SECURE CONFIGURATION OF INTRUSION DETECTION SENSORS FOR DYNAMIC ENTERPRISE-CLASS DISTRIBUTED SYSTEMS

A Dissertation

Submitted to the Faculty

of

Purdue University

by

Gaspar Modelo-Howard

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

May 2013

Purdue University

West Lafayette, Indiana

This thesis is dedicated, first and foremost, to my muse Lourdes. I am so lucky to have you. Hope our ride has been fun so far and that we can enjoy more adventures together. The world is huge and there are still many more places to discover.

To David and Diego, you are my guiding light. Life is so wonderful thru your eyes.

To the perfect mom, Argy, your love and support knows no boundaries.

To my brother, Gabriel. My honest, true friend.

To Tio Ivan. You gave me my first computer, pushed me to dream big and chase those dreams.

ACKNOWLEDGMENTS

I would like to thank the many people who have contributed to this thesis. Without their support, this work would not have been possible.

Many thanks to my advisor, Professor Saurabh Bagchi, for giving me the opportunity to start my life as a computer security researcher. He took me into DCSL and provided me with interesting opportunities to explore and study the world of computer security.

My gratitude to the members of my Program Committee Professors Guy Lebanon, Sonia Fahmy, and Vijai Pai for taking time out of their busy schedules to consider this work.

Thanks to Dawn Weisman and the rest of the NEEScomm IT Team, for providing an opportunity to support my research and a rich computing environment that shaped it.

I also thank my friends Daniel Torres, Ruben Torres, and Oscar Garibaldi for the great times watching sports, talking about research and Panama, and just enjoying college life.

Many thanks to Julio Escobar for inspiring me to pursue a life of research in computing. You are the epitome of the Panamanian who believes in his country and its people. Let's now "invent our future".

Thanks to the team of researchers and developers of the Bro Network Security Monitor system and to Matt Jonkman of the Emerging Threats IDS RuleSet. Both open source projects helped foster our research and provided an important framework to test our ideas.

My Ph.D. studies were partly supported by an IFARHU-SENACYT Scholarship from the Republic of Panama. Many thanks to these public institutions for providing a valuable opportunity to advance my professional career.

TABLE OF CONTENTS

Page

LIST OF TABLES

LIST OF FIGURES

Figure Page

ABSTRACT

Modelo-Howard, Gaspar Ph.D., Purdue University, May 2013. Secure Configuration of Intrusion Detection Sensors for Dynamic Enterprise-Class Distributed Systems. Major Professor: Saurabh Bagchi.

To secure todays computer systems, it is critical to have different intrusion detection sensors embedded in them. The complexity of distributed computer systems makes it difficult to determine the appropriate choice and placement of these detectors because there are many possible sensors that can be chosen, each sensor can be placed in several possible places in the distributed system, and overlaps exist between functionalities of the different detectors. For our work, we first describe a method to evaluate the effect a detector configuration has on the accuracy and precision of determining the systems security goals. The method is based on a Bayesian network model, obtained from an attack graph representation of the target distributed system that needs to be protected. We use Bayesian inference to solve the problem of determining the likelihood that an attack goal has been achieved, given a certain set of detector alerts. Based on the observations, we implement a dynamic programming algorithm for determining the optimal detector settings in a large-scale distributed system and compare it against a greedy algorithm, previously developed.

In the work described above, we take a (static) snapshot of the distributed system to determine the configuration of detectors. But distributed systems are dynamic in nature and current attacks usually involve multiple steps, called multi-stage attacks, due to attackers usually taking multiple actions to compromise a critical asset for the victim. Current sensors are not capable of analyzing multi-stage attacks. For the second part of our work, we present a distributed detection framework based on a probabilistic reasoning engine that communicates to detection sensors and can achieve two goals: (1) protect a critical asset by detecting multi-stage attacks and (2) tune sensors according to the changing environment of the distributed system, which includes changes to the protected system as well as changing nature of attacks against it.

Each node in the Bayesian Network model represents a detection signature to an attack step or vulnerability. We extend our model by developing a system called pSigene, for the automatic generation of generalized signatures. It follows a four-step process based on a biclustering algorithm to group attack samples we collect from multiple sources, and logistic regression model to generate the signatures. We implemented our system using the popular open-source Bro Intrusion Detection System and tested it for the prevalent class of Structured Query Language injection attacks. We obtain True and False Positive Rates of over 86% and 0.03%, respectively, which are very competitive to existing signature sets.

1. INTRODUCTION

1.1 Motivation

Intrusion Detection Systems (IDS) play an important role in the cybersecurity strategy of many organizations, to protect their distributed computing systems. The host and network activity experienced in these systems calls for continuous monitoring, and this role is usually satisfied by an IDS with one or more detection sensors. Today's distributed systems exhibit a very dynamic and complex nature as its components are incessantly modified, interconnected or replaced and the topology of the underlying network keeps changing. From a security perspective, it is impossible then to assume under this scenario that all security risks can be completely eliminated or that no errors will occur. Additionally, the system's components are not perfect and carry vulnerabilities and other security problems that eventually are exploited by malicious users. A sound, complete cybersecurity strategy must then include mechanisms to detect when security breaches happen. That is the main role of an IDS.

The complexity and dynamism found nowadays in distributed systems creates a conundrum for IDS designers and operators: how to configure and operate an IDS to effectively detect when bad things happen in a computing scenario that keeps changing and is composed from multiple components? Under this scenario, it is difficult to determine the appropriate choice and placement of the intrusion detections sensors because there are many possible sensors that can be chosen, each sensor can be placed in several possible places in the distributed system, and overlaps exist between functionalities of the different detection sensors.

In this thesis, we set out to provide mechanisms to help security professionals to design, configure, and operate an IDS for a distributed system in a dynamic environment. We develop techniques that evaluate the proposed or current configuration of a intrusion detection system, which require numerous, distributed sensors. For each of these components to effectively contribute, the IDS should continuously evaluate the sensors and reconfigure itself, based on the changes to the monitored distributed system. If the sensors are not properly managed, their contribution will decrease over time, jeopardizing the effectiveness of the IDS and ultimately, compromising the security of the distributed system.

An advantage to having multiple, distributed sensors in an IDS is the possibility to correlate the alerts generated by these sensors, in order to ultimately make more accurate detections than when using a single detection sensor. We present an alert correlation technique based on attacks graphs and Bayesian inference by using the probabilistic graphical model known as Bayesian network. It turns out that efficient alert correlation is very environment specific as the value of a particular set of alerts to help determine if an intrusion is happening, highly increases based on the location of the corresponding sensors and the ability of the alerts to describe a possible intrusion scenario.

The problem of dealing with complex and dynamic distributed systems makes it necessary to consider the automation of many tasks to manage intrusion detection systems. An IDS increasingly produces alerts that make it impossible for humans to respond effectively and in a timely manner. It is then necessary to develop algorithms that can scale to promptly determine if an intrusion has occurred and to allow for the reconfiguration of the IDS when changes happen.

The detection method used by an IDS is a critical aspect when determining its efficiency and performance. A 0-1 attitude by IDS developers when selecting the detection method to use, has made users question the merits of the detection systems. IDS are commonly divided into two groups, according to the detection method used: anomaly- and misuse-based detection. The former creates a profile of the normal (non-malicious) behavior observed in the monitored system while the latter uses signatures of attacks to detect malicious activity. Both methods present limitations and as we present in this thesis, the detection efficiency of an IDS can be improved by mixing both methods, producing signatures that represent generalized versions of malicious behavior.

Our work is motivated by the experience of designing and operating the open-source Bro IDS for NEEShub, a distributed system which is part of a National Science Foundation-sponsored Center at Purdue University. The system serves content and simulation tools for an engineering domain for thousands of users. While managing the Bro IDS, we experienced the growing complexity and dynamism of the system and served well as a reminder of the need to produce techniques for alert correlation, to use multiple sensors, and to automate to managing of the IDS and its sensors.

1.2 Outline

Here we briefly summarize the contents of the chapters presented in this work.

In **Chapter 2**, we present a method to evaluate the effect a detector configuration has on the accuracy and precision of determining the systems security goals. The method is based on a Bayesian network model, obtained from an attack graph representation of the target distributed system that needs to be protected. We use Bayesian inference to solve the problem of determining the likelihood that an attack goal has been achieved, given a certain set of detector alerts.

We explore two methods to determine the configuration of intrusion detection sensors in a distributed system. First, a greedy algorithm is introduced and tested against two electronic commerce scenarios. Then, we present a dynamic programming algorithm for determining the optimal detector settings in a large-scale distributed system and compare it against the previously developed greedy algorithm. We also report the results of five experiments, measuring the Bayesian networks behavior in the context of two real-world distributed systems undergoing attacks. Results show the dynamic programming algorithm outperforms the Greedy algorithm in terms of the benefit provided by the set of detectors picked. The dynamic programming solution also has the desirable property that we can trade off the running time with how close the solution is to the optimal.

In the work described in the previous chapter, we take a (static) snapshot of the distributed system to determine the configuration of detectors. But distributed systems are dynamic in nature and current attacks usually involve multiple steps, called multi-stage attacks, due to attackers usually taking multiple actions to compromise a critical asset for the victim. Current sensors are not capable of analyzing multi-stage attacks. In **Chapter 3**, we introduce a distributed detection framework based on a probabilistic reasoning engine that communicates to detection sensors and can achieve two goals: (1) protect a critical asset by detecting multi-stage attacks and (2) tune sensors according to the changing environment of the distributed system, which includes changes to the protected system as well as changing nature of attacks against it.

The Bayesian Network model described in the previous chapters, represents every vulnerability found in the monitored distributed system as a node. That is, the exact vulnerability is represented as a node in the Bayesian Network. To extend the detection capability of the model, a node can be generalized so it represents a group of similar vulnerabilities, not only a single one. To achieve this, in **Chapter 4** we present a system, called *pSigene*, for the automatic generation of intrusion signatures by mining the vast amount of public data available on attacks. Each signature represents then a group of similar vulnerabilities. The system follows a four-step process to generate the signatures, by first crawling attack samples from multiple public cybersecurity web portals. Then, a feature set is created from existing detection signatures to model the samples, which are then grouped using a biclustering algorithm which also gives the distinctive features of each cluster. In the last step, the system automatically creates a set of signatures using regular expressions, one for each cluster. We tested our architecture for the prevalent class of SQL injection attacks and found our signatures to have a True and False Positive Rates of over 86% and 0.03%, respectively and compared our findings to other SQL injection signature sets from popular IDS and web application firewalls. Results show our system to be very competitive to existing signature sets.

Finally, **Chapter 5** presents our future plans and work.

1.3 Published Work

Part of this thesis have been published:

- Gaspar Modelo-Howard, Jevin Sweval, and Saurabh Bagchi:
  Secure Configuration of Intrusion Detection Sensors for Changing Enterprise Systems. In: Proc. of the 7th ICST Conference on Security and Privacy for Communication Networks (SecureComm'11). London, United Kingdom, September 2011.

- Gaspar Modelo-Howard, Saurabh Bagchi, and Guy Lebanon:
  Approximation Algorithms for Determining Placement of Intrusion Detectors in a Distributed System. CERIAS Technical Report 2011-01, Purdue University.

- Gaspar Modelo-Howard, Saurabh Bagchi, and Guy Lebanon:
  Determining Placement of Intrusion Detectors for a Distributed Application through Bayesian Network Modeling. In: Proc. of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08). Boston, MA, September 2008.

2. DETERMINING PLACEMENT OF INTRUSION DETECTORS FOR A DISTRIBUTED APPLICATION THROUGH BAYESIAN NETWORK MODELING

2.1 Introduction

It is critical to provide intrusion detection to secure today's distributed computer systems. The overall intrusion detection strategy involves placing multiple detectors at different points of the system. Examples of specific locations are network ingress or combination points, specific hosts executing parts of the distributed system, or embedded in specific applications that form part of the distributed system. At the current time, the placement of the detectors and the choice of the detectors are more an art than a science, relying on expert knowledge of the system administrator.

The choice of the detector configuration has substantial impact on the accuracy and precision of the overall detection process. There are many choices to consider, including the placement of detectors, their false positive (FP) and false negative (FN) rates, and other detector properties. This results in a large exploration space which is currently explored using ad-hoc techniques. This chapter presents an important step in constructing a principled framework to investigate this exploration space.

At first glance it may seem that increasing the number of detectors is always a good strategy. However, this is not the case and an extreme design choice of a detector at every possible network point, host, and application may not be ideal. First, there is the economic cost of acquiring, configuring, and maintaining the detectors. Detectors need tuning to achieve their best performance and to meet the targeted needs of the application (specifically in terms of the balance between false positive and false negative rates). Second, a large number of imperfect detectors means a large number of alert streams in benign conditions that could overwhelm the manual

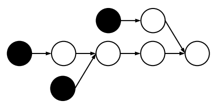


Fig. 2.1. Attack graph model for a sample web server. There are three starting vertices, representing three vulnerabilities found in different services of the server from where the attacker can elevate the privileges in order to reach the final goal of compromising the password file.

or automated response process. Third, detectors impose a performance penalty on the distributed system as they typically share bandwidth and computational cycles with the application. Fourth, system owners may have varying security goals such as requiring high sensitivity or ensuring less tolerance for false positive alerts.

