actions for ART trees with up to 45 nodes within about 2 minutes. Second, an online finite-lookahead Markovian decision model with an expansion limit of 4 steps was generated and solved. As illustrated in Figure 2.8(b), limited expansion improves a solution's convergence speed and increases the solvable ART size to trees with up to 120 nodes within 30 seconds. Third, to further improve RRE's scalability, we evaluated how fast a decision process is solved with an upper expansion limit of 2. Figure 2.8(c) shows that ART trees with more than 900 nodes are still solvable in less than 40 seconds. By solving ART trees with about 900 nodes in a minute, RRE can protect large-scale computer networks.

2.7.2 Comparison

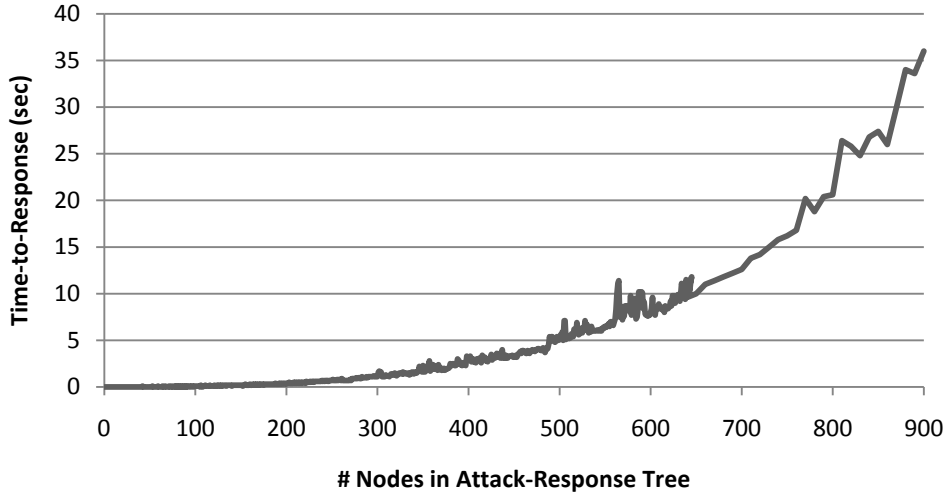
This section evaluates how beneficial RRE is compared to static intrusion response systems, particularly those that statically choose and take response actions from a lookup table that stores alert-response mappings. In our experiments, both IRS systems, i.e., RRE and static engines, were given a sample ART tree with $\mid \mathcal{L} \mid = 6$ leaf nodes based on which they computed response actions. RRE's response action selection has already been explained in detail; the static IRS maps each alert, i.e., a leaf node in ART, to a response action that resets that particular leaf node with minimum cost. Given the current network state, we compared how much cost RRE and the static IRS spent toward the end of the game, i.e., the point at which all leaf nodes had been cleaned. ART parameters and result graphs are omitted due to space constraint; however, final results are



(a) Complete State Space Generated



(b) Finite Lookahead: 4 Steps



(c) Finite Lookahead: 2 Steps

Figure 2.8: Scalability Improvement Using Finite Lookahead Solution

described here.

We modeled the attacker to be completely intelligent; in other words, in each step, he or she took the most harmful possible adversarial action. There were a total of $2^{|\mathcal{L}|} = 64$ starting scenarios (states) for two different game schemes. In the first scheme, the action ratio between IRS and the attacker was 1; in other words, for each action taken by the response system, the attacker was allowed to pick one adversarial action. As expected, for all initial scenarios, in picking the optimal action, RRE required a recovery cost less than or equal to what the static IRS did. In the second game scheme, we fortified the attacker's strength, and set the action ratio to 1/2 meaning that for each action by the IRS, the attacker was allowed to take 2 actions. In 5 scenarios (out of 64), RRE caused more recovery cost than its static competitor, the reason being that the RRE chooses the optimal response action under the assumption that the action ratio is 1.

2.8 Related Work

FLIPS [29], a host-based application-level firewall, uses *selective transactional* technique to emulate selected application pieces prior to their real execution, and takes static response actions if any anomalous behavior is detected. Musman et al. [30] presented an agent-based response architecture called SoSMART, in which user-designed incident cases are mapped to the appropriate response actions. Additionally, the SoSMART employs *case-based reasoning* to match the current system state to the situations previously identified as intrusive. A statically mapped response selection scheme makes FLIPS and SoSMART efficient and easy to implement, but diminishes their flexibility when deployed in a dynamically changing real-world environment.

EMERALD [31], a dynamic cooperative response system, introduces a layered approach to deploy monitors through different abstract layers of the network. Analyzing IDS alerts and coordinating response efforts, the response components are also able to communicate with their peers at other network layers. AAIRS [32] provides adaptation through a confidence metric associated with IDS alerts and through a success metric corresponding to response actions. Although EMERALD and AAIRS offer great infrastructure for automatic IRS, they do not attempt to balance intrusion damage and recovery cost.

αLADS [33], a host-based automated defense system, uses a partially observable Markov de-

cision process to account for imperfect state information; however, αLADS is not applicable in general-purpose distributed systems due to their reliance on local responses and specific profile-based IDS. Balepin et al. [11] address an automated response-enabled system that is based on a resource type hierarchy tree and a directed graph model called a *system map*. Both αLADS and the IRS in [11] can be exploited by the adversary, since none of them takes into account the malicious attacker's potential next actions while choosing response actions.

Game theory in an IRS-related context has also been utilized in previous papers. Lye et al. [34] use a game-theoretic method to analyze the security of computer networks. The interactions between an attacker and the administrator are modeled as a two-player simultaneous game in which each player makes decisions without the knowledge of the strategies being chosen by the other player; however, in reality, IDSes help administrators probabilistically figure out what the attacker has done before they decide upon response actions, as in sequential games. AOAR [35], created by Bloem et al., is used to decide whether each attack should be forwarded to the administrator or taken care of by the automated response system. Use of a single-step game model makes the AOAR vulnerable to multi-step security attacks in which the attacker significantly damages the system with an intelligently chosen sequence of individually negligible adversarial actions.

Foo et al. [36] present ADEPTS, an automated IRS that uses I-GRAPH, i.e., graphs of intrusion goals, to determine the spread of the intrusion and the appropriate response. A subtle, yet significant, difference between I-GRAPHS and the ART graph is that the former is designed according to intrusion scenarios, while the latter is based on consequences regardless of whatever attack scenarios cause them. SARA [37], a response engine architecure, consists of several components that function as sensors (information gathering), detectors (analysis of sensor data), arbitrators (selection of appropriate response actions), and responders (implementation of response); however, the authors do not study a particular response strategy.

2.9 Conclusions

A game-theoretic intrusion response engine, called the Response and Recovery Engine (RRE), was presented. We modeled the security maintenance of computer networks as a Stackelberg stochastic two-player game in which the attacker and response engine try to maximize their own benefits

by taking optimal adversary and response actions, respectively. Using an extended attack tree structure, called the attack-response tree (ART), RRE explicitly takes into account inaccuracies associated with IDS alerts in estimating the security state of the system. Moreover, RRE explores the intentional malicious attacker's next possible action space before deciding upon the optimal response action, so that it is guaranteed that the attacker cannot cause greater damage than what RRE predicts. Experiments show that RRE appropriately takes countermeasure actions against ongoing attacks, and brings an insecure network to its normal operational mode with the minimum possible cost.

CHAPTER 3

A CONSEQUENCE-CENTRIC SECURITY METRIC

Having discussed the mathematical framework for an intrusion response system in Chapter 2, we now start an in-depth explanation of the specific challenges typically faced by such a system, and our proposed solutions. In this chapter, we discuss the security assessment problem, and propose an information flow-based algorithm that a response system can use to assess and measure the security of each system state while choosing the optimal response action.

The current techniques for handling the security assessment problem generally fall short in three major aspects. First of all, the existing solutions are heavily reliant on human knowledge and involvement [38, 39, 40, 41], such that the system administrator must observe the triggered IDS alerts (possibly in a visual manner) and manually evaluate their criticality, which mainly depends on the alerts' accuracy and the underlying system configurations. As the size and complexity of the computer networks increase, manual inspection of the alerts becomes very tedious or even impossible in practice. Second, the current model-based approaches [42, 43, 44, 45] try to compute security metrics based on a manually designed model and a strong set of assumptions about the availability of knowledge about the attackers' behaviors and the vulnerabilities within the system. Finally, and probably the most important and subtle weakness of the previous techniques for IDS alert correlation and security state[1] estimation is that they often focus only on the attack paths [46] and subsequent privilege escalations [47, 48] without considering dependencies among system assets. They define the security metric of a given system state to be the smallest number of vulnerability exploitations (privilege escalations) needed to get from that state to the goal state in which the attacker can get the necessary privileges to cause his or her ultimate malicious consequence, e.g., a sensitive file modification. We call such metrics *attack-centric* metrics. Therefore, the met-

[1] A security state in the attack graph literature is usually defined to be the set of an attacker's privileges in that state, and the state transitions represent vulnerability exploitations that lead to privilege escalations.

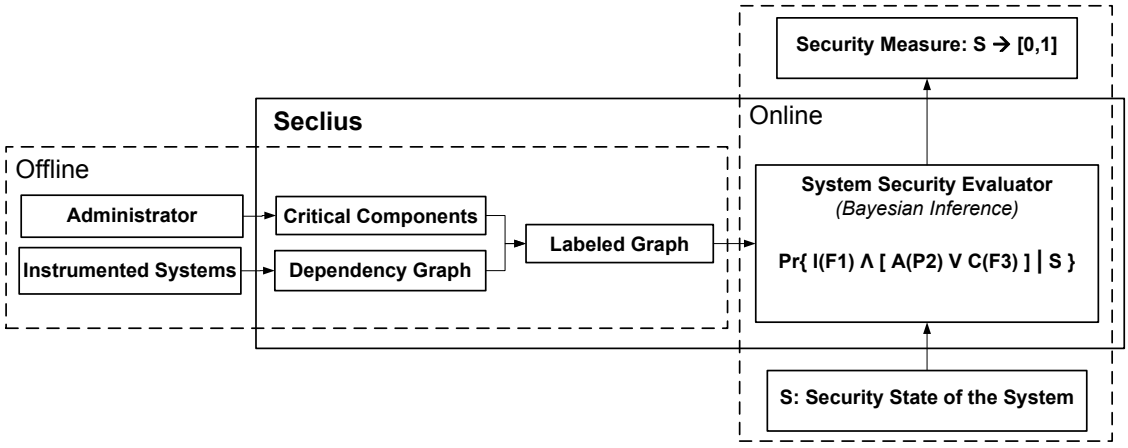


Figure 3.1: The Components of the Seclius Framework

ric is not defined for a non-goal state (regardless of the transitions); equivalently, it is assumed that all attackers will pursue exploitations until they get to the goal state (which is insecure by definition). However, in practice, there are often unsuccessful attacks that cause some level of damage, such as a Web server crash that results from an unsuccessful buffer overflow exploitation. Hence, it is important to consider the damage already caused by the attacker, and not only vulnerability exploitations, while evaluating the security measure of a system state.

In this chapter we present *Seclius* , an online system security evaluation engine that provides both a high-level security measure of the system, according to the received IDS alerts, and a ranking of malicious events and affected components of the system based on how crucial they are for the high-level security requirements of the organization. Separation of system characteristics from organizational security requirements enables Seclius to greatly minimize the manual inputs required while leveraging automated instruments to capture component dependencies. Administrators can focus on defining the high-level security requirements while the Seclius engine automatically learns the low-level system characteristics. Additionally, Seclius takes into account uncertainties of the detection systems and provides criticality assessment of the components affected by attackers.

3.1 System Overview

Seclius's high-level goal is to assess the security of each possible system state with minimal human involvement. In particular, we define the security of a system state as a binary vector in which each bit indicates whether a specific malicious event has occurred in the system.

We consider two types of malicious events. First, there are vulnerability exploitations, which are carried out by an attacker to obtain specific privileges and improve his or her control over the system. Therefore, the first set of bits in a state denote the attacker's privileges in that state, e.g., root access on the web server host. Those bits are used to determine what further malicious damage the attacker can cause in that state. Second, there are attack consequences, which are caused by the attacker after he or she obtains new privileges. Specifically, we defined consequences as the violations of the "CIA" criteria (i.e., Confidentiality, Integrity, and Availability) applied to critical assets in the organization, such as specific files and processes. For example, the integrity of a file F_2, denoted by $I(F_2)$, is compromised if F_2 is either directly or indirectly modified by the attacker.

According to [49], the security of any given system is characterized by a set of its identifiable attributes such as security criteria of the critical assets, e.g., integrity of a database file; the notion of a security metric is defined as a quantitative measure of how much of that attribute the system possesses. Our goal is to provide administrators with a framework to compute such measures in an online manner. We believe that there are two major challenges to achieving this goal. First, while the critical assets are system-specific and should be defined by administrators, a framework that requires too much human involvement is prone to errors and has limited usability. As a result, a formalism is needed so that administrators can define assets simply and unambiguously. Second, low-level IDS alerts usually report on local consequences with respect to a specific domain. Consequently, we need a method that provides understanding of what the low-level consequences represent in a larger context and quantifies how many of the security attributes the whole system currently possesses, given the set of triggered alerts.

We developed the Seclius framework (see Figure 3.1) to address those two challenges. Seclius works based on two inputs: a manually defined consequence tree, and an automatically learned dependency graph. The hierarchical and formal structure of the consequence tree enables administrators to define the critical assets easily and unambiguously with respect to the subjective mission

of the organization. The dependency graph captures the dependencies between these assets and all the files and processes in the system during a training period. In production mode, Seclius receives low-level alerts from intrusion detection sensors and uses a taint propagation analysis method to evaluate online the probabilities that the security attributes of the critical assets are affected.

To illustrate how Seclius works in practice, consider a scenario with an e-commerce organization. Administrators would first define critical assets and organizational security requirements through the consequence tree. We emphasize that this task does not require deep-knowledge expertise about the IT infrastructure. In our example, the critical assets would include the web server and the database server, and the security requirements would likely consist of the availability of the web server and the integrity and confidentiality of the database files. The second step would be to run a training phase with no ongoing attack in order to collect data on intra- and inter-host dependencies between files and processes. After a few hours, the results of this training phase would be automatically stored in the dependency graph, and the instruments used to track the dependencies would be turned off. The third and final step would be to start Seclius in production mode to start processing alerts from IDSes and probabilistically determine whether the critical assets are compromised. If a malicious web server exploit was detected by some IDS, Seclius would update not only the security measure of the web server, but also that of the database files that depend on the web server, according to the learned dependency graph. By observing cross-asset dependencies, Seclius can precisely measure 1) how the integrity, confidentiality, or availability of the two assets has been affected by the exploit, and 2) the privileges gained by the attacker and which security domains he or she was able to reach.

In the next section, we further describe the mathematical tools and the formalism used by the various components of Seclius to provide that information.

3.2 Consequence Tree

We discuss in this section the first manual input required by Seclius, namely the consequence tree (CT). The goal of the CT is to capture critical IT assets and organizational security requirements. The criticality of assets in an organization depends on its mission. For instance, in an online

shopping infrastructure, the availability of a web server process is usually more critical than the availability of an email server. The opposite would likely be true for an email service provider. Hence, Seclius requires system administrators to manually provide the list of organizational critical assets.

Generally, the critical assets could be provided using any function: a simple list (meaning that all items are equally important), a weighted list, or a more complex combination of assets. In this chapter, we chose to use a logical tree structure. We believe that it offers a good trade-off between simplicity and expressiveness, and the fact that it can be represented visually makes it a particular helpful resource for administrators. The formalism of the CT follows the traditional fault tree model [50]; however, unlike fault trees, the leaf nodes of the CTs in Seclius address security requirements (confidentiality, integrity, and availability) of critical assets, rather than dependability criteria.

The CT formalism consists of two major types of logical nodes, namely `AND` and `OR` gates. To design an organizational CT, the administrator starts with the tree's root node, which identifies the main high-level security concern/requirement, e.g., "Organization is not secure." The rest of the tree recursively defines how different combinations of the more concrete and lower-level consequences can lead to the undesired status described by the tree's root node. The recursive decomposition procedure stops once a node explicitly refers to a consequence regarding a security criterion of a system asset, e.g., "availability of the Apache (apache2d) is compromised." Those nodes are in fact the CT's leaf consequence nodes, which take on binary values indicating whether their corresponding consequence has happened (1) or not (0). Throughout the chapter, we use the C, I, and A function notations to refer to the CIA criteria of the assets. For instance, $C(F_2)$ and $I(P_6)$ denotes confidentiality of file F_2 and integrity of process P_6, respectively. The leaves' values can be updated by IDSes, e.g., Samhain [51]. The CT is derived as a Boolean expression and the root node's value is consequently updated to indicate whether the organizational security is still being maintained.

A CT indeed formulates subjective aspects of the system. Its leaf nodes list security criteria of the organization's critical assets. Additionally, the CT implicitly encodes how critical each asset is using the depth of its corresponding node in the tree; that is, the deeper the node is, the less critical the asset is in the fulfillment of the organizational security requirements. Furthermore, the CT

formulates redundant assets using AND gates. Seclius requires administrators to explicitly mention the redundancies because it is often infeasible to discover redundancies automatically over a short learning phase [52].

The CT formulation is different from the attack tree formalism, as the CTs formulate how past *consequences* contribute to the overall security requirements, whereas attack trees usually address attackers' potential *intents* against the organization. In other words, at each time instant, given the consequences already caused by the attackers in the system, Seclius employs the CT to estimate the *current* system security, while the system's attack tree is often used to probabilistically estimate how an attacker could or would penetrate the system in the *future*.

3.3 Dependency Graph

As mentioned in the previous section, the CT captures only the subjective security requirements, and does not require deep-knowledge expertise about the IT infrastructure, thanks to the dependency graph (DG). The goal of the DG is to free the administrator from providing low-level details about the organization. These details are automatically captured by Seclius during a learning phase, during which the interactions between files and processes are tracked in order to probabilistically identify direct or indirect dependencies among all the system assets. For instance, in a database server, the administrator only needs to list the sensitive database files, and Seclius later marks the process `mysqld` as critical because it is in charge of reading and modifying the databases. Such a design greatly reduces the resources and time spent by administrators in deploying Seclius.

Each vertex in the DG represents an object, namely a file, a process, or a socket, and the direct dependency between two objects is established by any type of information flow between them. For instance, if data flow from object o_i to o_j, then object o_j becomes dependent on o_i; the dependency is represented by a directed edge in the DG, i.e., $o_i \rightarrow o_j$. To capture this information, Seclius intercepts syscalls and logs them during the learning phase. In particular, we are interested in the syscalls[2] that cause data dependencies among the OS-level objects. A dependency relationship is

[2]Specifically, we log the following syscalls: `openat`, `dup`, `dup2`, `close`, `write`, `writev`, `read`, `readv`, `ipc`, `clone`, `fork`, `vfork`, `execve`, `open`, `creat`, `mmap`, `mmap2`, `socketcall`, `recv`, `recvfrom`, `recvmsg`, `send`, `sendto`, and `sendmsg`.

stored by three elements: a source object, a sink object, and their security contexts. When the learning phase is over, syscall logs are automatically parsed and analyzed line by line to generate the DG. Each dependence edge is tagged with a frequency label indicating how many times the corresponding syscalls were called during the execution.

We make use of the Bayesian network formalism to store probabilistic dependencies in the DG; a conditional probability table (CPT) is generated and stored in each vertex. This CPT encodes how the information flows through that vertex from its parents (sources of incoming edges) to its children. For example, if some of the parent vertices of a vertex become tainted directly or indirectly by attacker data, the CPT in the vertex saves the probability that the tainted data will propagate to any of the children through the vertex.

More specifically, each DG vertex is modeled as a binary random variable (representing a single information flow), equal to either 1 (true) or 0 (false) depending on whether the vertex has been tainted; the CPT in a vertex v stores the probability that the corresponding random variable will take the true value ($v = 1$), given the binary vector of values of the parent vertices $P(v)$. Formally, the probability value is computed using the following equation: $Pr(v|P(v)) = 1 - \prod_{p_v^i \in P(v)} \{1 - \mathbf{1}_{(p_v^i)}.Pr(p_v^i \rightarrow v)\}$ where $\mathbf{1}_{(.)}$ is the indicator function that takes on the value 1 if the condition inside the parentheses holds, and 0 otherwise. $Pr(p_v^i \rightarrow v)$ is the probability that the information will flow from the parent node p_v^i to the vertex v. This probability represents the fraction of times during which information flows from p_v^i to v. It is calculated using the frequency labels on the p_v^i's outgoing edges. In summary, the vertex v takes on the value 1 if information flows from any of its parents that have the value 1.

Figure 3.2 illustrates how a CPT for a single flow (with 1-bit random variables) is produced for a sample vertex, i.e., the file F_4. The probabilities on the edges represent $Pr(. \rightarrow .)$ values. For instance, the process P_1 writes data to the files F_4 and F_7 with probabilities 0.3 and 0.7, respectively. As shown in the figure, the file F_4 cannot become tainted if none of its parents are tainted, i.e., $Pr(F_4|\bar{P}_1, \bar{P}_9) = 0$. If only the process P_1 is tainted, F_4 can become tainted only when the information flows from P_1, i.e., $Pr(F_4|P_1, \bar{P}_9) = 0.3$. If both of the parents are already tainted, then F_4 would get tainted when information flows from either of its parent vertices. In this case, the probability of F_4 being tainted would be the complement probability of the case when information flows from none of its parents. Therefore, $Pr(F_4|P_1, P_9) = 1 - (1 - 0.3) \times (1 - 0.8) = 0.86$.

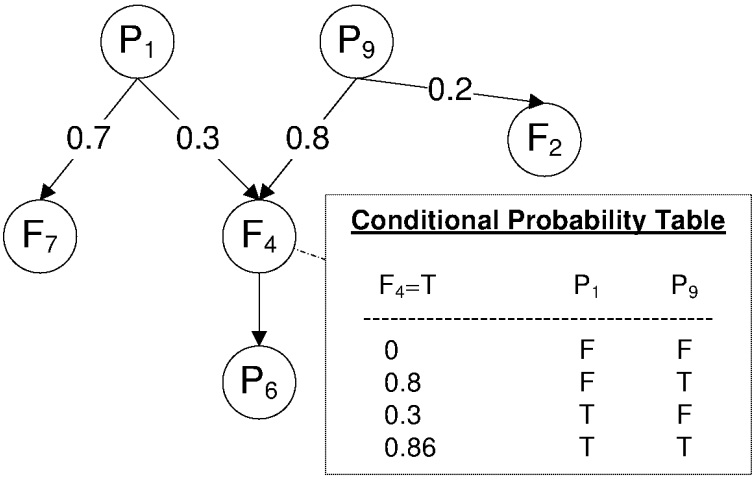


Figure 3.2: Conditional Probability Table Construction

Each CT leaf node that represents a CIA criterion of a critical asset is modeled by Seclius as an information flow between the privilege domains controlled by the attacker (according to the current system state) and those that are not yet compromised. Confidentiality of an object is compromised if information flows from the object to any of the compromised domains. Integrity of an object is similarly defined, but the flow is in the reverse direction. Availability is not considered as an information flow by itself; however, an object's unavailability causes a flow originating at the object, because once an object becomes unavailable, it no longer receives or sends out data as it would have if it had not been compromised. For instance, if a process frequently writes to a file, once the process crashes, the file is not modified by the process, possibly causing inconsistent data integrity; this is modeled as a propagation of tainted data from the process to the file. We consider all the leaf nodes that concern the integrity criterion of critical assets as a single information flow, because they conceptually address the same flow from any of the compromised domains to the assets. However, confidentiality flows cannot be grouped as they originate individually at separate sources.

If each information flow is represented as a bit and to completely address n concurrent information flows, we define the random variable in each vertex as an n-bit binary vector in which each bit value indicates whether the vertex is already tainted by the bit's corresponding flow. In other words, to consider all the security criteria mentioned in a CT with n leaves, every vertex denotes an n-bit random variable (assuming integrity bits are not grouped), where each bit addresses a single

flow, i.e., a leaf node. The CPTs are generated accordingly; a vertex CPT stores the probability of the vertex's value given the value of its parents, each of which, instead of true/false, can take on any n-bit value.

3.4 System Security Evaluation

Given the DG generated during the learning phase, all the syscall interception instrumentations are turned off, and the system is put in its operational mode. The learned DG is then used in an online manner to evaluate the security of any system security state. The goal of this section is to explain how this online evaluation works in detail. We first assume that the IDSes report the exact system state with no uncertainty. We discuss later how Seclius deals with IDS inaccuracies.

At each time instant, to evaluate the security of the system's current state s, DG vertices are first updated according to s, which indicates the attacker's privileges and past consequences (CT's leaf nodes). For each consequence in s, the corresponding flow's origin bit in DG is tainted. For instance, if file F_4 is modified by the attacker (integrity compromise), the corresponding source bit in DG is set to 1 (evidence bit).

The security measure for a given state s is defined to be the probability that the CT's root value is still 0 ($Pr(root\bar{(}CT)|s)$), which means that the organizational security has not yet been compromised. More specifically, if the CT is considered as a Boolean expression, e.g., $CT = (C(F10) \wedge A(P_6)) \vee I(F_2)$, Seclius calculates the corresponding marginal joint distribution, e.g., $Pr[(C(F10) \cap A(P_6)) \cup I(F_2)]$, conditioned on the current system state (tainted evidence vertices).

Seclius estimates the security of the state s by calling a belief propagation procedure, namely, the Gibbs sampler [13], on the DG to probabilistically estimate how the tainted data are propagated through the system while in state s. The Gibbs sampler uses the DG's CPT to generate a large number of samples from the $Pr[CT|s]$ distribution without directly calculating the density function [13]. Similarly, the security measure is estimated for each system state individually. Therefore, if the attacker modifies any other object and/or gets more privileges, the system would switch to a new state, whose security measure would be separately evaluated.

It is worth emphasizing that Seclius does not use the DG model to estimate how the attacker con-

tacts other objects from a compromised object, such as a tainted process, to exploit a vulnerability and/or escalate his or her privileges. Seclius uses the DG only to estimate how the tainted data would propagate through other *non compromised* system assets, which would behave normally as they did during the learning phase. For every asset already compromised, Seclius assumes a pessimistic behavior model, i.e., the asset deterministically contacts all other assets in its privilege domain.

This approach can evaluate the security of each system state. However, the exact current security state of the system is usually not completely observable, due to IDS inaccuracies, i.e., false positive and negative rates. We define the notion of the *information state* of the system, which formally is a probability distribution over all states in the state space of the system $s \in S$. The information state of the system is estimated, based on the IDS alerts and the false positive and negative rates of the IDSes, using the following equation: $Pr(s) = \prod_{i=1}^{|s|-1}(\mathbf{1}_{s_i=1} \cdot [1 - FP(s_i)] + \mathbf{1}_{s_i=0} \cdot [1 - FN(s_i)])$, where $\mathbf{1}$ is the indicator function, and s_i is the binary state variable in state s. $FP(s_i)$ and $FN(s_i)$ denote the false positive and negative rates, respectively, that depend on the intrusion detection system by which the corresponding alerts are triggered. The false positive and negative rates can be set to be deterministic, i.e., 0, if the detectors are quite accurate. Seclius can also take qualitative values, i.e., $FP : s_i \rightarrow \{low, medium, high\}$, which are later translated into crisp values, i.e., $\{0.25, 0.5, 0.75\}$.

Once the information state of the system has been estimated, Seclius computes the expected security measure of the information state using the following equation: $\sum_{s \in S}(Pr(s) \cdot Sec(s))$ where the *Sec* function is evaluated for the exact state s, as described above.

3.5 Implementation

Seclius uses syscall interception to capture information flow and learn data dependencies. We implemented it as a loadable kernel module, which currently works with the Linux kernels up to 2.6.32-22. However, for recent kernels, before replacing the syscall table entries with our wrapper syscall function pointers, we had to automatically find the address of `sys_call_table`, because its address is no longer exported. Furthermore, we had to make the corresponding memory pages

writable as, for security purposes, the syscall table is read-only by default. Once deployed, the module logs called syscalls, and information about the calling processes, e.g., PIDs.

Instead of syscall interception, a byte-by-byte information flow tracer, e.g., the TEMU [53] instruction-level taint tracker, could alternatively be used. But, TEMU's overhead is usually too high for a production system (see Section 3.6). However, we used TEMU as a ground truth to evaluate the accuracy of the flow tracer in Seclius. To actually use TEMU in our experiments, we modified it to produce higher-level information (not only instructions) regarding file-system objects, such as files that are getting dynamically tainted. Using The Sleuth Kit (TSK) [54], our implementations read the virtual machine's file system, and dynamically translate disk-level tainted addresses (generated by TEMU) to file system object names, such as file names and their absolute addresses.

Seclius employs the DlibC++ library [55], which performs approximate Monte Carlo inference in the DG. Briefly, the Gibbs sampler in DlibC++ starts with an arbitrary initial state of the network, using `set_node_value`, and then iterates by randomly sampling possible values from the network and incrementally estimating the probability of a given expression using the `node_value` function.

3.6 Evaluation

We evaluate the accuracy and performance of the various components of Seclius. In particular, we designed a set of experiments to empirically evaluate the following questions: How accurate are the DG-generating instruments, and how much overhead do they generate? How long should the training phase be to build the DG with sufficient coverage? How much time does Seclius require to process alerts and assess the security metric value of the different assets? How does an inaccurate CT affect the results? How does Seclius automatically prioritize past security events for administrators?

We start by describing our experimentation setup, and then proceed to examine these five questions.

Experimentation Setup. The experiments in this section were conducted on a virtual machine so that we could easily reset our testing environment to a clean state. The host computer had a 2.20

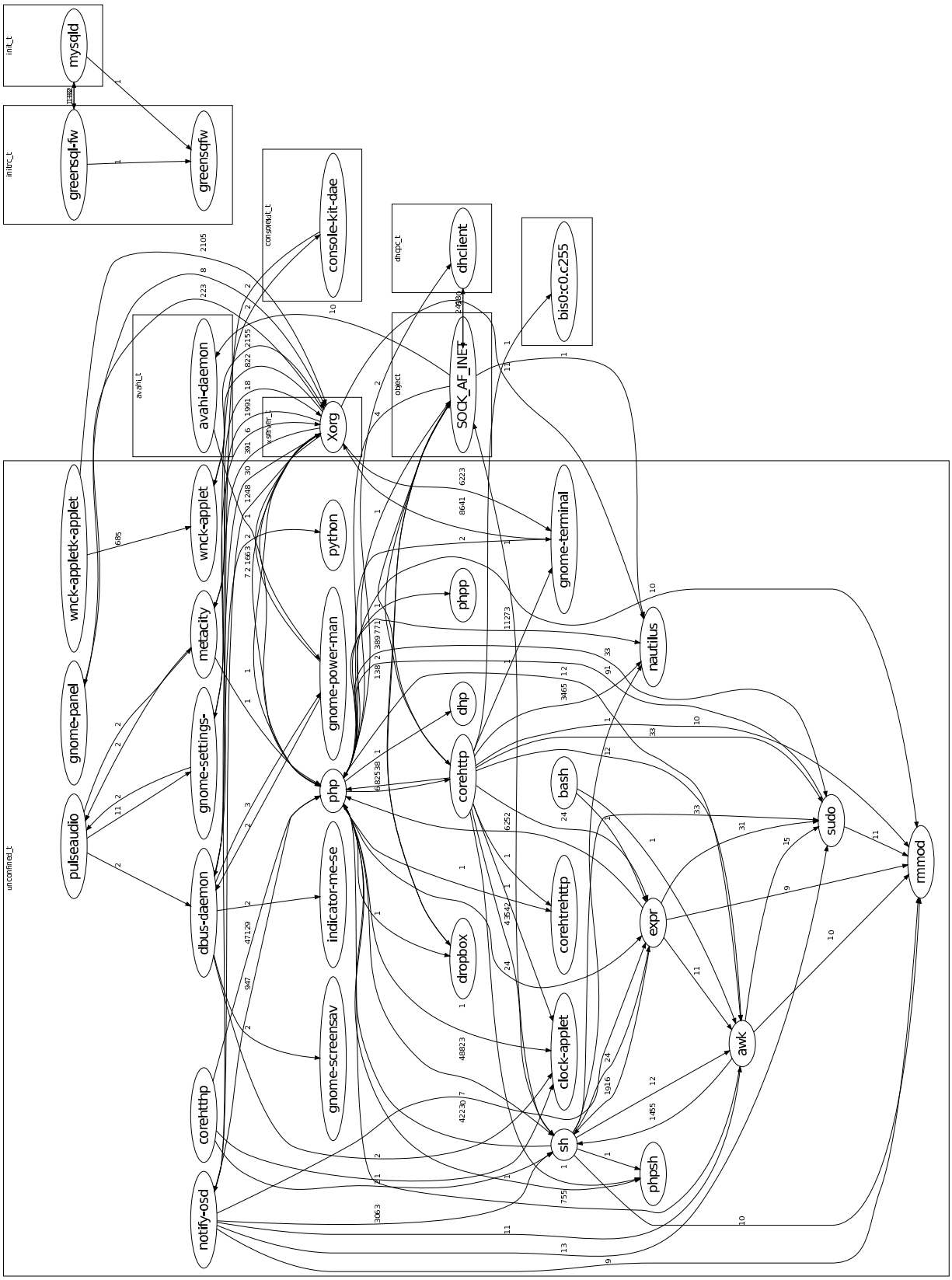


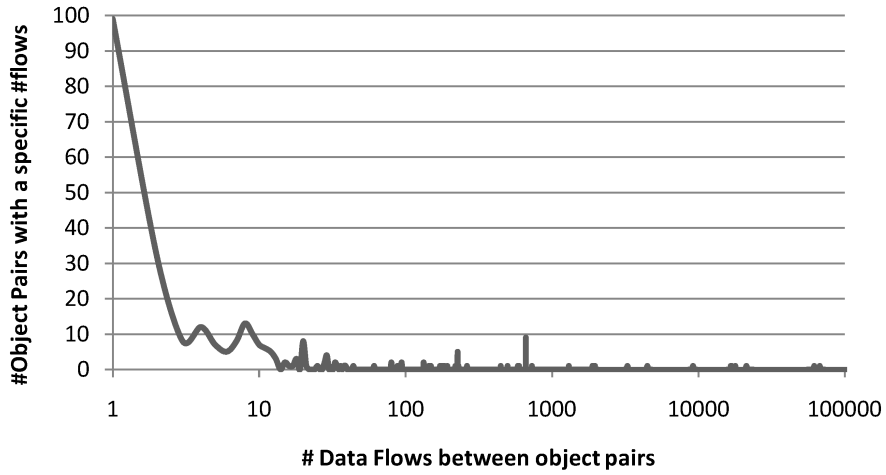
Figure 3.3: Dependency Graph Including Processes Only

GHz AMD Athlon^{TM}64 3700+ CPU, 1 MB of cache, and 2.0 GB of RAM. The guest OS, using VirtualBox, was running Ubuntu 9.04 with a Linux 2.6.22 kernel. We installed several applications, including OpenSSH server, Apache2, MySQL, the eVision content management system, and the RoomPHPlanning web scheduling application. We additionally changed the `php.ini` file to make the system vulnerable to PHP code injection attacks.

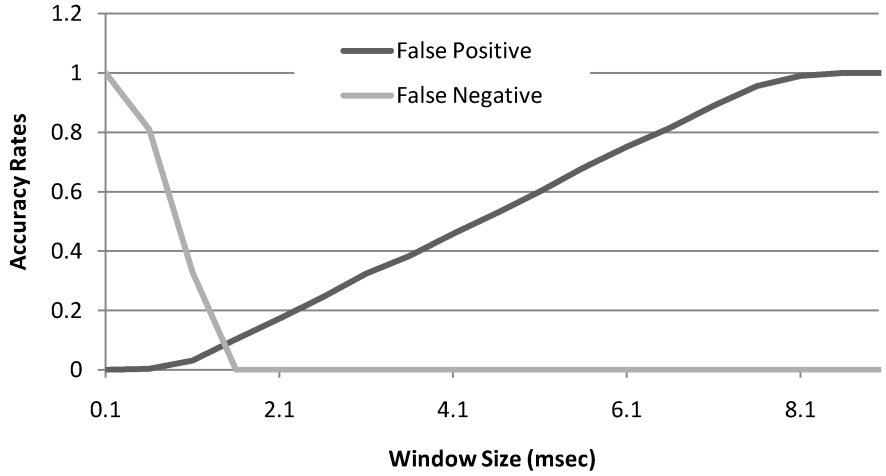
To feed intrusion detection alerts to Seclius, we installed the following sensors: Snort [56]: a lightweight network-based IDS that uses a database of known attack signatures. In our experiments, we ran Snort in fast-alerting detection mode with its standard 74 rule-sets; DazukoFS [57]: a cross-platform device driver that allows applications to control file accesses on the system; ClamAV [58]: an open-source antivirus toolkit. In particular, we used Sigtool in ClamAV to update the virus signatures and Clamscan for scanning downloaded suspicious files. Samhain [51]: a host-based IDS that periodically checks file integrity; Zabbix [59]: a statistical anomaly-based IDS to detect process or network failures and resource intensive activities; and PHPIDS [60]: a monitoring tool for PHP applications to detect cross-site-scripting and remote code execution attacks.

Accuracy and Overhead of the DG Instruments. As mentioned in the previous section, we developed a syscall interception kernem module to automatically generate the DG. Seclius also captures detailed security contexts through SE-Linux, which we enabled and configured with the default Ubuntu policy `policy.24`. The experiments were done on the Apache web server process while it was benchmarked from a remote machine using the command `ab -n 50000`, which subsequently generates repeated HTTP requests. Figure 3.3 shows a sample DG for our experimental web server; rectangles in the figure represent different SE-Linux privilege domains within the system, and the vertices are OS-level objects. for clarity, only the process nodes are shown. The original graph, including all FIFOs and other OS-level objects, consisted of 8,396 nodes. According to the syslogs, the process had periodically been reading from the network sockets `SOCK_AF_INET`, and then accessed some FIFO pipes, the `/var/www/index.html` and `/var/log/apache2/access.log` regular files. Figure 3.4(a) shows the data-flow histogram of the DG. More specifically, the vertical axis depicts how many vertex pairs in the DG experienced a specific data-flow frequency given by the horizontal axis in the figure.

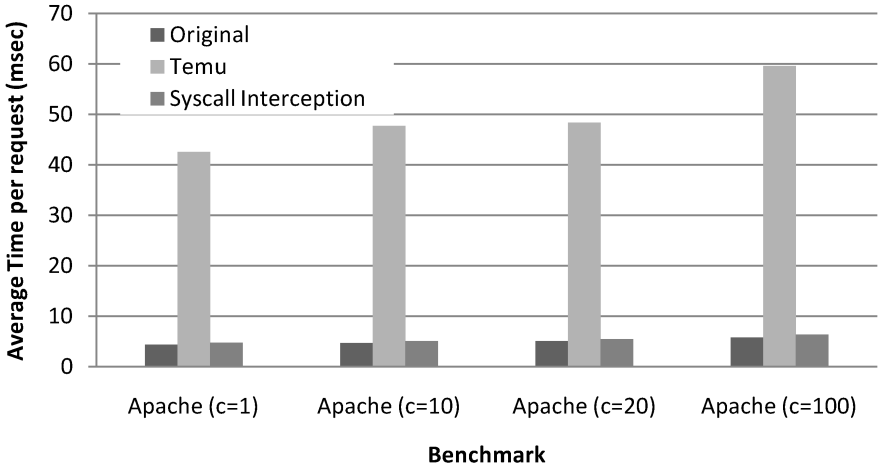
We define the DG's accuracy as the portion of dependencies correctly identified among system assets. Our choice of using syscall interception rather than byte-by-byte instruction-level taint-



(a) The DG's Behavior Histogram



(b) Accuracy of Syscall Taint Tracking



(c) Overhead Syscall vs. TEMU

Figure 3.4: Accuracy and Overhead Analysis of the Dependency Graph Instruments

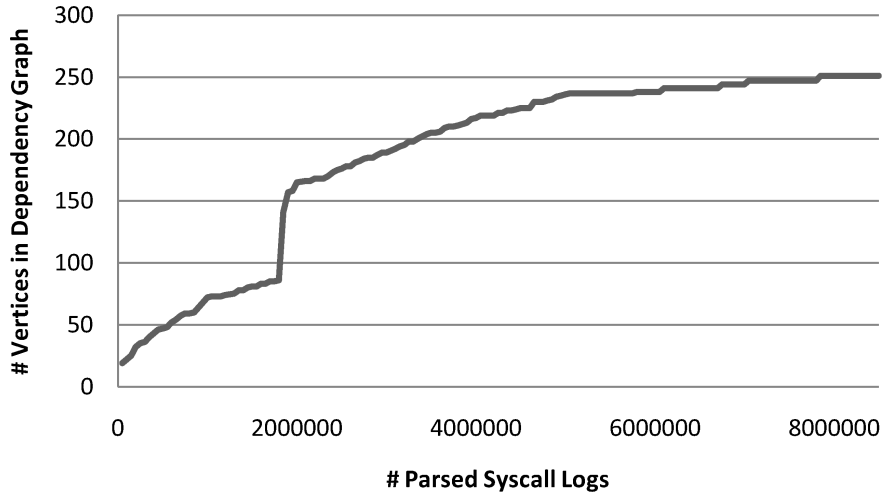
tracking can generate false positive, i.e., outgoing data flows wrongly marked as tainted, or false negatives. Indeed, our approach uses a sliding window of size t to associate a DG vertex's outgoing data flow after each incoming input. The numbers of false positives and false negatives directly depends on the size of t.

To evaluate the inaccuracies of our syscall interception approach, we varied t when traacking the Apache process and we compared the results to ground truth provided by TEMU. The false positive rate was calculated as $\#FalsePositives/(\#FalsePositives + \#TrueNegatives)$. Similarly, the false negative rate was estimated using $\#FalseNegatives/(\#FalseNegatives + \#TruePositives)$. Figure 3.4(b) shows the results for the various window sizes employed. The false positive rate grows linearly up to 1.0 as the window size exceeds 8.1 milliseconds. The false negatives, on the other hand, quickly drop to zero before the window size gets to 2 milliseconds. Based on these results, 1.5 milliseconds seems to be a reasonable window size for our environment, as both the false positive and negative rates remain fairly low.

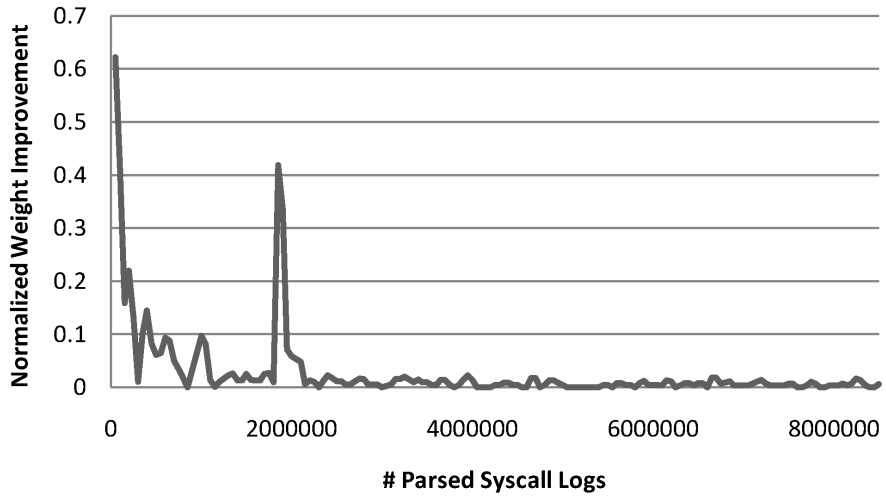
We also evaluated the overhead of our syscall tracking module compared to TEMU. Figure 3.4(c) shows how much overhead each technique caused within the system in terms of end-point response time for the Apache web server. The server was benchmarked against different numbers of concurrent clients. As shown in the figure, the syscall interception caused less than 10% overhead on average compared to the original execution, which was not instrumented, whereas the TEMU engine caused an approximately 850% slowdown in the web server response time, which is clearly not suitable for practical uses.

Required Training Time for Sufficient Dependency Coverage. To configure Seclius, the normal system behavior (i.e., the DG) should be captured and learned. The training duration should be long enough to create a model representing the actual system. On the other hand, to accelerate the setup phase of Seclius and avoid large syscall logs, the training mode should be reasonably short. In this section, we evaluate how long it takes for the DG parameters to converge in the context of our Apache web server running a PHP application, namely the eVision content management system.

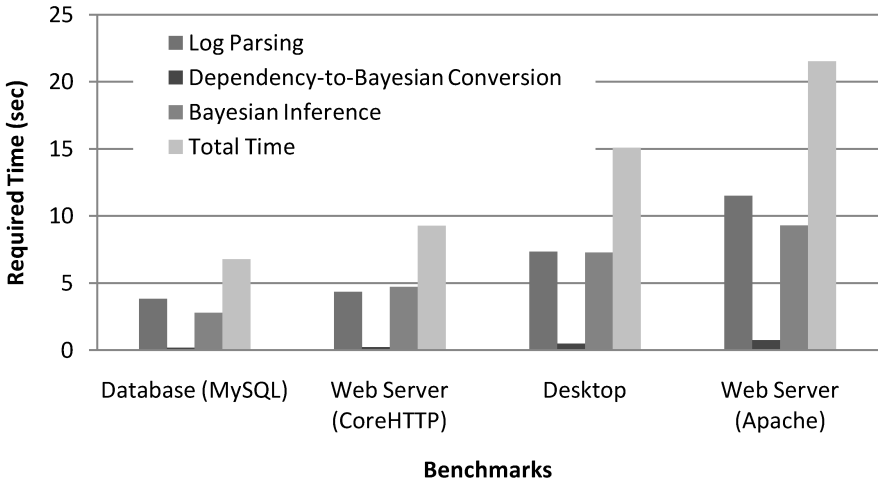
First, we focused on how the DG cardinality, i.e., number of vertices, gets updated and finally converges. We initially set the system up for the training without any incoming web client requests. Figure 3.5(a) shows the DG cardinality during the experiment vs. the number of syslogs, i.e., 1.8E6



(a) Dependency Graph Converge



(b) DG's Edge Label Converge



(c) Time Requirements

Figure 3.5: Convergence and Time Requirement Analysis of the Dependency Graph

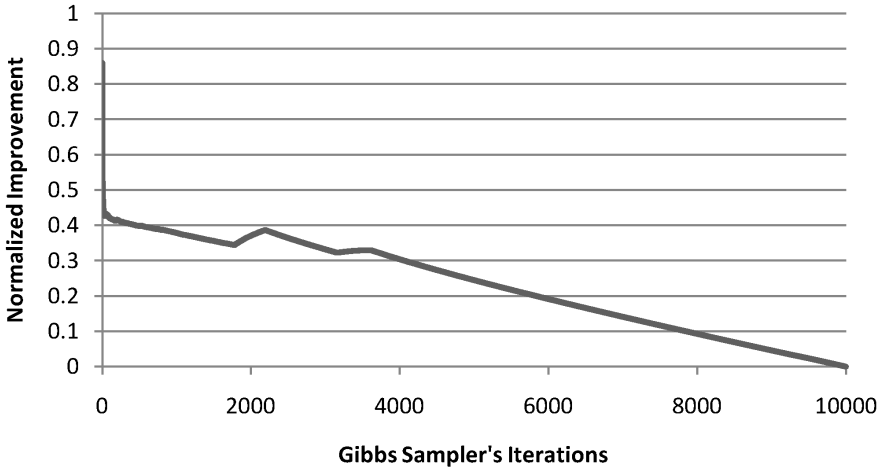


Figure 3.6: Convergence of the Gibbs Sampler Algorithm

syscalls per minute on average. The DG cardinality approached 87 within 67 seconds. Then, we started the `ab` application on a remote machine to send subsequent requests to the server; the request flow caused a 64-vertex jump in the DG cardinality. The reason for the jump was that Apache started accessing other OS-level objects, such as `/var/www/eVision/index.php`. It took approximately 14 more minutes for the DG cardinality to again completely converge to its stable value. The whole learning phase took about 19 minutes, and approximately 2.3 GB of syscalls were logged.

Second, we evaluated how the DG edge labels, i.e., probability values, converged during the training mode. Figure 3.5(b) shows the normalized label updates, i.e., the difference between the last and the updated values, while syslogs were being parsed. As shown in the figure, it took about 3 minutes on average for the edge labels to converge. The launched request flow caused a spike in the graph, but the normalized label updates quickly converged back to zero again.

Online Processing Time to Generate Security Metric Values. Once the training mode is complete, Seclius translates the DG to a Bayesian network (by producing CPT tables), and then uses it to evaluate the system security in an online manner. In this section, we evaluate how much each step of the algorithm contributes to the overall processing time.

The four major automated steps by Seclius are 1) syscall interception (the learning phase), 2) parsing of the syslogs and the DG creation, 3) conversion of the DG to the Bayesian network (CPT generation), and 4) online security evaluation via implementation of the Gibbs sampler.

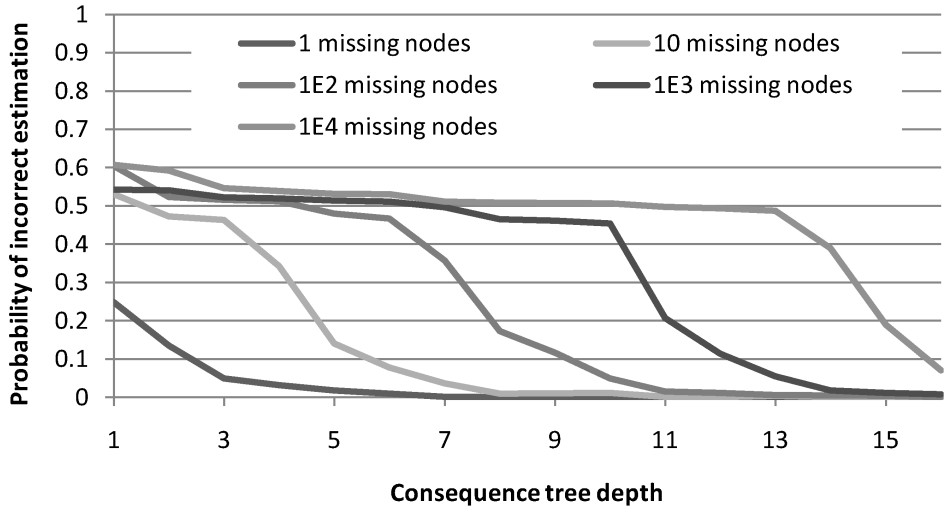


Figure 3.7: Incorrect Estimation Due to Incorrect CT

The learning phase was discussed in detail in the previous section. Figure 3.5(c) shows the time required for the steps after the system model has been learned on four machines. As demonstrated, parsing of syscall logs (43% on average) and the online security evaluation (53% on average) constitute two major portions of the total time, and the translation to the Bayesian network takes less than 4% of the total time.

Like any other Monte Carlo-based sampling technique, the Gibbs sampler algorithm terminates once the iterative conditional probability estimation improvement has satisfied a convergence threshold. Figure 3.6 shows how the delta value, i.e., the absolute value of the improvement by two subsequent sampling iterations, approaches zero (the Y axis is shown in the logarithmic scale base 10) with respect to the finished iterations, i.e., approximately 1,200 iterations per second.

Impact of Inaccurate Security Requirements. We also evaluated how much the estimated security is affected if the system operator provides a partially incorrect CT. For a given CT and for various leaf node values, we counted how many times the CT's root value was updated. In particular, we compared situations in which the full tree was used to get the root node value to situations in which the tree was partially missing randomly chosen nodes. We ran this comparison for a large number (1,000) of randomly generated full binary CTs in which nodes between the leaves and the root were chosen to be either `AND` or `OR` gates with a probability of 0.5. We did the experiment for full trees of a depth between 1 and 16. In each case, we compared the results when various numbers of nodes were missing.

Figure 3.7 shows the results for different scenarios. When the depth of the generated tree approaches 16, i.e., 2^{16} leaf nodes, the probability of getting an incorrect security estimation goes to zero for a fixed number of missing nodes. Note that when the number of missing nodes, shown on the graph, is larger than the tree order, all nodes are in fact missing; in other words, security is estimated without use of the CT, and hence, the estimated security is correct with an approximately 0.5 chance.

3.7 Case Study

We now illustrate how Seclius can assist administrators in gaining situational awareness through two realistic case studies. We start by showing how Seclius updates the security value of a set of processes running on an e-commerce machine under attack. We then describe how a multi-step attack occurring in a process control network can be precisely tracked by security administrators. It is worth emphasizing that to evaluate Seclius during the entire duration of our experiments, there will be no explicit response or recovery actions against attacks, i.e., all attacks will be accomplished successfully.

Online Security Evaluation for an E-commerce System. For this experiment, a single system instrumented with SE-Linux hosted a web server, a content management system that reads and writes from/to a MySQL database file. Due to the SE-Linux kernel module, important processes, e.g., `mysql` and `apache2`, were running in different privilege domains. The administrator defined a CT with a single critical asset, namely the integrity of the database file. Consequently, during the Bayesian inference analysis, Seclius was only concerned about information flow from any privilege domain controlled by the attacker to the database file.

Seclius later used the DG for the online security evaluation. Table 3.1 shows, for each system asset, i.e., DG vertex, the probability that the asset will be tainted by the data coming from the attacker. Each column represents a single attack step, and due to space limits, the table only shows the vertices with nonzero values; however, the main value to consider is the tainting probability of the database file.

Before the attack, Seclius assumed that the attacker was sitting outside and marked all the net-

Table 3.1: DG Updates During a Multi-step Attack

Multi-Step Attack		
Not attacked	Web server (Buff-Overflow)	SQL injection
database file (0.001)	**database file (0.012)**	**database file (0.43)**
nautilus (0.003)	nautilus (0.004)	nautilus (0.005)
avahi-daemon (0.697)	avahi-daemon (0.734)	avahi-daemon (1)
apache2 (0.126)	apache2 (1)	apache2 (1)
dropbox (0.945)	dropbox (0.945)	dropbox (0.96)
SOCK_AF_INET (1)	SOCK_AF_INET (1)	SOCK_AF_INET (1)
avahi-daahi-daemon (0.693)	avahi-daahi-daemon (0.732)	avahi-daahi-daemon (1)
/var/www/ (0.013)	avahi-daen (0.02)	avahi-daen (0.03)
mysqld (0.053)	/var/www/ (0.07)	/var/www/ (0.093)
/usr/share/ (0.002)	mysqld (0.468)	mysqld (1)
/var/log/ (0.121)	/usr/share/ (0.009)	/var/log/ (0.951)
VBoxService (0.006)	/var/log/ (0.946)	apache2us (1)
dhclient (0.677)	/etc/group (0.004)	/var/ (0.001)
avahi-daehi-daemon (0.002)	dhclient (0.678)	dhclient (0.679)
avahi-da-daemon (0.003)	avahi-daehi-daemon (0.004)	avahi-daehi-daemon (0.004)
nautilushi-daemon (0.001)	apache2us (1)	avahi-da-daemon (0.006)
avahi-daen (0.02)	avahi-da-daemon (0.005)	/etc/group (0.007)

work sockets tainted by the attacker before starting the security evaluation. The results shown in the first column of the table are the tainting probability values after the security evaluation was completed. As shown, before the attack, the database file was secure (i.e., not tainted by data from the attacker) with probability $1 - 0.001 = 0.999$. However, as the attacker penetrated further into the system and escalate privileges, getting closer to the critical database file, the probability of its integrity being compromised went up (first row of the table), even if it was not directly modified by the attacker. This increase came from the Gibbs sampler algorithm that recursively traversed the DG and tried to find all the assets on which any of the critical assets depend either directly or indirectly.

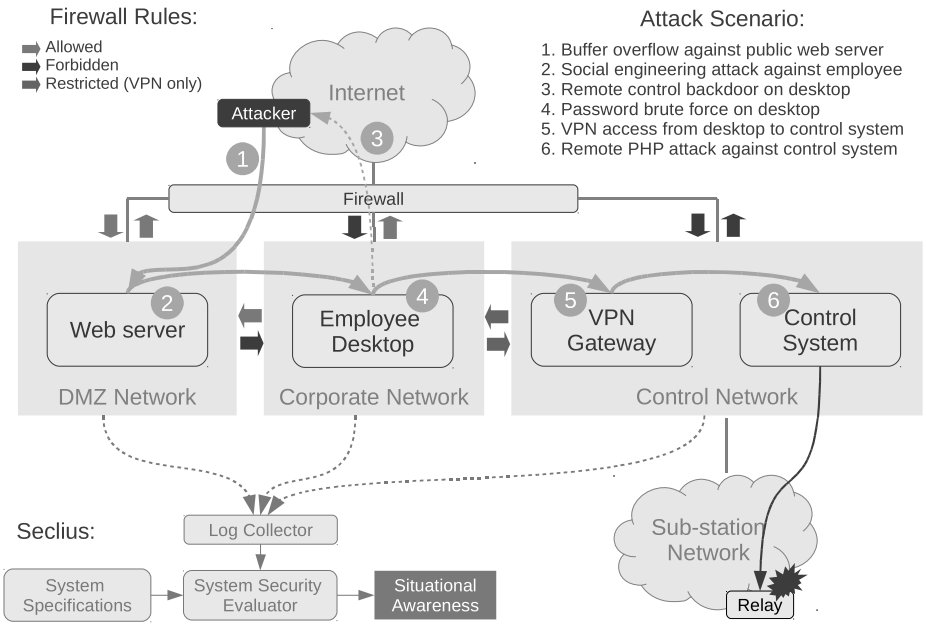
Power-Grid Organization with SCADA Network. To illustrate how the Seclius framework works in a critical infrastructure, we implemented it in a power supervisory control and data acquisition (SCADA) environment, namely the Trustworthy Cyber Infrastructure for the Power Grid (TCIPG) testbed [61]. We picked the SCADA environment because cyber-security in such critical infrastructures is paramount [62]. Furthermore, such process control networks display rather deterministic behaviors [63] that can be learned efficiently. The TCIPG testbed is compatible with the control systems reference architecture that has been proposed by the Department of Homeland Security [64]. Figure 3.8(a) shows the high-level architecture of the testbed, which consists of

three network zones, namely a DMZ, a corporate network, and a control networks. The traffic is compartimented by firewall rules that are depicted with color-encoded arrows in Figure 3.8(a). The control network is in charge of receiving sensory information from the power system and sending control commands, i.e., power generation set points, back. In particular, we connected the control system to a PowerLinc X10-TW523 [65] device to turn off or on a remote power system component represented by an electric fan in our experiments. The corporate network is responsible for business management and customer interaction. As shown in the figure, all the data flow between the corporate and control networks is allowed only through VPN tunnels. The corporate-customer interaction is done through the two web servers, which are running in a demilitarized zone for improved security.

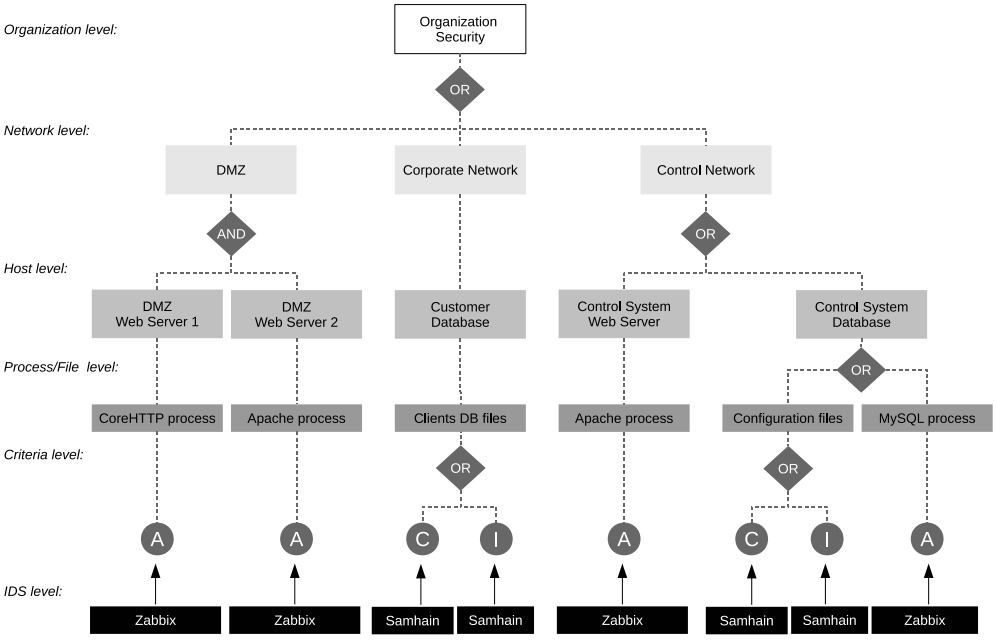
The testbed has intentionally been set up to include several real vulnerabilities. In particular, the CoreHTTP web server that resides in the DMZ zone is vulnerable to a buffer overflow exploitation based on CVE-2007-4060. The desktop machine in the corporate network suffers from a weak root password, i.e., `root2008`, which makes the host vulnerable to password brute-force attacks. The control system is running the eVision content management system to store data, including configuration parameters, that pertain to the underlying power system relay. According to CVE-2008-6551, multiple directory traversal vulnerabilities in the eVision 2.0 content management system allow attackers to include and execute arbitrary PHP files.

Figure 3.8(b) shows the CT for the TCIPG testbed. The root node defines the highest-level security requirement, i.e., "Organization Security," decomposed into the underlying structural Boolean tree that formulates how the root criteria would be affected with lower-level consequences. As shown in the figure, starting from the organization level, the CT has six different granularity levels and includes the three networks zones and their corresponding hosts. We employed the Nmap network scanner to discover hosts within each network and automatically provide a template to the system administrator. The administrator then specified the main critical asset(s) within each host. Figure 3.8(b) also shows the set of IDSes that we deployed in the testbed to monitor and report on modifications made to the critical assets; we used Zabbix to check for the health of processes, and Samhain to check whether any sensitive file was read or modified.

As mentioned earlier, a system state is defined as the set of past consequences from attackers' actions and his or her privileges over the system. Zabbix and Samhain are responsible for captur-



(a) Experimental Case Study: SCADA Testbed



(b) Consequence Tree for the SCADA Testbed in Figure 3.8(a)

Figure 3.8: TCIPG Supervisory Control and Data Acquisition Testbed and the Corresponding Consequence Tree

ing the attack consequences. However, they do not provide any information about how attackers penetrate into the system, i.e., what vulnerabilities are exploited to gain privileges over the system. To detect the vulnerability exploitations and privilege escalations, we deployed Snort, LibSafe, ClamAV, and PHP-IDS.

Figure 3.8(a) also reveals the path through which we launched a multistep attack to get access to the control system. Our attack scenario includes the following steps. The remote attacker obtained access to the DMZ zone by exploiting the vulnerable CoreHTTP web server to install a fake plugin request website. The goal of this malicious website is to lure an employee into downloading a malware that we developed. This social engineering attack allowed the attacker to take control of an employee desktop inside the corporate network. We note that only such an indirect attack could work because the firewall rules deny direct access/penetration from the DMZ zone or the Internet to the corporate network. The malware installed a back-door shell that sends periodic requests to the attacker. Once the shell access was gained on the corporate network machine, the attacker ran the *John the Ripper* password cracker to obtain the local root password. This step took about 18 seconds to complete. Then, using the root access, the attacker initiated a VPN session to the VPN gateway in order to launch a PHP code injection against a web service hosted in the control system. This last penetration step enabled the remote attacker to open a shell in the control system and modify a sensitive configuration file used to control the underlying power system relay. In the testbed, the configuration file modification was physically demonstrated by the turning off of a relay-controlled light bulb.

Finally, we illustrate the benefits of Seclius by comparing four different detection solutions applied to the same attack replayed three times against the testbed. First, we deployed the IDSes to monitor the critical assets *only*, i.e., those listed as the leaf nodes of the CT, and we do not use dependencies. Second, the intrusion detectors were configured to detect consequences for any (critical or noncritical) asset within the system, still without dependencies. Third, the detection scenario was identical to the previous one but Seclius was also deployed to consider dependencies. The last detection scenario was identical to the previous one but detection inaccuracies were injected by probabilistically discarding some alerts. Using expert judgment, false positive and negative rates were set to 8 and 5 percent for Samhain, and 6 and 3 percent for Zabbix, respectively. For scenarios 3 and 4, Seclius was adjusted such that whenever a CT node value changed (due to

Table 3.2: Case Study: Comparing Three Different Detection Scenarios

Detection Scenarios Comparison		Attack Steps				
		Initial	Web Server	Desktop	Gateway	Relay
Consequence Tree Nodes	CoreHTTP	(0.00, 0.00, 0.00, 0.03)	(1.00, 1.00, **1.00**, **0.94**)	(1.00, 1.00, 1.00, 0.94)	(1.00, 1.00, 1.00, 0.94)	(1.00, 1.00, 1.00, 0.94)
	Apache-DMZ	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)
	Clients-DB	(0.00, 0.00, 0.00, 0.00)	(0.00, 0.00, **0.04**, **0.03**)	(0.00, 0.00, **1.00**, **0.85**)	(0.00, 0.00, 1.00, 0.85)	(0.00, 0.00, 1.00, 0.85)
	Apache-Ctrl	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)
	Config-Files	(0.00, 0.00, 0.00, 0.00)	(0.00, 0.00, 0.00, 0.00)	(0.00, 0.00, 0.00, 0.00)	(0.00, 0.00, **0.40**, **0.37**)	(0.00, 0.00, 0.40, 0.37)
	MySQL	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)	(0.00, 0.00, 0.00, 0.03)
	DMZ	(0.00, 0.00, 0.00, 0.00)	(0.00, 0.00, 0.00, **0.02**)	(0.00, 0.00, 0.00, 0.02)	(0.00, 0.00, 0.00, 0.02)	(0.00, 0.00, 0.00, 0.02)
	Corporate	(0.00, 0.00, 0.00, 0.00)	(0.00, 0.00, **0.04**, **0.03**)	(0.00, 0.00, **1.00**, **0.85**)	(0.00, 0.00, 1.00, 0.85)	(0.00, 0.00, 1.00, 0.85)
	Control	(0.00, 0.00, 0.00, 0.07)	(0.00, 0.00, 0.00, 0.07)	(0.00, 0.00, 0.00, **0.06**)	(0.00, 0.00, **0.40**, **0.38**)	(0.00, 0.00, 0.40, 0.38)
	Organization	(0.00, 0.00, 0.00, 0.07)	(0.00, 0.00, **0.04**, **0.11**)	(0.00, 0.00, **1.00**, **0.87**)	(0.00, 0.00, 1.00, **0.92**)	(0.00, 0.00, 1.00, 0.92)
Ranked Alerts			1	3,2,1	3,2,1,4,5	3,2,1,4,5

the underlying IDS alerts), a *meta-alert* was triggered to notify the administrator.

As a result, if a critical asset was touched by the attacker, all four detection cases mentioned above would detect it. If a noncritical asset was touched, cases 2, 3, and 4 would detect it but the detectors in case 2 would have had no information regarding how much any of the critical assets might be affected, and therefore, would not have been able to distinguish among the noncritical assets being accessed. On the other hand, making use of the DG, Seclius calculated the probability that the critical assets would be affected even if none of them were directly accessed by the attacker.

Table 3.2 shows the results for all the four cases right after each step of the attack. Each value in the table represents the probability of a particular node being true in the CT (see Figure 3.8(a)). For clarity, the zero values are grayed out and the meta-alerts are highlighted with bold fonts. Initially, when the attacker was not inside the system, all the DG vertices were set to false, except the socket on the web servers in the DMZ network (i.e., the only entities interacting with the Internet and where potential attackers resided). As demonstrated in the first column of the table, all the nodes in the CT were initially zero. Once the web server was compromised, Seclius updated the relevant nodes in the Bayesian network, namely all nodes in the web server and the socket on the employee desktop, to true. The reason is that once a privilege domain was compromised, Seclius pessimistically marked all the nodes in that privilege domain as tainted, and triggered the corresponding meta-alerts. In our experiments, the web server buffer overflow exploitation and the CoreHTTP process crash were detected by Snort and Zabbix, respectively.

As shown in the second column of Table 3.2, the first attack step was detected in all the cases as the availability of the CoreHTTP was directly affected by the attacker. It is important to note that the probability of the MySQL files being affected on the corporate network (in cases 3 and 4) also

increased, although the file was not directly touched by the attacker. This increase makes sense because the attacker controlled the web server, and thus, could flow data to the MySQL files (i.e., those listed as critical, integrity-wise). As shown, from the point of view of cases 1 and 2, only the web server compromise was detected, and the database files were not affected. While assessing system security, Seclius took such potential adversarial flow under consideration, since it made use of the dependencies among the system components.

Then, the attacker installed the malware using a social engineering attack and obtained control over the corporate network machine. ClamAV was able to detect the malware as an illegitimate process and reported it. Within the newly controlled machine, the attacker replaced the `/usr/bin/mysql` executable with a downloaded executable `mysql` that, after being spawned, accessed the clients' database[3], listed as a critical asset in the CT. Later, the attacker's attempt to crack the root password was detected by Zabbix due to the attack's high CPU consumption. As shown in the third column of Table 3.2, the probability values in cases 1 and 2 were not affected as a result of the attack; however, Seclius pessimistically marked all the nodes in the compromised privilege domains as tainted, as processes within such domains were no longer trusted to follow their normal behavior.

Getting closer to the target, the attacker then got access to the VPN gateway. This step was detected by Snort, which had been configured to detect any connection initialized from the control network to the corporate network. Normally, the other direction was used in SCADA architectures to transfer the field sensory data into the corporate network. As there was no critical asset listed in the gateway, no node in the CT was affected in cases 1 and 2. However, as shown in the fourth column of Table 3.2, node probabilities were updated by Seclius as the attacker got closer to the assets in the control system (the Bayesian network vertices corresponding to the gateway were marked as true), increasing the probability of their security criteria being indirectly affected (according to the DG). Consequently, the attacker started scanning the control network to find his or her next target and this activity was detected by Snort.

Finally, using a legitimate service, the attacker remotely sent relay set points to the control

[3]The access to the clients' database files did not get detected even though the DazukoFS module had already been loaded and was monitoring accesses to the clients' database files, because as the clients' databases are frequently accessed by the legitimate users, accesses by the `mysql` process were configured to be white-listed.

system without compromising it. This action turned off the physical relay, representing a potential threat for a large-scale blackout in the power grid. More specifically, the set points were first stored in a configuration database file, which was then used by a cron job to send relevant commands to the power component. The modification of the configuration database was not detected, as it was accomplished through legitimate channels. We note that to limit the false positive rate, the legitimate channels had already been white-listed. Therefore, neither case 1 nor case 2 changed any value in the CT; however, from the previous step and using the captured DG, Seclius had updated the CT values. Consequently, once the attack was completed, from the point of view of cases 1 and 2, the value of the root node, i.e., organizational security, was still 0. On the other hand, Seclius estimated the root node value as 1.0 after the attack, which better represents reality.

The last row of the table shows the list of alerts ranked from high to low after every attack step. The indices are mapped to alerts as follows: 1) [Snort: web-server, buffer-overflow]; 2) [ClamAV: desktop, malware-installation]; 3) [Zabbix: desktop, passwd-bruteforce]; 4) [Snort: gateway, VPN-connection]; 5) [Snort: gateway, network-scan]. Alerts 2 and 3 flooded the screen, but here, for clarity, we map each one to a single index. After the attack's last step, Seclius ranked alert 3 the first, since it contributed the most to the organizational security compromise, i.e., CT's root node value. On the other hand, alert 5 was ranked the least important since its corresponding event was just a network reconnaissance that did not affect the CT's root value and would usually be less crucial than explicit privilege escalations or malicious consequences.

3.8 Related Work

Extensive research has been conducted over the past decade on the topics of system situational awareness and security metrics. In this section, we survey the related literature and discuss how Seclius compares to them.

Security metrics and evaluation techniques fall into two categories. First, static solutions, such as FLIPS [29], in which an IDS alert scoring value is hard-coded on each detection rule; the (alert, score) mappings are stored in a lookup table to be used later to prioritize alerts. The advantages of the static techniques are their simplicity and their rapidity. However, they suffer from a lack of

flexibility, mainly because they completely ignore the system configuration and scalability, since it is infeasible to predict all the alert combinations from IDSes in a large-scale network.

Second, there are dynamic methods, which are mostly based on attack graph analysis [47]. The main idea is to capture potential system vulnerabilities, and then extract all possible attack paths. The generated graph can be used to compute security metrics and assess the security strength of a network [66, 67]. The main issue with attack-graph-based techniques is that they require strong assumptions about attacker capabilities and vulnerabilities. While these approaches are important in planning for future attack scenarios, we take a different perspective by relying on past consequences, actual security requirements, and low-level system characteristics, such as file and process dependencies, instead of hypothetical attack paths. As a result, our method is defense-centric rather than attack-centric and does not suffer from the issues of unknown vulnerabilities and incomplete attack coverage.

Other important defense-centric approaches include M-Correlator [68] and FuzMet [69]. These techniques use manually filled knowledge bases of alert applicability, system configuration, or target importance to associate a context with each alert and to provide situational awareness accordingly. Seclius provides more practicality by minimizing and simplifying the required manual inputs via automatically learning the low-level system characteristics in order to evaluate precisely the extent to which alerts are affecting the critical components of the organization.

Such a damage assessment feature has previously been explored via file-tainting analysis for malware detection [70], for offline forensic analysis using backtracking [71] or forward-tracking techniques [72], and for online damage situational awareness [73]. In [73], information flow is tracked across multiple layers, namely at the instruction level and at the OS process level. Compared to our approach, this cross-layer technique is more precise, but requires implementation in a virtual environment and can cause important performance degradation.

3.9 Discussion

We discuss three limitations of the current version of Seclius and the potential solutions to address each of them.

First, as in any learning algorithm, it is not certain that the learned DG actually captures every dependency. One trivial solution would be to make sure that the learning phase is long enough to capture all the dependencies. Alternatively, an active learning algorithm could be used. For instance, the configuration files could be parsed to extract potential dependencies, or a mechanism to make sure all the program paths are exercised. Replacing passive learning with an active algorithm would require application-specific knowledge; however, it would help to accelerate the learning phase.

Second, the evaluated security value will be affected if some malicious events are missed by the intrusion detectors. Our main contribution in this paper is in showing how to make use of the system dependency graph and the security requirements to evaluate the security of any *given* state; in other words, we do not claim to have come up with a new intrusion detection technique. However, our tool, which makes use of Seclius to evaluate system security, takes under consideration the intrusion detection inaccuracies, i.e., false positive and negative rates, if provided. Additionally, security evaluation by Seclius is done based on the past consequences, which are easier to detect than exploitations. As a case in point, detecting that the web server is unavailable is usually simpler than determining the exploitation that caused the server crash.

Finally, as Seclius is an information flow-based metric, when the system has not yet been attacked, Seclius usually evaluates the system security to be close to absolute, but not completely secure. Because information often flows from external end points, where attackers potentially reside, to the critical assets even during the system's normal operational mode. A possible solution to this problem would be to normalize the evaluated security measure based on the measure of the non-compromised system.

3.10 Conclusion

We proposed *Seclius*, an online security evaluation framework that leverages dependencies between OS-level objects to measure the probability that critical assets have been directly or indirectly compromised. The different components of our framework addresses three important limitations faced by traditional security evaluation techniques. First, a consequence tree captures

the subjective security requirements and minimizes administrator input. Second, Seclius processes IDSes alerts online to measure actual attack consequences and does not rely on assumptions about attacker' behaviors or system vulnerabilities. Third, a dependency graph is combined with a taint tracking method to probabilistically evaluate the system-wide impact of locally detected intrusions as well as attacker privileges and security domains, without being constraints to a pre-defined attack path. Experiments showed that Seclius efficiently learns the system's normal behavior with only 6% overhead, and evaluates the system security within 10 seconds. Finally, we illustrated how Seclius can be applied in a control system environment to provide a comprehensive situational awareness solution to security administrators.

CHAPTER 4

MANAGING UNKNOWN EXPLOITATIONS

The main responsibility of an intrusion response system is to decide upon response and recovery actions given the current state of a system, which is estimated based on the alerts received from distributed intrusion detection systems. However, one of the major challenges in intrusion tolerance research is how to provide online protection against "zero-day" attacks [74, 75] that utilize previously unknown vulnerabilities. Because zero-day exploitations by definition have been previously unknown, they cannot usually be detected and reported by the IDS sensors to a response system. In this chapter, we discuss an information-theoretic iterative forensics solution to identify source vulnerabilities of zero-day attacks right after they occur within the system. Knowledge about source vulnerabilities helps response systems deploy relevant IDSes, and be notified if a similar attack happens again.