In this chapter we address the problem of determining where (and how many) to place detectors in a distributed system, based on situation-specific security and performance goals. We also show that this is an intractable problem. The security goals are determined by requiring a certain trade-off between the true positive (TP) – true negative (TN) detection rates.

Our proposed solution starts with attack graphs, as shown in Figure 2.1, which are a popular representation for multi-stage attacks [1]. Attack graphs are a graphical representation of the different ways multi-stage attacks can be launched against a specific system. The nodes depict successful intermediate attack goals with the end nodes representing the ultimate attack goal. The edges represent the notion that some attack goals serve as stepping stones to other attack goals and therefore have to be achieved first. The nodes can be represented at different levels of abstraction; thus the attack graph representation can bypass the criticism that detailed attack methods and steps need to be known a priori to be represented (which is almost never the case for reasonably complex systems). Research in the area of attack graphs has included automation techniques to generate these graphs [3], [2], to analyze them [4], [5], and to reason about the completeness of these graphs [4].

We model the probabilistic relations between attack steps and detectors using the statistical formalism of Bayesian networks. Bayesian networks are particularly appealing in this setting since they enable computationally efficient inference for unobserved nodes (such as attack goals) based on observed nodes (detector alerts.) The important question that Bayesian inference can answer for us is, given a set of detector alerts, what is the likelihood or probability that an attack goal has been achieved. A particularly important advantage is that Bayesian network can be relatively easily created from an attack graph structure which is often assumed to be provided.

We formulate two Bayesian inference algorithms, implementing a greedy approach for one and dynamic programming for the other, to systematically determine the accuracy and precision a specific detector configuration has. We then proceed to choose the detector placement that gives the highest value of a situation-specific utility function. We show the Greedy algorithm has an approximation ratio of $\frac{1}{2}$. The dynamic programming solution falls in the algorithm category of the *fully polynomial time approximation scheme* (FPTAS) and has the desirable property that we can trade off the running time with how close the solution is to the optimal.

We demonstrate our proposed framework in the context of two specific systems, a distributed E-commerce system and a Voice-over-IP (VoIP) system, and compared both algorithms. We experiment with varying the quality of the detectors, the level of knowledge of attack paths, and different thresholds set by the system administrator for determining whether an attack goal was reached. Our experiments indicate that the value of a detector in terms of determining an attack step degrades exponentially with its distance from the attack site.

The rest of this document is organized as follows. Section 2.2 presents the related work and section 2.3 introduces the attack graph model and provides a brief presentation of inference in Bayesian networks. Section 2.4 describes the model and algorithms used to determine an appropriate location for detectors. Section 2.5 provides a description of the distributed systems used in our experiments. Sections 2.6 and 2.7 present a complete description of the experiments along with their motivations to help determine the location of the intrusion detectors. Finally, section 2.8 concludes the chapter and discusses future work.

2.2 Related Work

Bayesian networks have been used in intrusion detection to perform classification of events. Kruegel et al. [6] proposed the usage of Bayesian networks to reduce the number of false alarms. Bayesian networks are used to improve the aggregation of different model outputs and allow integration of additional information. The experimental results show an improvement in the accuracy of detections, compared to threshold-based schemes. Ben Amor et al. [7] studied the use of naive Bayes in intrusion detection, which included a performance comparison with decision trees. Due to similar performance and simpler structure, naive Bayes is an attractive alternative for intrusion detection. Other researchers have also used naive Bayesian inference for classifying intrusion events [8].

To the best of our knowledge, the problem of determining an appropriate location for detectors has not been systematically explored by the intrusion detection community. However, analogous problems have been studied to some extent in the physical security and the sensor network fields.

Jones *et al.* [9] developed a Markov Decision Process (MDP) model of how an intruder might try to penetrate the various barriers designed to protect a physical facility. The model output includes the probability of a successful intrusion and the most likely paths for success. These paths provide a basis to determine the location of new barriers to deter a future intrusion.

In the case of sensor networks, the placement problem has been studied to identify multiple phenomena such as determining location of an intrusion [10], contamination source [11], [12], and atmospheric conditions [13]. Anjum *et al.* [10] determined which nodes should act as intrusion detectors in order to provide detection capabilities in a hierarchical sensor network. The adversary is trying to send malicious traffic to a destination node (say, the base node). In their model, only some nodes called tamper-resistant nodes are capable of executing a signature-based intrusion detection algorithm and these nodes cannot be compromised by an adversary. Since these nodes are expensive, the goal is to minimize the number of such nodes and the authors provide a distributed approximate algorithm for this based on minimum cut-set and minimum dominating set. The solution is applicable to a specific kind of topology, widely used in sensor networks, namely clusters with a cluster head in each cluster capable of communicating with the nodes at the higher layer of the network hierarchy.

In [11], the sensor placement problem is studied to detect the contamination of air or water supplies from a single source. The goal is to detect that contamination has happened and the source of the contamination, under the constraints that the number of sensors and the time for detection are limited. The authors show that the problem with sensor constraint or time constraint are both NP-hard and they come up with approximation algorithms. They also solve the problem exactly for two specific cases, the uniform clique and rooted trees. A significant contribution of this work is the time efficient method of calculating the sensor placement. However, several simplifying assumptions are made—sensing is perfect and no sensor failure (either natural or malicious) occurs, there is a single contaminating source, and the flow is stable.

Krause *et al.* [13] also point out the intractability of the placement problem and present a polynomial-time algorithm to provide near-optimal placement which incurs low communication cost between the sensors. The approximation algorithm exploits two properties of this problem: submodularity, formalizing the intuition that adding a node to a small deployment can help more than adding a node to a large deployment; and locality, under which nodes that are far from each other provide almost independent information. In our current work, we also experienced the locality property of the placement problem. The proposed solution *learns* a probabilistic model (based on Gaussian processes) of the underlying phenomenon (variation of temperature, light, and precipitation) and for the expected communication cost between any two locations from a small, short-term initial deployment.

In [12], the authors present an approach for determining the location in an indoor environment based on which sensors cover the location. The key idea is to ensure that each resolvable position is covered by a unique set of sensors, which then serves as its signature. They make use of identifying code theory to reduce the number of active sensors required by the system and yet provide unique localization for each position. The algorithm also considers robustness, in terms of the number of sensor failures that can be corrected, and provides solutions in harsh environments, such as presence of noise and changes in the structural topology. The objective for deploying sensors here is quite different from our current work.

For all the previous work on placement of detectors, the authors are looking to detect events of interest, which propagate using some well-defined models, such as, through the cluster head *en route* to a base node. Some of the work (such as [13]) is focused on detecting natural events, that do not have a malicious motive in avoiding detection. In our case, we deal with malicious adversaries who have an active goal of trying to bypass the security of the system. The adversaries' methods of attacking the system do not follow a well-known model making our problem challenging. As an example of how our solution handles this, we use noise in our BN model to emulate the lack of an accurate attack model.

There are some similarities between the work done in alert correlation and ours, primarily the interest to reduce the number of alerts to be analyzed from an intrusion. Approaches such as [14] have proposed modeling attack scenarios to correlate alerts and identify causal relationships among the alerts. Our work aims to closely integrate the vulnerability analysis into the placement process, whereas the alert correlation proposals have not suggested such importance.

The idea of using Bayes theorem for detector placement is suggested in [15]. No formal definition is given, but several metrics such as accuracy, sensitivity, and specificity are presented to help an administrator make informed choices about placing detectors in a distributed system. These metrics are associated to different areas or sub-networks of the system to help in the decision process.

Many studies have been done on developing performance metrics for the evaluation of intrusion detection systems (IDS), which have influenced our choice of metrics here. Axelsson [16] showed the applicability of estimation theory in the intrusion detection field and presented the Bayesian detection rate as a metric for the performance of an IDS. His observation that the base rate, and not only the false alarm rate, is an important factor on the Bayesian detection rate, was included in our work by using low base rates as part of probability values in the Bayesian network. The MAFTIA Project [17] proposed precision and recall to effectively determine when a vulnerability was exploited in the system. A difference from our approach is that they expand the metrics to consider a set of IDSes and not only a single detector. The idea of using ROC curves to measure performance of intrusion detectors has been explored many times, most recently in [18], [19].

Extensive work has been done for many years with attack graphs. Recent work has concentrated on the problems of generating attack graphs for large networks and automating the process to describe and analyze vulnerabilities and system components to create the graphs. The NetSPA system [2] uses a breath-first technique to generate a graph that grows almost linearly with the size of the distributed system. Ou et al. [3] proposed a graph building algorithm using a formal logical technique that allows to create graphs of polynomial size to the network being analyzed.

2.3 Background

2.3.1 Attack Graphs

An attack graph is a representation of the different methods by which a distributed system can be compromised. It represents the intermediate attack goals for a hypothetical adversary leading up to some high level attack goals. The attack goal may be in terms of violating one or more of confidentiality, integrity, or availability of a component in the system. It is particularly suitable for representing multi-stage attacks, in which a successful attack step (or steps) is used to achieve success in a subsequent attack step. An edge will connect the antecedent (or precondition) stage to the consequent (or postcondition) stage. To be accurate, this discussion reflects the notion of one kind of attack graph, called the exploit-dependency attack graph [2], [4], [3], but this is by far the most common type and considering the other subclass will not be discussed further in this chapter.

Recent advances in attack graph generation have been able to create graphs for systems of up to hundreds and thousands of hosts [2], [3].

For our detector-location framework, exploit-dependency attack graphs are used as the base graph from which we build the Bayesian network. For the rest of this chapter, the vertex representing an exploit in the distributed system will be called an attack step.

2.3.2 Inference in Bayesian Networks

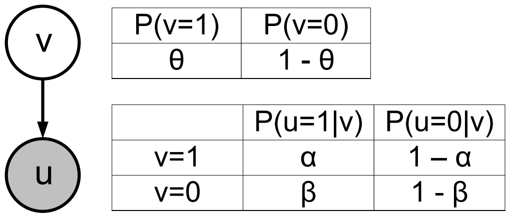
Bayesian networks [20] provide a convenient framework for modeling the relationship between attack steps and detector alerts. Using Bayesian networks we can infer which unobserved attack steps have been achieved based on the observed detector alerts.

Formally, a Bayesian network is a joint probabilistic model for n random variables $(x_1, \ldots, x_n)$ based on a directed acyclic graph $G = (V, E)$ where V is a set of nodes corresponding to the variables $V = (x_1, \ldots, x_n)$ and $E \subseteq VxV$ contains directed edges connecting some of these nodes in an acyclic manner. Instead of weights, the graph edges are described by conditional probabilities of nodes given their parents that are used to construct a joint distribution $P(V)$ or $P(x_1, \ldots, x_n)$.

There are three main tasks associated with Bayesian networks. The first is inferring values of variables corresponding to nodes that are unobserved given values of variables corresponding to observed nodes. In our context this corresponds to predict-ing whether an attack step has been achieved based on detector alerts. The second task is learning the conditional probabilities in the model based on available data which in our context corresponds to estimating the reliability of the detectors and the probabilistic relations between different attack steps. The third task is learning the structure of the network based on available data. All three tasks have been extensively studied in the machine learning literature and, despite their difficulty in the general case, may be accomplished relatively easily in the case of a Bayesian network.

We focus in this chapter mainly on the first task. For the second task, to estimate the conditional probabilities, we can use characterization of the quality of detectors [21] and the perceived difficulty of achieving an attack step, say through risk assessment. We consider the fact that the estimate is unlikely to be perfectly accurate and provide experiments to characterize the loss in performance due to imperfections. For the third task, we rely on extensive prior work on attack graph generation and provide a mapping from the attack graph to the Bayesian network.

In our Bayesian network, the network contains nodes of two different types $V = V_a \bigcup V_b$. The first set of nodes V_a corresponds to binary variables indicating whether specific attack steps in the attack graph occurred or not. The second set of nodes V_b corresponds to binary variables indicating whether a specific detector issued an alert. The first set of nodes representing attack steps are typically unobserved while the second set of nodes corresponding to alerts are observed and constitute the evidence. The Bayesian network defines a joint distribution $P(V) = P(V_a, V_b)$ which can be used to compute the marginal probability of the unobserved values $P(V_a)$ and the conditional probability $P(V_a|V_b) = P(V_a, V_b)/P(V_b)$ of the unobserved values given the observed values. The conditional probability $P(V_a|V_b)$ can be used to infer the likely values of the unobserved attack steps given the evidence from the detectors. Comparing the value of the conditional $P(V_a|V_b)$ with the marginal $P(V_a)$ reflects the gain in information about estimating successful attack steps given the current set of detectors. Alternatively, we may estimate the suitability of the detectors by computing classification error rate, precision, recall and Receiver Operating Characteristic (ROC) curve associated with the prediction of V_a based on V_b.