In general, detecting intrusions early enough for the response systems to do something about them can be a challenging and expensive endeavor. In particular, there are two main points to consider in effective attack detection: cost and coverage. For instance, in the case of detecting buffer overflows, there are relatively inexpensive techniques with imperfect coverage, e.g., address space randomization, and expensive methods with excellent coverage, e.g., strict bounds checking. However, it is difficult to find a technique that achieves both low cost and perfect coverage. The scope of a detection mechanism also impacts its coverage. Techniques that only monitor a single process (e.g., a Web server) may not be able to detect attacks such as SQL injection, because the attacks do not produce symptoms in the monitored process. Finally, as one broadens the range of vulnerability types to be detected, multiple types of detection mechanisms are often required, and the costs associated with each individual detector become an even more important factor.

While intrusion detection techniques exist for many types of software vulnerabilities, deploying all of them to catch the small number of vulnerabilities and exploits that might actually exist for a

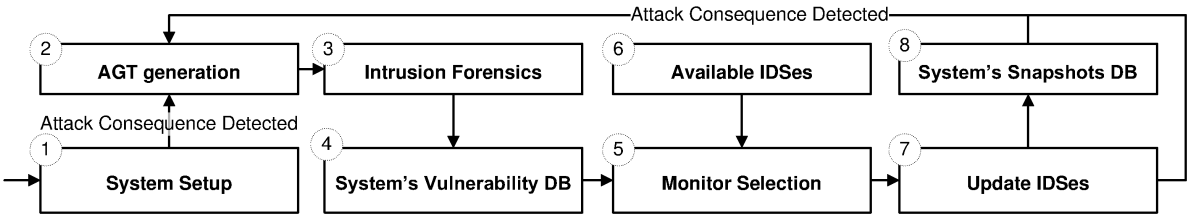


Figure 4.1: FloGuard Architecture

given system is often not cost-effective. Hence, in practice, IDSes are usually deployed to monitor the known vulnerabilities within a system. Consequently, unknown vulnerability exploitations may remain undetected and thus be overlooked by the intrusion response system. In this chapter, we describe an algorithm that response systems can use to manage unknown exploitations. It makes use of an online forensics analysis by dynamically turning IDSes on and off within the system. Specifically, we present FloGuard, an on-line intrusion forensics and on-demand detector selection framework that provides systems with the ability to deploy the right intrusion detectors dynamically in a cost-effective manner when the system is threatened by an exploit. FloGuard relies on often easy-to-detect symptoms of attacks, e.g., participation in a botnet or DDoS, and works backwards by iteratively deploying off-the-shelf detectors closer to the initial attack vector.

4.1 Architecture

We begin by describing FloGuard's architecture and its overall operation. Figure 4.1 shows a bird's-eye view of different components in FloGuard, their responsibilities, and how they are logically interconnected. FloGuard assumes a virtual machine environment and requires the target system to operate as a guest VM. FloGuard itself operates in the hypervisor/host OS of the VM environment to protect itself from attack and facilitate secure snapshotting facilities. Our prototype makes use of the Qemu [76] system emulator. The main inputs that FloGuard requires from a system administrator are a Vulnerability-Detector database (VulDB) and an initial set of "attack consequence detectors."

Attack consequence detectors are lightweight detectors that can operate continuously or periodically and detect the eventual symptoms of an intrusion and which cannot be disabled by an attack. Examples include hypervisor-based file-integrity or malware checkers (for known malicious

payloads), ISP-based in-network DDoS, BotNet, and worm detection mechanisms, or statistical anomaly detectors for detecting abnormal resource usage or network patterns. The output of an attack consequence detector is a process, port, or file that exhibits the symptoms of an attack. In our experiments, we use the Samhain file integrity checker [51] as the attack consequence detector.

The VulDB encodes information about the kinds of vulnerabilities that FloGuard is to guard against and the intrusion detection systems available to it. The database does not require knowledge of the specific vulnerabilities that may actually be present in the target system (which are unknown), but just the high-level types, e.g., buffer overflow or SQL injection, that are possible and for which detection and/or prevention mechanisms are available. Information about the intrusion detectors and prevention mechanisms includes their cost in terms of the additional CPU overhead and the vulnerability types that they can guard against, i.e., the detector coverage. Finally, the database may optionally include a mapping of processes in the target system to the kinds of vulnerabilities that may or may not occur in them (e.g., SQL injection cannot occur in an application that does not use a SQL database). Section 4.2 describes the contents of the VulDB in detail.

During normal operation, FloGuard turns on the consequence detectors and periodically collects incremental snapshots of the system. It also keeps an append-only log of all system calls[1] and privilege transitions made by processes in the system. When one of the attack consequence detectors produces an alert (i.e., called the detection point), FloGuard parses the system call logs from bottom to the top to generate a data flow graph like in BackTracker [77], and determines set of all potential attack sources (entry points) based on a reverse data flow analysis on the generated data flow graph similar to Taser [78]. For instance, an internet socket, from which there has been a data flow path to the detection point, is marked as a potential attack source. Once the set of possible attack sources is determined, the last system snapshot taken before all the potential attack sources is marked as the last clean snapshot, in which attack had not yet started, and is used by the algorithm for forensics purposes as discussed below.

FloGuard then uses the syscall logs from the clean snapshot to the detection point in order to produce an Attack Graph Template (AGT) that contains all possible information flows that have

[1] All the syslogs are sent to a secure backend through a uni-directional communication link.

occurred between different privilege domains in the system since the last clean snapshot[2]. These information flows may occur directly between processes, or through intermediate files, shared memory, pipes, or sockets. These information flows are a superset of any actual attack path, modeled as a series of privilege escalations through exploiting specific types of vulnerabilities, from a state in which an attacker has no access to the system, to a state detectable by the attack consequence detector. The AGT and its construction are discussed in detail in Section 4.3.

Subsequently, FloGuard invokes intrusion detectors to refine the AGT from possible attack paths into an actual attack path. This is done through an iterative forensics analysis process that uses the AGT node associated with the consequence detector alert as a starting point. It uses the outputs of the intrusion detectors (starting with the consequence detectors) to determine which paths in the AGT can be implicated or eliminated, and produces a refined AGT that is a subset of the original one. To do so, it relies on the detector coverage matrix from the VulDB and uses the algorithm described in Section 4.4. Then, FloGuard picks the best new set of detectors to deploy (in addition to the consequence detectors) that provide the best balance of cost and detection coverage over the refined AGT. The goal is to pick detectors that will best help refine the AGT even further if the same attack happens again, and ideally isolate a single attack path starting from the initial exploit to the detected attack consequence. Section 7.2.2 describes the algorithm used for this purpose.

After selecting a new set of detectors, FloGuard rolls the system back to a past clean snapshot, deploys the detectors, and waits for a repeat attempt of the attack. On the next alert, either from the consequence detectors or the newly deployed detectors, FloGuard reinvokes the forensics engine. If the same attack was repeated, it will be detected sooner than the previous one, and the actual detector outputs provide more evidence to help the forensics engine further refine the AGT. If the alert is due to a new attack that cannot be explained by the refined AGT, the forensics engine produces a new AGT refinement using the system's behavior during the attack and the actual detector outputs. In this manner, a repeated attack is detected successively closer to its exploit source until eventually, the detection point is close enough to the actual attack vector that a mitigation mechanism can then be used to block the attack. Thus, only the detectors related to vulnerabilities

[2]It is important to mention that kernel vulnerabilities are also addressed using this approach. Since the last possible exploitation by the attacher, giving him or her the highest privilege, is anyway the root escalation which is also logged, as logs are sent realtime to a backend server, and later used for forensics analysis.

that are present in the system and for which an exploit actually exists need to be deployed in the process.

There are several important points to mention. First as FloGuard makes use of securely logged syscalls, it does not rely on any knowledge of what the attacker may do in future. Second, we have currently prototype implementation of FloGuard for server systems (See Section 4.6) where an extra computer system could typically be provided to FloGluard for its offline forensics analysis right after the detection point without bringing the actual system down for long. Finally, regarding the attack types against which FloGaurd could be employed, FloGuard works based on logged past activities within the system; therefore, when being used against social engineering attacks, it can trace the attack path back to the executable which was downloaded during the social engineering phase of the attack.

Limitations. In particular, a limitation of our approach is that useful work is lost when the system is reverted back to a previous snapshot. However, this is not a fundamental limitation of the approach itself. If it is feasible to record a log of the target system's interactions with the external world (e.g., network connections and keyboard inputs) since the last snapshot, the communication log can be replayed to avoid any lost work. Such logging has been shown to be feasible in some contexts, e.g., Taser [78]. An additional advantage is that if the communications log also contains the attack vector, consecutive replays can be used to make quick progress on the attack forensics without having to wait for the next attack. We have used this technique successfully in our experiments.

As any other forensics solutions, FloGuard starts its analysis to identify the exploited unknown vulnerabilities after an attack happens; therefore, it does not protect the system against exfiltration attacks.

Finally, FloGuard does not detect any new kinds of exploits by itself; its detection coverage is ultimately limited by the set of detectors that are available to it. Nevertheless, we believe that by deploying expensive detectors on-demand only when they are truly needed, FloGuard enables many expensive detection mechanisms to be used that would otherwise be impractical. Thus, the overall coverage it achieves is superior to what would be practically feasible otherwise. Furthermore, it allows for easy incorporation of additional detector types when they become available.

4.2 Vulnerability-Detector DB

The vulnerability-detector database encodes all the domain knowledge about vulnerability types, detection mechanisms, and the applications in the system that are used by the forensics engine to make its detection decisions. In this section, we describe each of the components in this database using the target systems used in our experiments (Section 4.6) as examples. The target systems used standard Ubuntu 9.04 distributions with a number of different services enabled in each scenario including two LAMP-based (Linux-Apache-MySQL-PHP) user applications: 1) RoomPHPPlanning [79], an application used for making room reservations, and 2) eVision [80], a content management system. Several off-the-shelf intrusion detection systems were also employed, as described below.

Vulnerability Types. In general, vulnerabilities are software flaws that are used by an adversary in the penetration step of an attack to obtain a privilege domain on a machine. The *vulnerability types* set in the VulDB includes all "possible" types of vulnerabilities that could potentially exist in different parts of the target system. A vulnerability type represents general vulnerability classes, e.g., buffer overflow, that encompass all target systems, without mentioning their context. It does not represent a *specific* vulnerability in the system, e.g., the vulnerable application's name and the exact location in the application's code. It is possible and quite likely that an individual target system, in fact, is not susceptible to most of the vulnerability types. Furthermore, even if a particular type of vulnerability is present in a system, it is likely that only one or a small number of processes will actually be vulnerable to it.

In addition, the VulDB also contains a target-system-specific mapping of vulnerabilities to processes in the system. The mapping can be positive or negative (e.g., the eVision PHP application could be vulnerable to a SQL injection attack while the sshd daemon cannot). Since the system's actual vulnerabilities are unknown, these mappings represent *potential* vulnerabilities, not known ones. The mappings are optional, and FloGuard assumes that every process can be vulnerable to every vulnerability type if it is missing. But if present, it significantly reduces the number of escalation paths that FloGuard's forensics algorithm must consider. Table 4.1 shows the types of vulnerabilities currently used by our prototype, and the processes in our experimental system that could possibly be vulnerable to each type.

Table 4.1: Vulnerability Categorization

Symbol:Vulnerability Type	**Processes**
Memory safety violations	
Buff:Buffer overflows	PHP, sshd, Http, DB
DngPtr:Dangling pointers	PHP, sshd, Http, DB
Input validation errors	
FmtStr:Format string bugs	PHP, sshd, Http, DB
ShlMC:Shell metachar bugs	
SQLIn:SQL injection	eVision, RoomPHPlan
CodIn:Code injection	eVision, RoomPHPlan
DirTrv:Directory traversal	eVision, RoomPHPlan
CSS:Cross-site scripting	
HttpHdr:HTTP header injection	
HttpRsp:HTTP response splitting	
Race conditions	
TcTu:Time-of-check-time-of-use	kernel
SymRc:Symlink races	kernel
Privilege-confusion	
CSFor:Cross-site request forgery	
ClkJk:Clickjacking	
FTPBnc:FTP bounce attack	
User interface failures	
WrnFtg:Warning fatigue	
BlmVic:Blaming the Victim	
Race:Race Conditions	DB
Weak authentication	
PwdDic:Password Dictionary	sshd, OSAuth
Encypt:Encryption Bruteforce	sshd, OSAuth

Detection Mechanisms. The second element in the VulDB is the set of intrusion detection systems (IDSes) that are available to FloGuard to monitor different parts of the target system. Different kinds of detection systems that together cover as many different vulnerability types as possible are preferable. In general, any detectors that produce intrusion alerts associated with a particular process, file, or communication port are supported. For each detection mechanism, FloGuard also requires a relative cost measure associated with the detector when it is deployed. The cost measure is only used to compare one intrusion detector against another, so as long as a uniform measure is used for all the detectors, it is not necessary for the cost to represent any specific performance or overhead measure. In our experiments, we used the additional CPU overhead imposed by each detector as the cost measure. If a generic cost measure is used, the IDS descriptions are system-agnostic, and can be reused.

Table 4.3 provides examples of the different types of intrusion detection techniques that FloGuard supports. Additionally, for each detection mechanism, the table provides its average cost, i.e., performance overhead, and an example IDS that specifically uses that type of detection mechanism. We used a subset of these detectors in our experiments; a detailed description of their settings and cost measures is presented in Section 4.6.

Ultimately, FloGuard's main objective is to protect a system from attack. While each detector can be converted into a rudimentary intrusion mitigator (by restarting the target process as soon as the detector detects an intrusion), specialized mitigation mechanisms that may not detect attacks but can block them can also be included in the VulDB database. For example, a "disable account" action cannot detect attacks per-se, but can block password attacks. Similarly, a "Enable Sweeper" action can enable the Sweeper input attack-signature generator and filter on a target process. The mapping between such attack mitigators and vulnerability types is made in the same way as for regular detection mechanisms using the detector-capability matrix described below. In this manner, the monitor selection algorithm (Section 7.2.2) can deal with mitigators and detectors in the same framework. If a process/file is known to be along the attack path with high probability and a low cost mitigator is available, the optimization algorithm will choose it instead of a detector.

Detector-Capability Matrix. The final element in the VulDB is the detector-capability matrix, which indicates the ability of a given IDS to detect various vulnerability types. The matrix is defined over the Cartesian product of the vulnerability type set and the set of IDSes, and shows

Table 4.2: The Detector-Capability Matrix

	Buff	DngPtr	FmtStr	ShlMC	SQLIn	CodIn	DirTrv	CSS	HttpHdr	HttpRsp	TcTu	SymRc	CSFor	ClkJk	FTPBnc	WrnFtg	BlmVic	Race	PwdDic	Encrypt
Tnt	H	M	H	M	C	L	H	C	M	M	L	L	H	H	H	N	N	N	N	N
FW	L	N	L	N	N	L	N	N	L	L	N	N	L	L	L	N	N	N	M	M
Snrt	M	N	M	M	M	M	M	N	M	M	N	N	N	N	M	N	N	N	H	H
App	H	L	H	H	H	H	H	L	C	C	N	N	N	N	H	N	N	N	H	H
ClSt	C	M	H	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
CtFl	C	H	H	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N	N
DtFl	L	L	L	M	C	L	H	C	M	M	L	L	H	H	H	N	N	N	N	N
AV	N	N	N	N	N	M	N	N	N	N	H	H	N	N	N	L	L	L	N	N
Hst	L	L	L	N	N	H	N	N	N	N	H	H	N	N	H	M	M	L	N	N
Stat	M	M	L	N	N	L	N	N	N	N	N	N	N	N	H	N	N	N	H	H

how likely it is that each IDS could detect an exploitation of a specific vulnerability type. Table 4.2 illustrates a sample detection capability matrix for the different IDS and vulnerability types that were discussed earlier. The notations used in the table are as follows. C (or N) means that the detection technique can always (or never) detect the exploit, while L/M/H means that it can detect a few/medium/large instances of the exploit. C/N detectors are use to confirm or eliminate attack paths, while L/M/H values are only used to prioritize detector deployment. So, an incorrect choice between L, M, or H (which can be hard to quantify accurately) will only impact the efficiency of the forensics, not its correctness. Some of the detection systems, e.g., OSSEC [81], monitor system-wide events, whereas others only observe one or a set of processes in the system, e.g., Memcheck [82]. Entries in Table 4.2 report detection capabilities assuming that the vulnerability exploitation context is being observed by the corresponding detection system. The detection capability matrix is later employed by the forensics and detector selection algorithms in deciding on the minimum-cost set of intrusion detection systems with maximum exploit detection capability.

4.3 Attack Graph Templates

Generally, every cyber attack scenario (path) consists of a number of subsequent exploits, which are basically atomic malicious actions, including NOP (no-operation). In other words, the adversary, with initially no access to the system, can launch various attacks by taking a sequence

Table 4.3: Detection Algorithm Categorization

Detection Policy	**Symbol**: **Mechanism**	**Cost**	**Detector**
Information flow analysis	Tnt: Taint tracking	Very High	TEMU [53]
Input investigation	FW: Feature-based packet monitoring	Very Low	Firewalls [83]
	Snrt: Content-based packet monitoring (stateless)	Medium	Snort [56]
	App: Application-based IDS (stateful)	Medium	Secerno [84]
Execution monitoring	ClSt: Control Violation: call stack monitoring	High	CS [85]
	CtFl: Control Violation: control integrity monitoring	High	CFI [86]
	DtFl: Data Violation: data flow monitoring	Very High	MemCheck [82]
Consequence detection	AV: Malicious code: executable integrity checking	Low	ClamAV [58]
	Hst: Host-based detection systems	Low	Samhain [51]
	Stat: Statistical anomaly-based	Low	Zabbix [59]

of exploit actions to achieve the privilege domain required for his or her malicious goal, e.g., modifying a sensitive system file. In particular, each exploit action is designed to take advantage of a specific vulnerability to breach a defensive security line in the system. An example exploit would be to smash the stack and inject the shell code while the vulnerable application is running. Throughout this chapter, exploits are represented as (application, vulnerability type) pairs in which the first and second elements denote, respectively, the vulnerable application and the vulnerability type in the application that is being exploited.

Traditionally, an attack graph for a computer system represents a collection of known penetration scenarios according to a set of known vulnerabilities in the system [87]. In this section, we present the attack graph template (AGT), i.e., an extended attack graph, which is intended to cover all "possible" *attack types*. As an example, the attack graph template for a web server addresses the possibility that the server application might be vulnerable to buffer overflow attacks, even if there is no such known vulnerability in the application. The attack graph template is a state-based graph in which each state is defined to be the set of the attacker's privileges in that state. State transitions in AGT (each mapped to a malicious action, i.e., vulnerability exploitation) represent privilege escalations. Formally, the AGT encodes all possible attack paths from the initial state, in which the attacker has no control over any privilege domain, to the goal state, in which the attacker has achieved the required privilege domains to accomplish his or her malicious goals. Additionally, the monotonicity property [88] is assumed; in other words, an attacker never backtracks, and hence does not need to relinquish privileges he has already gained.

In general, unknown vulnerability exploitations in a given application could be in any part of the

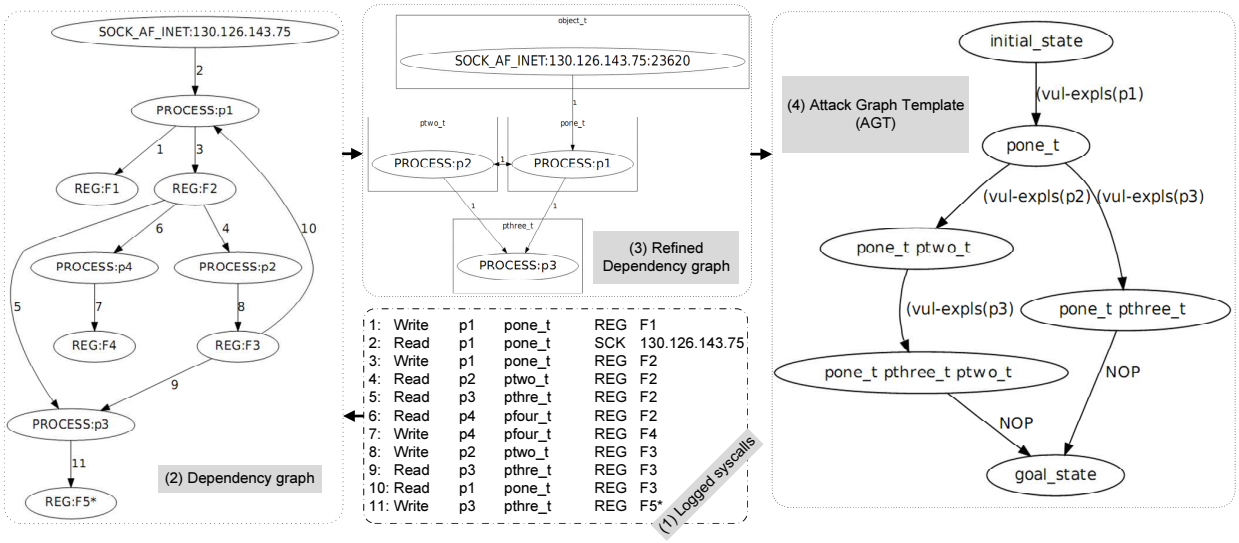


Figure 4.2: Attack-Graph-Template Generation for a Sample System

application code; however, almost all of them are of known *types* such as buffer overflow or SQL injection. Furthermore, as each IDS can detect certain *types* of exploits, generating AGTs that consider all possible vulnerability exploitations in applications allows FloGuard to determine which state transitions in an AGT get detected if a particular set of IDSes are deployed. Additionally, as the AGT is designed to consider all exploit types which constitute a finite set, the order (#states) and size (#state transitions) of AGT are finite, and often manageable in practice (as shown in the experiments).

Automatic AGT Generation. Generating attack graph templates manually for real-world systems can get very tedious. Furthermore, the AGT needs to be continuously updated by the admins to catch the system's internal dynamic behavior over time. Here, we present an algorithm to automatically generate the AGT for the system within a past time interval (i.e., from a time instant before the attack to the detection point) by observing the system's behavior, i.e., system call logs.

To automatically generate the AGT, FloGuard is particularly interested in the syscalls that cause data dependencies among the OS-level objects[3]. A dependency relationship is specified by three parts: a source object, a sink object, and an occurrence time[4]. For example, the reading of a file

[3]Throught the chapter, we use the term OS-level objects for processes, regular files, directories, symbolic links, character devices, block devices, FIFO pipes, and all thirty five types of the sockets including internet sockets, i.e., `AF_INET`, interchangeably.

[4]We also log the security context of the objects. As a case in point, on a system with the SE-Linux operating system, the web server directory is associated with the security type `httpd_sys_content_t`. As mentioned later, that

by a process causes that process (sink) depend on that file (source). We classify dependency-causing events based on the source and sink objects for the dependency they induce: process-process, process-file, and process-filename. The first category of events are those for which one process directly affects the execution of another process. One process can affect another directly by creating it, sharing memory with it, or signaling it. The second category of events are those for which a process affects or is affected by data or attributes associated with a file. System calls like write and writev cause a process-to-file dependency, whereas syscalls like read, readv cause a file-to-process dependency. The third category of events cause a process to affect or be affected by a filename object. Any syscall that includes a filename argument, e.g., open, causes a filename-to-process dependency, as the return value of the syscall depends on the existence of that filename in the file system directory tree.

Given a detection point and the syscall logs, the dependency graph is generated using an algorithm similar to BackTracker [77]. Event logs are parsed and analyzed line by line from the beginning to the detection point; their corresponding source and sink objects are created or updated; and a directed edge, labeled with the event's occurrence time, is created between those nodes. Number of directed edges between every node pair denotes how many times the corresponding syscall was called during the execution.

Having generated the dependency graph, we run two optimizations before converting it to AGT. First, using time-sensitive backward reachability analysis, we try to eliminate irrelevant vertices/edges that do not causally affect the state of the detection point [89]. Starting from the detection point, we traverse the directed graph in chronologically reverse direction (considering the time tags on the edges) and mark the encountered nodes until we get to any attack entry point, such as the keyboard or a network socket. Consequently, none of the unmarked nodes are on any path between any entry point and the detection point (they cannot causally affect the detection point); hence, the unmarked nodes in the graph are safely removed, leading to a significantly smaller graph in practice. Then, we create a *dummy* process-node (representing all the attackers) and connect it to all the entry point nodes in the graph; the time labels on those edges are assigned to be less than the minimum of all edges in the original graph. Second, by calculating transitive closure of the graph between process pairs, we get rid of all the non-process nodes; any pair of

information is used when converting a dependency graph to its corresponding attack graph template.

processes get connected if there is a causal directed path [89] between them consisting of only non-process nodes. For instance, the indirect dependency path (between two processes with a file in the middle) $p_1 \rightarrow f_1 \rightarrow p_2$ is converted to $(p_1 \rightarrow p_2)$ labeled with the first time the file was read by p_2. After all process pairs have been considered, non-process nodes are safely removed, leading to a dependency graph consisting of only processes.

Finally, the refined dependency graph is used to generate the AGT. The idea is to consider any edge in the dependency graph a potential vulnerability exploitation by the attacker to obtain the privilege of the process directed to by the edge. Vulnerabilities are exploited by attackers to improve their privileges on the system; therefore, it is reasonable to consider only the edges that connect two processes with different privileges (security types). However, edges connecting process pairs of the same privilege are not completely ignored; instead, as they provide useful information about how processes within a privilege domain interacted with each other during the execution, we use those edges to avoid including impossible exploitations, based on inter-process communications, in the final AGT.

To convert the dependency graph to AGT, we traverse the dependency graph and concurrently update the AGT. Starting from the dummy process-node in the dependency graph, we create the initial state of AGT with an empty privilege set. We run a causal depth-first search (DFS), i.e., with increasing time tags on the edges of the paths being traversed. While DFS is recursively traversing the dependency graph, it keeps track of the current state in the AGT, i.e., the set of privileges already gained through the path traversed so far by DFS. When DFS traverses a dependency graph edge which crosses over privilege domains, a state transition in AGT happens if the current state in AGT does not include the privilege domain of the process directed by the edge. The transition is between the current state and the state[5] that includes exactly the same privilege set as the current state plus the privilege of the process directed by the dependent graph edge. In fact, between those two states in the AGT, more than one transition edge might be created depending on how many vulnerability types could potentially exist in the process, according to the VulDB.

Once the depth-first search is over, AGT is generated such that all its terminal states include the privilege domain of the process that had caused the detection point event during the attack. The last step would be to add a *goal state* to the AGT and connect all the terminal states to it using NOP

[5]This state is created if it has not been created before.

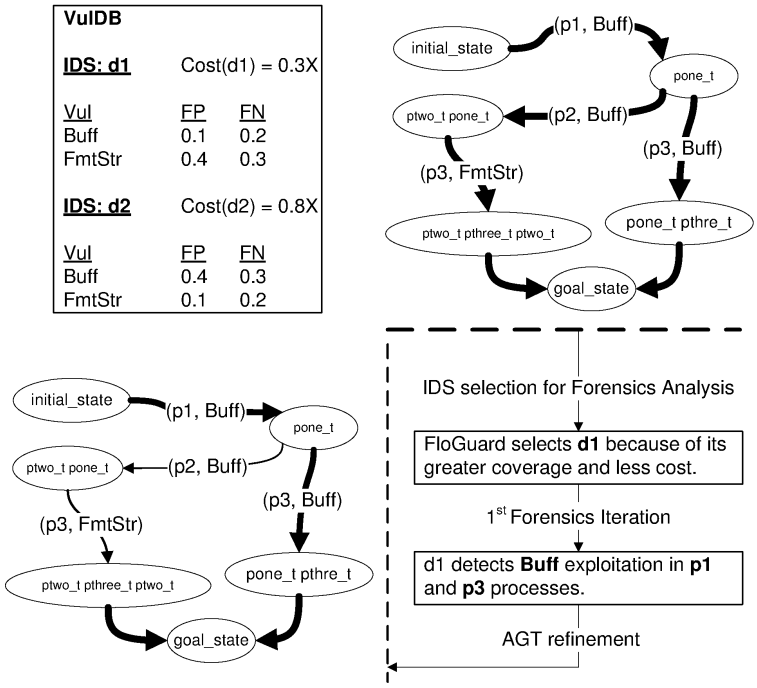


Figure 4.3: A Sample AGT Refinement During a Forensics Iteration

(No-OPeration) edges, meaning that no action is needed to make the transition. Now the AGT is complete; however at this point, the path traversed by the attacker is not known, and the forensics analysis (Section 4.4) tries to identify the exact path.

Figure 4.2 shows an example of how the AGT for a sample system is automatically generated. Area 1 in the figure shows the syscalls logged during the attack. Each log line consists of the time, the caller process, its privilege, the object type (e.g., regular file or socket), and a filename or socket address. Line 11 is assumed to be reported as the detection point. The corresponding dependency graph is shown in area 2 of the figure. Numbers tagged on the edges show occurrence times of the syscalls. After the reachability analysis based on the detection point (`F*`), the graph is demonstrated in area 3 of Figure 4.2. The produced AGT (as shown in area 4 of the figure) enumerates all possible paths from the initial state to the goal state in which the attacker has the necessary privilege (`pthre_t`) to accomplish the detection point event. It is here assumed that the attacker has no physical access to the machine, and hence, the attacks are launched remotely.

4.4 Intrusion Forensics

As mentioned above, FloGuard initially turns on the attack consequence detectors, and the system starts its normal operational mode. Deploying the consequence detectors guarantees that most of the attack paths in the system are eventually detected once the attacker, having gained the necessary privileges, perpetrates his or her malicious goal action causing a consequence; we call this the *detection point* event.

In this section, we present an automated intrusion forensics analysis algorithm that analyzes past attacks. More specifically, once an attack is detected at one detection point, first its corresponding AGT is generated; then, the forensics analysis attempts to figure out the exact vulnerability exploitation (privilege escalation) sequence through which the attacker went from the initial state of the AGT to the goal state. The forensics analysis by FloGuard requires two logging subroutines. First, we assume that an incremental snapshot of the system is taken periodically, e.g., once a day; therefore, we could go back to any time instant in the past that coincides with one of the snapshot times. Second, syscalls are being logged and stored in a secure back-end storage device while the system is operating. That enables FloGuard to generate the AGT for any past time interval.

In particular, to figure out the attack path actually taken by the attacker, FloGuard employs an iterative forensics algorithm; it restores a clean system snapshot and waits for attackers to launch similar attacks (exploiting the same vulnerability) several times. During each iteration, it deploys a different IDS to gather further evidence regarding the attack. The deployed IDSes are chosen based on their cost, detection capabilities, and the generated AGT. In particular, FloGuard chooses the intrusion detector d^* for each forensics iteration using the following equation: $d^* \leftarrow \arg\max_{d \in D}\{Coverage(AGT, d)/Cost(d)\}$, where D is the set of available IDSes; $Coverage(AGT, d)$ denotes the expected number of already-unmonitored transitions in AGT that are monitored by d. Using VulDB, FloGuard knows what vulnerability exploitations (state transitions) each IDS can detect; therefore, after each iteration, it can prune the AGT based on the deployed IDS and its alerts during the attack. More specifically, the detected vulnerability exploitations are marked, and the rest (the vulnerabilities whose exploitations did not get reported, while being monitored, by the deployed IDS) can safely be removed from the AGT. The refined AGT is used to choose the IDS for the next iteration (attack repetition). FloGuard continues the

forensics iterations until the refined AGT consists of one marked path from its initial state to the goal state. The marked path is the actual path traversed during the attack.

In practice, detection systems are not always accurate, and sometimes either produce false positives or miss some misbehaviors (false negatives). FloGuard takes such inaccuracies into account using their corresponding rates provided in VulDB[6] (otherwise, a default value is used), and defining the edge weights w^e (which are all initially set to 1). In particular, if an IDS d reports a vulnerability exploitation e during a forensics iteration i, the corresponding edge (state transition) in AGT is not completely removed; instead, its weight is updated using the equation $w_i^e \leftarrow w_{i-1}^e \times [1 - FPR(d,e)]$, according to its previous weight and the false positive rate (FPR) of the IDS d in monitoring the exploit e. Similarly in case the IDS does not report any incident, the weight is updated with consideration of the IDS's false negative rate using the equation $w_i^e \leftarrow w_{i-1}^e \times FNR(d,e)$. Note that knowledge regarding the false positive rates in FloGuard is more critical than the false negative rates; lack of knowledge about the false negatives would make FloGuard choose the IDSes somewhat blindly, increasing overall overhead of the forensics, whereas FloGuard may, at the end, report an attack path incorrectly if false positive rates are completely ignored. Figure 4.3 demonstrates how a sample AGT is pruned during two forensics iterations; edge weights, in the figure, are represented by line thicknesses.

Essentially, to provide a precise automated forensics analysis, FloGuard must traverse the time dimension back and forth. FloGuard's implementation provides two different solutions, which are conceptually identical. 1) If the infrastructure supports system-wide restore/replay, FloGuard uses it to restore the past snapshot and replay the whole system several times, running the same forensics analysis as mentioned above, until the exact attack path in AGT is identified. 2) If the system-wide restore/replay is not supported, as described in this section, instead of making use of a restore/replay mechanism, FloGuard waits for the attacker to repeat the attack in the future. Note that in the second situation, the attacker can possibly cause more damage to the system, as he or she actually repeats the attack several times until FloGuard identifies the attack path and blocks future similar attacks. In this chapter, we focused on the second situation only due to the space

[6]Before the calculations, the qualitative values (as provided by VulDB) are mapped to their corresponding statically defined crisp values as follows. N and C are mapped to 0 and 1, respectively. And, L, M, and H are, respectively, mapped to 0.25, 0.5, and 0.75.

limit; however, the main concept is the same for both solutions.

4.5 Monitor Selection

Once the forensics analysis identifies the attack path in the AGT, FloGuard chooses the cost-optimal set of IDSes, as mitigation mechanisms, (unless it is statically defined in VulDB for the identified vulnerability exploitation, as discussed in Section 4.2) to deploy in the system permanently until the administrators install suitable patches. In particular, FloGuard selects and deploys a set of intrusion detectors that cooperatively detect and block exploitations of the *known* vulnerabilities in the system.

The monitor selection algorithm works on an AGT that consists of only known vulnerability exploitations (known state transitions) that have been identified at some point by the forensics analysis[7]. To select the intrusion detectors, FloGuard considers the trade-off between the overall detection cost of each IDS set vs. the cost of damage by the attackers due to the exploitations not monitored by that set. Formally, FloGuard uses the following equation to decide upon the subset of detection systems to handle known vulnerabilities:

$$D_k^* = \arg\min_{D_i \subseteq D} [\arg\max_{AP \in AGT} C(AP, D_i)]^8 \tag{4.1}$$

where AP represents an attack path (sequence of exploits) from the initial state to the goal state in AGT. D is the set of available intrusion detection systems. The $C(AP, D_i)$ function denotes the overall cost if the system operates with intrusion detectors $d \in D_i$ turned on. In particular, the overall cost is determined through consideration of two factors: 1) damage cost, which is caused by the attacker trying to get from the initial to the goal state through the attack path AP; and 2) detection cost, e.g., performace overhead caused by the intrusion detectors that are deployed and running. Depending on the exploits in AP and the detection capability of the intrusion detectors in D_i, AP might, but would not necessarily, be cut at some point between the initial and goal state by

[7] As discussed earlier, the attack-graph template that includes only known vulnerability exploitations is equivalent to the traditional attack graph formalism.

[8] The equation basically picks the subset of IDSes, which minimze the maximum possible cost caused (according to the AGT's structure) if the system operates with that IDS subset deployed.

one of the detectors. Formally, the C function is defined as follows:

$$C(AP, D_i) = \left\{ \sum_{e \in AP} \left[Depth(AP_e). \prod_{e' \in AP_e} \left(w^{e'} . \min_{d \in D_i} [FNR(d, e')] \right) \right] \right\} \times \sum_{d \in D_i} Cost(d) \qquad (4.2)$$

where the first term of the C function corresponds to the damage to the system that the attacker causes before he or she gets caught by any of the deployed intrusion detectors D_i. Briefly, we used the detection latency from the initial exploit node to the detection point as the damage cost, since it determines how much rollback (and data loss) is needed. More formally, the first term in Equation (4.2) formulates the expected depth of penetration (number of subsequent vulnerability exploitations) the attacker can cause following the attack path AP without being detected by any of the deployed intrusion detectors D_i; to do so, the algorithm considers penetration depth of each subattack AP_e (defined as the subpath of AP from the initial state to e) and the likelihood of it not being detected. The penetration depth for a subattack AP_e, denoted by $Depth$, is defined as the length of the path from the initial state to e through the attack path AP. The probability that the attack at a specific step AP_e is not yet detected is calculated by considering 1) the weights on each exploit e' in the subpath AP_e that have been updated through the forensics iterations; and 2) the false negative rates (denoted by FNR) of the deployed IDSes (more specifically, the IDS with the least false negative rate) regarding each exploit e' in the subpath AP_e The second term in Equation (4.2) represents overall detection cost by the deployed intrusion detectors, which is defined as the performance penalty measured as the additional amount of time required to process a fixed amount of user workload. Consequently, Equation (4.1) solves the abovementioned tradeoff and selects the IDS set, which minimizes the overall cost according to the AGT's structure.

4.6 Evaluation

We implemented FloGuard and evaluated it on different attack scenarios against four applications each with a specific vulnerability. The experimented vulnerability exploitations include buffer overflow exploitation, SQL injection vulnerability, PHP remote code execution, and password attacks.

Experimentation Setup. The experiments were conducted on a computer system with 2.20

GHz AMD Athlon ™64 Processor 3700+ CPU, 1 MB of cache, and 2.0 GB of RAM. The host operating system (OS) running on the machine was Ubuntu 9.04 with a Linux 2.6.22 kernel. We changed some of the default configuration files in the OS, namely `php.ini`, in order to make the system vulnerable to various attacks. The system studied was run in a virtual machine over the Qemu [76] system emulator, accelerated by the Kqemu kernel module. In particular, the production system included a web server with several PHP applications, including eVision CMS, which is a web content management system, and RoomPHPlanning, which is a web scheduling application. Furthermore, the applications could connect to a MySQL database, and the trusted remote clients made use of SSH to obtain access to the system. Using Qemu, we could implement our periodic snapshot mechanism and do a restoration whenever a type of misbehavior is reported by one of the intrusion detection systems.

Throughout the evaluation section, we make use of a set of IDSes that fall into the following categories. To identify and block malicious network packets, we used Snort [56], which is a fairly lightweight, network-based detection system that uses known malware signatures. In our experiments, we ran Snort in fast-alerting detection mode with 74 different rule-sets. We made use of LibSafe [90] to report malicious use of library functions in C/C++ applications. Most of the attacks include an egg download step in which the attacker either downloads malware to the target machine or uses social engineering techniques to make a legitimate user do so. To detect malware and virus downloads, we employed the ClamAV antivirus [58], which is an open-source toolkit for Unix machines. In particular, we used Sigtool in ClamAV to update the virus signatures and Clamscan for scanning downloaded suspicious files. Also, we inserted one additional virus signature for the malicious file uploaded to the server during the remote code execution attack (see next section). To detect attackers' local malicious actions on file system objects, such as sensitive file modification, we used Samhain [51], which is a host-based detection system that is essentially a file integrity checker. We employed Zabbix [59], i.e., a statistical anomaly-based detection system, to detect other anomalous local activities, such as local DoS or brute-force attacks that intensively consume system resources. To detect general memory access violations, we employed the Memcheck tool in the Valgrind suite [82], which is a heavyweight, dynamic instrumentation and memory access violation tool set. In particular, we ran Valgrind under the target application with the `--leak-check` option enabled. The most comprehensive and expensive detection system that we used in our eval-

uations was a modified version of the TEMU [53] system-wide taint-tracking engine, which runs below the virtual machine on the host OS. More specifically, using TEMU, one can mark some input data, such as keyboard keys and network interfaces, as tainted, and then TEMU will track the information flow and store the executed instructions in a trace file on the host OS. Tomake use of TEMU in our experiments, we had to improve its capability to produce higher-level information (not only instructions) regarding file-system objects, such as files that are being dynamically tainted. In order to do that, we improved its implementation by using The Sleuth Kit (TSK) [54] to read the file system in the virtual machine's image file; this enabled us to dynamically translate disk-level tainted addresses, which are generated by TEMU, to file system object names, such as file names and their absolute addresses.

In our evaluations, all activities on the system were done remotely; we assumed that no client modified the system through physical access to the machine. Therefore, taking a snapshot of the virtual machine at a time point and logging all the network traffic since then, one could later restore and replay the whole system completely several times, in order to do iterative intrusion forensics as described earlier in the chapter. To consider users with physical access to the machine, we also had to capture and log inputs from devices such as the keyboard and mouse. We employed Qemu to take snapshots of the whole system periodically while the production system was under its normal operational mode, i.e., receiving and responding to client queries. To replay the whole system again, Qemu was used to restore the stored snapshot, and Wireplay [91], running on the host OS, was employed to replay the pcap file, i.e., network traffic captured by Tcpdump, and feed the packets into the virtual machine.

Services and Vulnerabilities. Next, we describe the vulnerabilities and the affected services in our experiments.

SQL injection is a code injection technique that exploits an input validation vulnerability occurring in the database layer of an application. According to CVE-2009-4669 [92], multiple SQL injection vulnerabilities in RoomPHPlanning 1.6 allow attackers to execute arbitrary SQL commands 1) via the loginus parameter to `Login.php` or 2) via the `Old Password` field to `changepwd.php`, and allow 3) authenticated administrators to execute arbitrary SQL commands via the `id` parameter to `admin/userform.php`.

Buffer overflow is an anomaly in which a process writes more data, e.g., malicious shell code,

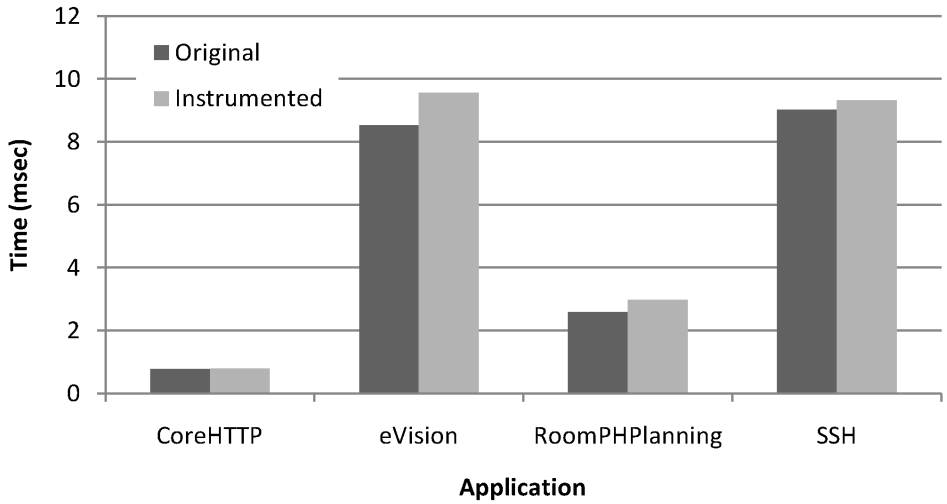


Figure 4.4: Performance Overhead of FloGuard's Instrumentation

to a stack buffer than the buffer's size. The extra data overwrite adjacent memory that might be program flow control data, e.g., a return address. Based on CVE-2007-4060 [92], an off-by-one error in the `http.c` in the CoreHTTP 0.5.3.1 web server allows remote attackers to execute arbitrary code via an intelligently handcrafted HTTP request.

PHP remote code execution, mostly via file syscalls, allows attackers to inject and execute machine code in the machine. According to CVE-2008-6551, multiple directory traversal vulnerabilities in the e-Vision 2.0 content management system allow attackers to include and execute arbitrary PHP files.

The weak password vulnerability arises when a user account with a weak password, e.g., "server-root," exists on a host that provides a login service to other users. Our production system includes accounts with weak passwords that open up the opportunity to make use of password-cracking software tools, such as John the Ripper [93], to get control over the system. As John the Ripper needs to have local access to password hashes, for some of the experiments we implemented a password brute-force script in Perl that reads a text file storing a large number of passwords and try them subsequently against a remote system.

Instrumentation Overhead. As mentioned earlier in the chapter, the syscall interception component of FloGuard is always deployed in the system and intercepts and logs the syscalls between the snapshots while the system is operating. We have implemented the syscall interception as a loadable kernel module; however recently, there have been other approaches like [94] proposed

intercept system calls from within the hypervisor using the previously-generated signatures of the process memory images, but their approach also assumes the root domain is tamper-proof since the attacker with the root access can modify kernel data structures, and consequently, make the signatures useless. We configured the `/etc/syslog.conf` file such that all the syslogs are directly sent through a uni-directional link to a secure backend machine; therefore, all the system calls logged before the attacher gets access to the root domain are trusted. Our implementation currently works with the Linux kernels up to 2.6.32-22; however for recent kernels, before replacing the syscall table entries with our wrapper syscall function pointers, we had to automatically find the address of `sys_call_table` (as its address is not exported anymore), and furthermore, we had to make the corresponding memory pages writable as, for security purposes, the syscall table in recent Linux kernels is read-only. Once deployed, the syscall interception module tries to extract and log as much information as possible about the calling process, such as its PID and security context, and the arguments to the syscall, such as filepaths and inode numbers for a given file descriptor.

In the FloGuard framework, as the syscall inteception module is always loaded while the system is operating, it is important for its overhead to the overall system throughput to be low. Here, we have measured how throughput of the different applications in the system are affected by FloGuard's permanent instrumentation, i.e., insertion of the syscall interception module. The results are demonstrated in Figure 4.4. For the first three applications, the results denote the average response time for subsequent requests that were static HTML page requests for the CoreHTTP web server, and PHP page requests for the eVision and RoomPHPlanning web applications. The results for SSH show how the time required to transfer 100 KB of data from the server is affected by the FloGuard instrumentations.

Attack-Graph Template Generation. While the instrumented system is operating, if one of the attack consequence detectors reports a malicious behavior, FloGuard's first responsibility is to retrieve and parse the syslogs that have been intercepted by our syscall interception module and logged by the kernel (using `printk`). Some of the syscall logs become corrupted while the kernel is trying to write them in the syslog, i.e., less than 6% on average. To precisely parse the syslogs and ignore the corrupted lines, we extended our module to put extra paddings in the logs to help the parser identify and ignore the corrupted log lines. Figure 4.5 shows a sample syscall log line in FloGuard.

```
Jun 29 14:10:42 Nash kernel:  [11869.281696] ::FloGuard::  ACTION: Read readv PROCESS:
gnome-terminal PID: 1824 SECCON: unconfined_u:unconfined_r:unconfined_t:s0-s0:c0.c255
OBJTYPE: REG 26 INODE: 138548 /tmp/vte6991GV
```

Figure 4.5: A Sample Syscall Log Line

All the required information is collected from kernel data structures and logged. The line above starts with the date, host name, and kernel time. "`::FloGuard::`" padding tells the parser that the line is a FloGuard log line. The main data-flow-related action, which is either read or write, and the actual called syscall are also logged. The caller-process-related information, such as process ID and the security context, is reported as well. Finally, information regarding the object on which the process took the action is extracted and logged; in this example, it is a regular file with the file descriptor 26, inode number 138548, and absolute path /tmp/vte6991GV.

We have collected the intercepted syscalls for the four abovementioned attacks on the system. Figure 4.6(a) shows the number of intercepted syscalls for each attack on the system (see the first column for each attack type). For each attack, once the attack consequence detector reports the detection point event, which was a sensitive file modification for all the attacks, the parser starts reading the syscall logs line by line (approximately 300K lines per second on average) and automatically creates the directed dependency graph. In the graph, nodes are objects that represent OS-level objects, such as regular files, pipes, sockets, and processes, and store information about them, e.g., PID and security contexts for the processes. Each directed edge in the dependency graph represents a data-flow event between two nodes in the graph and is implemented as an object storing all the time instances when the edge's corresponding syscall was called. The second column in Figure 4.6(a) shows the number of nodes in the generated dependency graph for each attack.

To increase the accuracy of the data dependency in the graph, in the syscall interception module, we create a new object, e.g., pipes between process pairs, whenever some data are written to an empty object that has already been created. This enables the parser to distinguish the two separate data flows that are using the same memory/disk region. As a result, the generated dependency graph includes a few fake objects, which can be safely removed. Once the dependency graph is produced, FloGuard starts refining the dependency graph by removing such unnecessary nodes while maintaining the data consistency of the graph by adding a few extra edges to the graph (see

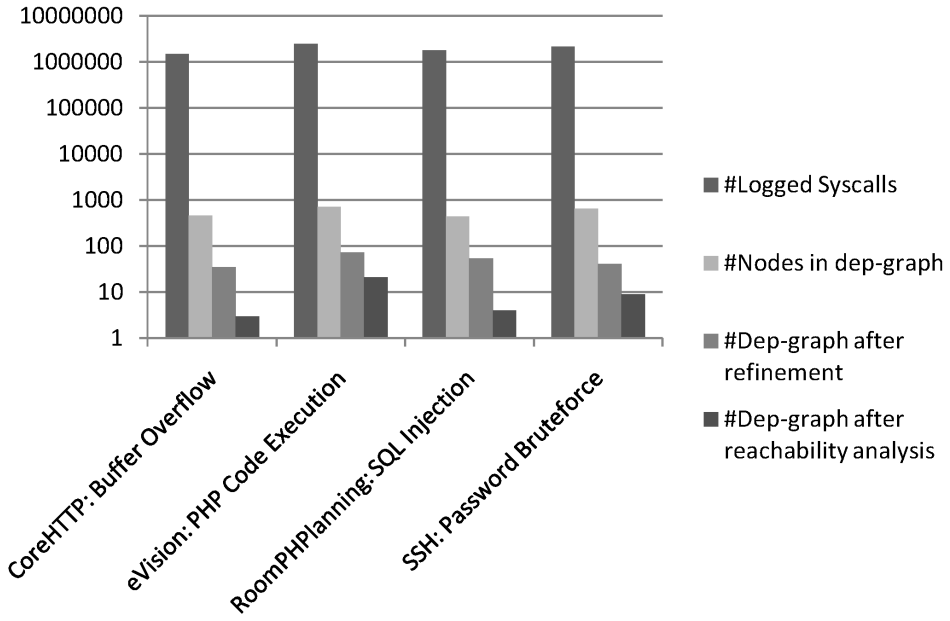
Section 4.3). The results for our attacks are shown in the third column of Figure 4.6(a). Due to large range of demonstrated numbers, the graph depicts numbers in a logarithmic scale.

The last phase before generating the AGTs is to further prune the dependency graph using the reachability analysis. The automated reachability analysis is accomplished according to the detection point that is reported by the attack consequence detectors. More specifically, assuming the attackers have no physical access to the machine, FloGuard starts with the detection point node in the graph and traces back considering occurrence time tags on the edges, until it reaches an attack entry point, which consists of the objects of type `SOCK_AF_INET` or `SOCK_AF_INET6`. As expected, the time-sensitive reachability analysis by FloGuard results in a fairly significant reduction in the number of nodes in the dependency graph. The reachability analysis result for each attack type is shown in Figure 4.6(a). Upon completion of the reachability analysis, the graph is ready to be converted to an AGT graph.

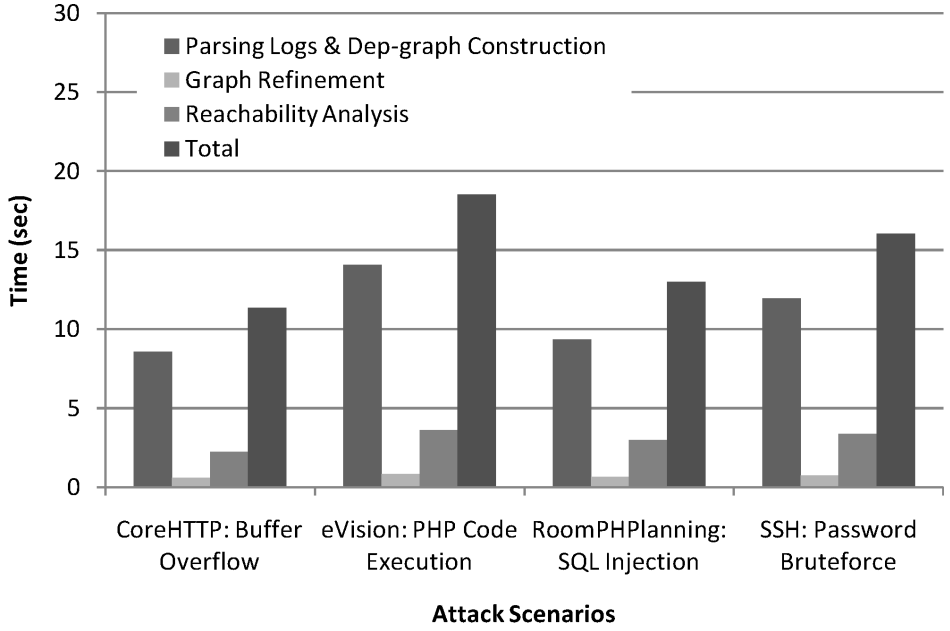
Figure 4.6(b) shows how long each of the abovementioned steps takes. Because in our implementations, the logs are parsed and the initial dependency graph is concurrently constructed, we report the time they took in a single column, i.e., the first columns in Figure 4.6(b). The second and third columns report the time needed for the graph refinement, i.e., removing unnecessary nodes, and the reachability analysis. As shown, most of the total time is spent to parse the syscall logs and produce the initial dependency graph.

Once the reachability analysis on the graph is done, the dependency graph is converted to its corresponding AGT as described in Section 4.3. As the reachability analysis does a fairly good job of pruning the dependency graph, the time required to convert the resulting dependency graph to the corresponding AGT was less than 1 second for all the attack types in our experiments. As discussed earlier, the generated AGT is guaranteed to include the attack path that was traversed by the attacker, who initially did not have any access to the machine, before perpetrating the detection point event. Figure 4.7 shows the automatically generated AGT for the remote PHP code execution attack scenario in our experiments[9]. Internet sockets were assumed to be the only attack entry point in the AGT generation procedure; as the syscall interception module in FloGuard stores source and destination IP/port addresses of the sockets, it is possible to generate a further, more

[9]FloGuard is using the `dot` tool in the Graphviz framework to draw the figures, and can automatically generate and logs the dependency graphs and AGTs, as configured, for later visual inspection by the system administrators.



(a) Logs and Graph Sizes



(b) Required Time for Each Step

Figure 4.6: Automated AGT Generation for Four Attack Scenarios

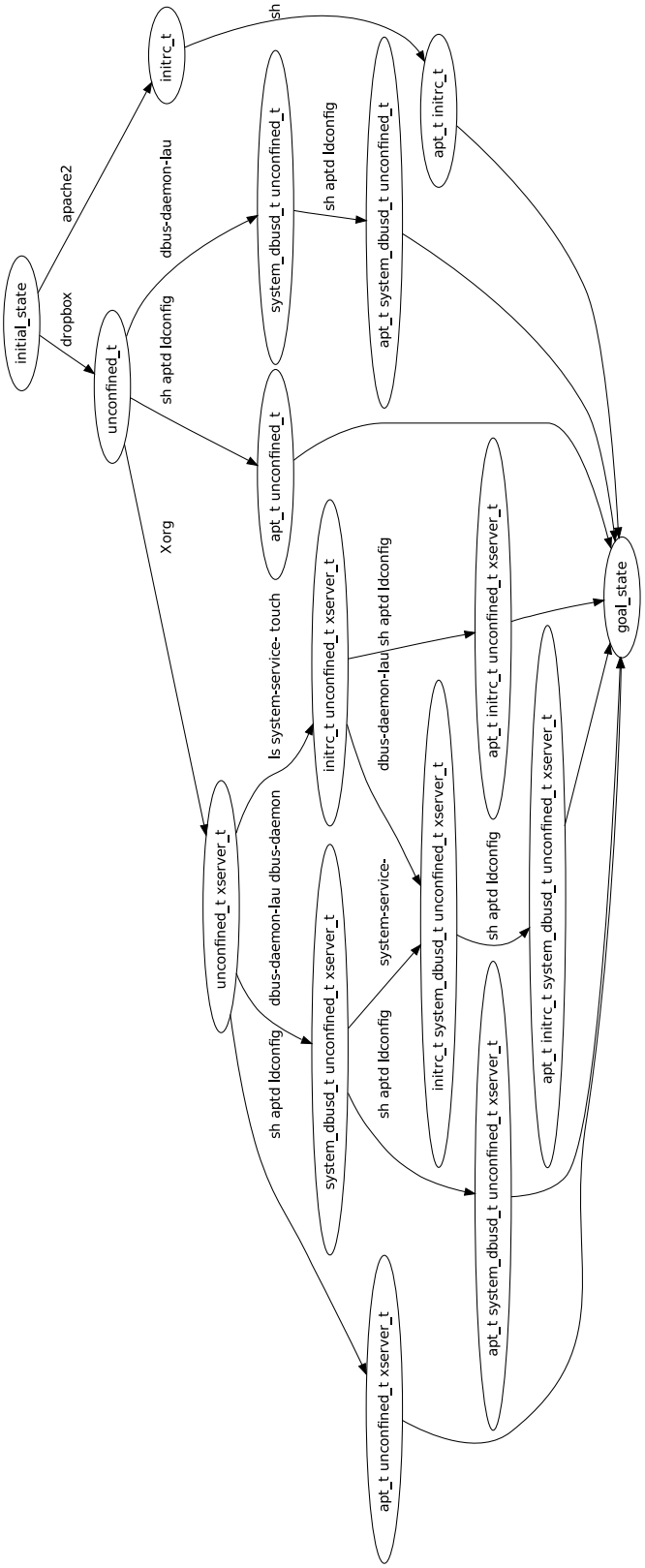


Figure 4.7: An Automatically generated AGT for Remote PHP Code Execution

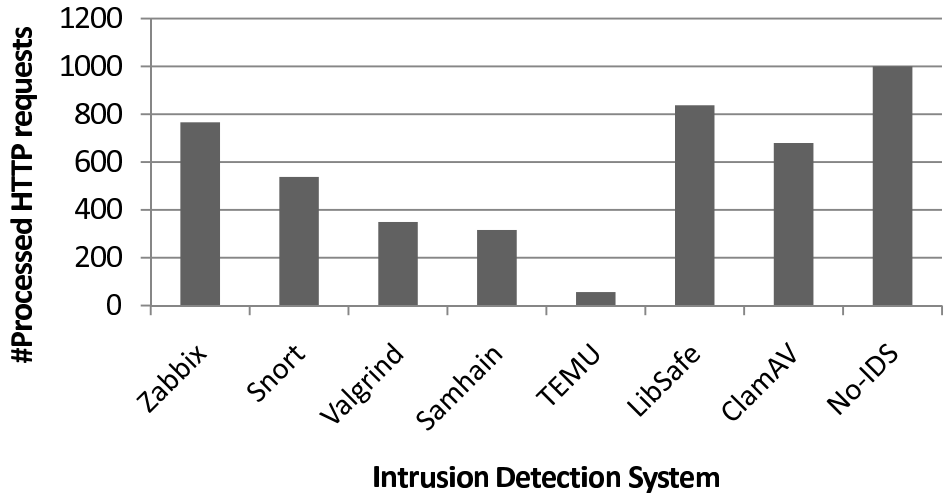
accurate dependency graph and AGT by distinguishing between sockets according to their source and destination addresses. As demonstrated in the figure, the produced AGT is represented using the SE-Linux privilege domains; however, in case the target system uses the traditional two-level discretionary access control, i.e., user and root, the nodes in AGT will have those two privilege levels only. Having generated the AGT for the system, FloGuard uses it to start the automated forensics analysis.

IDS Computational Cost. We first need to evaluate how much computational overhead each IDS causes in our system setup; in other words, we have to quantify and populate the cost function that later will be used to obtain the optimal set of detection systems. To do so, we measured a fine-grained metric that measures the computational overhead of individual detectors on the system's CPU. As illustrated in Figure 4.8(b), the taint analysis engine, i.e., TEMU, puts the highest load on the system's computing resources. Regarding the Samhain IDS, it is important to mention that the measurements show the CPU overhead during its normal operational mode and after its integrity database, i.e., *samhain_file*, was created. The initial database creation took 472 seconds on average for the whole system.

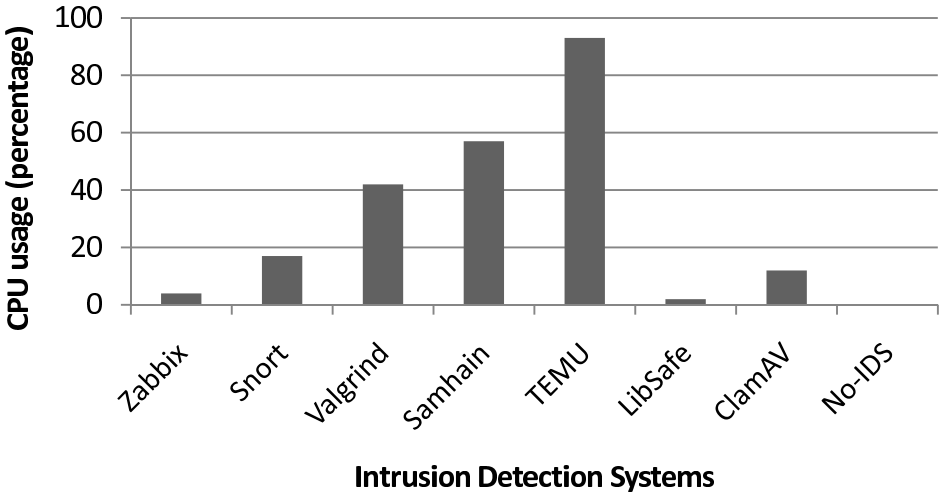
Second, we deployed all the IDSes in the system and turned them on one by one, each time evaluating their impact on the system's overall throughput[10] while the system was operating. In particular, during each round of experiments, we turned on one intrusion detector, e.g., Samhain, and employed HTTPTrafficGen [95] on a remote host within the same network to send subsequent HTTP requests to the system. As shown in Figure 4.8(a), TEMU slowed down the system the most, and the system's maximum throughput was achieved when there was no IDS running in the system.

Periodic Snapshots. As explained earlier, two components in FloGuard put overhead on the system's throughput even if no attack occurs in the system: the syscall interception module and the periodic snapshots. Having discussed the overall overhead caused by the syscall interception and logging earlier in this section, here we evaluate how periodic snapshots impact the system's availability. In particular, we measured the average performance overhead by the system-wide snapshotting for a set of general scenarios. It is worth mentioning that the snapshots are taken and

[10]Here, we define the system's throughput to be the maximum number of client requests that could be processed and responded to by the server within a fixed amount of time.



(a) IDS Impact on System Availability



(b) IDS Impact on System CPU Usage

Figure 4.8: Cost Evaluation of Individual Intrusion Detection Systems

stored incrementally. Figure 4.9(a) illustrates the time it takes for the engine to snapshot the whole system given the amount of data that is modified in the system since the previous snapshot. As a case in point, if 2 GB of data are modified, e.g., downloaded, in the virtual machine between two successive snapshots, i.e., 30 minutes in our experiments, it takes about 13.4 seconds on average to pause the system and take and store a complete system-wide snapshot[11].

Intrusion Forensics. Having evaluated the cost of each detection system, here we explain how FloGuard accomplishes the forensics analysis for each attack scenario against the system using the system's AGT, automatically generated as explained earlier. Figure 4.9(b) shows the high-level overview of how experiments for the forensics analysis were done. In this section, we present the forensics analysis for each attack separately. For each attack scenario, the system was first setup i.e., the syscall interception module was inserted, the attack consequence detector (the Samhain file integrity checker) was deployed, and the periodic snapshotting capability was turned on. Then, the system started its normal operation (running the web server serving the remote clients). Each attack was launched from a remote machine against the system, and modified a sensitive file in the system (as discussed below) that was reported by the Samhain consequence detector. Once the detection point is reported, FloGuard starts the forensics analysis by retrieving a clean snapshot of the system, and selection and deployment of the detection systems according to the logged syscalls during the attack. The snapshot restoration step was pretty fast (3.2 seconds on average). As shown in Figure 4.9(b), FloGuard picks the best detection system based on the system's AGT, which was generated using the syscall logs, and the information statically stored in VulDB, e.g., the detector capability matrix. Upon deployment of the selected IDS during each forensics iteration, FloGuard waits for the attackers to repeat the attack against the system. Once repeated (as reported by Samhain), FloGuard investigates the AGT to see if the exact attack path is identified according to the deployed IDS and the evidence set gathered during the past forensics iterations. If so, forensics is done and a mitigation technique based on the identified vulnerability is permanently deployed in the system until the administrators install suitable patches. Otherwise, the syslogs, collected during the repeated attack, are parsed and AGT is generated; the intersection of the recent AGT and the AGTs generated during the past repetitions of the attack is calculated and

[11]It is important to note that, with the recent support by the Virtual Box framework, it is also possible to take live snapshots, and hence, the system is not paused while the snapshot is taken.

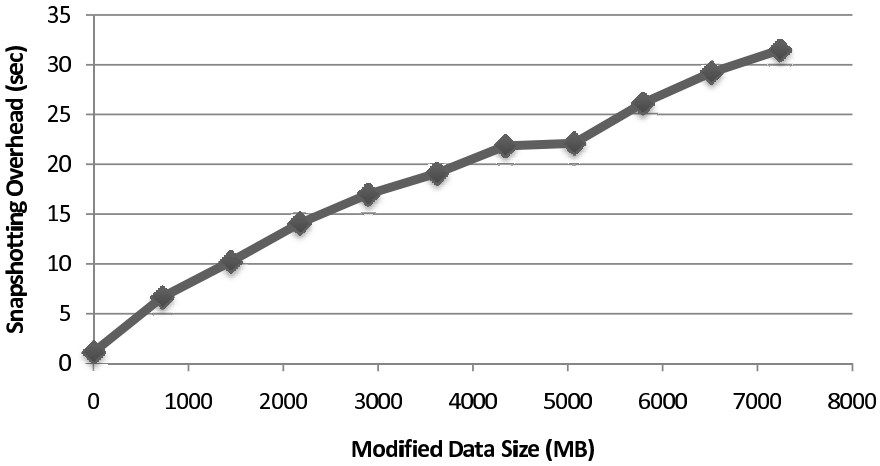
used to pick the next best IDS to deploy in the system for the next forensics iteration[12]. To further clarify, each state transition, i.e., vulnerability exploitation (privilege escalation) represented by a (*process, vulnerability*) edge between two states in the intersection of AGTs, is the intersection of all exploitations between the same state pairs in all the AGTs.

Finally, we present our experimental results regarding iterative intrusion forensics analysis for four different attack scenarios. As mentioned above, Samhain was deployed as the attack consequence detection system throughout the experiments. Specifically, we modified its configuration, i.e., `/etc/samhain/samhainrc`, to monitor the files and directories in which we are interested, and configured it to report events with at least `crit` severity level. Before the experiments, we created its initial database, i.e., `/var/state/samhain/samhain_file`, using `samhain -t init`, and its database was updated, using `-t update`, right after every time a snapshot of the system was taken. During the normal operational mode of the system, Samhain was configured to check the marked sensitive files and directories against its database and fire an alert upon identifying a modification or access (depending on the policy defined in the configuration file).

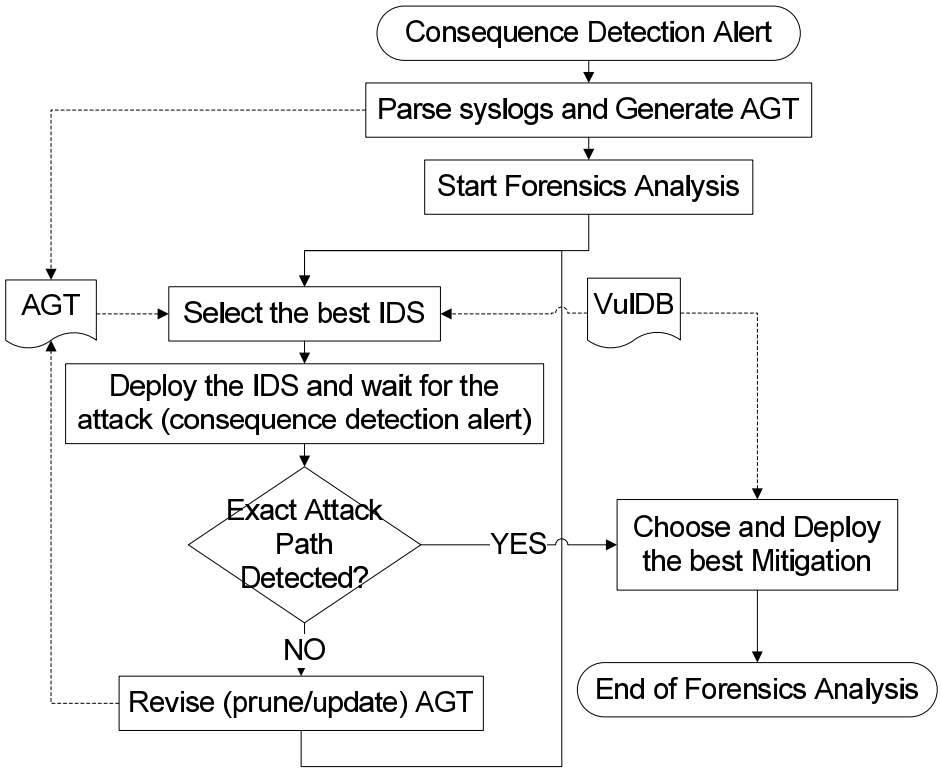
First, we start with the buffer overflow attack scenario. Once the CoreHTTP web server was up and operating after the snapshot, from a remote machine we launched a buffer overflow exploit, which we had created manually using GDB, and the shell code to spawn and get access to a remote shell on the target machine. Using the shell, we modified the web server's configuration file, i.e., `chttp.conf`, which had been marked to be monitored by Samhain; consequently, once Samhain started performing its periodic check using its database, it fired the alert illustrated by Figure 4.10 regarding the `chttp.conf` modification.

Upon receiving the abovementioned alert from Samhain, FloGuard started its forensics analysis by retrieving and parsing the syscall logs from the last snapshot to the detection point, and generating the AGT. As shown in Figure 4.11(a), the initial AGT consisted of 6 possible attack paths based on the intercepted syscalls during the attack. Having employed the monitor selection algorithm (discussed earlier), FloGuard picked Valgrind as the first detection system, as it maximized the coverage/cost measure, to deploy in the system to monitor the web server. Consequently, the

[12] As the attackers go through the same attack path in AGT, the intersection of the AGTs is also guaranteed to include the actual path traversed by the attackers while including fewer vertices, thus accelerating the forensics process. To support forensics for the case in which several attack paths can exist, the recently generated AGT is used for the next forensics iteration.



(a) Snapshotting Overhead on the System



(b) Forensics Experiments Setup

Figure 4.9: Snapshots Overhead and Experiment Setup

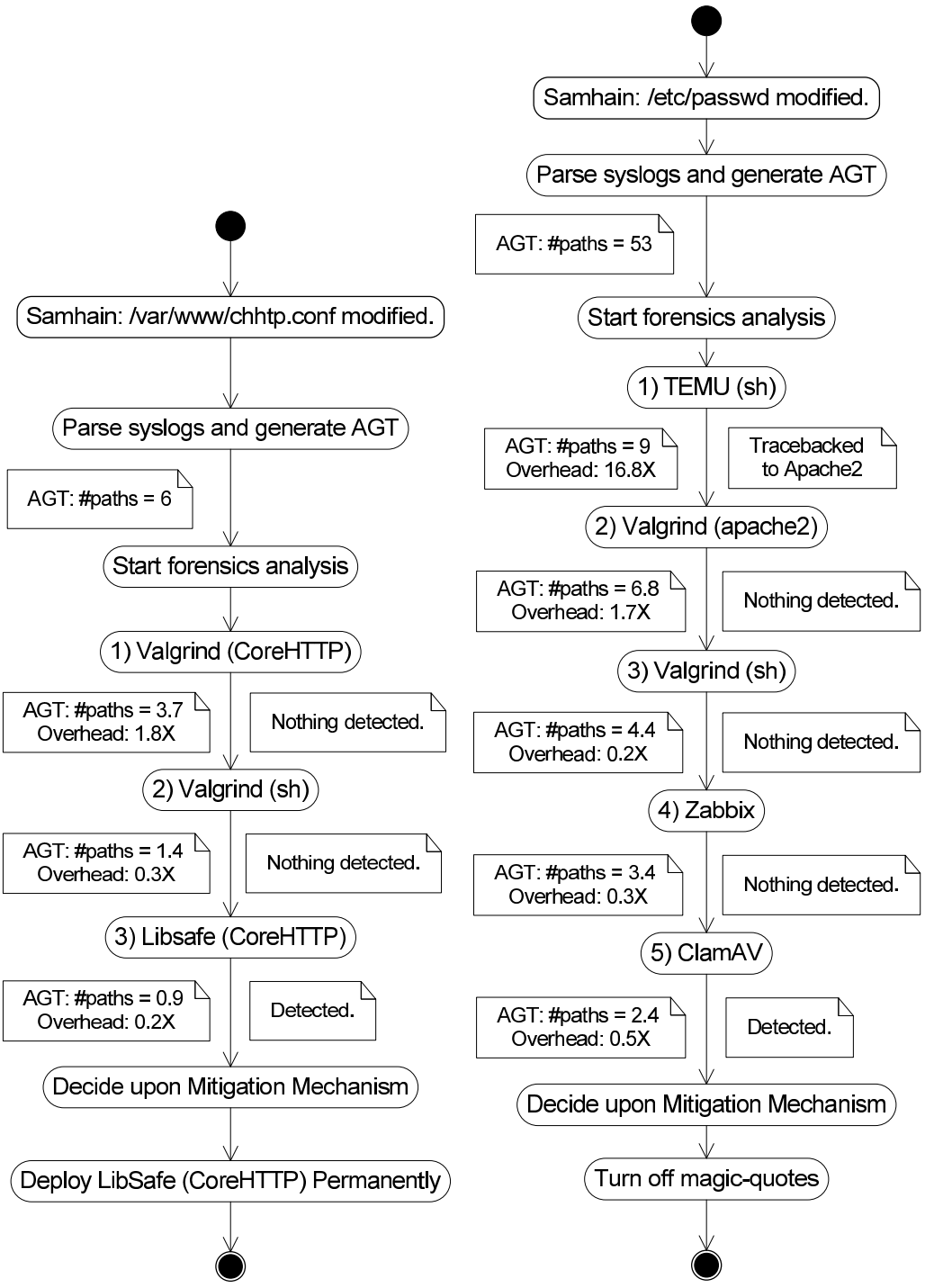
```
CRIT: [2010-07-05T21:53:01-0500] msg= POLICY [IgnoreNone], path=/var/www/chttp.conf,
inode_old=824046, inode_new=789469, dev_old=8,1, dev_new=8,1, size_old=123,
size_new=127, atime_old=[2010-07-06T02:43:54], atime_new=[2010-07-06T02:50:58],
mtime_old=[2010-07-06T02:43:45], mtime_new=[2010-07-06T02:49:44],
chksum_old=87C08...50EB7, chksum_new=F38...3E92
```

Figure 4.10: A Sample Alert by the Samhain File Integrity Checker

past clean snapshot was retrieved, Valgrind was deployed using the `valgrind --tool=memcheck --leak-check=yes corehttp chttp.conf` command, and the system started its normal operation while FloGuard was waiting for the next repetition of the attack. On the other hand, we relaunched the attack from the remote host against the web server again, and again modified a sensitive file in the system. Valgrind was not able to detect the buffer overflow exploitation in CoreHTTP since Valgrind, by design, does not perform bounds checking on static arrays (allocated on the stack). Once Samhain fired the sensitive modification alert for the second time, as shown in the figure, FloGuard produced an AGT that was identical to its previous version in this case, and was pruned based on (the detection capability of) the deployed detection system, i.e., Valgrind. As demonstrated in the figure, the resulting AGT consisted of 3.7 expected number of attack paths from the initial state to the goal state. Using the refined AGT and the detection capability matrix, FloGuard chose the next detection system, i.e., Valgrind on the `sh` process, and the iterative forensics continued until Libsafe was picked to monitor the web server that successfully detected the buffer overflow caused by the `scanf` function in CoreHTTP. Consequently, LibSafe was permanently turned on to detect and block similar attacks exploiting the buffer overflow vulnerability in CoreHTTP until the administrator manually patches/updates the vulnerable web server.