P(v=1)	P(v=0)
θ	1 - θ

	P(u=1|v)	P(u=0|v)
v=1	α	1 – α
v=0	β	1 - β

Fig. 2.2. Simple Bayesian network with two types of nodes: an observed node (u) and an unobserved node (v). The observed node correspond to the detector alert in our framework and its conditional probability table includes the true positive (α) and false positive (β).

Note that the analysis above is based on emulation done prior to deployment with attacks injected through the vulnerability analysis tools, a plethora of which exist in the commercial and research domains, including integrated infrastructures combining multiple tools.

Some attack steps have one or more detectors that specifically measure whether an attack step has been achieved while other attack steps do not have such detectors. We create an edge in the Bayesian network between nodes representing attack steps and nodes representing the corresponding detector alerts. Consider a specific pair of nodes $v \in V_a, u \in V_b$ representing an attack step and a corresponding detector alert. The conditional probability $P(v|u)$ determines the values $P(v = 1|u = 0), P(v = 0|u = 1), P(v = 0|u = 0), P(v = 1|u = 1)$. These probabilities representing false negative, false positive, and correct behavior (last two) can be obtained from an evaluation of the detectors quality.

2.4 System Design

2.4.1 Framework Description

Our framework uses a Bayesian network to represent the causal relationships between attack steps and also between attack steps and detectors. Such relationships are expressed quantitatively, using conditional probabilities. To produce the Bayesian network[1], an attack graph is used as input. The structure of the attack graph maps exactly to the structure of the Bayesian network. Each node in the Bayesian network can be in one of two states. Each attack stage node can either be achieved or not by the attacker. Each detector node can be in one of two states: alarm generated state or not. The leaf nodes correspond to the starting stages of the attack, which do not need any precondition, and the end nodes correspond to end goals for an adversary. Typically, there are multiple leaf nodes and multiple end nodes.

The Bayesian network requires that the sets of vertices and directed edges form a directed acyclic graph (DAG). This property is also found in attack graphs. The idea is that the attacker follows a monotonic path, in which an attack step does not have to be revisited after moving to a subsequent attack step. This assumption can be considered reasonable in many scenarios according to experiences from real systems.

A Bayesian network quantifies the causal relation that is implied by an edge in an attack graph. In the cases when an attack step has a parent, determined by the existence of an edge coming to this child vertex from another attack step, a conditional probability table is attached to the child vertex. As such, the probability values for each state of the child are conditioned by the state(s) of the parent(s). In these cases, the conditional probability is defined as the probability of a packet from an attacker that already achieved the parent attack step, achieving the child attack step. All values associated to the child are included in a conditional probability table (CPT). As an example, all values for node u in Figure 2.2 are conditioned on the

[1]Henceforth, when we refer to a node, we mean a node in the Bayesian network, as opposed to a node in the attack graph. The clarifying phrase is thus implied.

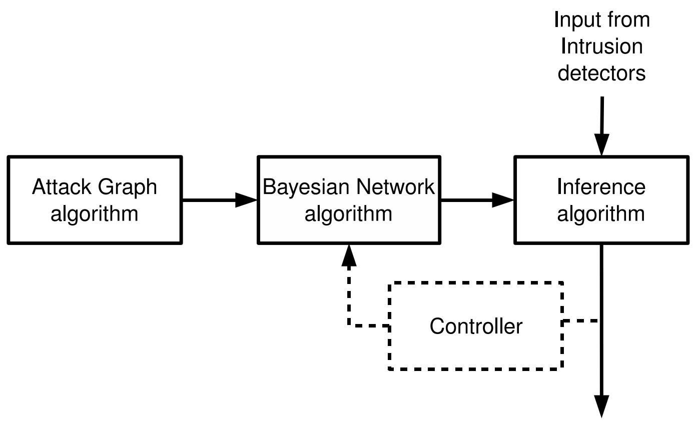


Fig. 2.3. A block diagram of the framework to determine placement of intrusion detectors. The dotted lines indicate a future component, controller, not included currently in the framework. It would provide for a feedback mechanism to adjust location of detectors.

possible states of its parent, node v. In conclusion, we are assuming that the path taken by the attacker is fully probabilistic. The attacker is following a strategy to maximize the probability of success, to reach the security goal. To achieve it, the attacker is well informed about the vulnerabilities associated to a component of the distributed system and how to exploit it. The fact that an attack graph is generated from databases of vulnerabilities support this assumption.

The CPTs have been estimated for the Bayesian networks created. Input values are a mixture of estimates based on testing specific elements of the system, like using a certain detector such as IPTables [22] or Snort [23], and subjective estimates, using judgment of a system administrator. From the perspective of the expert (administrator), the probability values reflect the difficulty of reaching a higher level attack goal, having achieved some lower level attack goal.

A potential problem when building the Bayesian network is to obtain a good source for the values used in the CPTs of all nodes. The question is then how to deal with possible imperfect knowledge when building Bayesian networks. We took two approaches to deal with this issue: (1) use data from past work and industry sources and (2) evaluate and measure in our experiments the impact such imperfect knowledge might have.

For the purposes of the experiments explained in sections 2.6 and 2.7, we have chosen the *junction tree* algorithm [**?**] to do inference, the task of estimating probabilities given a Bayesian network and the observations or evidence. There are many different algorithms that could be chosen, making different tradeoffs between speed, complexity, and accuracy. Still, the junction tree engine is a general-purpose inference algorithm well suited for our experiments since it works under our scenario: allows discrete nodes, as we have defined our two-states nodes, in direct acyclic graphs such as Bayesian networks, and does exact inference. This last characteristic refers to the algorithm computing the posterior probability distribution for all nodes in network, given some evidence.

2.4.2 Greedy Algorithm

We present here an algorithm to achieve an optimal choice and placement of detectors. It takes as input (i) a Bayesian network with all attack vertices, their corresponding CPTs and the host impacted by the attack vertex; (ii) a set of detectors, the possible attack vertices each detector can be associated with, and the CPTs for each detector with respect to all applicable attack vertices.

The DETECTOR-PLACEMENT algorithm (2.1) starts by sorting all combinations of detectors and their associated attack vertices according to their benefit to the overall system (line 1). The system benefit is calculated by the BENEFIT function (2.2). This specific design considers only the end nodes in the BN, corresponding to the ultimate attack goals. Other nodes that are of value to the system owner may also be considered. Note that a greedy decision is made in the $Benefit$ calculation each detector is considered singly. From the sorted list, (detector, attack vertex) combinations are added in order, until the overall system cost due to detection is exceeded (line 7). Note that we use a $costBenefit$ table (line 4 of $Benefit$ function), which is

Algorithm 2.1 DETECTOR-PLACEMENT (BN, D)

Input: (i) Bayesian network $BN = (V, CPT(V), H(V))$ where V is the set of attack vertices, $CPT(V)$ is the set of conditional probability tables associated with the attack vertices, and $H(V)$ is the set of hosts affected if the attack vertex is achieved.

(ii) Set of detectors $D = (d_i, V(d_i), CPT[i][j])$ where d_i is the ith detector, $V(d_i)$ is the set of attack vertices that the detector d_i can be attached to (i.e., the detector can possibly detect those attack goals being achieved), and $CPT[i][j]$ $\forall j \in V(d_i)$ is the CPT tables associated with detector i and attack vertex j.

Output: Set of tuples $\theta = (d_i, \pi_i)$ where d_i is the ith detector selected and π_i is the set of attack vertices that it is attached to.

```
    $systemCost = 0$
 1: sort all $(d_i, a_j), a_j \in V(d_i), \forall i$ by Benefit$(d_i, a_j)$. Sorted list = $L$
 2: length$(L) = N$
 3: **for** $i = 1N$ **do**
 4:   systemCost = systemCost + Cost$(d_i, a_j)$
 5:   /* Cost$(d_i, a_j)$ can be in terms of economic cost, cost due
 6:   to false alarms and missed alarms, etc. */
 7:   **if** $systemCost > threshold\ \tau$ **then**
 8:     break
 9:   **end if**
10:   **if** $d_i \in \Theta$ **then**
11:     add $a_j$ to $\pi_i \in \Theta$
12:   **else**
13:     add $d_i, \pi_i = a_j$ to $\Theta$
14:   **end if**
15: **end for**
```

likely specified for each attack vertex at the finest level of granularity. One may also specify it for each host or each subnet in the system.

The worst-case complexity of this algorithm is $O(dv\ B(v, CPT(v)) + dv \log(dv) + dv)$, where d is the number of detectors and v is the number of attack vertices. $B(v, CPT(v))$ is the cost of Bayesian inference on a BN with v nodes and $CPT(v)$ defining the edges. The first term is due to calling Bayesian inference with up to d times v terms. The second term is the sorting cost and the third term is the cost of going through the for loop dv times. In practice, each detector will be applicable to only a constant number of attack vertices and therefore the dv terms can be replaced by a constant times d, which will be only d considering order statistics.

The reader would have observed that the presented algorithm is greedy-choice of detectors is done according to a pre-computed order, in a linear sweep through the sorted list L (the for loop starting in line 3). This is not guaranteed to provide an optimal solution. For example, detectors d_2 and d_3 taken together may provide greater benefit even though detector d1 being ranked higher would have been considered first in the DETECTOR-PLACEMENT algorithm. This is due to the observation that the problem of optimal detector choice and placement can be mapped to the $0-1$ knapsack problem which is known to be NP-hard. The mapping is obvious consider $D \times A$ (D: Detectors and A: Attack vertices). We have to include as many of these tuples so as to maximize the benefit without the cost exceeding , the system cost of detection.

2.4.3 Cost–Benefit Analysis

We address the problem of determining the number and placement of detectors as a cost-benefit exercise. The system benefit is calculated by the BENEFIT function shown below. This specific design considers only the end nodes in the BN, corresponding to the ultimate attack goals. Other nodes that are of value to the system owner may also be considered in alternate designs.

Algorithm 2.2 BENEFIT (d_i, a_j)

```
1: //This is to calculate the benefit from attaching detector $d_i$ to attack vertex $a_j$
2: //$F$ is the set of end attack vertices $f_k$
3: $F \leftarrow \bigcup_{k=1}^{M} f_k$
4: **for all** $f_k \in F$ **do**
5:     perform Bayesian inference with $d_i$ as the only detector in the network and connected to attack vertex $a_j$
6:     calculate $Precision(f_k, d_i, a_j)$
7:     calculate $Recall(f_k, d_i, a_j)$
8:     $systemBenefit \leftarrow \sum_{i=1}^{m} \frac{(1+\beta_{d_i}^2)\left(Precision(f_k,d_i,a_j)\times(Recall(f_k,d_i,a_j)\right)}{\left(\beta_{d_i}^2\times Precision(f_k,d_i,a_j)+Recall(f_k,d_i,a_j)\right)}$
9:
10: **end for**
11: return $systemBenefit$
```

The BENEFIT function is used to calculate the benefit from attaching a detector to an attack vertex in the Bayesian network. To evaluate the performance of a detector, the algorithm uses two popular measures from statistical classification, *precision* and *recall*. Precision is the fraction of true positives (TP) determined among all attacks flagged by the detection system. Recall is the fraction of TP determined among all real positives in the system. Then, the BENEFIT function combines both measures into a single measure, $F_\beta - measure$ [32], which is the weighted harmonic mean of precision and recall and a popular method to evaluate predictors. β is the ratio of recall over precision, defining the relative importance of one to the other. The resulting $F_\beta - measure$ constitutes the output of the BENEFIT function and is called the *systemBenefit*, provided from attaching the detector to the Bayesian network.

The cost model for the system under analysis is defined by the following formula, corresponding to the expectation (in the probabilistic sense) of the cost:

$$COST(d_i, a_j) = \sum_{k=1}^{M} \big(Prob_{f_k}(TP) \times (cost_{respond}) + Prob_{f_k}(FP) \times (cost_{respond}) \\ + Prob_{f_k}(FN) \times (cost_{notrespond}\big)$$

We calculate the cumulative cost associated by selecting a detector, based on its different outcomes with respect to the end nodes: true positive (TP), false positive (FP), and false negative (FN). True negatives (TN) are not considered to compute the detector cost as we believe there should not be any penalty for correct classification of non-malicious traffic. The cost of positive (FP and TP) outcome is related to the response made by the detection system, whereas the FN cost depends on the damage produced by not detecting the attack.

In our design, all probability values (TP, FP, and FN) are first computed by performing sampling on the Bayesian network, since there are no real data (logs) when the system starts and placement of detectors is calculated for the first time. After the initial configuration is done and the system has been monitored for some time, the detection system can be reconfigured by using the log files collected to compute new probability values.