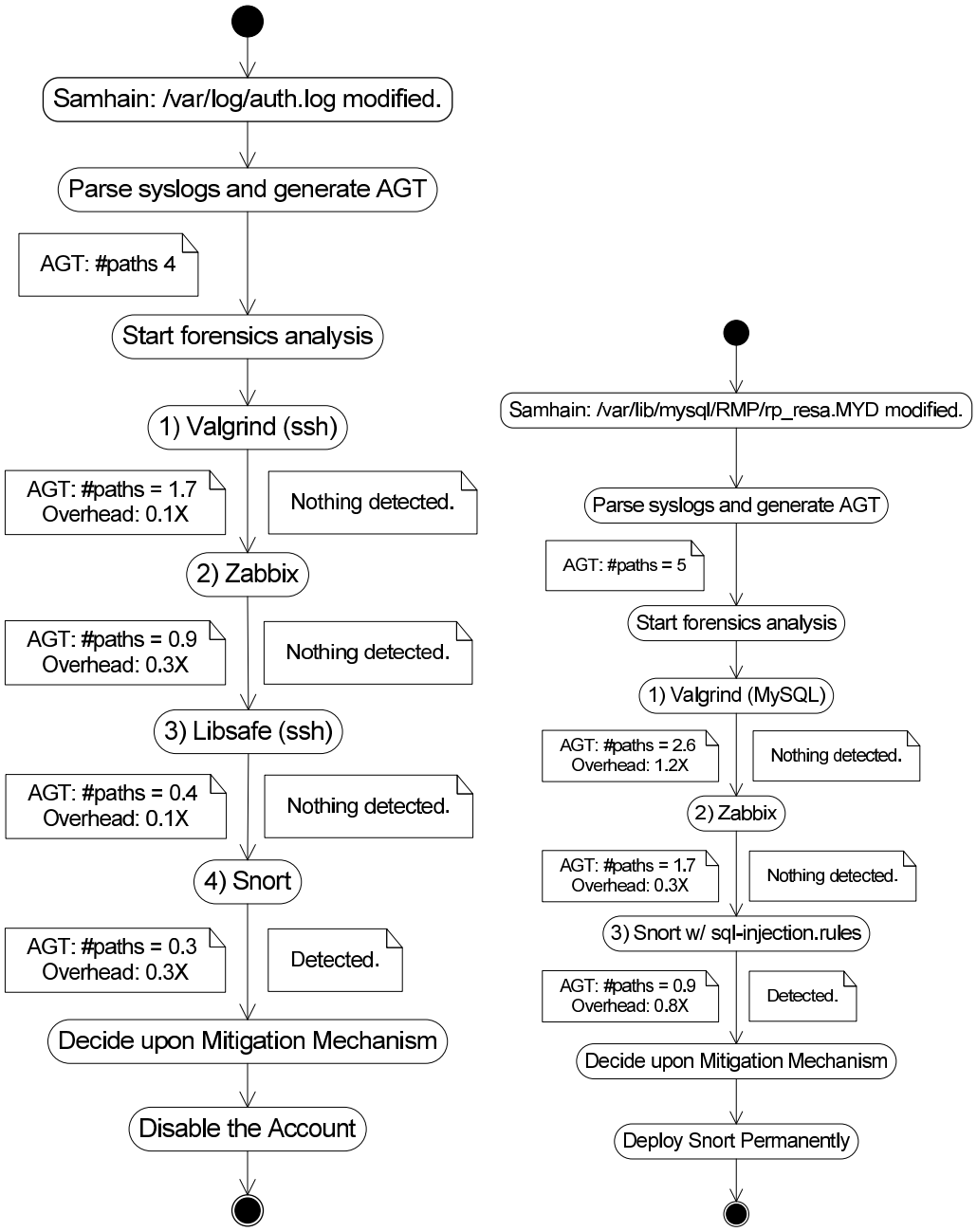
The second attack scenario in our experiments was privilege escalation through exploitation of the PHP remote code execution vulnerability in the eVision-2.0 content management system application. To attack the system, we employed the exploit in Perl, which we got from exploit-db[13]. Much as in the previous scenario, we launched the attack from a remote machine against eVision-2.0, which was vulnerable to arbitrary file upload and local file inclusion, so during the attack we included the gif file to the folder `eVision-2.0/modules.conf/brandnews/showpart/icons` and executed it on the server. The included file had the code `<?php system($_GET[cmd]); ?>` which enabled us to execute any arbitrary command from the remote machine on the server. Specifically, as shown in Figure 4.11(b), we modified the `/etc/passwd` file, which was marked to be monitored by Samhain, after the privilege escalation. The first detection system to turn on for the forensics analysis was TEMU, because of its capability in tracing back the data from the process `sh` that caused the detection point event (see Figure 4.7). In particular, to use TEMU, the tracecap plugin was loaded using the `load_plugin tracecap/tracecap.so` command, and then all input

[13]http://www.exploit-db.com/exploits/7947.



(a) CoreHTTP: Buffer Overflow (b) eVision: Remote Code Execution

Figure 4.11: Iterative Intrusion Forensics Analysis



(a) SSH: Password Brute-force (b) RoomPHPlanning: SQL Injection

Figure 4.12: Iterative Intrusion Forensics Analysis

```
alert tcp $EXTERNAL_NET any → $HOME_NET 22 (msg:``Bruteforce attack'';
content:``pass''; threshold:  type threshold, track_by_src, count 5, seconds 60;
sid:301;)
```

Figure 4.13: A Sample Alert by the Snort Network-based Intrusion Detection System

data to the process were tainted (`taint_file file_path 0 tainting-ID`). The sh process was then traced by TEMU's `tracebyname` command. Finally, the actual data source, i.e., the apache2 process (see Figure 4.7), was identified via the `list_tainted_files` command, which we had implemented by translating disk-level addresses to filenames (as discussed earlier in this section). Using TEMU helped to prune the AGT to include only the paths exploiting possible vulnerabilities in the apache2 process. Finally, the ClamAV detection system detected the uploaded file during the attack, i.e., `osirys.txt.gif`[14], and FloGuard, using the VulDB, decided to turn off the `magic_quotes` (even though this might have affected other applications).

The third attack scenario in our experiments was the password brute-force attack against the sshd process in the system. The attack was launched from a remote machine using a Perl script that subsequently tried thousands of frequently used passwords, given a huge file storing the passwords as an argument, against the server. In our case, the actual password was "administrator." Subsequent password trials made Samhain fire an alert upon the `/var/log/auth.log` file modification. In forensics analysis, Snort finally detected the password brute-force attack using the signature illustrated by Figure 4.13.

That, in plain text, makes Snort send an alert if subsequent passwords (packets) containing the term *pass* are tried more than 5 times within a 60-second window. Finally, FloGuard decided to disable the account using VulDB.

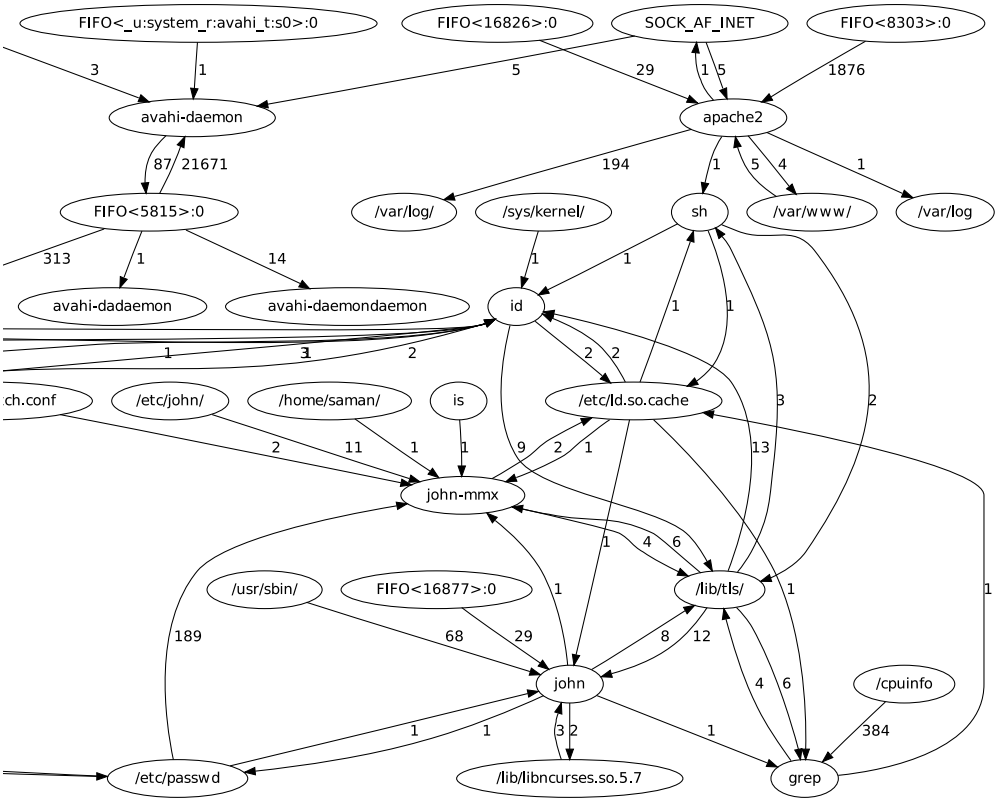
The next attack scenario in the experiments was designed to modify a sensitive database file through the exploitation of a SQL injection vulnerability in the `login.php` file of the RoomPHPlanning web application. Upon receiving the Samhain alert, FloGuard started the iterative forensics analysis. As shown in Figure 4.12(b), Zabbix was selected to be deployed in the second iteration after Valgrind was unable to detect anything during the first iteration; however, Zabbix did not detect any misbehavior in the system either as the SQL injection attack had not consumed

[14]We had already created the MD5 signatures, using `sigtool --md5 malware > signature-ID.hdb` and updated the virus database directory in ClamAV, i.e., `/usr/local/share/clamav/`.

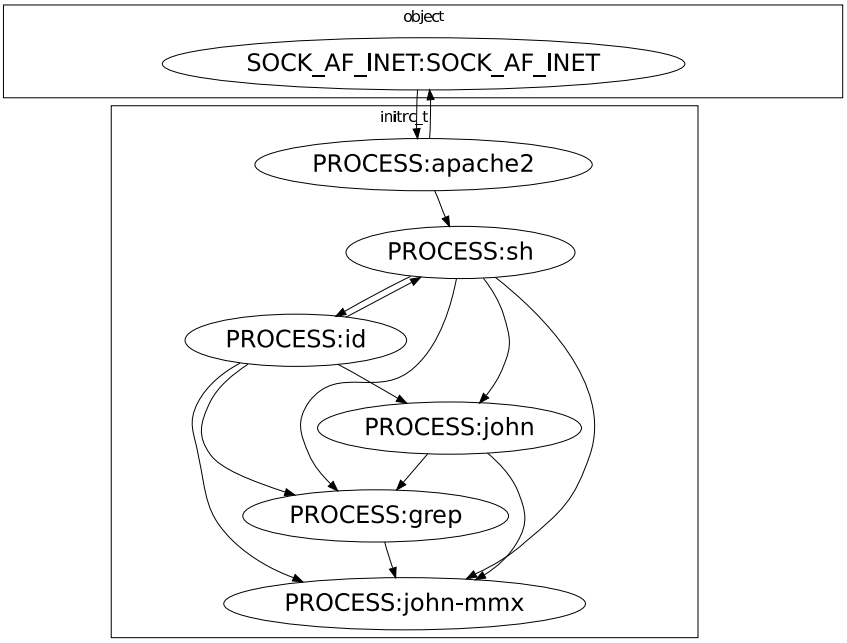
a considerable amount of system resources. Finally for the last forensics iteration, FloGuard chose to deploy Snort (with the updated `sql-injection.rules` enabled in `/etc/snort/snort.conf`) which detected the SQL injection by identifying SQL meta-characters in the incoming data.

Finally, we experimented FloGuard on a multi-step attack scenario. We initially inserted the syscall interception module in the kernel, and deployed the Zabbix consequence detector in the target machine. First, we launched the attack from a remote host, and got a shell access on the target system through the local file inclusion vulnerability in the eVision application. Then, to maintain the control over the system, through reboots and software patches, we tried to get the admin password of the system using the John the Ripper password cracker; that was possible as we had intentionally left the `/etc/shadow` readable. Before starting the password attack, we used the `unshadow` tool to combine the `/etc/passwd` and `/etc/shadow` and store the user-hash mappings in a seprate file. The last step was simply to spawn the `john` tool on the file; the tool started subsequent password trials over-consuming system's computational resources (96% CPU usage on average); this caused the Zabbix anomaly detector fire an alert making FloGuard start the iteration forensics analysis to identify the vulnerability exploitations during the attack.

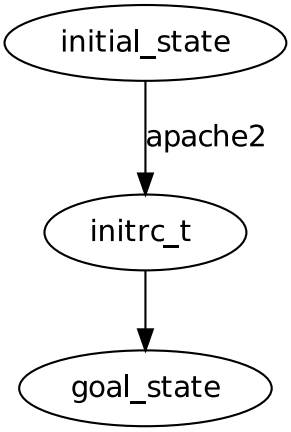
As discussed before, FloGuard first retrieved the logged syscalls, parsed them, and generated the dependency graph which is partially (due to space limits as there were 1286 vertices in the graph) shown in Figure 4.14(a). Then, given the process which caused the detection point `john-mmx`, FloGuard called the refinement procedure and the reachability analysis on the graph removing unnecessary nodes. Figure 4.14(b) demonstrates the resulting dependency graph which had totally 7 nodes. Using the refined dependency graph, FloGuard generated the AGT to later investigate possible privilege escalation paths in the system. As shown in Figure 4.14(c), the `apache2` process in the only possible entry point (initial vulnerability exploitation) for the attack. Trying to identify the vulnerability type which was exploited in the apache2 process, FloGuard went through the iterative forensics analysis (as discussed before in this section) while we were remotely replaying the attack. During the first iteration, FloGuard picked the Valgrind intrusion detector which was unable to detect any misbehavior; however, the second chosen detection system, i.e., ClamAV, detected the downloaded file `osirys.txt.gif` during the PHP file inclusion exploitation (using `clamscan -d signature-83.hdb /var/www/`).



(a) A Snapshot of the Dependency Graph



(b) The Dependency Graph after the Reachability Analysis



(c) The Attack Graph Template

Figure 4.14: The Dependency Graph and Attack Graph Template for the Multi-step Attack

4.7 Related Work

There are several results in the literature that propose individual elements of what FloGuard achieves. However, we are not aware of any other approach that can perform on-demand, cost-aware intrusion detector deployment to defend against multi-stage attacks affecting multiple parts of a system.

Several approaches perform correlation of outputs across multiple IDSes. For example, Debar et al. [96] introduce an aggregation algorithm to receive and correlate detection alerts from different detection *probes*. BotHunter [97] uses an intrusion report correlation model and can recognize successful bot infections through the communication sequences that occur during the infection process. However, both methods assume that all the appropriate intrusion detectors have already been chosen and deployed.

The concept of on-demand re-execution in a modified environment has also received some attention. Rx [98] and First-Aid [99] re-execute applications in a different execution environment, but they are not targeted towards security exploits. Bouncer [74], Vigilante [100] and Sweeper [75] combine an intrusion detector along with program slicing or symbolic execution to trace-back from the detection point and produce input filters that can block the exploit packets. FLIPS [29] employs re-execution with instruction set randomization to detect root vulnerabilities. Shadow Honeypots [101] use re-execution in space (i.e., another machine) instead of time. They use anomaly detectors to identify anomalous incoming traffic that is later processed by a shadow honeypot, i.e., an instrumented version of the application, to determine the accuracy of the anomaly prediction. However, most of these techniques only support detection of a single type of vulnerability (usually memory errors), and rely on the ability to detect attacks within the same process as the bug or exploit entry-point. They cannot trace multi-step attacks that are detected in other parts of the system. Moreover, they do not consider multiple types of detectors and associated cost factors. FloGuard complements systems such as Sweeper by tracing detection points across multiple processes.

Attack trees and graphs have been extensively used by security researcher to document known system vulnerabilities and attack paths. Two main drawbacks of the current attack graph generation approaches are: 1) their disability in addressing unknown attacks, e.g. zero-day attacks; and 2) to improve scalability, their logic-based state notion does not represent system-level detailed

information that significantly limits their usage in practice.

There has also been related work on intrusion forensics analysis. Mukkamala et al. [102] introduce a network forensics algorithm based on artificial neural networks to discover sources of information breaches. Carrier [54] presents file-system-based forensics techniques to determine the source of security breaches by investigating their effects on the file-objects, e.g., file modification dates. Taser [78] is an operator-assisted intrusion forensics system based on pessimistic taint tracking; hence, it may result in a large set of possible attack sources. BackTracker [77] and Panorama [70] aid off-line forensic analysis by producing taint-traces of file and process connections that led to a detected security breach. Because these forensics tools are based on passive data collection, they are either very pessimistic, marking most activities as malicious, or optimistic, thus missing many malicious activities that occur during an attack. In comparison, because FloGuard can actively deploy additional detection mechanisms to validate or refine its suspected attack paths, it can support much more realistic analysis. Nevertheless, pessimistic techniques that automatically produce system-level taint-graphs can be used to automatically produce initial AGTs for FloGuard.

4.8 Conclusion

We presented FloGuard, a cost-aware intrusion detection system that uses online forensics and on-demand IDS deployment. FloGuard enables systems to defend against attacks that exploit various classes of previously unknown vulnerabilities. Upon detection of the high-level consequences of an attack, an automated forensics analysis, based on the attack-graph-template formalism, is used to iteratively deduce an attack path, comprising a sequence of vulnerability exploitations, that led to the detected consequence. An optimal set of monitors is then deployed to allow future attacks using the same exploit to be detected close enough to the intrusion point for automated mitigation mechanisms to be utilized. Our experiments show that FloGuard can deploy off-the-shelf IDSes only when they are needed and help protect systems against previously unknown vulnerabilities with minimal snapshotting overheads during normal operation.

CHAPTER 5

SECURITY-STATE ESTIMATION

One of the key problems that should be solved by an intrusion response system is how to determine the current security state of the system according to the set of past triggered IDS alerts and sensor measurements. The problem is even harder when the past alerts and other information come from different types of sensors. As a case in point, in a cyber-physical system like a power grid infrastructure, sensory information comes from both cyber IDSes and physical power sensors, e.g., power measurement units (PMUs). Gathering all the information from different sources and unifying it to identify the current system security state is a challenging problem. In this chapter, we address the problem of estimating a system's security state given the uncertain reports obtained from various types of sensors.

In related work, Bothunter [97] extends ideas from multi-sensor data fusion to probabilistically correlate triggered alerts by intrusion detection systems (IDSes). The main goal is to identify the set of compromised hosts [103]; however, using such solutions in power grid cyber-physical systems may result in incorrect reports, as they ignore information from power sensors, and hence do not consider the underlying power system dynamics. On the other hand, [104, 105] focus on detecting corrupted measurements using only power sensors. Such bad-data detection techniques have two major limitations. Detection accuracy of some approaches, e.g., the least-square error-based algorithms [105], is usually low against security attacks, as they consider *all* the measurements to be correct, so-called *good*, initially. Furthermore, some other approaches, e.g., combinational-based techniques [104], often do not scale well enough as they totally ignore information from cyber sensors, and hence their search space for detecting bad measurements grows exponentially with the number of measurements [104].

In this chapter, we present a cyber-physical intrusion detection system (*CPIDS*), which, at each time instant, identifies the compromised set of computers in the cyber network and maliciously

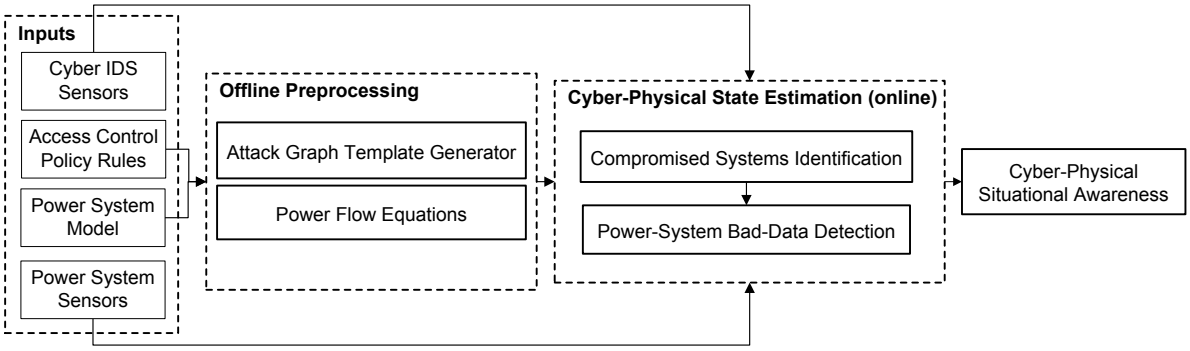


Figure 5.1: CPIDS's High-level Architecture

modified set of power system measurements. The main challenge is in fusing *uncertain* information from *different* types of distributed sensors, such as power system measurement units and cyber-side intrusion detection probes, to detect malicious activities within the cyber-physical system. The proposed framework investigates the cyber and physical parts of the system separately; therefore, for a system that includes only computer networks, the intrusion response system could use the cyber portion of the introduced solution by itself.

5.1 Overview

Figure 5.1 presents a high-level overview of CPIDS's architecture and how its components are interconnected. Before CPIDS starts its online operation, it uses the power network's access control policies, e.g., firewall rules, and automatically generates a pessimistic attack graph, called an *attack graph template (AGT)*. The state transitions in an AGT encode all possible attack paths that an attacker can traverse by sequences of vulnerability exploitations. Furthermore, CPIDS takes an underlying power system model and calculates the corresponding power flow equations (Figure 5.1) to determine how the correct power measurements should be correlated.

As illustrated, during the online operation, CPIDS uses both cyber IDS reports and power measurements to estimate the cyber-physical security state of the power grid that encodes the compromised host systems and the maliciously modified power measurements. Those two sets of information, encoded by the cyber-physical security state notion, gives a necessary and sufficient view of the system security, regarding both of the cyber and power sides, for automated intrusion

response systems or network administrators to decide upon response strategies if a security attack occurs.

In particular, to estimate the set of compromised host systems, at each time instant, CPIDS estimates the attack path in AGT that has been traversed by the adversary based on the past triggered sequence of IDS alerts. Specifically, due to inherent uncertainties in reported IDS alert notifications, determining the exact traversed attack path is not always feasible. Instead, at each time instant, a posterior probability distribution over the AGT's state space is calculated according to the false positive and negative rates of the triggered and non-triggered IDS alerts, respectively. That estimated probabilistic state knowledge reveals the set of privilege domains, i.e., host systems, believed to be compromised in the control network (Figure 5.1).

Then, by using the power grid topology and the estimated probabilistic knowledge about the set of compromised hosts, CPIDS identifies the *potentially corrupted* power measurements. To clarify, the potentially corrupted power measurements are the measurements which could or might have been modified based on the given topological information regarding which power sensors are controlled and maintained by the estimated set of compromised hosts. CPIDS then estimates the power system state, using the traditional Newton-Raphson algorithm, by initially ignoring the potentially corrupted power measurements. Finally, CPIDS employs the estimated power system state and the power system model to estimate the correct measurement vector. As illustrated in Figure 5.1, CPIDS identifies the most likely set of *actually corrupted* measurements by comparing the correct measurement vector and the real measurements.

Consequently, the actually corrupted measurements and the set of compromised hosts define the current cyber-physical security state of the power grid.

5.2 Cyber Security-State Estimation

We will now discuss how CPIDS generates the power network attack graph template from the access control policies and uses it to estimate the compromised set of hosts, given the IDS alerts.

The power network's access control policies, such as firewall rulesets, are composed of rules about sources (IP/port addresses) that are or are not allowed to reach a destination. CPIDS parses

the rulesets and creates a binary network connectivity matrix that is a Cartesian product of host systems. The $[i, j]$ entry of the matrix takes on a true value if the traffic from host h_i to host h_j is allowed, and a false value if it is not allowed. The connectivity matrix always includes an Internet node representing a group of hosts outside of the network where attackers are assumed to initially reside.

Attack graph template generation. Generally, every cyber attack path consists of an escalating series of vulnerability exploitations by the adversary, who initially has no access to the system and then achieves the privilege required to reach his or her malicious goals, e.g., modifying a power sensor measurement. We present the *attack graph template (AGT)*, i.e., an extended attack graph [87], which represents all *possible* attack paths (unlike traditional attack graphs [87], which only address previously *known* paths). To further clarify, an AGT, by design, would address a zero-day (previously unknown) buffer overflow exploitation of a historian server process, while a traditional attack graph would be unaware of it. An AGT is a state-based directed graph, in which a state is defined as the compromised privilege domains in that state. Therefore, the initial state is $(\varnothing)$, in which the attacker does not yet have any privileges over the power network. Each state transition represents a privilege escalation which is achieved through a vulnerability exploitation. Therefore, any path on the AGT graph represents an attack path in the power network.

To generate an AGT, CPIDS pessimistically considers every host within the power network to be a single potentially vulnerable privilege domain. In particular, CPIDS automatically generates an AGT by traversing the connectivity matrix and concurrently updating the AGT. First, CPIDS creates the AGT's initial state $(\varnothing)$ and starts the AGT generation with the network's entry point (Internet) node in the connectivity matrix. Considering the connectivity matrix as a directed graph, CPIDS runs a depth-first search (DFS) on the graph. While the DFS is recursively traversing the graph, it keeps track of the current state in the AGT, i.e., the set of privileges already gained through the path traversed so far by the DFS. When the DFS meets a graph edge $[i, j]$ that crosses over privilege domains h_i to h_j, a state transition in AGT is created if the current state in AGT does not include the privilege domain of the host to which the edge leads, i.e., h_j. The transition in AGT is between the current state and the state that includes exactly the same privilege set as the current state plus the host h_j directed by the graph edge $[i, j]$. The AGT's current state in the algorithm is then updated to the latter state, and the algorithm proceeds until no further updates to

AGT are possible according to the connectivity matrix. At that point, the offline AGT generation is complete, and by design, the AGT includes all possible attack paths launching from remote (Internet) host systems against the network.

AGT-to-HMM conversion. The AGT is converted to a hidden Markov model (HMM) [106], which will be used later to determine the attack path traversed by the attacker at each time instant, given the past set of triggered IDS alerts.

To generate the HMM model, CPIDS enhances the AGT using the network's topology to encapsulate knowledge about deployed cyber-side IDSes. In other words, each AGT edge is tagged by a (possibly empty) set of IDSes that monitor the edge's corresponding network link within the power network. CPIDS later uses these tags to map IDS alerts (observations) to their corresponding state transitions to estimate the attack path traversed by the attacker. In practice, IDSes usually report many false positives and may miss some incidents, i.e., false negatives. To account for the inherent uncertainties in IDS alert notifications, CPIDS labels the IDS tags on state transitions with their false positive and negative rates.

Cyber security-state estimation. We now describe how CPIDS, during its online operation, makes use of the HMM model and online IDS alerts to probabilistically deduce the attacker's previous actions (vulnerability exploitations), and hence the set of already compromised host systems. In particular, CPIDS makes use of the HMM to track the attacker's action sequence while the IDS alerts are sequentially triggered. To do so, CPIDS uses an HMM smoothing algorithm [106] to estimate the network's current security state given the past triggered IDS alerts. In an HMM, unlike a regular Markov model, states are not directly visible, but observations (IDS alerts) are visible. The goal is to get the past observation sequence and probabilistically estimate the traversed state sequence (attack path) considering the false positive and negative rates of the monitoring IDS probes.

Formally, CPIDS models each attack scenario as a discrete-time hidden Markov process, i.e., event sequence $Y = (y_0, y_1, \cdots, y_{n-1})$ of arbitrary lengths. $y_i = (s_i, o_i)$, where s_i is an HMM state at the ith step of the attack and is unobserved, and the observation o_i is the triggered IDS alerts at that step. The initial state is defined as $s_0 = (\varnothing)$, as discussed above.

CPIDS's main responsibility is to compute $Pr(s_t \mid o_{0:t})$, that is, the probability distribution over hidden states at each time instant, given the HMM model and the past IDS alerts $o_{0:t} = (o_0, \cdots, o_t)$.

In particular, CPIDS makes use of the forward-backward smoothing algorithm [106], which, in the first pass, calculates the probability of ending up in any particular HMM state given the first k IDS alerts in the sequence $Pr(s_k \mid o_{0:k})$. In the second pass, the algorithm computes a set of backward probabilities that provide the probability of receiving the remaining observations given any starting point k, i.e., $Pr(o_{k+1:t} \mid s_k)$. The two probability distributions can then be combined to obtain the distribution over states at any specific point in time given the entire observation sequence:

$$Pr(s_t \mid o_{0:t}) = Pr(s_k \mid o_{1:k}, o_{k+1:t}) \propto Pr(o_{k+1:t} \mid s_k) \cdot Pr(s_k \mid o_{1:k}), \tag{5.1}$$

where the last step follows from an application of Bayes's rule and the conditional independence of $o_{k+1:t}$ and $o_{1:k}$ given s_k. Having solved the HMM's smoothing problem for $Pr(s_t \mid o_{0:t})$, CPIDS probabilistically knows about the current cyber security state, i.e., the set of compromised host systems. Next, our goal is to use the knowledge of current cyber security state to accurately estimate the underlying power system state.

5.3 Power System State Estimation

As discussed before, we define the cyber-physical security state of the power grid as a set of compromised host systems and maliciously modified power measurements in that state. In Section 5.2, we introduced an algorithm to probabilistically determine the set of compromised hosts at each time instant by calculating the probability of being in each HMM state that encodes the set of compromised hosts. In this section, we present how CPIDS uses the knowledge about compromised hosts to identify the set of maliciously modified power measurements, so-called *bad data*. As discussed later, the bad-data detection enables CPIDS to estimate the underlying power system state correctly.

Background. Before presenting the bad-data detection algorithm, here we briefly give a background review on power system flow equations and state estimation. In a power-grid infrastructure, the underlying power system is represented as a set of nonlinear AC equations for active and reac-

tive power flows:

$$\mathbf{P}_{ij} = \mathbf{V}_i^2[-\mathbf{G}_{ij}] + \mathbf{V}_i\mathbf{V}_j[\mathbf{G}_{ij}cos(\theta_i - \theta_j) + \mathbf{B}_{ij}sin(\theta_i - \theta_j) \tag{5.2}$$

$$\mathbf{Q}_{ij} = -\mathbf{V}_i^2[-\mathbf{B}_{ij}] + \mathbf{V}_i\mathbf{V}_j[\mathbf{G}_{ij}sin(\theta_i - \theta_j) - \mathbf{B}_{ij}cos(\theta i - \theta_j) \tag{5.3}$$

where $\mathbf{P}_{ij}$ and $\mathbf{Q}_{ij}$ are active and reactive power flows from bus i to bus j, respectively. $\mathbf{G}_{ij}$ and $\mathbf{B}_{ij}$ denote the elements in the i,j position of the real and imaginary components of the system admittance matrix $\mathbf{Y_{bus}} = \mathbf{G} + j\mathbf{B}$, which contains the network line parameters ($\mathbf{I} = \mathbf{Y_{bus}}\mathbf{V}$) [107].

In general, the power system state estimation problem involves estimation of the current conditions in a power system based on snapshots of real-time measurements, i.e., real and reactive power magnitudes. The estimated quantities include bus voltage magnitudes and angles that constitute the power system state variables. The estimate is computed using known equations, which relate the power system measurements to the unknown states that are to be estimated. The equations depend on the power flow equations that are derived from the power system topology. In particular, in Equations (5.2) and (5.3), the values of $\mathbf{P}_{ij}$ and $\mathbf{Q}_{ij}$ are measured by the power sensors, such as the power measurement units, and the values of the power system state vector, $\mathbf{V}$ (voltage magnitudes) and θ (phase angles), are estimated using the iterative Newton-Raphson state estimation equations [107]. Once the state variables, i.e., bus voltage phasors, are known, all other quantities, such as currents and non-measured real and reactive line flows, can be computed [107].

In general, power system state estimation is typically an overdetermined problem, since there are more measurements available than are needed to solve for the unknown voltage magnitudes and angles. In other words, the power system state estimation server can still estimate the power state correctly if the redundant measurements are completely ignored. However, in a practical attack-free situation, power measurements usually include zero-mean Gaussian noise due to natural and accidental faults. Therefore, deployment of redundant power sensors improves power system state estimation accuracy.

In power system state estimation, it is critical that data from power sensors are correct and match the power system model (represented by Equations (5.2) and (5.3)). The reason is that modified measurements could mislead the estimation equations so that they produce incorrect power system

estimates without noticing anything wrong with the measurements. That fact is exploited by data modification attacks, in which once the attacker compromises a particular power network host system, he or she maliciously modifies a subset of power measurements before they are fed to the power state estimation servers. Consequently, system operators or intrusion response systems would consider an incorrect view of the power system while deciding upon response strategies.

To protect power estimation servers against such targeted attacks, CPIDS identifies maliciously modified measurements, and ignores them while estimating the power system state. Consequently, the resulting power state estimate gives a correct view of the underlying power system as the state estimation is done using correct measurements only. As discussed next, to identify the modified (bad) measurements, CPIDS makes use of the calculated knowledge about the compromised hosts, given the past IDS alerts (Section 5.2).

Bad-Data Detection. Many proposed schemes exist for bad-measurement identification [108]; residual-based approaches [107] are the most widely used techniques against non-malicious *accidental* failures. In summary, those algorithms examine the L_2-norm of the measurement residual $||\mathbf{z} - \mathbf{H}\hat{\mathbf{x}}||$, i.e., the difference between the true measurements $\mathbf{z}$ and the estimated values of the measurements $\mathbf{H}\hat{\mathbf{x}}$ which are calculated using the power system state estimate $\hat{\mathbf{x}}$ and the system matrix $\mathbf{H}$. The measurements with the L_2-norm greater than a certain threshold τ are marked as bad data. However, false-data injection attacks [109] prove the inability of residual-based techniques to handle *interacting* or malicious bad-data modifications [104]. The failure of such techniques results from their heavy dependence on computation of an initial estimate $\hat{\mathbf{x}}$, using *all* the measurements, which may be affected by the bad data.

To identify malicious data modifications, we present a new scalable and combinatorial-based bad-data detection algorithm. The algorithm makes use of the power measurements as well as the cyber security state estimation result, i.e., the posterior distribution over the HMM's state space $Pr(s_t \mid o_{0:t})$ (Section 5.2). The main idea is to circumvent the problem of needing to compute the first power system estimate $\hat{\mathbf{x}}$ from the full data set by initially throwing out the set of suspicious measurements. A trivial solution would be to blindly consider any combination of the sensors to be corrupted, to estimate the power system state for each combination without using measurements from those sensors, and finally to calculate $||\mathbf{z} - \mathbf{H}\hat{\mathbf{x}}||$ to identify the true corrupted measurements. However, that approach is not generally scalable enough to be used in large-scale power systems,

Algorithm 1: Power System Bad-Data Detection Algorithm

```
   Input: P(s_t | o_{0:t}), z, deadline
   Output: [pwr_state, bad_data]
 1 cybr_state;
 2 pwr_state;
 3 bad_data;
 4 ε_m ← 0;
 5 List ← Order_{P(s_t|o_{0:t})}(S);
 6 while get_time() ≤ deadline do
 7     s ← List.pop();
 8     c ← measurement_combination(s);
 9     [z_c, H_c] ← Update_c(z, H);
10     if Observable(z_c, H_c) then
11         x̂ ← Newton_Raphson(z_c, H_c);
12         ε ← ||z − Hx̂||;
13         if ε_m < ε then
14             [pwr_state, bad_data] ← [x̂, c];
15             ε ← ε_m;
16         end
17     end
18 end
```

as M sensors yield 2^M possible combinations. As discussed below, CPIDS uses the posterior distribution $Pr(s_t \mid o_{0:t})$ to order and limit the number of combinations to check.

Algorithm 1 shows the algorithm that CPIDS implements to detect maliciously bad power measurements. The main inputs (Line 1) are the cyber security state estimation result $Pr(s_t \mid o_{0:t})$, the power system measurements, and a timeout threshold for the algorithm. CPIDS initially orders the HMM states according to the estimated posterior probability $P(s_t \mid o_{0:t})$ in a descending order (Line 5). Then, CPIDS iteratively checks different combinations of measurements (Line 6). In particular, the most likely HMM state s is first picked from the list (Line 7). Using the power grid topology, CPIDS knows which measurements could or might have been corrupted, given the set of compromised hosts encoded by s. The set of *potentially* corrupted measurements is stored in a binary vector c (Line 8). To clarify, assuming that there is a total of m measurements, $c_{m\times 1}$ is a binary vector in which 1s and 0s represent bad and good measurements, respectively. For instance, none of the measurements are marked as potentially corrupted in the following measurement combination $c = (0,0,\ldots,0)^T$.

The idea is to throw away the measurements that correspond to the 1 values in c, and proceed with the normal state estimation routine using the remaining measurements. Given the calculated c, rows of the $\mathbf{z}$ and $\mathbf{H}$ matrices that correspond to the 1 values in c are deleted, and the results are saved in $\mathbf{z}_c$ and $\mathbf{H}_c$ (Line 9). Using the dimensionally reduced matrices $\mathbf{z}_c$ and $\mathbf{H}_c$, the power system state is then estimated (Line 11). The state estimate $\hat{\mathbf{x}}$ is used to reconstruct the estimated measurement vector $\hat{\mathbf{z}} = \mathbf{H}\hat{\mathbf{x}}$, which is compared to the actual measurements $\mathbf{z}$ (Line 12). During each iteration of the algorithm, the most deviating $\hat{\mathbf{z}}$ so far and the related values are stored (Line 14). In essence, each iteration (Line 6) checks a specific set of potentially bad measurements to determine whether or not they differ significantly from the values they should have, which are computed based on the remaining (good) measurements. Finally, the procedure returns the best estimates for the power system state, and the set of measurements that were identified as corrupted (Line 1).

One main point in the algorithm is the *observability* condition (Line 10), which checks whether the power system state can be estimated while ignoring a particular subset of measurements c. Otherwise, if too many measurements are compromised and must be removed, the system will no longer be observable, and the algorithm cannot proceed with that particular iteration (Line 10). In general, a necessary requirement for an observable power system is that the number of available measurements be equal to or larger than the number of power system state variables. However, it may be that only parts of the network are observable and some other parts of the system are not observable, even if the total number of good measurements is sufficient. Hence, it is not only important that there be enough good measurements, but also that they come from well-distributed parts of the underlying power system. The entire power system is said to be *observable* if all state variables can be estimated based on the given measurements. Further discussion of observability analysis is beyond the scope of this chapter. The interested reader is referred to the literature concerning measurement placement for observability [108].