2.4.4 FPTAS Algorithm

The mapping of our DETECTOR-PLACEMENT problem to the 0-1 Knapsack problem allows us to utilize the existing algorithms for the popular NP-hard optimization problem. In particular, the Knapsack problem allows approximation to any required degree of the optimal solution by, as previously mentioned, using an algorithm classified as (FPTAS) since the algorithm is polynomial in the size of the instance n and the reciprocal of the error parameter ϵ. An FPTAS is the best possible solution for an NP-hard optimization problem, assuming of course that $P \neq NP$. The original FPTAS for the 0-1 Knapsack problem was given in [33].

A description of the FPTAS implemented for our experiments follows and is adapted from [34], [35]. The scheme is composed of two steps: first, the scaling of the benefit space to reduce the number of different benefit values to consider and second, running a pseudo polynomial time algorithm based on the dynamic programming technique on the scaled benefit space.

Step 1 - Scaling Step

To obtain the FPTAS, the benefit space is scaled to reduce the number of different profit values and effectively bound the profits in n, the input size. By scaling with respect to the error parameter ϵ, the algorithm produces a solution that is at least $(1-\epsilon)$ times the optimal value, in polynomial time with respect to both n and ϵ. The algorithm is as follows:

Algorithm 2.3 BENEFIT SPACE SCALING

```
1: Let $B \leftarrow$ benefit of the most profitable object
2: Given $\epsilon > 0$, let $E = \frac{\epsilon B}{n}$
3: $n \leftarrow$ length$[L]$
```

Step 2 - Dynamic Programming Step

Let W be the maximum capacity of the knapsack. All n items under consideration are labeled $i \in 1, \ldots, k, \ldots, n$ and each item has some weight w_i and a scaled benefit value b'_i. Then the Knapsack problem can be divided into sub-problems to find an optimal solution for S_k; that is the solution for when items labeled from 1 to k have been considered, but not necessarily included, in the solution. Then, let $B[k, w]$ be the maximum profit of S_k that has total weight $w \leq W$. Then, the following recurrence allows to calculate all values for $B[k, w]$:

$$B[k,w] = \begin{cases} B[k-1,w] & if w_k > w \\ max B[k-1,w], B[k-1,w-w_k] + b'_k & else \end{cases}$$

The first case of the recurrence is when an item k is excluded from the solution since if it were, the total weight would be greater than w, which is unacceptable. In the second case, item k can be in the solution since its weight (w_k) is less than the maximum allowable weight(w). We choose to include item k if it gives a higher benefit than if we exclude it. In the formula, if the the second term is the maximum value, then we include item k, and we exclude it if the first term is the maximum. The final solution $B[n, W]$ then corresponds to the set $S_{n,W}$ for which the benefit is maximized and the total cost is less or equal to W.

The running time of FPTAS is given by $O\left(\frac{n^2 B}{\epsilon}\right)$, and its design is based on the idea of trading accuracy for running time. The original benefit space of the 0-1 Knapsack problem is mapped to a coarser one, by ignoring a certain number of least-significant bits of benefit values, which depend on the error parameter ϵ. The mapped coarser instance is solved optimally through an exhaustive search by using a dynamic programming-based algorithm. The intuition, then, is to allow the algorithm to run in polynomial time by properly scaling down the benefit space. This thus provides a trade-off between the accuracy and the running time.

2.5 Experimental Systems

We created three Bayesian networks for our experiments modeling two real systems and one synthetic network. These are a distributed electronic commerce (e-commerce) system, a Voice-over-IP (VoIP) network, and a synthetic generic Bayesian network that is larger than the other two. The Bayesian networks were manually created from attack graphs that include several multi-step attacks for the vulnerabilities found in the software used for each system. These vulnerabilities are associated with specific versions of the particular software, and are taken from popular databases [24], [25]. An explanation for each Bayesian network follows.

2.5.1 E-Commerce System

The distributed e-commerce system used to build the first Bayesian network is a three tier architecture connected to the Internet and composed of an Apache web server, the Tomcat application server, and the MySQL database backend. All servers are running a Unix-based operating system. The web server sits in a de-militarized zone (DMZ) separated by a firewall from the other two servers, which are connected to a network not accessible from the Internet. All connections from the Internet and through servers are controlled by the firewall. Rules state that the web and application servers can communicate, as well as the web server can be reached from the Internet. The attack scenarios are designed with the assumption that the attacker is an external one and thus her starting point is the Internet. The goal for the attacker is to have access to the MySQL database (specifically access customer confidential data such as credit card information node 19 in the Bayesian network of Figure 2.4). A complete description of the Bayesian network used in the experiments is presented in Appendix A (Figures A.1 and A.2).

As an example, an attack step would be a portscan on the application server (node 10). This node has a child node, which represents a buffer overflow vulnerability present in the rpc.statd service running on the application server (node 12). The other attack steps in the network follow a similar logic and represent other phases of an attack to the distributed system. The system includes four detectors: IPtables, Snort, Libsafe, and a database IDS. As shown in Figure 2.4, each detector has a causal relationship to at least one attack step.

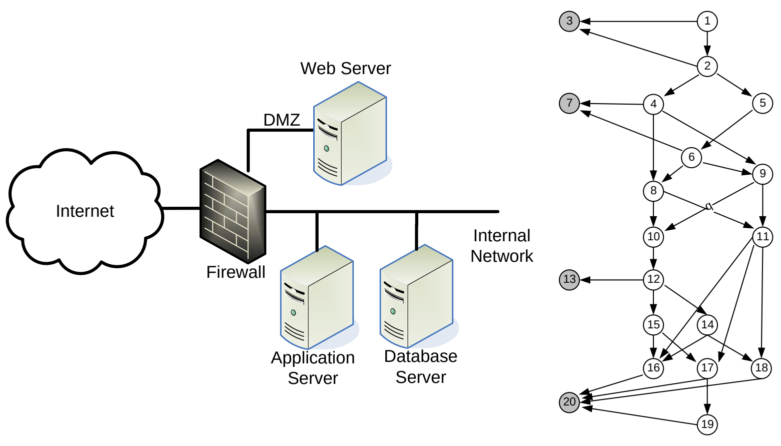


Fig. 2.4. Network diagram for the e-commerce system and its corresponding Bayesian network. The white nodes are the attack steps and the gray nodes are the detectors.

2.5.2 Voice-over-IP (VoIP) System

The VoIP system used to build the second network has a few more components, making the resulting Bayesian network more complex. The system is divided into three zones: a DMZ for the servers accessible from the Internet, an internal network for local resources such as desktop computers, mail server and DNS server, and an internal network only for VoIP components. This separation of the internal network into two units follows the security guidelines for deploying a secure VoIP system [26].

The VoIP network includes a PBX/Proxy, voicemail server and software-based and hardware-based phones. A firewall provides all the rules to control the traffic between zones. The DNS and mail servers in the DMZ are the only accessible hosts from the Internet. The PBX server can route calls to the Internet or to a public-switched telephone network (PSTN). The ultimate goal of this multi-stage attack is to eavesdrop on VoIP communication. There are 4 detectors: IPtables, and three network IDSs on the different subnets.

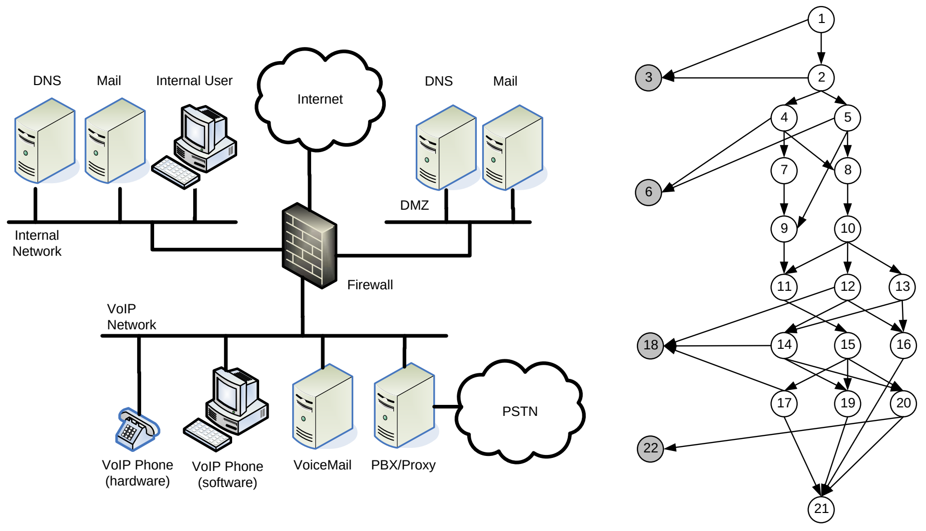


Fig. 2.5. VoIP system and its corresponding Bayesian network.

A third synthetic Bayesian network was built to test our framework for experiments where a larger network, than the other two, was required. This network is shown in Figure 2.7(a).

2.6 Experiments for Greedy Algorithm

The correct number, accuracy, and location of the detectors can provide an advantage to the systems owner when deploying an intrusion detection system. Several metrics have been developed for evaluation of intrusion detection systems. In our work, we concentrate on the precision and recall. Precision is the fraction of true positives determined among all attacks flagged by the detection system. Recall is the fraction of true positives determined among all real positives in the system. The notions of true positive, false positive, etc. are shown in Figure 2.6. We also plot the ROC curve which is a traditional method for characterizing detector performanceit is a plot of the true positive against the false positive.

For the experiments we create a dataset of 50,000 samples or attacks, based on the respective Bayesian network. We use the Matlab Bayesian network toolbox [27]

	Attack = True	*Attack = False*
Detection = True	*TP*	*FP*
Detection = False	*FN*	*TN*

$$Recall = \frac{TP}{TP+FN} \qquad Precision = \frac{TP}{TP+FP}$$

Fig. 2.6. Parameters used for our experiments: True Positive (TP), False Positive (FP), True Negative (TN), False Negative (FN), precision, and recall.

for our Bayesian inference and sample generation. Each sample consists of a set of binary values, for each attack vertex and each detector vertex. A one (zero) value for an attack vertex indicates that attack step was achieved (not achieved) and a one (zero) value for a detector vertex indicates the detector generated (did not generate) an alert. Separately, we perform inference on the Bayesian network to determine the conditional probability of different attack vertices. The probability is then converted to a binary determination whether the detection system flagged that particular attack step or not, using a threshold. This determination is then compared with reality, as given by the attack samples which leads to a determination of the systems accuracy. There are several experimental parameters which specific attack vertex is to be considered, the threshold, CPT values, etc. and their values (or variations) are mentioned in the appropriate experiment. The CPTs of each node in the network are manually configured according to the authors experience administering security for distributed systems and frequency of occurrences of attacks from references such as vulnerability databases, as mentioned earlier.

2.6.1 Experiment 1: Distance from Detectors

The objective of experiment 1 was to quantify for a system designer what is the gain in placing a detector close to a service where a security event may occur. Here we used the synthetic network since it provided a larger range of distances between attack steps and detector alerts.

The CPTs were fixed to manually determined values on each attack step. Detectors were used as evidence, one at a time, on the Bayesian network and the respective conditional probability for each attack node was determined. The effect of the single detector on different attack vertices was studied, thereby varying the distance between the node and the detector. The output metric is the difference of two terms. The first term is the conditional probability that the attack step is achieved, conditioned on a specific detector firing. The second term is the probability that the attack step is achieved, without use of any detector evidence. The larger the difference is, the greater is the value of the information provided by the detector. In Figure 2.7(b), we show the effect due to detector corresponding to node 24 and in Figure 2.7(c), we consider all the detectors (again one at a time). The effect of all the detectors shows that the conclusions from node 24 are general.

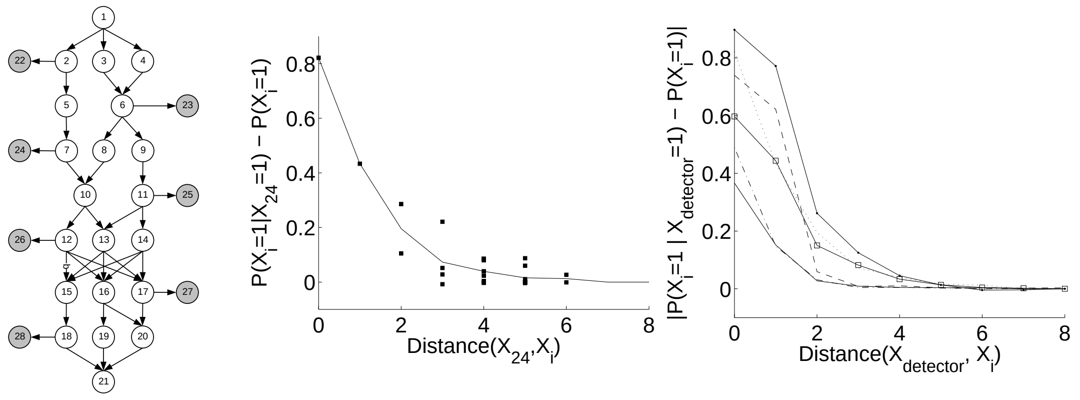


Fig. 2.7. Results of experiment 1: Impact of distance to a set of attack steps. (a) Generic Bayesian network used. (b) Using node 24 as the detector (evidence), the line shows mean values for rate of change. (c) Comparison between different detectors as evidence, showing the mean rate of change for case.