It is worth highlighting that Algorithm 1 provides bad-data detection mainly for malicious cases and is a supplement rather than a replacement for residual-based approaches, which are suitable for detecting non-interacting and natural errors. The main reason is that the proposed algorithm is, in essence, a combinatorial-based solution that makes use of cyber-side IDS reports to improve its scalability. In case of natural errors, IDS reports would not provide any useful information, and

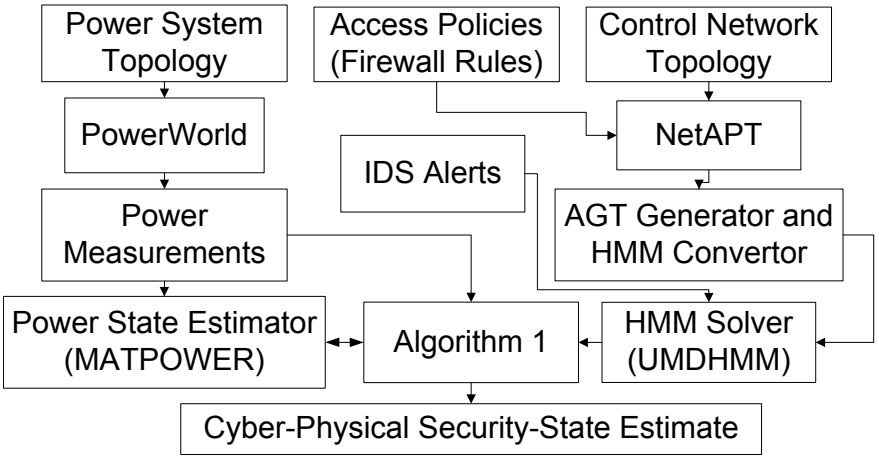


Figure 5.2: CPIDS Implementation Setup

hence the proposed algorithm could not always identify corrupted measurements within a short amount of time. Consequently, the proposed approach and traditional residual-based techniques should be used together to achieve efficient detection of measurement corruptions due to both security attacks and accidental errors.

5.4 Evaluation

In this section, we discuss our implementations and present the experimental results. All our experiments were on a 32-bit system with an Intel Core 2 2.16 GHz CPU, 3.00 GB of memory, and the Windows 7 Professional operating system.

Implementations. Figure 5.2 shows a high-level overview of our implementation setup. A unified XML format was used to describe the network topology and global access control policies. During the offline phase, CPIDS uses the NetAPT tool [110] to perform a comprehensive security analysis of the access policy rules and to produce the network connectivity matrix according to the control network topology input. The matrix is later translated into an HMM model through an AGT generation step (Section 5.2). As illustrated in Figure 5.2, during the online phase, CPIDS feeds the past triggered IDS alerts to an HMM solver (the UMDHMM tool [111]) to solve the HMM model for the posterior distribution and estimate the cyber security state of the system.

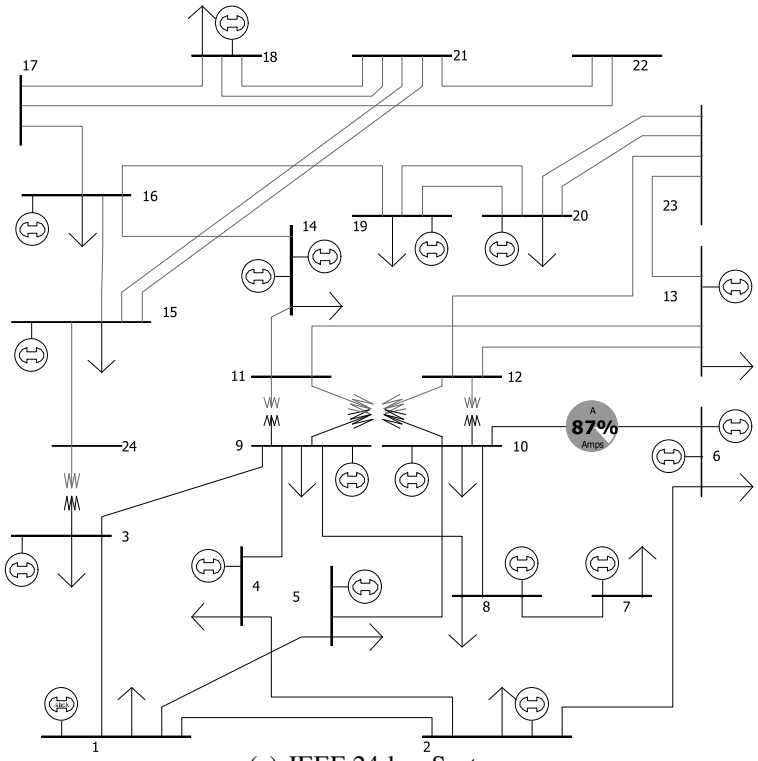
On the power side, we employed the PowerWorld Simulator [112] to simulate the underlying power system, represented by the power system topology input in Figure 5.2. PowerWorld was

used to produce online power measurements and sending them in real-time to the power state estimation component (the MATPOWER Matlab toolbox [113]). To set up a real-time connection to Matlab, PowerWorld used the SimAuto toolbox in Matlab. Finally, the power system state estimate by MATPOWER and the cyber security state estimate by UMDHMM were used by the Algorithm component, which implements Algorithm 1, to determine the cyber-physical security state of the power grid.

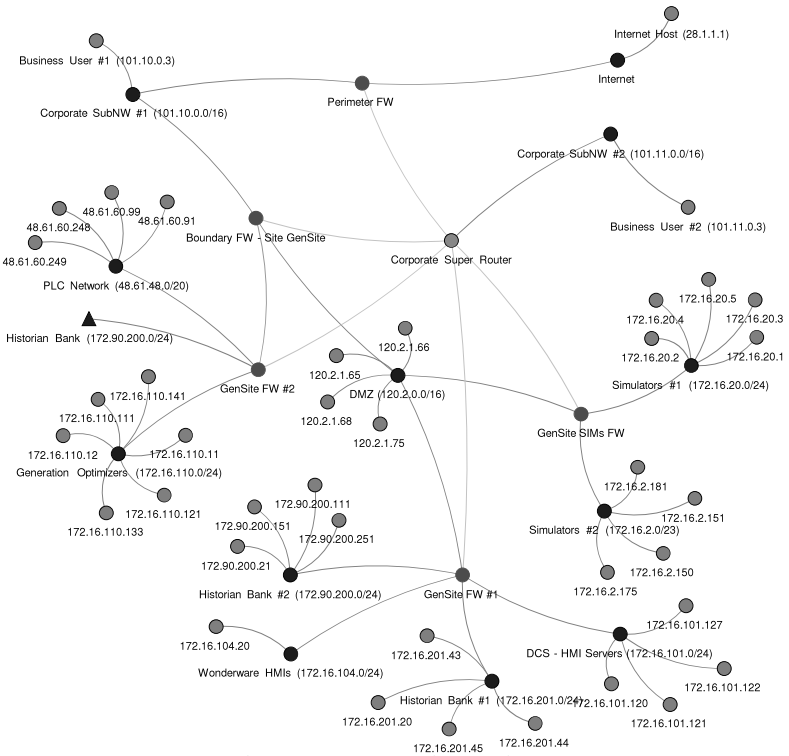
In our experiments, we evaluated CPIDS on a simulated power grid infrastructure. The underlying power system was the IEEE 24-bus reliability test system [114] (Figure 5.3(a)). The power system consisted of 38 transmission lines, and each line had two power sensors on each of its ends, measuring real and reactive power. The power system was monitored and controlled by two control networks with identical network topologies and access control policies. Figure 5.3(b) shows the topology of a single control network that has 59 nodes, e.g., host systems and firewalls. The first control network monitors and controls buses $1-12$ in the power system (Figure 5.3(a)), and the second network monitors and controls buses $13-24$. In particular, each power bus is monitored and controlled by a single host system in the corresponding control network. That mapping is later used by the implementation of the proposed bad-data algorithm to determine which measurements the attacker could have modified, given that he or she has compromised a particular host system.

AGT generation. Given the power network topology and the access policy rules, i.e., 100 firewall rules, CPIDS constructed the network connectivity matrix and generated the corresponding AGT model. Figure 5.4 illustrates a simplified version of the generated AGT. For presentation clarity, only a single host in each network was considered during the AGT generation, and host names on Figure 5.4 are represented by h_x encodings. Table 5.1 shows the mappings between the encodings in Figure 5.4 and the host systems in Figure 5.3(b). As shown on the generated AGT, the attacker initially resides remotely in the Internet with no privileges on the power network (AGT's state 0) and could traverse different attack paths to access a particular host, e.g., h_8, in the power network. Each AGT edge represents an allowed access (i.e., possibly a vulnerability exploitation) from a source to a destination host in the power network.

Cyber security-state estimation. The generated AGT was then converted to its corresponding hidden Markov model (Section 5.2) to allow probabilistic deduction of the attack path that revealed the set of compromised hosts. The generated HMM maintained the same state space and



(a) IEEE 24-bus System



(b) Power Network Topology

Figure 5.3: Experimental Power Grid Testbed Architecture

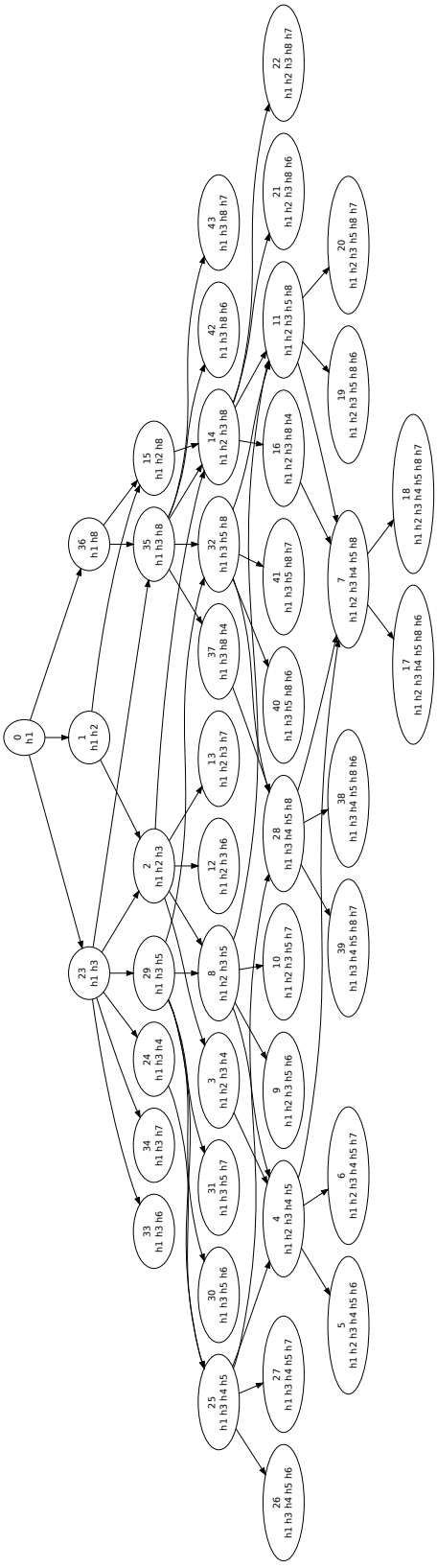


Figure 5.4: Automatically Generated AGT for the IEEE 24-bus Power Control Networks

Table 5.1: (IP, host) Mappings from Figure 5.3(b) to Figure 5.4

Host	**IP Address**	**Host**	**IP Address**
h_1	28.1.1.1	h_2	101.10.0.3
h_3	120.2.1.65	h_4	172.16.101.122
h_5	172.16.104.20	h_6	172.16.201.45
h_7	172.90.200.251	h_8	101.11.0.3

Table 5.2: Cyber Security-State Estimation Results

Probability	**HMM's State ID**	**Compromised Hosts**
0.032141	0	28.1.1.1
0.953099	1	28.1.1.1, 101.10.0.3
0.001001	15	28.1.1.1, 101.10.0.3, 101.11.0.3
0.016759	36	28.1.1.1, 101.11.0.3

namings as the AGT illustrated in Figure 5.4. Then, we launched an attack to compromise the host system 101.10.0.3 within the power network. The attack caused the IDS, which was monitoring the host's incoming traffic, to trigger an alert. The UMDHMM tool used the generated HMM and the triggered IDS alert to estimate the cyber security state. Table 5.2 shows the probability distribution over the HMM's state space. The most likely current state in HMM marks the host systems 28.1.1.1 and 101.10.0.3 as compromised. From the cyber-physical network's topology input and given the compromised hosts, CPIDS marks the real and reactive power measurements on transmission lines $1-2$ and $16-17$ as potentially corrupted[1].

Bad-data detection. We evaluated how efficiently the proposed bad-data detection algorithm performs compared to the traditional residual-based approaches.

The first attack modified the measurements from a single real power sensor on the $1-2$ line after compromising a critical power network host. Figure 5.5 shows different parameters observed after we ran both of the bad-data detection algorithms. The vertical axis shows the real power per-unit values for the 38 power system sensors (indicated by the horizontal axis). For each sensor, four values are reported. The first column shows the actual (correct) measurements from the PowerWorld Simulator before they were maliciously modified, as reflected in the second column. The third column shows the measurements estimated using the proposed framework, which used the cyber-side intrusion detection (ID) information. The last column reports the measurements

[1]"$1-2$" denotes the power system line that connects bus 1 to bus 2.

estimated using the traditional residual-based approaches. As shown in the figure, during the first scenario, only the measurement from the first sensor on the $1-2$ line was corrupted 1 p.u. before being sent to the estimation server. The proposed ID-based solution's estimation of the first sensor's measurement 0.1224, was almost equal to its correct value, 0.1247 (i.e., with only 0.002 p.u. difference), and hence far from its modified value, resulting in a large measurement residual (i.e., 1.002 p.u.). The residual-based algorithm was also able to detect the data corruption, as its calculated measurement residual value, 0.502, was above the predefined threshold ($\tau = 0.1$ p.u.). However, its estimated value was not as accurate as that of the ID-based algorithm. The estimated value was affected by the corrupted value that was wrongly considered good and used by the state estimator during the residual-based approach's first power state estimation.

The second attack aimed to cause non-interacting measurement modifications on two power sensors. In particular, measurements from the bus 1 sensor on the $1-2$ line, and from the bus 16 sensor on the $16-17$ line, were corrupted. Thus, the corruptions were both 1 p.u., and were not intentionally designed to match the underlying power system equations. In practice, such non-interacting bad data usually result from non-malicious natural and accidental failures. Figure 5.6 shows the measurement estimations resulting from rach of the two algorithms. Much as in the case above, the proposed algorithm and the residual-based approach were both able to detect the data corruption. However, the residual-based approach did not estimate the power system measurement on the compromised sensors accurately.

Finally, during a more complicated attack scenario, the attacker intentionally modified two measurements from sensors 1 (on bus 1) and 14 (on bus 1), which were monitoring the two ends of the $1-2$ power line. The data modifications were intentionally designed in such a way that they still satisfied the power flow equations. In particular, the measurement corruptions on sensors 1 and 14 were $+1$ and -1 p.u., respectively. Figure 5.7 shows the results for the interacting measurement corruption scenario. The proposed ID-based algorithm was still able to detect the measurement corruption by ignoring the set of potentially bad measurements. However, the locally consistent bad measurements deceived the residual-based approach into wrongly marking those measurements as correct, since the measurement residual value was 0.002, i.e., below the predefined threshold τ. When such malicious interacting measurement corruption attacks occur, residual-based approaches do not necessarily give an accurate estimate, even if redundant mea-

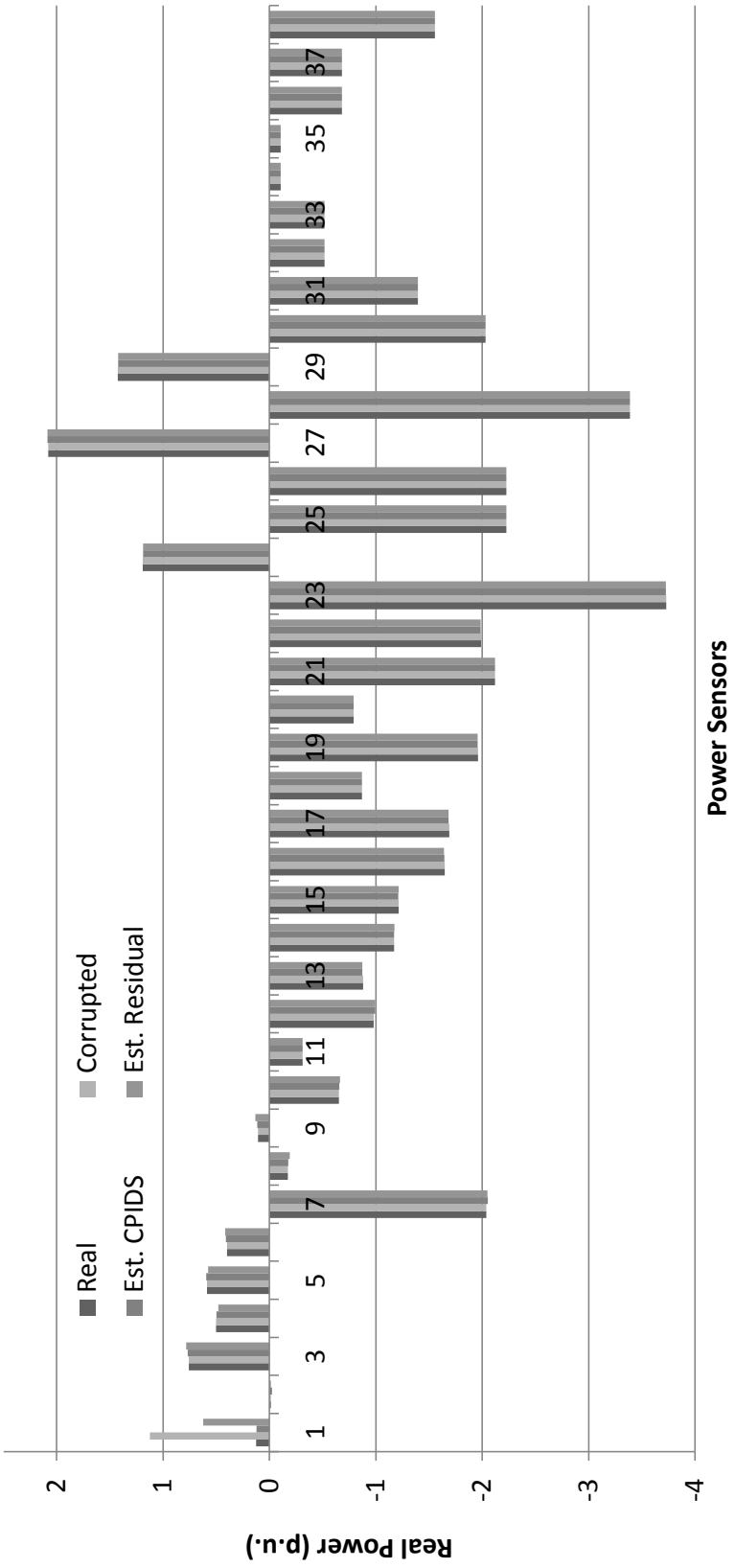


Figure 5.5: Single Measurement Corruption

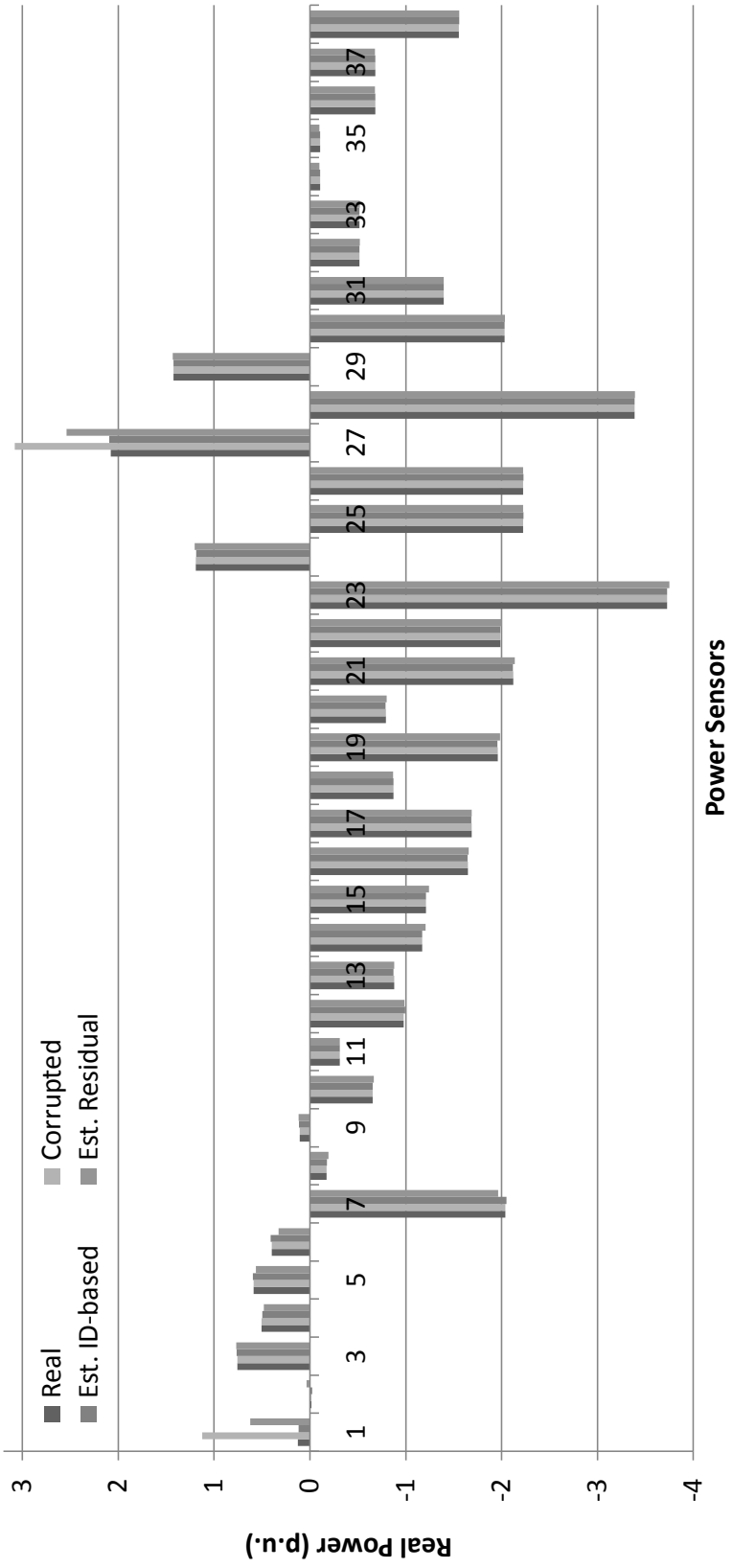


Figure 5.6: Multiple Non-interacting Measurement Corruption

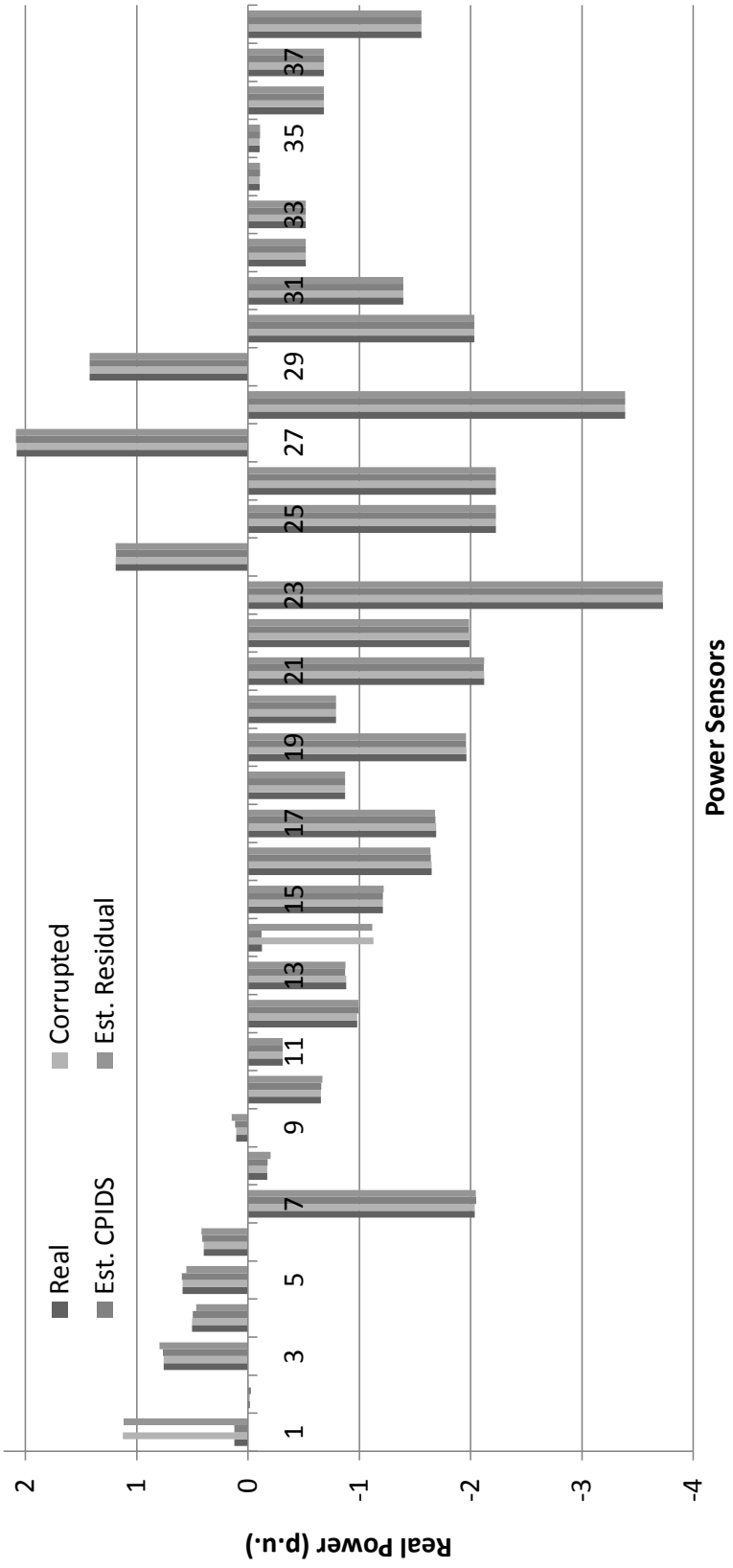


Figure 5.7: Multiple Interacting Measurement Corruption

surements are deployed.

5.5 Related Work

Recently, there has been increasing interest in security incident detection in power-critical infrastructures [115]; however, most of the past work has focused on either the cyber or power side of the power grid [116, 117].

The first group of techniques focus on dependability and security analysis of power systems [105]. Volkanovski et al. [118] introduce a power system reliability analysis algorithm using fault trees generated for each load point of the system. The proposed method focuses only on accidental failures due to natural causes, and hence does not consider maliciously failed power components. Zhou et al. [119] present a sequential power system state estimation algorithm using the reports from synchronized phasor measurement units. Lo et al. [120] propose a power system bad-data detection algorithm based on rotation of measurement order for sequential state estimation. Both the Zhou and Lo techniques completely ignore the cyber network topology, which might be the root cause of the problem; hence, they do not often consider compromises on the cyber side [121].

On the other hand, cyber-based diagnostics mechanisms try to estimate the security state of a computer network [122]. Ten et al. [123] propose a vulnerability assessment technique to evaluate the vulnerabilities of SCADA systems at three levels: system, scenarios, and access points. Mohajerani et al. [124] introduce a method to detect and improve the vulnerability of power networks against the intrusion and malicious acts of cyber hackers by calculating the risk of each asset being compromised. Both the abovementioned frameworks perform in an offline manner, and hence cannot monitor the system for malicious activities while it is in its operational mode. Wilken et al. [125] propose a software fault diagnosis solution that uses data redundancy to detect faults that have been caused by probabilistic system failures [126]. Therefore, software crashes that result from vulnerability exploitations cannot be completely detected using the proposed approach. In cyber systems, there have been extensive investigations into intrusion detection techniques such as anomaly-based [127], signature-based [128], and (recently) specification-based solutions [129]. However, those traditional cyber diagnostics solutions completely ignore topology and configura-

tion of the underlying physical power system [130].

Cardenas et al. [131] investigate an intrusion detection technique for process control networks in which the attack's final target is assumed to be given. This assumption could simply be exploited by the attackers to further damage the process control network by targeting other critical goals. CPIDS, while generating the attack graph, considers all possible attack paths, even including those that do not end up in a critical asset, e.g., an internal Web server. Furthermore, unlike the previous detection techniques, CPIDS takes into account the cyber security sensory reports to improve scalability of the bad-data detection algorithm in the underlying power system.

5.6 Conclusions

The purpose of seeking improved system state knowledge from the data is to facilitate better situational awareness and, ultimately, better system control. By using available data as part of the proposed cyber-physical intrusion detection system, we are accomplishing a step towards that goal. In this chapter, we presented CPIDS, the cyber-physical security state estimation framework, to identify malicious activities and accurately estimate the cyber-physical security state of a power grid. CPIDS exploits all the available offline information, like power network access policies, to create a comprehensive model of the cyber-physical system. During operational mode, CPIDS makes use of the available online information from both the cyber security sensors and the power measurements and efficiently fuses all the information using the generated system model. The experimental results show that CPIDS can efficiently estimate the cyber security state of the system, identify malicious measurement corruptions, and, consequently, calculate a correct state estimate of the underlying power system.

CHAPTER 6

CASE STUDY: ENTERPRISE NETWORK

Having already discussed some of the main challenges of designing an intrusion response system, in this chapter we describe a proof-of-concept implementation of an extended version of the game-theoretic intrusion Response and Recovery Engine (RRE), which we presented in Chapter 2, in an enterprise network environment. In such an environment, business metrics play a critical role in determining the best IT-level system setup and reconfiguration options to achieve an organizational business-level goal. We focus on how to map pure IT-level measures to higher-level business metrics, so that RRE considers the business health of the whole system while deciding upon the optimal response and recovery actions.

Optimizing business value in the presence of IT system-level attacks is a very complex problem for three reasons. First, it requires understanding of how the various business processes are mapped to the underlying IT resources. Until recently, this required extensive design documentation, e.g., design specifications. In practice, such documentation is not only rare but, even if present, difficult to keep up-to-date. With the advent of automated discovery tools for fine-grained application dependencies, like Galapagos [132, 133], the ability to perform this mapping for large-scale IT infrastructures is only now starting to become a reality. Second, optimization of business value requires understanding and quantification of the impact of violation of the dependability and security (i.e., business health) supported by the IT system components that underly the business processes. Third, it is inherently more difficult to optimize the business value under security attacks than to achieve other system goals, such as performance and availability of a server, because a single security attack, such as one resulting in a compromised credential, may have diverse implications for the business and may affect multiple business processes.

In this chapter, we present a framework for managing business health of an enterprise despite security attacks In particular, the framework addresses situations in system administration that in-

volve multi-objective decision-making while taking into account both business-level and IT-level metrics. The attack-response tree (ART) formalism [116] is extended and used to translate underlying IT-level measures into high-level business metrics, thereby enabling assessment of overall business health. The proposed framework is then validated, in the context of an online enterprise model executing multiple business processes, against three high-impact attack classes that are common for such enterprises. Experimental results, given later in the chapter, show that considering business-level metrics in addition to IT-level metrics improves the decision quality in selecting response and recovery actions, such that the system's high-level business health is recovered faster because more appropriate actions are taken first.

6.1 Related Work

The related work falls into two categories: 1) intrusion response techniques, and 2) techniques for operational risk quantification and control. Our work draws from both categories, but to our knowledge, it is the first work that unites the concepts in those categories.

Intrusion-response systems were generally focused on handling disruptions to the IT infrastructure components, but do not account for business-level implications. Consequently, these systems may choose responses that may well mitigate the effects of an attack on individual hosts, but, in doing so, may worsen the overall business health.

Work on operational risk has delved into the financial impact of disruptive events occurring at the IT system-level, but the main focus has been on events that affect resource availability. While that may suffice in the context of benign failures, it is inadequate in the context of security attacks. Systems for handling intrusions in enterprise IT infrastructures can be broadly categorized into intrusion prevention, intrusion detection, and intrusion response systems. An effective enterprise strategy for security needs to integrate and coordinate all three approaches.

While significant research has been performed in the last decade on effective techniques for handling intrusions, until recently, much of that research has focused on intrusion detection and intrusion prevention approaches. Recent works on intrusion response focus on automated mechanisms to minimize the damage inflicted to the system by the attacker (e.g., [36, 32, 11, 29, 134, 31,

30, 37]). These works can be categorized based on the response selection method into static, dynamic, and cost-sensitive systems [10]. In static intrusion response systems, there is a predefined set of responses for each possible intrusion alert. These systems suffer from a lack of flexibility (since response selection is not adaptive) and scalability (since there may be numerous possible intrusion alerts in the system). Dynamic intrusion response systems address those drawbacks by choosing response, based on attack characteristics, like severity of the attack, and on intrusion detection system (IDS) characteristics, like confidence in the accuracy of the IDS. However, they still do not consider the cost of responses or the damage caused by the intrusion during response selection. Cost-sensitive intrusion response systems take into account such factors.

The effects of IT system component failures on business costs have been investigated in the context of operational risk. Supatgiat et al. [135] define a general cause-and-effect model and decomposition techniques, and use them to simulate loss distribution functions. The model has been applied to business models and could be applied to attacks. However, it does not address dynamic responses to a current situation. The work also does not offer methods to systematically derive cause-and-effect chains from business and IT system diagrams such as one might obtain from typical documentation or discovery technologies. Cheng et al. [136] have considered business processes and their direct resources (IT and other) as they can be modeled in advanced business process modeling tools. Losses occur when certain tasks are not available. The authors derive closed-form solutions to the loss distribution under certain resource failure arrival rates. The work does not consider dynamic reactions to a current situation, or malicious attacks, which can spread by paths other than standard business dependencies.

6.2 Case Study

To motivate and validate our business-value-driven intrusion response framework, we consider a typical web-based retail company architecture, called “Widgets R Us.” In this section, we describe three of the company’s business processes, its IT infrastructure, and how its business processes are mapped to its IT infrastructure. We also describe various attack scenarios.

Business Processes. We consider three business processes for ”Widgets R Us”: order man-

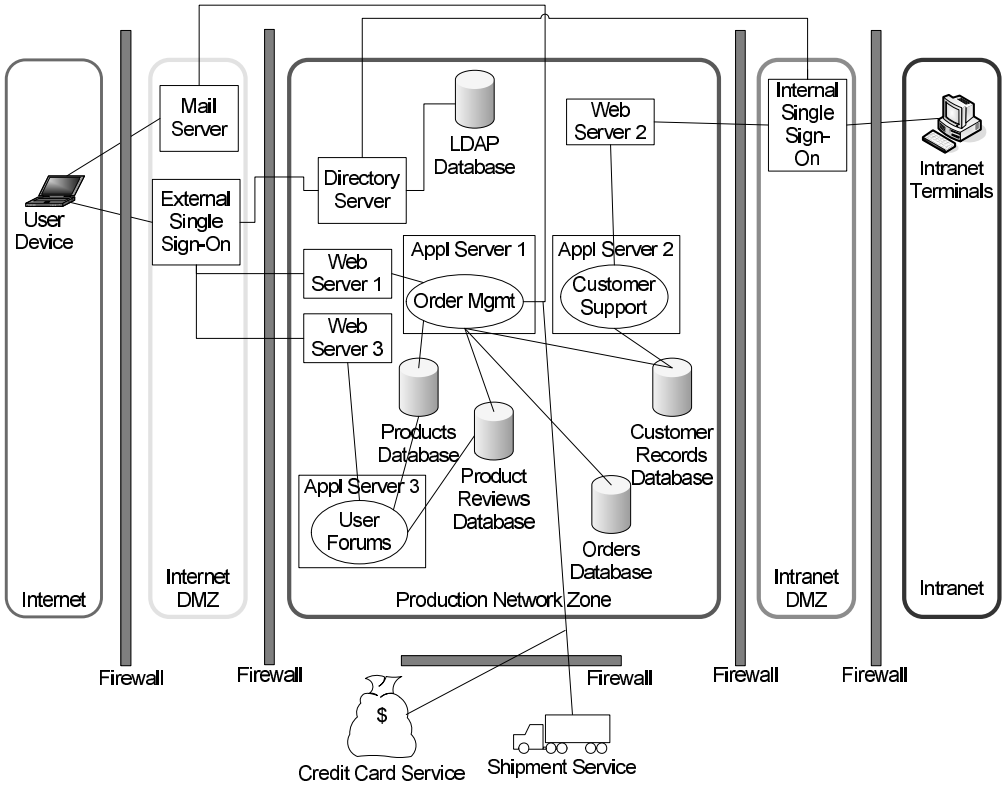


Figure 6.1: Enterprise IT Infrastructure

agement, customer support, and customer forum. The business processes are shown in Figure 6.4 using the business process modeling notation (BPMN) 1.1 standard [137]. The order management process (Figure 6.4(a)) is an abstraction of the typical high-level activities required to support a web-based retail transaction. The process includes activities performed by external service companies: a credit card verification service and a shipping service. For simplicity, the customer support process (Figure 6.4(b)) is limited to an abstraction of phone-based customer support. The process activities include answering general product questions and creating/updating customer account information. The customer forum process (Figure 6.4(c)) represents activities enabling customers to view, update, and discuss product reviews. When a customer browses the product catalog during the order management process, the order in which the products are presented to the customer is based, in part, on the user reviews.

These three retail-based business processes vary in the business value they provide to the company and, therefore, will have different security priorities associated with them. The order management process is the most critical. The company can incur significant costs associated with

stolen information, for example credit card information [138]. The confidentiality, integrity, and availability of all the entities associated with the process are vital to the company's reputation and income; disruptions to the process have direct revenue impact. The customer support process is important, but less critical from a direct revenue impact standpoint than order management. Customer support is, however, important from a reputation standpoint. Since customer information is accessed and manipulated during the process, the availability and integrity of this information are areas of concern. The customer forum process is helpful to potential customers, but is the least critical of the three processes from a business value standpoint. Since product reviews affect product rankings, maintaining the integrity of reviews is a concern.