The results show that a detector can affect nodes inside a radius of up to three edges from the detector. The change in probability for a node within this radius, compared to one outside the radius, can be two times greater when the detector is used as evidence. For all Bayesian networks tested, the results were consistent to the three edges radius observation.

2.6.2 Experiment 2: Impact of Imperfect Knowledge

The objective of experiment 2 was to determine the performance of the detection system in the face of attacks. In the first part of the experiment *(Exp. 2a)*, the effect of the threshold, that is used in converting the conditional probability of an attack step into a binary determination, is studied. This corresponds to the practical situation that a system administrator has to make a binary decision based on the result of a probabilistic framework and there is no oracle at hand to help. For the second part of the experiment *(Exp. 2b)*, the CPT values in the Bayesian network are perturbed by introducing variances of different magnitudes. This corresponds to the practical situation that the system administrator cannot accurately gauge the level of difficulty for the adversary to achieve attack goals. The impact of the imperfect knowledge is studied through a ROC curve.

For Experiment 2a, precision and recall were plotted as a function of the threshold value. This was done for all the attack nodes in the Bayesian network and the results for a representative sample of six nodes are shown in Figure 2.8. We used threshold values from 0.5 to 0.95, since anything below 0.5 would imply the Bayesian network is useless in its predictive ability.

Expectedly, as the threshold is increased, there are fewer false positives and the precision of the detection system improves. The opposite is true for the recall of the system since there are more false negatives. However, an illuminating observation is that the precision is relatively insensitive to the threshold variation while the recall has a sharp cutoff. Clearly, the desired threshold is to the left of the cutoff point. Therefore, this provides a scientific basis for an administrator to set the threshold for drawing conclusions from a Bayesian network representing the system.

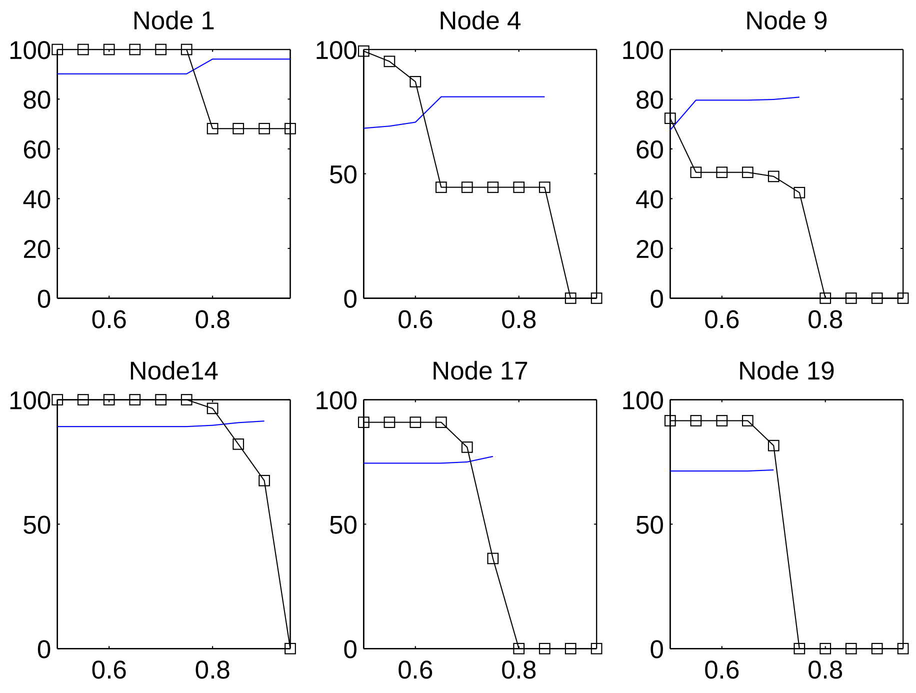


Fig. 2.8. Precision and recall as a function of detection threshold, for the e-commerce Bayesian network. The line with square markers is recall and other line is for precision.

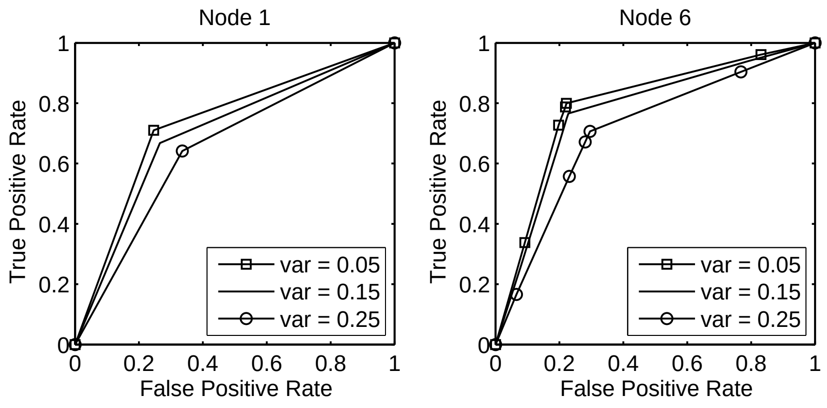


Fig. 2.9. ROC curves for two attack steps in e-commerce Bayesian network. Each curve corresponds to a different variance added to the CTP values.

In experiment 2b we introduced variance to the CPT values of all the attack nodes, mimicking different levels of imperfect knowledge an admin may have about the adversarys attack strategies. When generating the samples corresponding to the attacks, we used three variance values: 0.05, 0.15, and 0.25. Each value could be associated with a different level of knowledge from an administrator: expert, intermediate, and nave, respectively. For each variance value, ten batches of 1,000 samples were generated and the detection results were averaged over all batches.

In Figure 2.12, we show the ROC curves for nodes 1 and 6 of the e-commerce system, with all four detectors in place. Expectedly, as the variance increases, the performance suffers. However, the process of Bayesian inference shows an inherent resilience since the performance does not degrade significantly with the increase in variance. For node 1, several points are placed so close together that only one marker shows up. On the contrary, for node 6, multiple well spread out TP-FP value pairs are observed. We hypothesize that since node 1 is directly connected to the detector node 3, its influence over node 1 dominates that of all other detectors. Hence fewer number of sharp transitions are seen compared to node 6, which is more centrally placed with respect to multiple detectors.

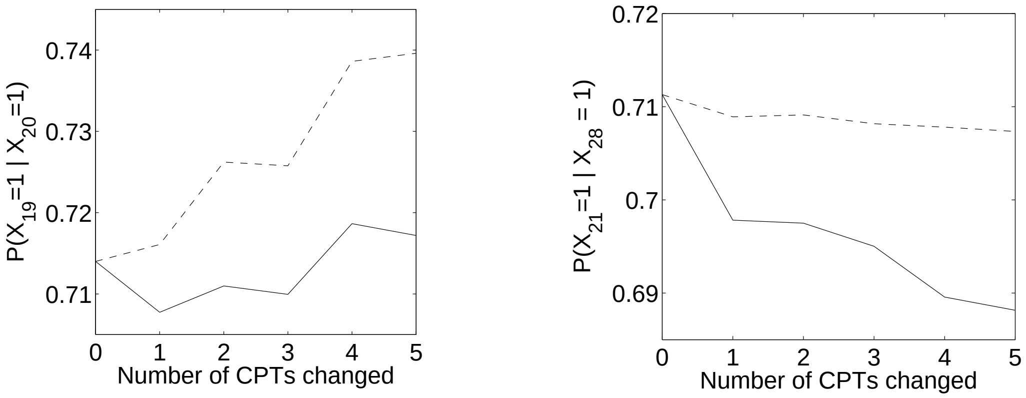


Fig. 2.10. Impact of deviation from correct CPT values, for the (a) e-commerce and (b) generic Bayesian networks.

Experiment 2c also looked at the impact of imperfect knowledge when defining the CPT values in the Bayesian network. Here we progressively changed the CPT values for several attack steps in order to determine how much we would deviate from the correct value. We used two values 0.6 and 0.8 for each CPT cell (only two are independent) giving rise to four possible CPT tables for each node. We plot the minimum and maximum conditional probabilities for a representative attack node for a given detector flagging. We change the number of CPTs that we perturb from the ideal values. Expectedly as the number of CPTs changed increases, the difference between the minimum and the maximum increases, but the range is within 0.03. Note that the point at the left end of the curve for zero CPTs changed gives the correct value.

Both experiments indicate that the BN formalism is relatively robust to imperfect assumptions concerning the CPT values. This is an important fact since it is likely that the values determined by an experienced system administrator would still be somewhat imperfect. Overall, as long as the deviation of the assumed CPTs from the truth is not overwhelming, the network performance degrades gracefully.

2.6.3 Experiment 3: Impact on Choice and Placement of Detectors

The objective of experiment 3 was to determine the impact of selecting the detectors and their corresponding locations. To achieve this, we ran experiments on the e-commerce and the VoIP Bayesian networks to determine a pair of detectors that would be most effective. This pair, called the optimal pair, is chosen according to the algorithm described in Section **??**. The performance of the optimal pair is compared against additional pairs selected at random. We show the result using the ROC curve for the two ultimate attack goals, namely node 19 and node 21 in the e-commerce and the VoIP systems.

To calculate the performance of each pair of detectors, we created 10,000 samples from each Bayesian network, corresponding to that many actual attacks. Then we performed Bayesian inference and calculated the conditional probability of the attack step, given the pair of detectors. We determined the true positive rate and false positive rate by sweeping across threshold values.

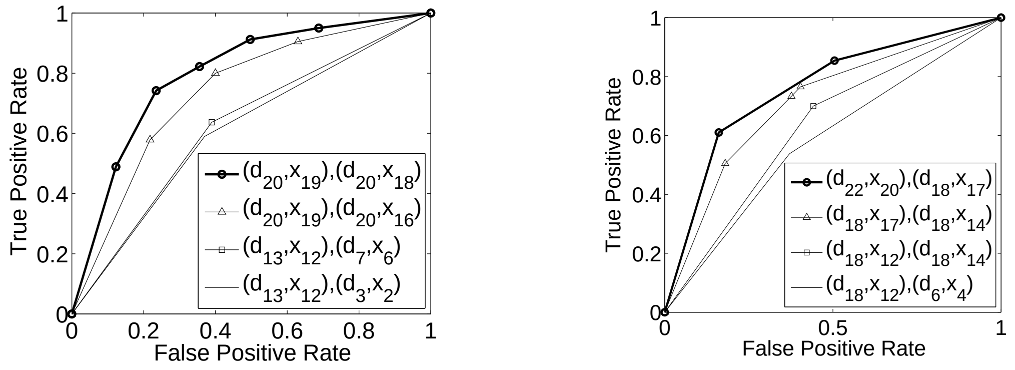


Fig. 2.11. ROC curves for detection of attack steps, using pairs of detectors, in the e-commerce network (left) and the VoIP network (right).

Results show that the pair of detectors determined from the algorithm performs better than the other randomly selected pairs. Figure 2.11a shows the situation in which a single detector (d_{20}) attached to two attack nodes (x_{19}, x_{18}) performs better than two detectors (d_{13} and d_7, or d_{12} and d_3). The placement of the detector d_{20} affects the performance. This can be explained by the fact that node 18 is more highly connected in the attack graph and therefore attaching detector d_{20} to that node, rather than node 16, provides better predictive performance.

There is a cost of adding detectors to a system, but there is also a cost of having a detector attached to more attack nodes, in terms of the bandwidth and computation. Thus adding further edges in the Bayesian network between a detector node and an attack node, even if feasible, may not be desirable. For the VoIP network, detector pair d_{22} and d_{18} performs best. This time two separate detectors outperform a single high quality detector (d_{18}) connected to two nodes.

Further details on all experiments performed, including all the probability values used for the Bayesian networks, are available at [28]. These are omitted here due to space constraints and the interested party is welcome to further read. All the experiments validate the intuition behind our algorithm that the greedy choice of the detectors also gives good results when multiple detectors are considered together and over the entire Bayesian network.

2.7 Experiments for FPTAS Algorithm

The correct number, accuracy, and location of the detectors can provide an advantage to the system's owner when deploying an intrusion detection system. Several metrics have been developed for evaluation of intrusion detection systems. In our work, as first presented in section 2.6, we concentrate on precision and recall. The notions of TP, FP, etc., are shown in Figure 2.6. We also plot the ROC curve which is a traditional method for characterizing detector performance – it is a plot of the true positive against the false positive.