IT Infrastructure. Illustrated in Figure 6.1 are the various artifacts of the "Widgets R Us" IT system architecture. It is a canonical three-tier architecture, partitioned into multiple network zones for defense-in-depth. The production systems reside in a production network zone, separated from the Internet by a demilitarized zone. The production network zone is also separated from the company intranet. This separation restricts access to the production system to only authorized, internal users; for example, senior administrators that perform application server updates and maintenance.

There are two web servers connected to an external single sign-on (SSO) server: one supports the web content for the order management process, and the other supports the customer forum process. A third web server, accessible only from the company intranet, is connected to an internal SSO server. This server is used by the customer support process. Both of the SSO servers perform the roles of authentication proxy and reverse HTTP proxy. Internal and external web requests are first directed to their respective SSO servers. The SSO server examines the URL in the user's web request and determines to which of the web servers the request should be forwarded.

The three business processes are executed on separate application servers. Customer records, customer orders, product records, and product reviews are stored in separate databases with each database running on a dedicated server. An LDAP user registry and authorization database stores information about users (both internal and external), groups, and roles and authorizations (for example, which user entity can perform what operation on which IT system component). A directory server is connected to the LDAP database. The SSO servers interact with the directory server to check a requested URL against LDAP and authenticating the user as required. Lastly, there is a

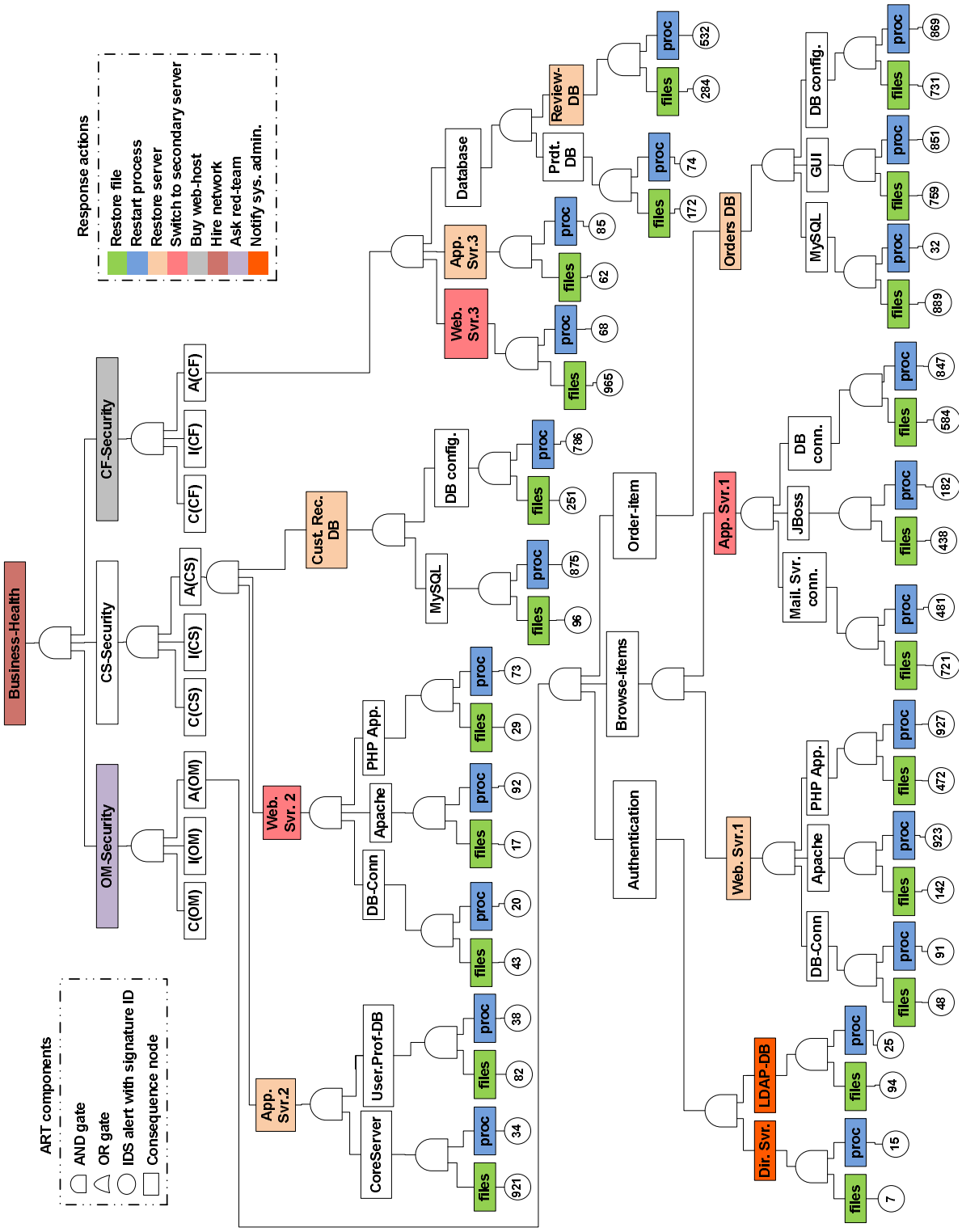


Figure 6.2: Partial ART for Sample Attacks on the IT Infrastructure of Widgets R Us

mail server that receives, routes, and delivers email.

Business-to-IT Mapping. Identifying what IT system components are needed to support each business process activity is a complex undertaking. Previous work (involving some authors of this chapter) [132] describes how such a mapping can be performed in an enterprise setting. The first step is the identification of the IT entry points for each business activity. IT entry points are system components (for example, a server, an application or middleware) by which a human, external contact, or machinery access the IT infrastructure. Typically, these points correspond to the inputs and outputs of the IT system. For example, the web server in Figure 6.1 is the IT entry point for the product catalog browsing activity of the order management process. The second step is to identify the dependencies that each IT entry point has on other IT system components. Automated discovery systems such as Galapagos [133] and TADDM [139] can be used to discover fine-grained dependency information between applications, middleware, and data. In the case of the catalog browsing activity, such a discovery system will indicate that the web server forwards requests to an application server, which in turn accesses data from a database server. Thus, the product catalog browsing activity depends on three IT components.

Using the aforementioned mapping approach, the three business processes in Figure 6.4 are mapped to the IT infrastructure in Figure 6.1 as follows: The order management process is mapped to Web Server 1, Application Server 1, Orders Database, Customer Records Database, and Product Reviews Database; the customer support process is mapped to Web Server 2, Application Server 2, and Customer Records Database; and the user forum process is mapped to Web Server 3, Application Server 3, Products Reviews Database, and Products Database.

Threat Model. We evaluate the effectiveness of our framework in the context of three attack classes: (1) user/admin credential compromise, (2) network sniffing, and (3) file system compromise. All three attack classes are prevalent, as evident in our incident analysis of IDS logs collected over the last five years at NCSA [12]. These three attack classes cover a broad range in parameters such as likelihood of their success, their potential disruption to the IT systems, and the cost to the company. Additionally, each of these attack classes could serve as a launch pad for several other attacks.

Credential Compromise: Attacks seeking user credentials are common. Supplemented with easily available local root escalation exploits (e.g., CVE-2009-2692 [140]), stolen user credentials

can help attackers obtain total control of the systems inside the network.

Network Sniffing: Once inside an enterprise network, the attacker can install a network sniffer [141] to help gather passwords for instant messengers, POP, IMAP, VNC, and similar services. Armed with such sensitive information, the attacker can carry out even more damaging attacks, e,g., session hijacking on HTTP traffic and gaining access to employee emails.

File System Attacks: Certain attacks on web-based ordering systems target the file system from which the web server is distributing web pages. For example, a vulnerability in the shared file system (e.g., NFS) may allow a remote attacker to read/write contents in a directory with user or root privileges [142]. Such an exploit would allow the intruder to run a fake web site on a genuine corporate server and lure customers into providing sensitive data.

Attack-Response Tree. Figure 6.2 shows the partial business-level ART for the Widgets R Us enterprise. The full ART considers confidentiality, integrity, and availability of each of the three business processes. However, due to space limitations, the figure shows only the availability aspect. The root node of the ART, viz. "Business Health", represents the top-level business goal. We have used abbreviated names for the node names in the ART. For instance, OM, CS, and CF denote the order management, customer support, and customer forum business processes respectively. C(OM), I(OM), and A(OM) refers to the confidentiality, integrity, and availability of the order management business process. The sub-trees for individual business processes is composed of sub-trees representing IT system components (e.g., web server) necessary for that process, which may in turn be composed of nodes representing sub-components. The hierarchical composition continues until the leaf consequence nodes, whose violations are detectable by intrusion detection system (IDS) alerts represented as circles tagged to those nodes. The leaf consequence nodes are concerned with the confidentiality, integrity, and availability of two types of OS-level objects, files and processes. Different response action types are color-coded differently. For example, the action "Restart Application Server 3 process" (attached to leaf consequence node 85) is encoded using blue. We use both network and host-based IDSes whose alerts notify RRE about different types of events in the network or files and processes in a system. For instance, an alert from OSSEC [143] is attached to the ART's leaf node 94 (shown in the figure) and informs RRE if "LDAP-DB files are modified."

$$V(s) = \min_{r \in R} \Big\{ \sum_{s' \in S} T(s,r,s').\{Cst(r) + Dmg(s,s') + \\ + \sqrt{\lambda}.\max_{a \in A}[\sum_{s'' \in S} T(s',a,s'').\big(Dmg(s',s'') + \sqrt{\lambda}.V(s'')\big)]\} \Big\}$$

Figure 6.3: The Dynamic Programming Equation to Solve for the Optimal Response Action

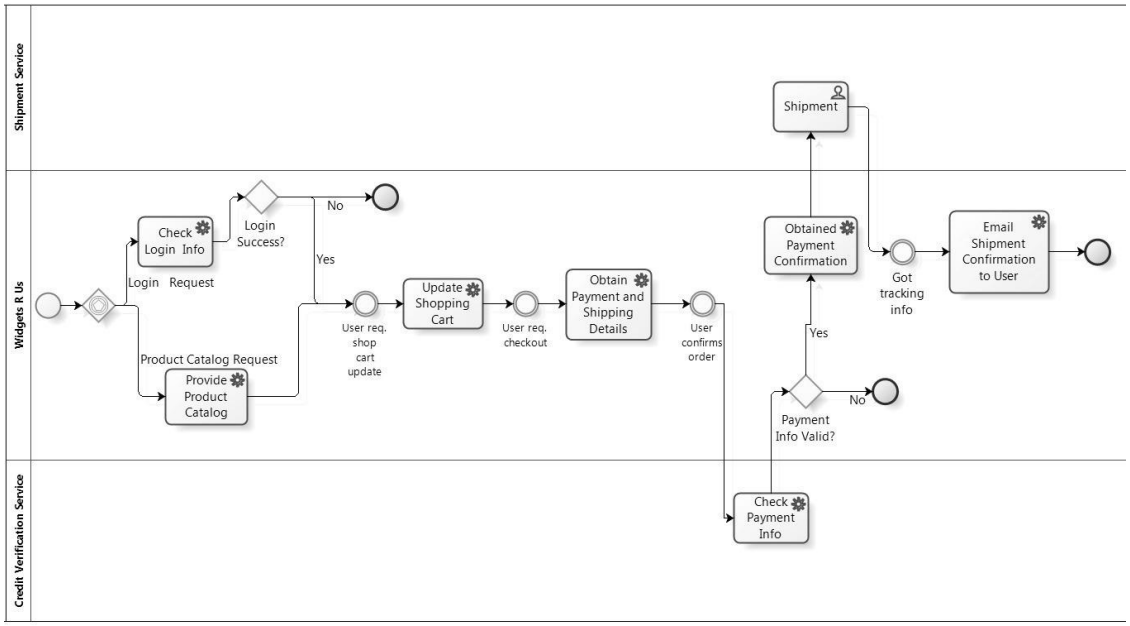
Table 6.1: Cost per One Hour Downtime [2]

Industry Class	Cost
Brokerage operations	$6,450,000
Credit card authorization	$2,600,000
Ebay	$225,000
Amazon.com	$180,000
Package shipping services	$150,000
Home shopping channel	$113,000
Catalog sales center	$90,000
Airline reservation center	$89,000
Cellular service activation	$41,000
On-line network fees	$25,000
ATM service fees	$14,000

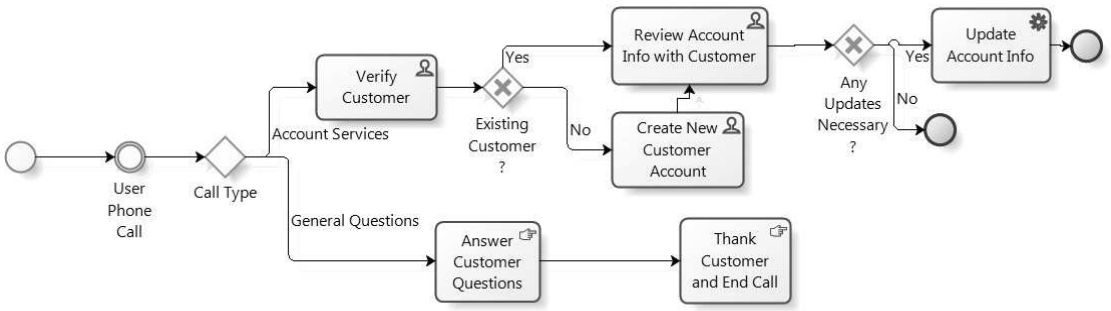
6.3 Business Health Management Framework

In this section, we describe how a given IT system-level RRE model is augmented to include business-level considerations. Using business-level metrics in conjunction with standard IT system-level metrics allows RRE to prioritize actions that would result in optimizing business health, e.g., by keeping alive the most critical business processes and minimizing the user-visible service interruption. From an IT system perspective, the integrity of two database files may be equally important. But if one contains client credit card information while the other contains users' product ratings, they represent different business criticalities. Factoring that information as business-level metrics into the RRE helps prioritize the recovery of the former over the latter. Towards that end, we extend the basic RRE to consider both IT system and business aspects of the system when determining responses to malicious attacks.

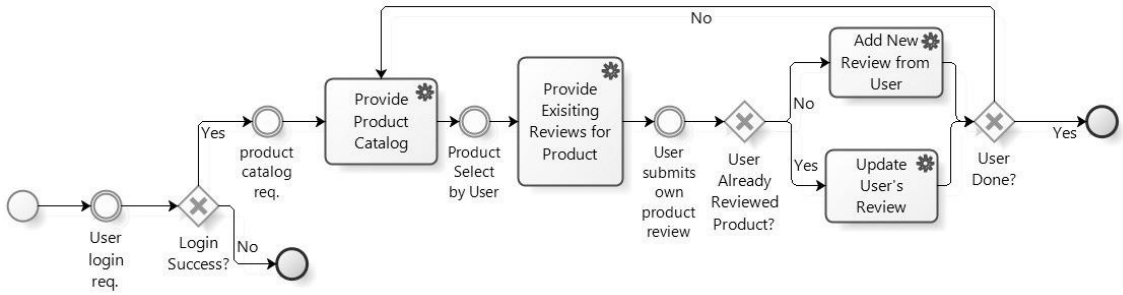
As in our previous work that considered only IT-level metrics (see Chapter 2), we formulate the cost function of RRE using dynamic programming. The equation in Figure 6.3 is a customized ver-



(a) Order Management Process



(b) Customer Support Business Process



(c) Customer Forums Business Process

Figure 6.4: Three Main Business Processes of Widgets R Us

Table 6.2: Cost per Compromised Record [3]

Industry Class and Costs					
Energy	Financial	Services	Research	Healthcare	Pharma
$237	$249	$256	$266	$294	$310

sion of Equation (2.6) to consider business costs, and formulates the cost function in RRE updated to include business-level metrics. Here, S is the state space; $V(.)$ is the value function used to solve for the optimal response action; T is the state transition probability function. The action set is partitioned into two subsets: R represents the set of response and recovery actions, and A represents the set of adversarial actions. *Dmg* or the damage function represents immediate business-level cost for each state transition $s \rightarrow s'$, which purely depends on the subset of consequences caused by the attacker as the result of the transition. *Cst* denotes the cost of taking a particular response action.

We now describe how the response action recommended by the RRE is the optimal one from a business perspective. The equation in Figure 6.3 recursively calculates the optimal cost function for each state s. The value function $V(s)$ is first initialized to zero for all $s \in S$; then, the value iteration algorithm [116] is employed to evaluate the correct value of the V function. The optimization involves two stages: the move of the attacker $a \in A$, and the move of the responder $r \in R$. At each step of the game s, RRE takes an action r making a transition to state s' from which the attacker takes next action $a \in A$ and makes a transition to state s''. The optimal response action is picked by minimizing over the response actions ($\min_{r \in R}$) the maximum possible damage caused by the attacker in next step (by maximizing over the adversarial actions in state s' using $\max_{a \in A}$). The $\lambda \in [0,1]$ is the a fixed discount factor, e.g., $\lambda = 0.9$, to put more weight on present actions compared to the actions taken in future [116].

For a given business, the damage function *Dmg* depends on the primary service type provided by the business. Studies by others have attempted to provide estimates for average cost of downtimes and data breaches for different industry classes [2, 3, 138]. For example, Table 6.1 shows the average cost of downtime for different businesses as estimated in [2]. In addition to downtimes, expectations of trust and privacy drive the costs of data breaches even higher. Table 6.2 shows the average cost per record compromised for each class of industry [3].

We now show how the numbers provided by studies such as [2, 3, 138] may be used to provide values for variables in the equation in Figure 6.3. Suppose that the current state is s, and that there are three possible response actions $R = \{r_1, r_2, r_3\}$ from which the optimal action should be selected. Each response action r is associated with a positive cost $Cst(r)$. Suppose that the system state s represents an online shopping Web server being down and unavailable, and response action $r_2 =$ *Restart the Web Server* (resulting in transition to state s_2) fixes that problem in 30 seconds (or 0.0083 hours). If the restart can be effected remotely by automatic means (involving no operator time), then we have $Cst(r_2) = 0$ and $Dmg(s, s_2) = 0.0083 \times \$90000 = \$750$, where \$90000 is the cost per hour downtime for a catalog sales center type of business (from Table 6.1). Similarly, for an attacker action $a_1 =$ *Compromise database* on a database that stores 50 customers' credit card records, resulting in transition from state s' to state s'_1, we will have $Dmg(s', s'_1) = 50 \times \$305 = \$15250$ where \$305 is the cost per breached credit card record (obtained from [138]).

We also modify the ARTs in the basic RRE [116] to account for business-level metrics. In the basic RRE, each business process in the enterprise would have its own set of ARTs corresponding to the IT-level components underlying the business process; we call these *IT-system level ARTs*. Each IT-system level ART would have been treated independently by the RRE to get its own corresponding optimal response action. We augment that model by introducing a *business-level ART*, which estimates the overall business health of the enterprise and is composed of *IT-system level ARTs* as follows: each leaf node in the business-level ART corresponds to a root node of an IT system-level ART. The value of a root node of an IT system-level ART indicates how likely it is that that component is currently secure. More specifically, it indicates violation of a security property in a corresponding IT system-level component of the enterprise. These values are later used to set the values of leaf nodes in the business-level ART. The overall business-level health of the enterprise is represented by root node of the business-level ART, and its value is calculated according to a bottom-up logical propagation of the values in its leaf nodes. Figure 6.2 shows a partial business-level ART for the case study provided in Section 6.2.

We update the response action set in the ARTs to include business- as well as IT system-level responses. For instance, business-level responses for a server intrusion may include hiring a red team, posting a "sorry, our web ordering is down" notice on the server, and sending coupons to clients. Additionally, the cost function in RRE is updated to address not only IT system-level

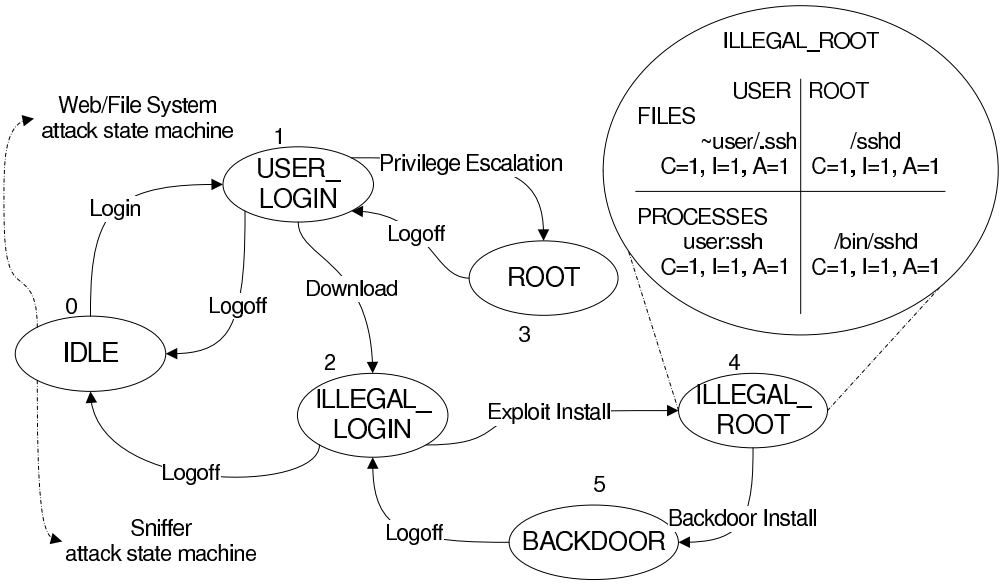


Figure 6.5: Credential Compromise Attack

```
Current privilege level: user USER_LOGIN → Download Exploit State → ILLEGAL_LOGIN
Choices: [4: EXPLOIT_INSTALL, 1: LOGOFF]
IDS alert: Jul 9 08:28:10 vv.ww.xx.yy:44619 → 195.22.100.56:80; GET /.0/ptrat.c 200
'OK' server5.xuna.nl
```

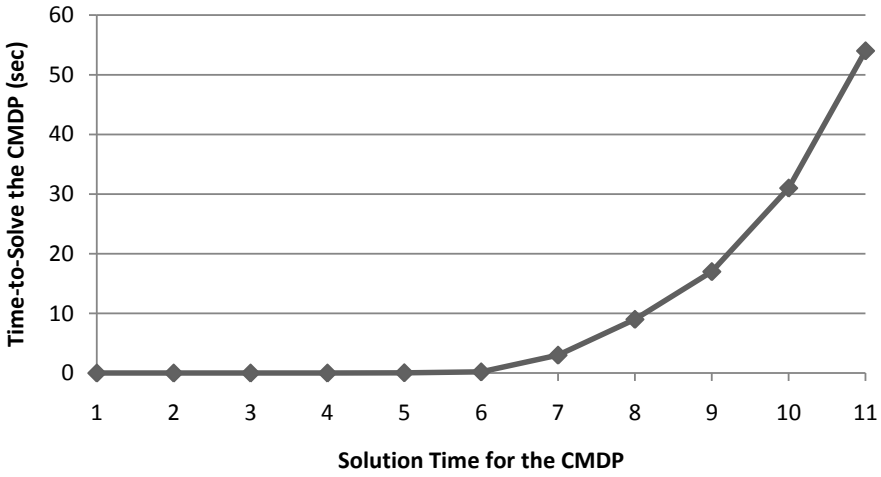
Figure 6.6: IRE Output: IDS Alert for Exploit Download

responses but also business-level responses that the core response engine can choose from; in other words, responses tagged to business-level consequence nodes (e.g., "order management business process is down.") in ARTs also have associated costs assigned to them.

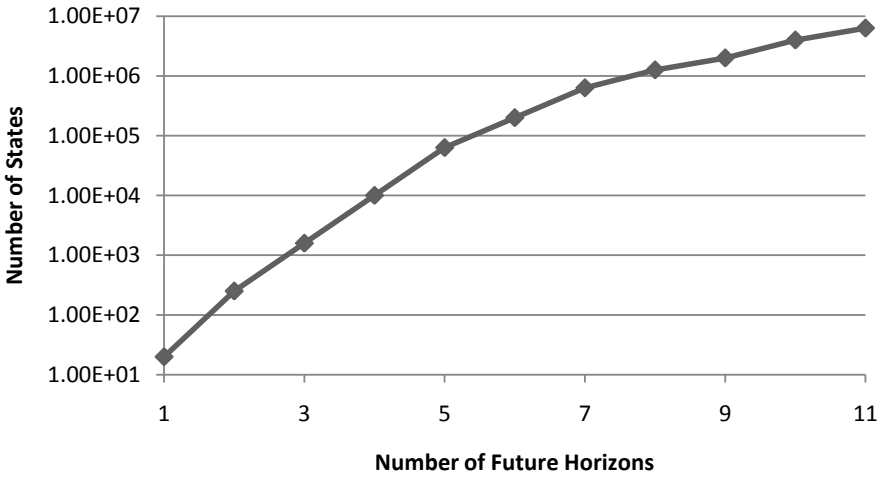
6.4 Implementation and Experimental Evaluation

We evaluate how considering business-level security metrics in conjunction with IT system-level metrics affects decision quality in terms of the cost of recovery after an attack. RRE is currently implemented for both Windows (C#.Net) and Linux (C/C++). All experiments were conducted on a system with a 2.2 GHz AMD Athlon 64 Processor 3700+ with 1 MB of cache, 2GB of memory, and the Ubuntu OS (Linux 2.6.24-21).

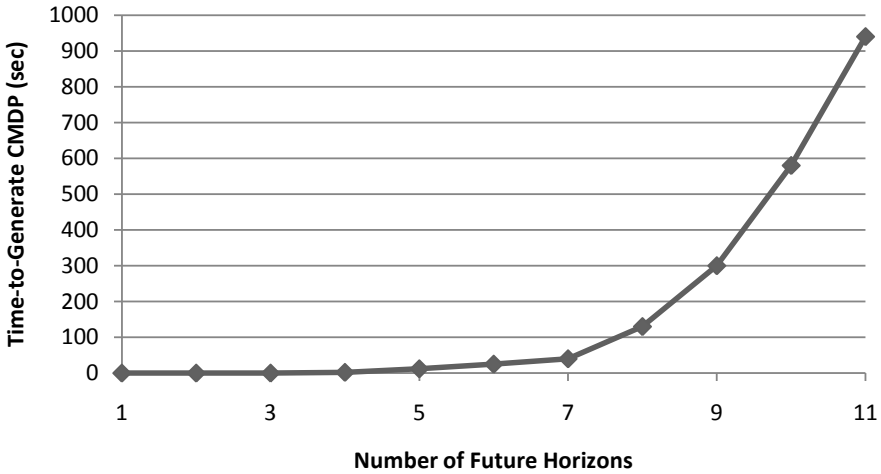
Incident Replay Engine (IRE). We constructed an incident-replay engine (IRE) based on attack patterns and monitor alerts obtained from the analysis of 150 attack incidents observed at NCSA



(a) Solution Time for the Competitive Markov Decision Process



(b) State-Space Size



(c) Time to Generate the Competitive Markov Decision Process

Figure 6.7: RRE Performance

(UIUC) over five years [12]. In essence, IRE mimics the world outside the RRE. At the heart of the IRE are sets of finite state machines. Each finite state machine (FSM) is an attack model for incidents belonging to the same attack class. For example, Figure 6.5 shows the state machine model for incidents grouped under credential compromise, which accounts for 26% of all analyzed incidents. Each node in the FSM depicts an IT system-level state. The arcs (transitions) are the actions taken by a genuine user or an attacker, or alerts output by the monitoring system. Some transitions can be performed in multiple ways, e.g., an attacker can use different exploits to achieve his/her goal. A specific alert is associated with each transition. While replaying incidents, IRE also accounts for deficiencies of the real-world monitoring infrastructure by generating false positives.

Building the FSM entails the following steps: (1) identification of all alerts associated with a given category of incidents; (2) identification of distinct system/application states traversed by the attacker in penetrating the network; and (3) assigning conditions to transitions between the system states identified in step 2.

IRE emulates the actions of both legitimate users and attackers using a state-based Markov model. It generates actions stochastically based on an exponential distribution as in the TPC-W benchmark [144]. Each action is associated with a set of pre-conditions and post-conditions; pre-conditions need to be met before the action can be taken, and post-conditions represent the action's effects on the system state variables. For instance, if IRE generates the attacker action "remove the `/etc/passwd`" (which requires root access), and the attacker only has user access privileges, then IRE simply ignores the action. On the other hand, if the attacker has already obtained root access, then the IRE updates the system state to reflect the removal of the file. At the end of each step, IRE doubling as a detection system, also sends reports to the RRE regarding attack incidents in the system. The reports that IRE sends to RRE include the confidentiality, integrity, and availability (CIA) binary values for files and processes of interest (e.g., the `/etc/passwd` file has been accessed and the `sshd` process terminated). Upon receiving reports from IRE, the response engine (RRE) concurrently (in a separate process) decides upon optimal response actions and sends the selected action back to IRE. Finally, IRE updates the system state to reflect the completion of the response action.

Figure 6.5 shows a part of the state machine based on which IRE replays a multi-stage credential compromise attack, one of the three attack classes (see Section 6.2) emulated by IRE. A genuine

user will escalate to root privilege (transition from state 0 to 1 to 3) through legitimate means (e.g., `sudo`, one-time password). On the other hand, once an attacker obtains user privileges, he or she may download a local root escalation exploit to obtain root privilege. Figure 6.6 shows the snippet of IRE output highlighting the current privilege level of the transition from user_login to illegal_login (transition from state 1 to 2) along with the IDS-generated alert for the exploit (ptrat.c) downloaded from `server5.xuna.nl`. Possible state transitions from the current state are `exploit_install` or `logoff`.

RRE Performance Analysis. The cost to a business of a successful attack is directly affected by the duration it takes for the failed system to recover back to its normal operational mode. In situations where an intrusion response engine such as RRE is employed, that duration consists of two factors: (1) the time it takes for the response engine to select the response action, and (2) the time it takes to effect that response action. Due to the first factor, it is important that RRE has good performance and select the optimal response action as quickly as possibly.

In our experiments, RRE uses the ART shown in Figure 6.2, which is fed into RRE in XML format. The ART is automatically converted to a state-based model, i.e., a game-theoretic competitive Markov decision process (CMDP), that is later solved for response actions. ART-to-CMDP conversion of the ART in Figure 6.2 results in a CMDP with 2^{42} states, which is computationally intractable; therefore, RRE makes use of an approximation algorithm, i.e., an *envelope* [116], to reduce the state-space size of the generated CMDP and hence make the model computationally feasible. Briefly, the envelope algorithm limits the number of future sequential actions, i.e., horizons, to consider by having RRE solve for the best response action [116]. For instance, if the envelope limits the number of horizons to two, RRE chooses the response action that minimizes the maximum damage that the attacker can cause in the next step; in other words, RRE considers all possible future action sequences of length two (i.e., response action $\rightarrow$ adversarial action).

Figure 6.7(a) demonstrates the time required to solve the generated CMDP given the number of horizons to explore. RRE can solve for response action within 10 seconds if the number of horizons to consider is kept below 8, and it takes about 1 minute if 11 horizons are considered. Clearly, the more horizons RRE considers, the closer the selected response action will be to the optimal one, but the more time it takes for the decision.

Figure 6.7(b) shows the number of states given the number of future horizons RRE takes into

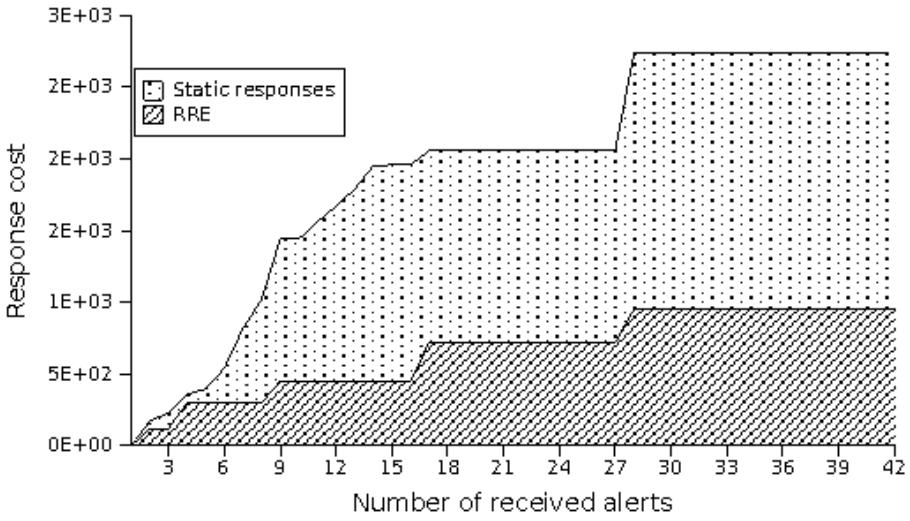


Figure 6.8: RRE vs. Static Intrusion Response

account when solving the CMDP model for the optimal action. For instance, considering only one future action sequence, the generated CMDP will have 43 states. The size of the state space exceeds one million when the number of horizons is set to 11.

RRE's most time-consuming task is the conversion of the business-level ART into its corresponding CMDP model. Figure 6.7(c) shows how much time RRE needs in order to generate a CMDP given a certain number of horizons to consider. It takes about 10 minutes for RRE to convert the ART to a CMDP when 10 future horizons are considered. However, it is important to mention that the ART-to-CMDP conversion step could be a one-time effort and done only once off-line, before the whole framework starts to operate.

Comparison with Static Response Mechanisms. We evaluated RRE against response systems that statically choose responses from a lookup table containing (IDS-alert, response) mappings. In our experiments, RRE solves the infinite horizon optimization problem to decide upon the action. Both RRE and static engines were run concurrently but independently fed by the same initial set of IDS alerts (initial system state). We compared the discounted cost spent by RRE and the static engine to completely recover the system, i.e., return to the point at which all leaf nodes were "cleaned up"

We consider the worst-case scenario and assume that the attacker, in each step of an attack progression, takes the most harmful possible adversarial action. There are a total of $2^{|\mathcal{L}|} = 2^{42}$ initial

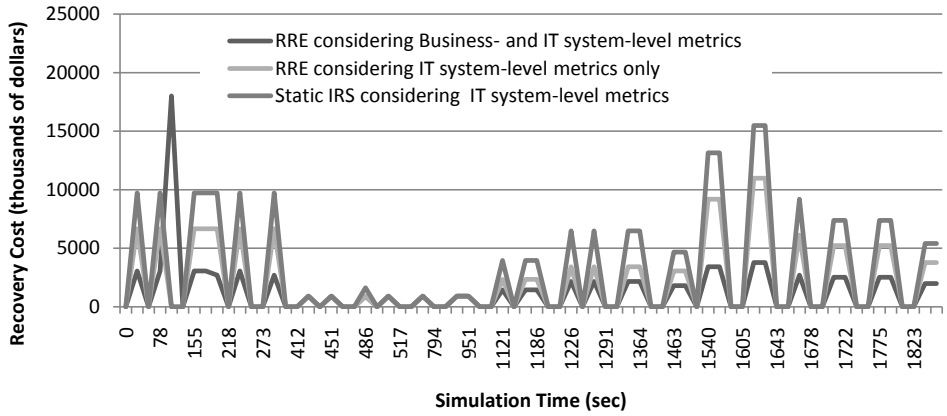


Figure 6.9: Recovery Cost With and Without Business-level Metrics

system states for the interaction; we investigate how RRE and the static engines perform, in terms of recovery cost, when $1 \leq i \leq 42$ of the IDS alerts are received in the starting state. Figure 6.8 shows the response cost (i.e, the total time to recover in seconds) for the RRE and the static intrusion response system. RRE takes advantage of its game-theoretic cost optimization engine to recover the system with lower cost. It is important to note that the recovery costs for the two engines are closer to each other when there are fewer IDS alerts received, and hence fewer leaf nodes fired in RRE; however, RRE is more helpful than static engines when larger numbers of IDS alerts from different parts of the system are received. The reason is that RRE solves the mathematical game-theoretic model to get the optimal action while the static intrusion response system simply looks up its manually prefilled table and sequentially takes response actions corresponding to each received alert.

RRE with and without Business-level ART. We evaluated the business-level impact of using the extended RRE (that includes the business-level ART), the basic RRE (presented in [116]), and a static intrusion response system for responding to attacks. We performed the evaluation in the context of the Widgets R Us enterprise example. Consider a situation in the Widgets R Us example where two servers, one mapped to the order management business process and the other mapped to the user forums business process, are simultaneously compromised by attackers. By exploiting the business-level metrics, the extended RRE could prioritize the compromised servers with respect to their criticality to overall business health and deal with them in that order, e.g., handling the server that deals with the order management business process first. The extended RRE does this

prioritization by making use of the costs of the consequences nodes in ART. That would not be possible in the case of the basic RRE, which considers only IT system-level metrics; with basic RRE, there is no way to distinguish the criticality of the various applications/servers to the overall business health of the whole infrastructure.

Figure 6.9 illustrates the discounted recovery cost during a 30-minute system-attacker interaction in which IRE emulated the attack incidents and the IT infrastructure of Widgets R Us and continuously sent IDS alerts out to RRE, which was running in a separate process. As explained before, the cost to recover from an attack depends both on the time it takes for the response engine to select the response action, and the time it takes to effect that response action. To permit fair comparison, the same cost function was used for both the IT-level ART and business-level ART models; similarly, the same IRE model was used for both. Given the received alerts, RRE started the game-theoretic decision engine to decide upon the optimal response action r using the equation in Figure 6.3. We assigned the cost per hour downtime for the Widgets R Us enterprise to be $\$180K$, which is the cost presented in Table 6.1 for Amazon.com.

RRE sends the chosen optimal action (which could be NOP) to the IRE, whose agents receive and execute the action(s) and update the state of the infrastructure. It is possible for the clients (or attackers) emulated by IRE to take their next normal (or adversarial) action(s) while RRE was still deciding upon what response action to take; in those situations, RRE's recommended response action was received by IRE when the infrastructure was in a state different from the one for which the response action was chosen. Therefore, the received response action were not always optimal, i.e., RRE payed a higher cost to recover the system. In our experiments, 3% of RRE's actions, on average, were received by IRE after the next state transition had already happened. As shown in Figure 6.9, basic RRE had a lower recovery cost than the static intrusion response system. The recovery cost was even lower with the extended RRE. In summary, the results show that adaptive intrusion response systems can benefit by factoring in business-level metrics along with standard IT system-level metrics.