For the experiments, we create a dataset of 50,000 samples or attacks, based on the respective Bayesian network. We use the Bayesian network toolbox for Matlab [27] for our Bayesian inference and sample generation. Each sample consists of a set of binary values, for each attack vertex and each detector vertex. A one (zero) value for an attack vertex indicates that attack step was achieved (not achieved), and a one (zero) value for a detector vertex indicates the detector generated (did not generate) an alert. Separately, we perform inference on the Bayesian network to determine the conditional probability of different attack vertices. The probability is then converted to a binary determination – whether or not the detection system flagged that particular attack step, using a threshold. This determination is then compared with reality, as given by the attack samples which leads to a determination of the system's accuracy. There are several experimental parameters – which specific attack vertex is to be considered, the threshold, CPT values, etc. – and their values (or variations) are mentioned in the appropriate experiment. The CPTs of each node in the network are manually configured according to the authors' experience administering security for distributed systems and frequency of occurrences of attacks from references such as vulnerability databases, as mentioned earlier.

2.7.1 Experiment 4: Comparison between Greedy algorithm and FPTAS

The objective of experiment 1 was to determine the performance of the FPTAS and compare it to the Greedy algorithm, using the Bayesian network for the e-commerce distributed system. The experiment was repeated for different capacity thresholds, representing cases for different numbers of detectors $(1, 2, 3$, or $4)$. For FPTAS, the algorithm used $\epsilon = 0.01$ since varying the parameter for different values, from 0.01 to 0.30, produced no relevant change on the resulting set of detectors and the running time was similar to the one from Greedy. More information on the running times is provided at the end of this section.

In all cases of the experiment, FPTAS performed better than Greedy algorithm in terms of achieving a higher benefit. FPTAS always first picked the (detector, location) pair $(d_{20,}a_{19})$ closer to the attack goal of interest and with the highest benefit, given the capacity constraint. Nevertheless, the selection of (detector, location) pairs was not accumulative as the capacity threshold was increased. As an example, when the threshold was set to 0.60 (representing the case for two detectors picked), FPTAS selected pairs $(d_{20,}a_{19})$ and $(d_{3,}a_{1})$ but when threshold was increased to 1.20 (three detectors), FPTAS selected $(d_{20,}a_{19})$, $(d_{20,}a_{17})$, and $(d_{3,}a_{2})$, removing $(d_{3,}a_{1})$ from the solution set. The reason for this situation is that as the capacity threshold is increased, it might include a detector with higher benefit and cost than one selected under the previous threshold considered.

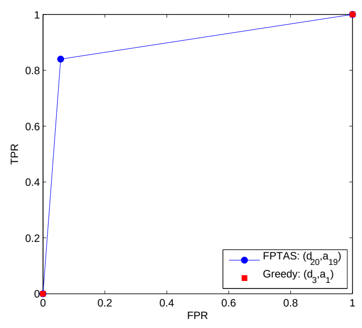
The performance of the Greedy algorithm was interesting as it always started selecting the $(d_{3,}a_{1})$ pair, regardless of the capacity threshold. This actually shows the drawback of the Greedy algorithm. It picks detectors that are accurate but are far from the ultimate attack goals that we are interested in. The case when the threshold was set to $W = 0.51$ (one detector) represented an example of a worst-case scenario

Table 2.1
Comparison between Greedy algorithm and FPTAS for different cost values.

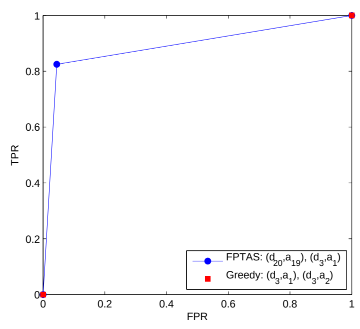
	Capacity (W=0.51)	Capacity (W=0.60)	Capacity (W=1.20)	Capacity (W=1.50)
Selections made by Greedy algorithm	(d_3, a_1)	(d_3, a_1) (d_3, a_2)	(d_3, a_1) (d_3, a_2) (d_{20}, a_{19})	(d_3, a_1) (d_3, a_2) (d_{20}, a_{19})
Benefit	0.64	0.61	1.46	2.05
Cost	0.12	0.42	0.88	1.27
Selections made by FPTAS algorithm ($\epsilon = 0.01$)	(d_{20}, a_{19})	(d_{20}, a_{19}) (d_3, a_1)	(d_{20}, a_{19}) (d_{20}, a_{17}) (d_2, a_3)	(d_{20}, a_{19}) (d_{20}, a_{17}) (d_3, a_1) (d_7, a_4)
Benefit	0.91	1.57	1.49	2.21
Cost	0.46	0.58	1.21	1,41

since the ratio between the optimal selection and the greedy choice was $\frac{1}{2}$. Still, as the threshold was increased, the Greedy algorithm seemed to correct itself and provide a solution closer to the optimal set. For cases of $W = 1.20$ and $W = 1.50$, the Greedy algorithm had all but one of the (detector, location) pairs that are part of the solution set chosen by FPTAS.

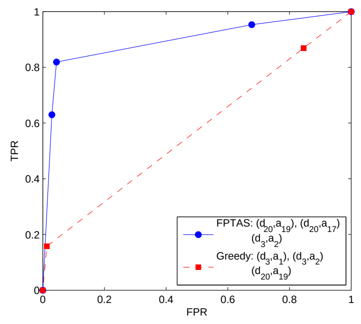
An interesting result is the cost associated to the detectors picked by the Greedy algorithm when compared to the choices made by FPTAS. In all our experiments, the selections made by Greedy had an overall lower cost and benefit than FPTAS. Although the cost value achieved by Greedy might be considered positive result, it is important to remember that our DETECTOR-PLACEMENT problem is an optimization problem where we try to maximize the benefit.



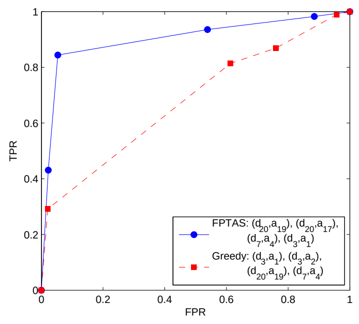
(a)



(b)



(c)



(d)

Fig. 2.12. ROC curves for detectors picked by Greedy (dashed line) and FPTAS (solid line) for different capacity values: (a) $W = 0.51$, (b) $W = 0.60$, (c) $W = 1.20$ and (d) $W = 1.50$.

The ROC curves shown in Figure 2.12 also represent the results from the experiment. In (a) and (b), the Greedy algorithm starts picking detectors too far away from the attack goal such that it doesn't have any (TPR,FPR) points, except for $(0, 0)$ and $(1, 1)$. We decided not to plot such lines because the performance of the detector(s) selected is no better than flipping a (fair) coin to determine if the attack goal has been achieved.

FPTAS performs better than the Greedy algorithm by immediately picking the detector closest to the attack goal, in the case of cost $= 0.51$ (one detector). In this case, FPTAS picks detector d_{20}, which is directly connected to the attack goal. In comparison, the Greedy algorithm starts by picking the detector farthest away from the attack goal. The reason for this is that such a detector has the highest benefit-to-cost ratio among all detectors. The problem is that this ratio does not reflect the actual performance of the detector for the attack goal. Such performance is shown in the corresponding ROC curve (one detector).

In the case of cost = 0.60 (two detectors), The Greedy algorithm follows a similar pattern as the previous case, picking the remaining detector with highest benefit-to-cost ratio. This detector is also far from the attack goal. In contrast, FPTAS picks a detector connected to an attack step, which is connected to attack goal a_{19}, and increasing the overall True Positive Rate of the detection system. Nevertheless, the same addition also increases the false positive rate.

For cases of cost = 1.20 (three detectors) and 1.50 (four detectors), the Greedy algorithm starts picking detectors closer to the attack goal that, as is shown in the corresponding ROC curves, perform relatively similar to the set of detectors selected by FPTAS.

In conclusion, FPTAS starts by selecting the closest (best) detector for the attack goal, and as it adds more detectors improves (marginally) the TPR of the detection system but with a price (also increasing the FPR). The Greedy algorithm selects from a decreasingly sorted list of detectors, according to their benefit-to-cost ratio. These are two examples of the first experiment we performed with Bayesian networks, where we demonstrated that as distance increases between detector and attack goal, the detection capability decreases.

We evaluated the running time for both algorithms by performing 100 execution runs for the Greedy algorithm and for each FPTAS with an error parameter (ϵ) from 0.0001 to 1. Figure 2.13 summarizes the findings on the running time for both algorithms. It shows the results for FPTAS, for ϵ from 0.0001 to 0.1 along with an exponential regression curve to fit the series of data points collected. In the case of the Greedy algorithm, we report a single value (represented by a straight line at 0.0523 seconds) since it is unaffected by ϵ. From the results, the Greedy algorithm ran faster than FPTAS, when the error parameter was less than 0.01. In our experiments, around that error value (0.01) they both showed similar running times. For the

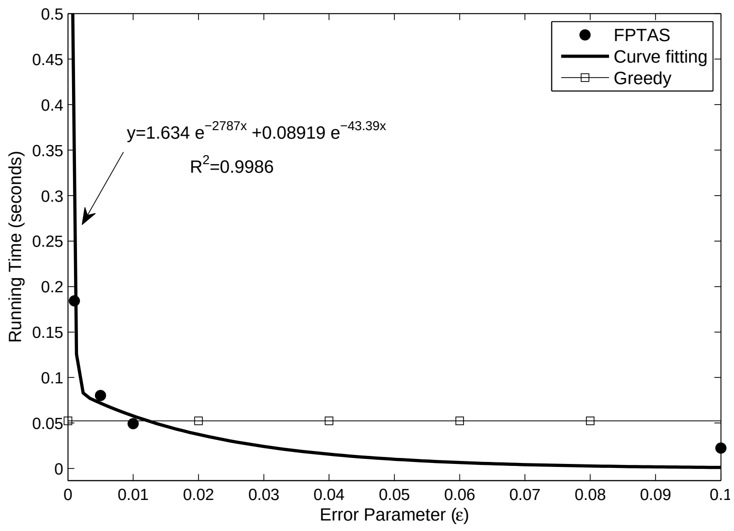


Fig. 2.13. Execution time comparison between Greedy algorithm and FPTAS, for different values of the error parameter (ϵ). In our experiments, values of ϵ equal or larger than 0.01 allow FPTAS to run faster than Greedy.

Table 2.2
Sensitivity analysis to different low cost values and Capacity $W = 0.90$.

Low	Greedy	FPTAS ($\epsilon = 0.01$)
0.1	(d_3, a_1)	(d_{20}, a_{19})
	(d_3, a_2)	(d_{20}, a_{17})
	(d_{13}, a_{12})	(d_3, a_2)
0.2	(d_3, a_1)	(d_{20}, a_{19})
	(d_3, a_2)	(d_{20}, a_{17})
0.3	(d_{20}, a_{19})	(d_{20}, a_{19})

Greedy algorithm it took on average 0.0523 seconds, while for FPTAS the average running time was 0.0499 seconds. We excluded from both algorithms the time taken to create the Bayesian network, the samples and to compute the probability values. All are necessary inputs for both algorithms and took 32.75 seconds on average to create and compute.

2.7.2 Experiment 5: Sensitivity to Cost Value

The objective of experiment 2 was to evaluate the impact of varying the quantitative value assigned to each cost category (low, medium, high). Three values were used for each category: low $(0.10, 0.20, 0.30)$, medium $(0.40, 0.50, 0.60)$ and high $(0.70, 0.80, 0.90)$. The experiment was repeated on both algorithms, FPTAS and Greedy algorithm, using the Bayesian network for the e-commerce distributed system and for different knapsack capacities $(0.51, 0.90, 1.20, 1.50, 2.00, 2.50,$ and $3.50)$. Such capacities correspond to the total resources available to the administrator to deploy and administer the detection system. For FPTAS, the algorithm used $\epsilon = 0.01$.

Tables 2.2 and 2.3 summarize the results for the low and medium cost values under two capacity scenarios. Table 2.2 is for capacity of 0.90, which corresponds to two detectors selected by both algorithms. Table 2.3 is for capacity of 2.0, which corresponds to five detectors selected.

Table 2.3
Sensitivity analysis to different medium cost values and Capacity $W = 2.00$.

Medium	Greedy	FPTAS ($\epsilon = 0.01$)
0.4	(d_3, a_1)	(d_{20}, a_{19})
	(d_3, a_2)	(d_{20}, a_{17})
	(d_{20}, a_{19})	(d_{13}, a_{12})
	(d_{20}, a_{17})	(d_3, a_2)
	(d_7, a_4)	(d_3, a_1)
0.5	(d_3, a_1)	(d_{20}, a_{19})
	(d_3, a_2)	(d_{20}, a_{17})
	(d_{20}, a_{19})	(d_{13}, a_{12})
	(d_{20}, a_{17})	(d_3, a_2)
	(d_{13}, a_{12})	(d_3, a_1)
0.6	(d_3, a_1)	(d_{20}, a_{19})
	(d_3, a_2)	(d_{20}, a_{17})
	(d_{20}, a_{19})	(d_{13}, a_{12})
	(d_{20}, a_{17})	(d_3, a_2)
	(d_{13}, a_{12})	(d_3, a_1)

Varying the quantitative value of a cost level seems to only slightly affect the outcome from the algorithms. In all cases, both algorithms are somehow consistent picking at least the first two or three (detector, attack node) pairs, while varying the quantitative value of the cost level. This is a positive result since the quantitative values are arbitrarily determined by the system administrator or person responsible for assessing the detection systems.

Comparing these results to the previous experiment, both the Greedy algorithm and FPTAS performed as they did in the previous experiment. In the case of Greedy algorithm, it keeps picking (d_3, a_1) before any other pair as this has the highest benefit-to-cost ratio. The FPTAS algorithm starts by picking (d_{20}, a_{19}) as it shows the highest benefit for the knapsack capacity constraint, regardless of the different values used for each cost level. In the case when a *(detector, attack node)* pair is changed because a level value has been changed, this can be explained from the impact the pair has on the individual cost assigned to each pair.

2.7.3 Experiment 6: ROC curves across Different Attack Graphs

The goal of this experiment is to show the performance of each algorithm, Greedy and FPTAS, by picking a pair of detectors for different attack goals. All attack goals in the e-commerce Bayesian network were used to evaluate the performance of the algorithms.

We decided to limit the size of the set of detectors picked to two, for each case and algorithm, since in our experience it is a reasonable number of detectors for a system administrator to use to defend a particular attack goal. Although such number would ultimately depend on several factors (for example: number of detectors available, the size of the network and its corresponding Bayesian network), we believe that the two-detector scenario allows us to show the behavior of each algorithm for the different attack goals considered.

Table 2.4 shows the detectors picked for each attack goal scenario and the corresponding algorithm used to picked the pair of detectors. Also, a ROC curve is created by averaging the FPR and TPR from the different attack goal scenarios.

The results from this experiment validate the observations from previous experiments. The Greedy algorithm starts by picking the detectors showing the highest benefits regardless of its distance from the attack goal. Still, since all attack nodes are considered as goals, there are several cases where the Greedy algorithm will pick

Table 2.4
Detectors selected by Greedy and FPTAS Algorithms for different attack goals (a_i)

Attack Goal	a_{19}	a_{18}	a_{17}	a_{16}	a_{15}	a_{14}	a_{12}	a_{11}
Greedy	(d_3, a_1)	(d_3, a_2)	(d_3, a_2)	(d_3, a_1)	(d_3, a_1)	(d_3, a_1)	(d_{13}, a_{12})	(d_3, a_1)
	(d_3, a_2)	(d_3, a_2)	(d_3, a_1)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)
FPTAS	(d_{20}, a_{19})	(d_{20}, a_{18})	(d_{20}, a_{17})	(d_3, a_1)	(d_3, a_1)	(d_3, a_1)	(d_{13}, a_{12})	(d_3, a_1)
	(d_3, a_2)	(d_3, a_2)	(d_3, a_1)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)
Attack Goal	a_{10}	a_9	a_8	a_6	a_5	a_4	a_2	a_1
Greedy	(d_3, a_2)	(d_7, a_4)	(d_3, a_2)	(d_7, a_6)	(d_3, a_2)	(d_7, a_4)	(d_3, a_2)	(d_3, a_1)
	(d_3, a_1)	(d_7, a_6)	(d_3, a_1)	(d_3, a_2)	(d_3, a_1)	(d_3, a_2)	(d_3, a_1)	(d_3, a_2)
FPTAS	(d_{13}, a_{12})	(d_7, a_6)	(d_{13}, a_{12})	(d_7, a_6)	(d_{13}, a_{12})	(d_{13}, a_{12})	(d_3, a_1)	(d_3, a_1)
	(d_7, a_4)	(d_7, a_4)	(d_3, a_2)	(d_3, a_2)	(d_3, a_2)	(d_7, a_4)	(d_3, a_2)	(d_3, a_2)

detectors close to the goal and with high benefit values. Therefore, the ROC curve (dashed line) shown in Figure 2.14 performs just slightly worse than in the case of the FPTAS algorithm. Looking at the choices made by FPTAS, it consistently picks detectors with a high benefit and close to the attack goal considered.

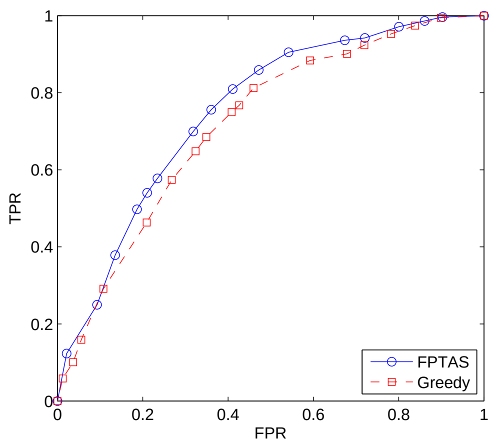


Fig. 2.14. ROC curves for detectors picked by Greedy (dashed line) and FPTAS (solid line) across all different attack goals in E-Commerce Bayesian network.

All the experiments validate the intuition that the algorithms can provide good results when multiple detectors are considered together and over the entire Bayesian network. The Greedy algorithm performed well under the scenarios considered, which we believe a good representation of the cases found in real-world systems. Still, as it is shown in Appendix B, there could be some scenarios for which the Greedy algorithm could produce results as low as half of the optimal solution. The FPTAS allows getting closer to the optimal solution as the algorithm is bounded by a polynomial in the size of the input and the reciprocal of the error parameter. In the experiments the FPTAS always selected a solution equal to or better than the Greedy algorithm, in terms of the benefit provided. As future work, we will test the algorithms under larger scenarios, which will determine the impact of the error parameter on the running time of the scheme.

2.8 Conclusions and Future Work

Bayesian networks have proven to be useful tools in representing complex probability distributions, such as in our case of determining the likelihood that an attack goal has been achieved, given evidence from a set of detectors. By using attack graphs and Bayesian inference, we can quantify the overall detection performance in the systems by looking at different choices and placements of detectors and the detection parameter settings. We also quantified the information gain due to a detector as a function of its distance from the attack step. Also, the effectiveness of the Bayesian networks can be affected by imperfect knowledge when defining the conditional probability values. Nevertheless, the Bayesian network exhibits considerable resiliency to these factors, as our experiments showed. Finally, we compared the performance of Greedy and FPTAS algorithms to determine a set of detectors given an attack goal. FPTAS consistently outperformed Greedy, although the latter could be used in scenarios where time constraints exist.

Future work will include looking at the scalability issues of Bayesian networks and its impact on determining the location for a set of detectors in a distributed system. The probability values acquisition problem can be handled by using techniques such as the recursive noisy-OR modeling [29] or using honeynets to monitor the behavior of attackers and compute the corresponding probability values. Experimentation is required to determine its benefits and limitations for our scenario.

3. SECURE CONFIGURATION OF INTRUSION DETECTION SENSORS FOR CHANGING ENTERPRISE SYSTEMS

3.1 Introduction

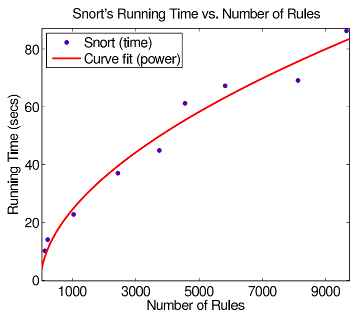
Current computer attacks against distributed systems involve multiple steps, thanks to attackers usually taking multiple actions to achieve their ultimate goal to compromise a critical asset. Such attacks are called multi-stage attacks (MSA). As today's enterprise systems are structured to protect their critical assets, such as, a mission-critical service or private databases, by placing them inside the periphery, MSAs have gained prominence. Examples include the breach of a large payment processing firm [37] and the breaches published by the U.S. Department of Health & Human Services [38]. MSAs are characterized by progressively achieving intermediate attack steps and progressing using these as stepping stones to achieve the ultimate goal(s). Thus, prior to the critical asset being compromised, multiple components are compromised. Logically, therefore, to detect MSAs, it would be desirable to detect the security state of various components in an enterprise distributed system—the outward facing services as well as those placed inside the periphery. Further, the security state needs to be inferred from the alerts provided by intrusion detection sensors (henceforth, shortened as "sensors") deployed in various parts of the system.

In the context of MSAs against distributed systems, this is challenging because sensors are designed and deployed without consideration for assimilating inputs from multiple detectors to determine how an MSA is spreading through the protected system. Prior work has shown that it is possible to determine the choice and placement of sensors in a systematic manner and at runtime, perform inferencing based on alerts from the sensors to determine the security state of the protected system components [36][1]. In achieving this, the solutions have performed characterization of individual sensors prior to deployment, in terms of their capability to detect specific attack step goals. At runtime, inferencing has been performed on the basis of the evidence—the alerts from the sensors—to determine the unobservable variables—the security state of the different components of the protected system. The sensors may be either network-based sensors, which observe incoming or outgoing network traffic, or host-based sensors, which observe activities within a host.

However, no existing solution has handled the various sources of dynamism that are to be expected in large-scale protected systems deployed in enterprise settings. The underlying protected system itself changes with time, with the addition or deletion of hosts, ports, software applications, or changes in connectivity between hosts. A static solution is likely to miss new attacks possible in the changed configuration of the protected system as well as throw off false alarms for attack steps that are just not possible under the changed configuration. The nature of attacks may also change with time or the anticipated frequencies of attack paths may turn out to be not completely accurate based on attack traces observed at runtime. Existing solutions cannot update their "beliefs" in an efficient manner and are therefore likely to be less accurate. Finally, when the compromise of a critical asset appears imminent, fast reconfiguration of existing sensors (such as, turning on some rules) may be needed to increase the certainty about the security state of the critical asset. Our contribution in this chapter is to show how the choice and placement of sensors can be updated through incremental processing when the above kinds of dynamism occur.

The solution we propose in this chapter called **D***istributed* **I***ntrusion and* **A***ttack* **D***etection* **S***ystem (DIADS)* is to have a central inferencing engine, which has a model of MSAs as attack graphs. DIADS creates a Bayesian Network (BN) out of an attack graph and observable (or evidence) nodes in the attack graph are mapped from sensor alerts. It receives inputs from the sensors and performs inferencing to determine

[1]In this chapter, we will refer to the distributed enterprise system that is being protected as the *protected system* and the set of sensors embedded in various components of the protected system as the *sensor system*.



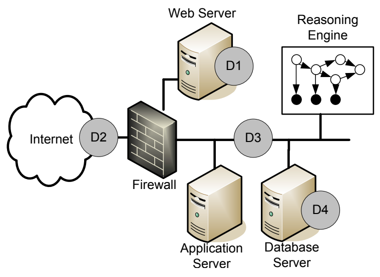


Fig. 3.1. (a) Results from curve fitting the data points from the Snort experiment. (b) General block diagram of the proposed DIADS. A wrapper (software) is used to allow communication from the sensors (circles labeled D1 to D4) and firewall to the reasoning engine and viceversa (only for sensors).

whether a rechoosing or replacement of sensors is needed. Further, it can reconfigure existing sensors, by turning on or off rules or event definitions based on the changed circumstances. Thus, the inferencing engine has a two-way communication path with the sensors. DIADS determines changes to the protected system by parsing changes to firewall rules at network points as well as at individual hosts and updates the BN accordingly. If on the basis of current evidence, it determines that a critical asset (also synonymously referred to as a crown jewel) will imminently be compromised, it determines what further sensors close to the asset should be chosen, or equivalently, what further rules in an already active sensor should be turned on.

One may think that a perfectly acceptable, and a much simpler, solution is to activate all the available sensors and turn on all the available rules at any sensor. Thus, there will be no reason to react to dynamic changes of the three types mentioned above. However, this will impose too high an overhead on the protected system in terms of the amount of computational resources that will be required and the frequency of false alerts that will be generated. For example, we determine empirically that for the popular Snort IDS [23] turning on the default set of rules will cause it to potentially take 85 seconds to match a single packet (corresponding to 9700 rules in

Figure 3.1). Therefore there is the motivation to dynamically reconfigure the sensors according to the activity observed in the network.

To sum up, in this chapter we make the following contributions:

1. We design a distributed intrusion detection system that can choose and place sensors in a distributed system to increase the certainty of knowledge about the security state of the critical assets in the system.

2. We imbue our solution with the ability to evolve with changes to the protected system as well as the kinds of attacks seen in the system.

3. Through domain-specific optimizations, we make our reasoning engine fast enough that it can perform reconfiguration of existing sensors while a multi-stage attack (MSA) is coursing through the protected system.