6.5 Discussion and Future Work

In addition to the approach described in this chapter, we have investigated qualitative models for business value and business health, such as "high", "medium", and "low." These values would be treated with fuzzy logic. However, a quantitative approach fits the cost-based rigorous analysis of RRE much better. For instance, in a quantitative approach we can compare whether a certain "medium" cost of a countermeasure against an attack that is only suspected with low probability yet is justified by the low-probability "high" cost of business unavailability if the attack succeeds. Many aspects of a business have very concrete values assigned to them, e.g., production or sales losses due to the unavailability of certain functions for certain periods. Even for not directly financial losses such as privacy breaches financial statistics are nowadays often available, e.g., the average cost for notifying affected people and helping them with follow-on problems. If an IT organization does not directly belong to a business organization, it often has SLAs (service-level agreements) that include financial consequences of missing IT targets. In the remaining cases, the costs have to be estimated, but even rough estimates should be better than trying to make comparison rules for complex terms as they arise in the Markov Decision Processes over values such as "high."

For future directions, we focus on two major areas. First, as an additional validation platform for our approach as well as for future intrusion response approaches that others may develop, we are planning an actual deployment of the enterprise infrastructure and the business processes described in Section 6.2. In such a deployment, the server hosting the RRE engine has to be strengthened against denial-of-service and other attacks. Second, we are working on generating ART models automatically, and on augmenting them by attacks on the business layer, as well as alerts on the business layer both for IT and business attacks.

6.6 Conclusion

In this chapter, we presented a framework for managing business health in the presence of malicious attacks. The framework addresses situations in system administration that involve multi-objective decision-making while taking into account both business- and IT system-level metrics.

We extended RRE's ART formalism to enable the assessment of overall business health (i.e., dependability and security). We demonstrated the use of the framework in the context of an online enterprise against three common attack classes. Our work highlights the importance of considering business-level metrics in developing automated intrusion response systems. Without such consideration, there is the danger that actions recommended by an intrusion response system may not be relevant from a business perspective even if they are optimal at the IT system-level.

CHAPTER 7

CONCLUSIONS AND FUTURE WORK

Through a myriad of applications, such as process control of power critical infrastructures, computer networks play an increasingly important role in many aspects of our lives. In response to computer networks' popularity and their critical responsibilities, attackers have also become interested in them. Security incidents, such as the targeted Stuxnet attack against nuclear power plants [145], demonstrate current computer systems and networks are still vulnerable to security attacks. A research result from CERT institute [146] shows that number of total cataloged vulnerabilities has increased from 171 in 1995 to 7236 in 2007. Another result from the same source states that 99 percent of all reported intrusions result through exploitation of known vulnerabilities or configuration errors. Therefore, if efficient intrusion detection and tolerance solutions existed most of the previous security attacks could have been terminated before they completely compromised sensitive organizational assets causing very high financial costs [3].

To provide computer networks with a mathematically provable and practically feasible intrusion tolerance solution, we presented an intrusion response and recovery engine that can reason about interactions between itself and an adversary. The engine accounts for the adversarial behavior of attackers, and takes into account uncertainties in reports from intrusion detection systems. Furthermore, we studied various challenges involved in the design and deployment of an intrusion response system in a large-scale computer network. Additionally, for each challenge, we proposed a theoretical solution that we empirically evaluated and validated on real-world systems. In this chapter, we briefly review the work described in this dissertation, describe some directions in which the work could be expanded, and conclude this dissertation with some final remarks.

7.1 Contribution Review

We first presented a game-theoretic intrusion response engine called the Response and Recovery Engine (RRE). In doing so, we modeled the security maintenance of computer networks as a Stackelberg stochastic two-player game in which the attacker and response engine try to maximize their own benefits by taking optimal adversary and response actions, respectively. Using the game-theoretic algorithm, RRE explores the intentional malicious attacker's next possible action space before deciding upon the optimal response action, so it is guaranteed that the attacker cannot cause greater damage than what RRE predicts.

To provide the response system's decision-making core with a security assessment capability, we proposed *Seclius*, an online security evaluation framework that leverages dependencies between OS-level objects to measure the probability that critical assets have been directly or indirectly compromised. The different components of our framework address three important limitations faced by the traditional security assessment techniques. First, a consequence tree captures the subjective security requirements and minimizes administrator input. Second, Seclius processes IDS alerts online to measure actual attack consequences and does not rely on assumptions about attacker behaviors or system vulnerabilities. Third, a dependency graph is combined with a taint tracking method to probabilistically evaluate the system-wide impact of locally detected intrusions as well as attacker privileges and security domains, without being constrained by previously known attack paths.

To allow intrusion response systems to deal with unknown vulnerabilities that are not usually monitored by intrusion detection systems, we presented FloGuard, which is an online forensics analysis framework. In particular, we presented an automated monitor selection engine that decides upon the optimal set of intrusion detection systems to turn on according to the system configuration and its known (and possibly unknown) vulnerabilities. In particular, once an attack consequence, e.g., a file modification, has been reported by one of the already deployed detection systems, the engine tries to discover the exact sequence of vulnerability exploitations during the attack through an iterative forensics analysis by restoring a clean system snapshot and replaying the system several times. Throughout each replay iteration, a different set of detection systems are turned on to increase forensics evidence until the exact vulnerability exploitation sequence is

inferred. Then the engine considers the capabilities and computational overhead of the available intrusion detectors and chooses the optimal set of them, which it then turns on to run permanently in the system.

To facilitate the identification of the system's current state by the response system, we presented CPIDS, which fuses information from different types of sensors and probabilistically determines the current system state. CPIDS supports both cyber and cyber-physical systems; in other words, it can take into account both cyber and physical sensory information. In our experiments, we focused on power grid critical infrastructures in which a cyber network (a so-called power control network) continuously monitors and controls an underlying physical power system. CPIDS exploits all the available offline information, like cyber network topology and enforced access control policies, and the underlying power system configuration to create a comprehensive model of the cyber-physical system. During the operational mode, CPIDS makes use of the available online information from both the cyber-security sensors and the underlying power system measurements to determine the current system state via efficiently fusion of all that information.

Finally, we presented an in-depth case study of the Recovery and Response Engine (RRE) to demonstrate its ability to choose attack responses that take into account overall business health, instead of just health of individual IT components. Our case study enterprise network included three different business processes with different levels of importance to the high-level organizational business mission. To represent a model enterprise IT infrastructure's state and to replay real attacks from an IDS alert archive, we designed and used a simulator that continuously simulates and sends the current system state and IDS alerts to the response system. Having received the state information and IDS alerts, the response system decides upon optimal response actions and sends them back to the simulator, which is in charge of updating the system state based on the selected response.

7.2 Future Work

Next we will describe potential future work that we would like to pursue. In brief, three directions require further investigation. In Section 7.2.1, we discuss our plans for a cyber-physical intrusion

tolerance solution that can not only provide a cyber intrusion response capability but also take physical recovery actions. In Section 7.2.2, we explain how domain-specific knowledge about computer networks and cyber attacks could be exploited to further accelerate the optimization procedure in order to reduce the response time of the RRE framework. Finally, in Section 7.2.3, we describe our plans for implementation of a tool to help expert system operators validate a response system, and help inexperienced operators learn from interacting with the response system while it is protecting a simulated environment.

7.2.1 Cyber-Physical Intrusion Tolerance

In this thesis, we proposed a method for security-state estimation for cyber-physical power grid infrastructures. Our developed response system (see Chapter 2) currently supports only cyber systems; in other words, there is no physical action involved in the framework. However, in real-world scenarios, to protect physical power equipment, the response system must sometimes perform physical actions, such as tripping some transmission lines within a power grid substation. Nowadays, to carry out physical actions, it is usually necessary to send power engineers to the relevant substations. Such manual reaction might not always be fast enough to guarantee end-customer safety, and to avoid physical equipment damage.

We are planning to investigate the possibility of using remotely controllable robotics systems in substations to facilitate and accelerate the physical response capability against attacks and accidental failures. The premise is that even if a robotics system is unable to perform all complicated actions needed to recover completely from attacks and failures, it could perform some simple physical actions that would limit the damaging effects of an attack or failure, e.g., it could provide a fail-safe capability or deaccelerate the failure process until it is manually handled by system operators.

7.2.2 Hierarchical CMDP Solution

In this thesis, we proposed the mathematical framework of a game-theoretic intrusion response system. To improve its scalability, as mentioned in Chapter 2, we used the envelope approximation

technique to accelerate the dynamic programming solution procedure. Although the envelope algorithm significantly speeds up the optimization, the response time is still not fast enough to react against quick scripted attacks in a timely manner.

We are currently investigating the possibility of exploiting some domain-specific knowledge and assumptions to further accelerate the optimization process. As a case in point, organizational cyber infrastructures usually follow a hierarchical topology; in other words, an organization typically includes many networks, that include many host systems that consist of several (traditionally two, root and user-level) privilege domains. The question is, would it be possible to group the state variables accordingly in the Markov decision processes of the response system in order to optimize all of the zones and hierarchies separately in parallel, and then aggregate the results? There are already papers, e.g., [147], proposing similar algorithms for general optimization problems; however, there has not yet been any mathematical analysis of how optimal the actions selected using those approaches are. Our goal is to do an in-depth mathematical analysis of such techniques in the intrusion tolerance domain.

7.2.3 Educational Tool Implementation

As the first step towards real deployment of the proposed intrusion response solution, we are currently planning to implement the game-theoretic RRE engine as a computer game for system operators. During the game, as in computer chess games, RRE would act as an attacker against the user, e.g., the system operator, who is playing the game. The user would be in charge of protecting the network against the attacker by taking appropriate response actions. The core decision-making engine in the game would be identical to our current implementation. A graphical user interface must be developed to provide the user with visualization of the underlying network and to provide, at each step of the game, a list of possible response actions to pick.

The tool could be used for different purposes depending on the expertise level of the user who is playing the game. For inexperienced clients, the tool would act as an educational framework by providing corrections or highlighting the optimal response actions at each step of the game, to help the system operators learn how to react in real-world situations. For experienced expert clients, the tool would be used as either a validation or learning platform for RRE. The expert user could

either double-check whether the response actions recommended by RRE are correct or pick the correct response actions so that RRE gradually learns the optimal response actions for different system states using a machine-learning algorithm, e.g., reinforcement learning.

7.3 Concluding Remarks

Developing intrusion tolerance solutions to make mission-critical cyber-physical systems and networks resilient, and providing them with self-healing capability are areas of research that have crucial importance in a modern society that is dependent on many cyber-physical systems for its day to day functioning.

However, traditional solutions have fallen short in addressing two major challenges. First, they have either focused on either theoretical or practical aspects of the intrusion tolerance problem that have resulted in either theoretically sound solutions with unrealistic practical assumptions, or extremely customized practical solutions mainly based on heuristics that are often hard to prove and/or expand. Our goal at the outset of the research that led to this dissertation was to develop a theoretically well-grounded, and at the same time, completely practical intrusion tolerance solution. After few unsuccessful practical attempts in employing various theoretically provable techniques, our empirical validations showed that game-theoretic approaches can practically facilitate the challenging problem of designing a smart automated intrusion response solutions against intelligent malicious human beings.

Second, previous individual efforts into providing resiliency to cyber-power networks have considered the cyber-side communication networks and the underlying physical power systems separately. Hence, those approaches have completely overlooked the information that result from logical and functional interconnections of the cyber and physical sides of a power critical infrastructure. However, correlated information from cyber intrusion detection systems and monitors, and power sensor measurements about ongoing security attacks could be exploited to address various challenges of the cyber-physical system resiliency problem such as state estimation and anomaly detection as we discussed earlier. We will be pursuing further research to come up with a completely unified, yet simple and practical, detection and response solution for cyber-physical

critical infrastructures.

In summary, there are several unsolved problems which need to be solved before complete deployment of our intrusion tolerance solution. Furthermore, we have no doubt that large-scale practical deployment of our solution will require the solution of additional, unanticipated challenges. But overall, we believe that our approach provides not only a concrete and practical model-driven intrusion response and recovery solution for cyber-physical systems, but also represents an exciting new line of research in the field of system resiliency that has the potential to provide rich rewards in the quest for systems that continue to do what they are supposed to do, despite attacks that occur.

REFERENCES

[1] A. Avizienis, J. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, pp. 11–33, 2004.

[2] R. W. Kembel, *Fibre Channel: A Comprehensive Introduction*. Northwest Learning Associates, Inc., 2009.

[3] "2009 annual study: Cost of data breach (understanding financial impact, customer turnover, and preventive solutions)," Ponemon Institute, Tech. Rep., 2010.

[4] Y. Qian, D. Tipper, and P. Krishnamurthy, *Information Assurance: Dependability and Security in Networked Systems*, ser. Computer Security. Elsevier/Morgan Kaufmann, 2008.

[5] R. Rehman, *Intrusion detection systems with Snort: advanced IDS techniques using Snort, Apache, MySQL, PHP, and ACID*. Prentice Hall PTR, 2003.

[6] F. Cohen, "Simulating cyber attacks, defences, and consequences," *Computers & Security*, vol. 18, no. 6, pp. 479–518, 1999.

[7] A. Somayaji and S. Forrest, "Automated response using system-call delays," in *Proceedings of the 9th Conference on USENIX Security Symposium*, vol. 9. Berkeley, CA, USA: USENIX Association, 2000, p. 14.

[8] D. Schnackenberg, K. Djahandari, and D. Sterne, "Infrastructure for intrusion detection and response," *Proceedings of DARPA Information Survivability Conference and Exposition*, pp. 3–11, 2000.

[9] G. White, E. Fisch, and U. Pooch, "Cooperating security managers: A peer-based intrusion detection system," *IEEE Network*, vol. 10, no. 1, pp. 20 –23, Jan/Feb 1996.

[10] N. Stakhanova, S. Basu, and J. Wong, "A taxonomy of intrusion response systems," *International Journal of Information and Computer Security*, vol. 1, pp. 169–184, Jan. 2007.

[11] I. Balepin, S. Maltsev, J. Rowe, and K. Levitt, "Using specification-based intrusion detection for automated response," in *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2003, pp. 136–154.

[12] A. Sharma, Z. Kalbarczyk, R. Iyer, and J. Barlow, "Analysis of credential stealing attacks in an open networked environment," in *Proceedings of the 4th International Conference on Network and System Security (NSS)*, 2010, pp. 144–151.

[13] G. Casella and E. I. George, "Explaining the Gibbs Sampler," *The American Statistician*, vol. 46, no. 3, pp. 167–174, 1992.

[14] S. Axelsson, "Intrusion detection systems: A survey and taxonomy," U.S Department of Energy, Tech. Rep. 99-15, 2000.

[15] G. Owen, *Game Theory*, 3rd ed. Academic Press, 1995.

[16] D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, and N. Weaver, "Inside the slammer worm," *IEEE Security and Privacy*, vol. 1, pp. 33–39, July 2003.

[17] "Cryopid: A process freezer for linux, available at: http://cryopid.berlios.de/index.html."

[18] D. Pradhan and N. Vaidya, "Roll-forward checkpointing scheme: A novel fault-tolerant architecture," *IEEE Transaction on Computers*, vol. 43, pp. 1163–1174, 1994.

[19] B. Schneier, *Secrets & Lies: Digital Security in a Networked World*. John Wiley & Sons, 2000.

[20] A. Valdes and K. Skinner, "Adaptive, model-based monitoring for cyber attack detection," *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, pp. 80–92, 2000.

[21] C. Kruegel, W. Robertson, and G. Vigna, "Using alert verification to identify successful intrusion attempts," *Information Processing and Communication*, vol. 27, pp. 220–8, 2004.

[22] J. Filar and K. Vrieze, *Competitive Markov Decision Processes*. Springer-Verlag, 1997.

[23] S. Hsu and A. Arapostathis, "Competitive Markov decision processes with partial observation," in *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, 2004, pp. 236–241.

[24] L. Kaelbling, M. Littman, and A. Cassandra, "Partially observable Markov decision processes for artificial intelligence," *Proceedings of the German Conference on Artificial Intelligence: Advances in Artificial Intelligence*, vol. 981, pp. 1–17, 1995.

[25] E. Sondik, "The optimal control of partially observable Markov processes," Ph.D. dissertation, Stanford University, 1971.

[26] R. Bellman, *Dynamic Programming*. Princeton University Press, 1957; republished 2003.

[27] A. Cassandra, "Exact and approximate algorithms for partially observable markov decision processes," Ph.D. dissertation, Brown University, 1998.

[28] T. Dean, L. Kaelbling, J. Kirman, and A. Nicholson, "Planning under time constraints in stochastic domains," *Artificial Intelligence*, vol. 76, pp. 35–74, July 1995.

[29] M. Locasto, K. Wang, A. Keromytis, and S. Stolfo, "FLIPS: Hybrid adaptive intrusion prevention," in *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2005, pp. 82–101.

[30] S. Musman and P. Flesher, "System or security managers adaptive response tool," *Proceedings of the DARPA Information Survivability Conference and Exposition*, vol. 2, pp. 56–68, 2000.

[31] P. Porras and P. Neumann, "EMERALD: Event monitoring enabling responses to anomalous live disturbances," *Proceedings of the 20th National Information Systems Security Conference*, pp. 353–365, 1997.

[32] D. Ragsdale, C. Carver, J. Humphries, and U. Pooch, "Adaptation techniques for intrusion detection and intrusion response system," in *IEEE lnternational Conference on Systems, Man, and Cybernetics*, 2000, pp. 2344–2349.

[33] O. P. Kreidl and T. M. Frazier, "Feedback control applied to survivability: A host-based autonomic defense system," *IEEE Transactions on Reliability*, vol. 53, no. 1, pp. 148–166, 2004.

[34] K. Lye and J. Wing, "Game strategies in network security," *International Journal of Information Security*, vol. 4, pp. 71–86, 2005.

[35] M. Bloem, T. Alpcan, , and T. Basar, "Intrusion response as a resource allocation problem," in *Proceedings of the Conference on Decision and Control*, 2006, pp. 6283–6288.

[36] B. Foo, Y. Wu, Y. Mao, S. Bagchi, and E. Spafford, "Adepts: Adaptive intrusion response using attack graphs in an e-commerce environment," in *Proceedings of the 2005 International Conference on Dependable Systems and Networks (DSN)*, 2005, pp. 508–517.

[37] S. Lewandowski, D. Hook, G. O'Leary, J. Haines, and M. Rossey, "SARA: Survivable autonomic response architecture," *Proceedings of the DARPA Information Survivability Conference and Exposition II*, vol. 1, pp. 77–88, 2001.

[38] C. Cranor, Y. Gao, T. Johnson, V. Shkapenyuk, and O. Spatscheck, "Gigascope: High performance network monitoring with a SQL interface," in *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD '02)*. New York, NY, USA: ACM, 2002, p. 623.

[39] M. J. Ranum, K. Landfield, M. T. Stolarchuk, M. Sienkiewicz, A. Lambeth, and E. Wall, "Implementing a generalized tool for network monitoring," in *Proceedings of the 11th Conference on Systems Administration (LISA '97)*. Berkeley, CA, USA: USENIX Association, 1997, pp. 1–8.

[40] M. Iliofotou, P. Pappu, M. Faloutsos, M. Mitzenmacher, S. Singh, and G. Varghese, "Network monitoring using traffic dispersion graphs (TDGs)," in *Proceedings of the 7th ACM SIGCOMM Conference on Internet Measurement (IMC '07)*, 2007, pp. 315–320.

[41] W. Barth, *Nagios: System and Network Monitoring*. San Francisco, CA, USA: No Starch Press, 2008.

[42] L. Wang, T. Islam, T. Long, A. Singhal, and S. Jajodia, "An attack graph-based probabilistic security metric," *Data and Applications Security XXII*, pp. 283–296, 2008.

[43] S. Jha, O. Sheyner, and J. Wing, "Two formal analyses of attack graphs," in *Proceedings of the 15th Computer Security Foundation Workshop*, 2002, pp. 49–63.

[44] R. Dantu and P. Kolan, "Risk management using behavior based Bayesian networks," in *Intelligence and Security Informatics*. Springer Berlin Heidelberg, 2005, vol. 3495, pp. 115–126.

[45] S. Noel and S. Jajodia, "Optimal IDS sensor placement and alert prioritization using attack graphs," *Journal of Network and Systems Management*, vol. 16, pp. 259–275, 2008.

[46] P. Xie, J. Li, X. Ou, P. Liu, and R. Levy, "Using Bayesian networks for cyber security analysis," in *Proceedings of IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2010, pp. 211–220.

[47] S. Noel and S. Jajodia, "Optimal ids sensor placement and alert prioritization using attack graphs," *Journal of Network and Systems Management*, vol. 16, no. 3, pp. 259–275, 2008.

[48] L. Wang, A. Liu, and S. Jajodia, "Using attack graphs for correlating, hypothesizing, and predicting intrusion alerts," *Computer Communications*, vol. 29, no. 15, pp. 2917–2933, 2006.

[49] "SSE-CMM: Systems Security Engineering Capability Maturity Model, Security Metrics, International Systems Security Engineering Association (ISSEA)," http://www.sse-cmm.org/metric/metric.asp.

[50] W. E. Vesely, F. F. Goldberg, N. H. Roberts, and D. F. Haasl, *Fault Tree Handbook*. Washington, DC: U.S. Nuclear Regulatory Commission, 1981.

[51] B. Wotring, B. Potter, M. Ranum, and R. Wichmann, *Host Integrity Monitoring Using Osiris and Samhain*. Syngress Publishing, 2005.

[52] X. Chen, M. Zhang, Z. M. Mao, and P. Bahl, "Automating network application dependency discovery: Experiences, limitations, and new solutions," in *Proceedings of the 8th USENIX Conference on Operating Systems Design and Implementation*. USENIX Association, 2008, pp. 117–130.

[53] D. Song, D. Brumley, H. Yin, J. Caballero, I. Jager, M. G. Kang, Z. Liang, J. Newsome, P. Poosankam, and P. Saxena, "BitBlaze: A new approach to computer security via binary analysis," in *Proceedings of the International Conference on Information System Security (ICISS)*, 2008, pp. 1–25.

[54] B. Carrier, *File System Forensic Analysis*. Addison-Wesley Professional, 2005.

[55] "Dlib C++, available at: http://dlib.net," 2010.

[56] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th USENIX Conference on System Administration*, 1999, pp. 229–238.

[57] J. Ogness, "Dazuko: An open solution to facilitate on-access scanning," in *Proceedings of the 13th Virus Bulletin International Conference*, 2003, pp. 1–5.

[58] T. Kojm, "Clamav, available at http://www.clamav.net/," 2009.

[59] "Zabbix available at http://www.zabbix.org/," 2010.

[60] "Phpids available at http://php-ids.org/," 2010.

[61] "Trustworthy Cyber Infrastructure for the Power Grid (TCIPG): http://www.tcipg.org," 2010.

[62] C.-W. Ten, G. Manimaran, and C.-C. Liu, "Cybersecurity for critical infrastructures: attack and defense modeling," *Proceedings of the IEEE International Conference on Systems, Man, and Cybernetics*, vol. 40, pp. 853–865, July 2010.

[63] H. Hadeli, R. Schierholz, M. Braendle, and C. Tuduce, "Leveraging determinism in industrial control systems for advanced anomaly detection and reliable security configuration," in *Proceedings of the IEEE International Conference on Emerging Technologies & Factory Automation*. Piscataway, NJ, USA: IEEE Press, 2009, pp. 1189–1196.

[64] K. Stouffer, J. Falco, and K. Kent, "Guide to supervisory control and data acquisition (SCADA) and industrial control systems security," in *SPIN*, 2006.

[65] K. Davidson, "The x-10 tw523 two-way power line interface," *Circuit Cellar*, vol. 5, pp. 34–35, 1988.

[66] L. Wang, S. Noel, and S. Jajodia, "Minimum-cost network hardening using attack graphs," *Computer Communications*, vol. 29, no. 18, pp. 3812–3824, 2006.

[67] R. Sawilla and X. Ou, "Identifying critical attack assets in dependency attack graphs," *Computer Security-ESORICS 2008*, pp. 18–34, 2008.

[68] P. Porras, M. Fong, and A. Valdes, "A mission-impact-based approach to INFOSEC alarm correlation," in *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*. Springer-Verlag, 2002, pp. 95–114.

[69] K. Alsubhi, E. Al-Shaer, and R. Boutaba, "Alert prioritization in intrusion detection systems," in *IEEE Network Operations and Management Symposium (NOMS)*, 2008, pp. 33–40.

[70] H. Yin, D. Song, M. Egele, C. Kruegel, and E. Kirda, "Panorama: Capturing system-wide information flow for malware detection and analysis," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2007, pp. 116–127.

[71] S. King and P. Chen, "Backtracking intrusions," *ACM Transactions on Computer Systems (TOCS)*, vol. 23, no. 1, pp. 51–76, 2005.

[72] N. Zhu and T. Chiueh, "Design, implementation, and evaluation of repairable file service." IEEE Computer Society, 2003, pp. 217–226.

[73] P. Liu, X. Jia, S. Zhang, X. Xiong, Y. Jhi, K. Bai, and J. Li, "Cross-layer damage assessment for cyber situational awareness," in *Cyber Situational Awareness*. Springer US, 2010, vol. 46, pp. 155–176.

[74] M. Costa, M. Castro, L. Zhou, L. Zhang, and M. Peinado, "Bouncer: securing software by blocking bad input," in *Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2007, pp. 117–130.

[75] J. Tucek, N. James, S. Lu, C. Huang, S. Xanthos, D. Brumley, Y. Zhou, and D. Song, "Sweeper: A lightweight end-to-end system for defending against fast worms," in *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems 2007*. New York, NY, USA: ACM, 2007, pp. 115–128.

[76] F. Bellard, "Qemu, a fast and portable dynamic translator," in *Proceedings of the USENIX Annual Technical Conference*, 2005, p. 41.

[77] S. T. King and P. M. Chen, "Backtracking intrusions," in *Proceedings of the Nineteenth ACM symposium on Operating systems principles*, vol. 37, no. 5, 2003, pp. 223–236.

[78] A. Goel, K. Po, K. Farhadi, Z. Li, and E. de Lara, "The taser intrusion recovery system," in *SOSP*, 2005, pp. 163–76.

[79] "RoomPHPlanning, available at http://www.beaussier.com/roomphplanning/," 2008.

[80] "eVision, available at http://sourceforge.net/projects/e-vision/," 2009.

[81] A. Hay, D. Cid, and R. Bray, *OSSEC Host-Based Intrusion Detection Guide*. Elsevier, 2008.

[82] N. Nethercote and J. Seward, "Valgrind: A program supervision framework," *Electronic Notes in Theoretical Computer Science*, vol. 89, no. 2, pp. 44–66, 2003.

[83] D. W. Chapman, *Cisco Secure PIX Firewalls*. Cisco Press, 2001.

[84] "Secerno, Available at http://www.secerno.com/," 2010.

[85] H. H. Feng, O. M. Kolesnikov, P. Fogla, W. Lee, and W. Gong, "Anomaly detection using call stack information," in *Proceedings of the 2003 Symposium on Security and Privacy*, 2003, p. 62.

[86] M. Abadi, M. Budiu, U. Erlingsson, and J. Ligatti, "Control-flow integrity," in *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS)*, 2005, pp. 340–353.

[87] B. Schneier, "Attack trees," *Dr. Dobb's Journal*, 1999.

[88] P. Ammann, D. Wijesekera, and S. Kaushik, "Scalable, graph-based network vulnerability analysis," in *Proceedings of the 9th ACM Conference on Computer and Communications Security (CCS)*, 2002, pp. 217–224.

[89] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Communications of the ACM*, vol. 21, no. 7, pp. 558–565, 1978.

[90] A. Baratloo, N. Singh, and T. Tsai, "Transparent run-time defense against stack smashing attacks," in *Proceedings of the USENIX Annual Technical Conference*, 2000, pp. 251–262.

[91] "Wireplay, available at http://code.google.com/p/wireplay/," 2007.

[92] D. E. Mann and S. M. Christey, "Towards a common enumeration of vulnerabilities," in *Proceedings of the 2nd Workshop on Research with Security Vulnerability Databases*, 1999.

[93] "John the Ripper, Available at http://www.openwall.com/john/," 2008.

[94] S. Krishnan, K. Z. Snow, and F. Monrose, "Trail of bytes: Efficient support for forensic analysis," in *Proceedings of the 17th ACM Conference on Computer and Communications Security (CCS)*. ACM, 2010, pp. 50–60.

[95] "Nsauditor: Httptrafficgen, Available at http://www.nsauditor.com/," 2008.

[96] H. Debar and A. Wespi, "Aggregation and correlation of intrusion-detection alerts," in *Proceedings of the Symposium on Recent Advances in Intrusion Detection (RAID)*, 2001, pp. 85–103.

[97] G. Gu, P. Porras, V. Yegneswaran, M. Fong, and W. Lee, "BotHunter: Detecting malware infection through IDS-driven dialog correlation," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*, 2007, pp. 1–16.

[98] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: Treating bugs as allergies: A safe method to survive software failures," in *Proceedings of the Twentieth ACM Symposium on Operating systems Principles (SOSP)*, 2005, pp. 235–248.

[99] Q. Gao, W. Zhang, Y. Tang, and F. Qin, "First-aid: Surviving and preventing memory management bugs during production runs," in *Proceedings of the 4th ACM European Conference on Computer systems*, 2009, pp. 159–172.

[100] M. Costa, J. Crowcroft, M. Castro, A. Rowstron, L. Zhou, L. Zhang, and P. Barham, "Vigilante: End-to-end containment of Internet worms," in *Proceedings of the Twenty-first ACM SIGOPS Symposium on Operating Systems Principles (SOSP)*, 2005, pp. 133–147.

[101] 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 USENIX Security Symposium*, 2005, p. 9.

[102] S. Mukkamala and A. H. Sung, "Identifying significant features for network forensic analysis using artificial intelligent techniques," *International Journal of Digital Evidence*, pp. 1–17, 2003.

[103] A. Valdes and K. Skinner, "Probabilistic alert correlation," in *Proceedings of the 4th International Symposium on Recent Advances in Intrusion Detection*, 2001, pp. 54–68.

[104] A. Monticelli, F. F. Wu, and M. Yen, "Multiple bad data identification for state estimation by combinatorial optimization," *IEEE Power Engineering Review*, vol. PER-6, no. 7, pp. 73–74, July 1986.

[105] W. Peterson and A. Girgis, "Multiple bad data detection in power system state estimation using linear programming," in *Proceedings of the Twentieth Southeastern Symposium on System Theory*, Mar. 1988, pp. 405–409.

[106] L. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," *Proceedings of the IEEE*, vol. 77, no. 2, pp. 257–286, 1989.

[107] A. Monticelli, *State Estimation in Electric Power Systems: A Generalized Approach*. Kluwer Academic Publishers, 1999.

[108] A. Wood and B. Wollenberg, *Power Generation, Operation, and Control*, 2nd ed. John Wiley and Sons, 1996.

[109] Y. Liu, M. K. Reiter, and P. Ning, "False data injection attacks against state estimation in electric power grids," in *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, New York, NY, USA, 2009, pp. 21–32.

[110] D. M. Nicol, W. H. Sanders, S. Singh, and M. Seri, "Usable global network access policy for process control systems," *IEEE Security and Privacy*, vol. 6, pp. 30–36, 2008.

[111] "UMDHMM Tool, Available at: http://www.kanungo.com/software/software.html."

[112] J. Glover, M. Sarma, T. Overbye, and T. Overbye, *Power system analysis and design*. Thomson, 2008.

[113] R. Zimmerman, C. Murillo-Sanchez, and R. Thomas, "MATPOWER: Steady-state operations, planning, and analysis tools for power systems research and education," *IEEE Transactions on Power Systems*, vol. 26, no. 1, pp. 12–19, Feb. 2011.

[114] R. T. S. T. F. of the Application of Probability Methods Subcommittee, "IEEE reliability test system," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-98, no. 6, pp. 2047–2054, Nov. 1979.

[115] F. C. Schweppe and J. Wildes, "Power system static-state estimation, part i: Exact model," *IEEE Transactions on Power Apparatus and Systems*, vol. PAS-89, no. 1, pp. 120–125, 1970.

[116] S. A. Zonouz, H. Khurana, W. H. Sanders, and T. M. Yardley, "'RRE: A game-theoretic intrusion response and recovery engine," in *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, 2009, pp. 439–448.

[117] S. Zonouz and W. Sanders, "A Kalman-based coordination for hierarchical state estimation: Algorithm and analysis," in *Proceedings of the 41st Annual Hawaii International Conference on System Sciences (HICSS)*, Jan. 2008, p. 187.

[118] A. Volkanovski, M. Cepin, and B. Mavko, "Application of the fault tree analysis for assessment of power system reliability," *Reliability Engineering and System Safety*, vol. 94, no. 6, pp. 1116–1127, 2009.

[119] M. Zhou, V. Centeno, J. Thorp, and A. Phadke, "An alternative for including phasor measurements in state estimators," *IEEE Transactions on Power Systems*, vol. 21, no. 4, pp. 1930–1937, Nov. 2006.

[120] K. Lo, P. Zeng, E. Marchand, and A. Pinkerton, "New bad-data detection and identification technique based on rotation of measurement order for sequential state estimation," *IEE Proceedings C on Generation, Transmission and Distribution*, vol. 139, no. 5, pp. 387–401, 1992.

[121] O. Kosut, L. Jia, R. J. Thomas, and L. Tong, "Malicious data attacks on smart grid state estimation: Attack strategies and countermeasures," in *Proceedings of the First IEEE International Conference on Smart Grid Communications (SmartGridComm)*, 2010, pp. 220–225.

[122] S. A. Zonouz, K. R. Joshi, and W. H. Sanders, "Cost-aware systemwide intrusion defense via online forensics and on-demand detector deployment," in *Proceedings of the 3rd ACM Workshop on Assurable and Usable Security Configuration*, 2010, pp. 71–74.

[123] C.-W. Ten, C.-C. Liu, and G. Manimaran, "Vulnerability assessment of cybersecurity for SCADA systems," *IEEE Transactions on Power Systems*, vol. 23, no. 4, pp. 1836–1846, Nov. 2008.

[124] Z. Mohajerani, F. Farzan, M. Jafary, Y. Lu, D. Wei, N. Kalenchits, B. Boyer, M. Muller, and P. Skare, "Cyber-related risk assessment and critical asset identification within the power grid," in *IEEE PES on Transmission and Distribution Conference and Exposition*, 2010, pp. 1 –4.

[125] K. D. Wilken and T. Kong, "Concurrent detection of software and hardware data-access faults," *IEEE Transactions on Computers*, vol. 46, pp. 412–424, April 1997.

[126] A. Avizienis, J.-C. Laprie, and B. Randell, "Dependability and its threats: A taxonomy," in *IFIP Congress Topical Sessions*, 2004, pp. 91–120.

[127] A. Patcha and J.-M. Park, "An overview of anomaly detection techniques: Existing solutions and latest technological trends," *Computer Networks*, vol. 51, pp. 3448–3470, August 2007.

[128] H.-K. Pao, C.-H. Mao, H.-M. Lee, C.-D. Chen, and C. Faloutsos, "An intrinsic graphical signature based on alert correlation analysis for intrusion detection," in *Proceedings of the International Conference on Technologies and Applications of Artificial Intelligence (TAAI)*, Nov. 2010, pp. 102–109.

[129] S. Niksefat, M. M. Ahaniha, B. Sadeghiyan, and M. Shajari, "Toward specification-based intrusion detection for web applications," in *Proceedings of the 13th International Conference on Recent advances in Intrusion Detection*, 2010, pp. 510–511.

[130] C. V. Zhou, C. Leckie, and S. Karunasekera, "A survey of coordinated attacks and collaborative intrusion detection," *Computer Security*, vol. 29, no. 1, pp. 124–140, 2010.

[131] A. A. Cárdenas, S. Amin, Z.-S. Lin, Y.-L. Huang, C.-Y. Huang, and S. Sastry, "Attacks against process control systems: Risk assessment, detection, and response," in *Proceedings of the 6th ACM Symposium on Information, Computer and Communications Security*, 2011, pp. 355–366.

[132] N. Joukov, B. Pfitzmann, H. V. Ramasamy, N. G. Vogl, M. V. Devarakonda, and T. Ager, "Itbvm: IT business value modeler," in *Proceedings of the IEEE International Conference on Services Computing*, 2009, pp. 128–135.

[133] K. Magoutis, M. Devarakonda, N. Joukov, and N. Vogl, "Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems," *IBM Journal of Research and Development*, vol. 52, pp. 367–378, 2008.

[134] S. Sidiroglou, M. Locasto, S. Boyd, and A. Keromytis, "Building a reactive immune system for software services," in *Proceedings of the USENIX Annual Technical Conference*, 2005, pp. 149–161.

[135] C. Supatgiat, C. Kenyon, and L. Heusler, "Cause-to-effect operational risk quantification and management," *Risk Management: An International Journal*, vol. 8, no. 1, pp. 16–42, 2006.

[136] F. Cheng, D. Gamarnik, N. Jengte, W. Min, and B. Ramachandran, "Modeling operational risks in business processes," *Journal of Operational Risk*, vol. 2, no. 2, pp. 73–98, 2007.

[137] *Business Process Modeling and Notation (BPMN), Available at http://www.bpmn.org*.

[138] *PCI Compliance and the cost of a credit card breach Available at http://www.braintreepaymentsolutions.com/blog/pci-compliance-and-the-cost-of-a-credit-card-breach*, Apr. 2008.

[139] "Tivoli Application Dependency Discovery Manager Available at http://www.ibm.com/software/tivoli/products/taddm," 2011.

[140] *National Vulnerability Database, Available at http://web.nvd.nist.gov/*, 2010.

[141] A. Ornaghi, *Ettercap, Available at http://ettercap.sourceforge.net*, 2009.

[142] L. Doorn, *NFS Auditing*, 1998, http://dir.filewatcher.com.

[143] A. Hay, D. Cid, and R. Bray, *OSSEC Host-Based Intrusion Detection Guide*, 1st ed. Syngress, 2008.

[144] D. Menasce, "TPC-W: A benchmark for e-commerce," *IEEE Internet Computing*, vol. 6, no. 3, pp. 83 –87, 2002.

[145] N. Falliere, L. O. Murchu, and E. Chien, "W32.Stuxnet Dossier," Symantic Security Response, Tech. Rep., Oct. 2010.

[146] CERT, "CERT statistics (historical), http://www.cert.org/stats/," *Software Engineering Institute, Carnegie Mellon University*, 2009.

[147] M. Hauskrecht, N. Meuleau, L. P. Kaelbling, T. Dean, and C. Boutilier, "Hierarchical solution of Markov decision processes using macro-actions," in *Proceedings of Uncertainty in Artificial Intelligence (UAI)*, 1998, pp. 220–229.