We structure the remainder of this chapter as follows. In Section 4.5 we review previous work on distributed intrusion detection systems (DIDS), MSA, and probabilistic approaches to intrusion detection. Section 3.3 states the problem studied and the threat model considered. Section 3.4 presents the proposed DIADS framework to detect MSAs and to reconfigure detection sensors, including a description of the different components and algorithms used. In Section 3.5 we provide a description of the experiments performed along with the results. Finally, Section 4.6 provides conclusions and future work.

3.2 Related Work

There has been previous work on developing and proposing DIDSs. Early examples of these systems are [39], [40], [41], and [42]. A starting point for DIDSs is the collaboration between Lawrence Livermore National Labs, U.S. Air Force and other organizations [39]. It represented the first attempt to physically distribute the detection mechanism, while centralizing the analysis phase in a single component, running a rule-based system.

Another distributed IDS is EMERALD [40]. It is a signature- and anomaly-based distributed IDS with statistical analysis capabilities (rule-based and Bayesian inference). The communication among sensors and monitors is structured in a hierarchy. NetSTAT [41] is a network-based IDS modeling intrusions as state transition diagrams and the target network as hypergraphs. By using both models, the system prioritizes which network events to monitor. AAFID [42] is a distributed framework based on software agents to collect and analyze data and used as a platform to develop intrusion detection techniques. An interesting policy-based proposal based on the popular Bro IDS [43] was presented in [44], using intrusion detection sensors in a distributed, collaborative manner.

Unfortunately there has not been much discussion about DIDS in the last few years so the impact of more complex distributed systems on the detection capabilities of IDS as well as the evolution of MSAs has been somewhat neglected. Previous work has primarily concentrated on increasing the accuracy of IDSs by improving their true positive (TP) rate on single step attacks. Additionally, it does not consider the dynamic nature of the protected system, one of our focus areas.

Previous work has considered MSAs [45], [46] but within a limited scope. [45] proposes an offline-method to correlate alerts using an attack graph, to improve detection rate, while reducing false positive (FP) and false negative (FN) rates. It is a rule-based method and does not consider a probabilistic approach. [46] presents a formal conceptual model based on Interval Temporal Logic (ITL) to express the temporal properties of MSAs.

A principal component for our framework is an attack graph, from which to create a corresponding Bayesian network. An example of previous work on using attack graphs for intrusion detection is found in [1]. Other works have previously focused on using attack graphs to evaluate (offline) the vulnerability state of the computer system [47].

Bayesian networks have been used for intrusion detection, examples include [36] and [48]. [36] models the potential attacks to a target network using a Bayesian network to determine (off-line) a set of detectors to protect the network. [48] presents a method based on Dynamic Bayesian networks to include the temporal properties of attacks in a distributed system.

Alert correlation is an area related to intrusion detection, that has received the attention of the research community. Schemes in this area can be classified under two basic groups: schemes that require patterns of actual attacks and/or alert interdependencies, and schemes that do not. Members of the first group include [14], [49], and [50]. Our proposed framework, can be classified as part of the first group. The second group of correlation schemes works without any specific knowledge of attacks. Examples include [51], [52].

In [14], the authors present a formal framework for alert correlation that constructs attack graphs by correlating individual alerts on the basis of the prerequisites and consequences manually associated to each alert. [49] presents techniques to learn attack strategies from correlated attack graphs. The basic idea is to compute how similar different attack graphs are by using error tolerant subgraph isomorphism detection. In [50] the authors built on the results from the previous two papers, integrating two alert correlation methods: correlation based on prerequisites and consequences of attacks and those based on similarity between alert attribute values. They used the results to hypothesize and reason about single attacks possibly missed by the IDSs. There are several similarities between their approach and ours. We both represent attack scenarios as graphs, assume attack steps are usually not isolated but rather part of an MSA. Still, there are also several differences between their alert correlation approach and ours. In a nutshell, our approach is adaptive, provides a larger visibility of the target network, follows a probabilistic model, and works online, while theirs is not.

3.3 Problem Statement and Threat Model

In this chapter, we answer two fundamental questions:
(1) How to update the configuration of sensors in an enterprise distributed system (i.e., one with many hosts and software applications and hence attack injection points) based on updated information that is obtained after the protected system and the sensor system have been deployed.
(2) When the imminent threat to a critical asset(s) is high, how to reconfigure existing sensors (such as, by activating new rules) to increase confidence in the estimate of the security state of the critical asset(s).

In terms of the model for the protected system, all the components fall target network under a single administrative domain and therefore, there is complete trust between the owners of the different assets.

The profile of the attackers includes highly motivated individuals that might have an economical incentive to compromise the distributed system. Attackers follow a multi-step approach to compromise a resource or acquire data. It could start with some reconnaissance, followed by exploitation of different hosts or services in the target network. This description also fits the cases where attack sources are botnets and malware, that does not include human intervention. We do not address intruders who steal data by physically connecting to a host (for example, an insider's attack using a USB memory stick).

In our framework, one or more *critical assets* are identified in the protected system by the system owner and become the main protection objective of our DIADS framework. Each critical asset is represented in the BN as a leaf node. An example of a critical asset is a database that contains personally identifiable information (PII). The above statement does not preclude having sensors that detect attacks at other assets (such as, at a network ingress point), but our inferencing uses such sensors to provide evidence of attacks leading up to a potential compromise of the critical assets. Also, our DIADS framework is not attempting to create better intrusion detection sensors; rather it is seeking to use existing sensors intelligently to obtain a better estimate of the security state of critical assets in an enterprise distributed system.

We consider only multi-stage attacks (MSAs) to distributed systems. An important example is an MSA to a three-tier system (web / application logic / database) which might allow an attacker to launch HTTP-based attacks to ultimately reach the database and the information stored in it.

3.4 DIADS Framework

In this document we propose a distributed intrusion detection framework that includes two components: (1) a probabilistic reasoning engine and (2) a network of detection sensors to detect various stages of MSAs, as shown in Figure 3.1. The second component comprises off-the-shelf sensors, augmented with a standard wrapper that allows the sensor to send alerts to the reasoning engine and receive commands back from the reasoning engine. The architecture is able to alert intrusion events related to potential MSAs and determine if any critical asset has been compromised, or is under imminent likelihood of being compromised based on current evidence of the spread of the attack. It also allows for reconfiguration of sensors according to changes to the protected system that is being monitored by the DIADS. Through this architecture, the DIADS can reduce the number of false positives that it would report if it were independently considering each step of the MSA. A block diagram of the proposed architecture is shown in Figure 3.2.

The reasoning engine represents different possible MSAs as a single Bayesian network, which is updated according to events reported by the detection sensors and the changing network configuration. The probabilistic engine can also request more information from sensors when necessary. The reasoning engine can estimate the security state of the critical assets given partial information about multi-stage attacks and from imperfect or noisy sensors.

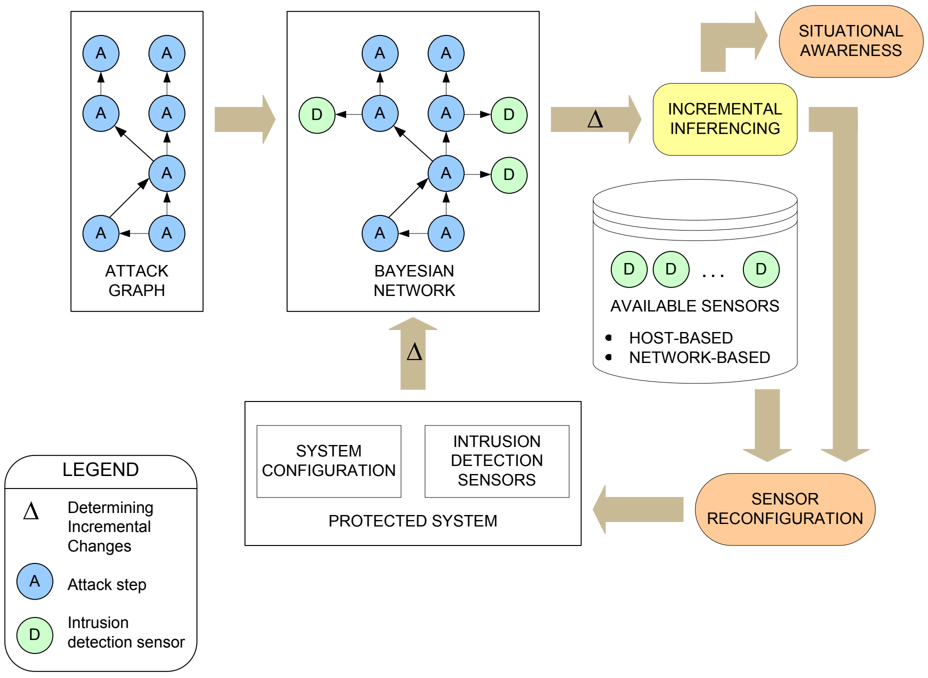


Fig. 3.2. Diagram of the proposed framework, providing details on the components of the reasoning engine.

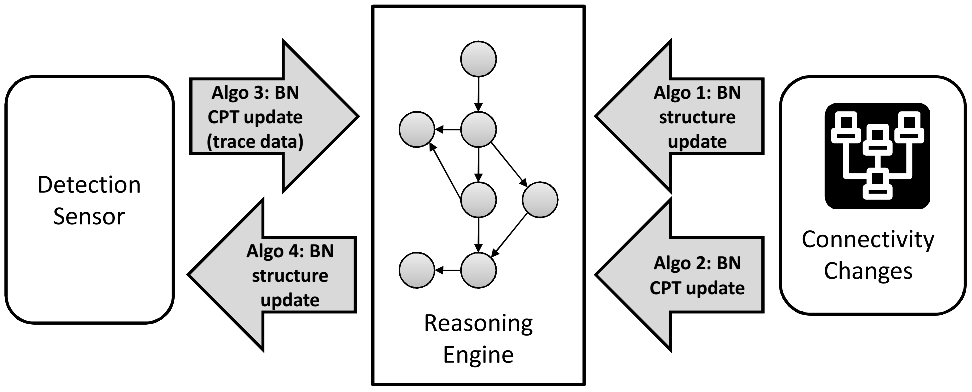

Detection Sensor
Algo 3: BN CPT update (trace data)
Algo 4: BN structure update
Reasoning Engine
Algo 1: BN structure update
Algo 2: BN CPT update
Connectivity Changes

Fig. 3.3. The framework uses four algorithms, three to update the reasoning engine and one to reconfigure the detection sensors.

The reasoning engine also collects background information about the distributed system so the model can be updated. As a starting point, we should consider the network and policy configurations stored in a firewall. The firewall can be at a network ingress-egress point as well as at individual hosts. The firewall configuration indicates which components are allowed to communicate with which components and thus has an important determining effect on the structure of the attack graph, and consequently, on the structure of the BN.

3.4.1 Probabilistic Reasoning Engine

To build our reasoning engine, we use Bayesian Network (BN), which is a popular probabilistic graphical model. It is a macro-language, representing joint distributions compactly by using a set of local relationships between random variables and specified by a graph. A key point is that the missing edges in the graph imply the conditional independence between the corresponding nodes. BN captures the characteristic in real-world data of *locality of influence*, the idea that most variables are influenced by only a few others. [36] shows the implications of this.

Bayesian networks combine graph theory with statistical techniques to model MSA scenarios. In our framework, we use an attack graph to create the structure of the BN, a directed acyclical graph. Each node in the graph represents a vulnerability, more specifically, a 3-tuple: $host \times port \times vulnerability$ existing in the target network. This means that the service running on that host and listening on that port has that vulnerability. The edges between nodes represent the direct precondition relationship between the attack steps. The BN also includes nodes to represent intrusion detection sensors. An edge $A \rightarrow D$ from an attack step node to a sensor node represents the possibility of the sensor detecting that attack step, with the CPT quantifying the accuracy and precision of the detection. Each node is parametrized by a set of probability values and represented as a *conditional probability tables (CPT)*. Proposed in previous work [36] and also suggested by [48], the Bayesian network representation can unify the information available from multiple sensors, in order to determine if an MSA is occurring.

There are several benefits of using Bayesian networks. First, it can be a more appropriate representation of reality than deterministic approaches, accounting for several sources of uncertainty—noisy sensors, unknown intentions of the adversary affecting the path of the MSA, and unknown difficulty of transitioning from one attack step node to the next. A potential drawback of probabilistic models is the combinatorial explosion faced when computing a joint probability distribution. In our work, we address this issue by using the Noisy-OR model [53] to represent the CPTs. Further details are provided in section 3.4.5. Our DIADS framework is composed of four algorithms, which are schematically shown in Figure 3.3. Pseudo-code for algorithms 1, 2, and 4 are provided in Appendix C.

3.4.2 Algorithm 1: BN update to structure based on Firewall rule changes

The algorithm produces a list of nodes and edges that should be added to (V_a, E_a) or deleted from (V_d, E_d) the Bayesian network to represent changes to the protected system. We use changes to firewall rules as a proxy for the changes to the protected system. The firewalls can be at a network ingress-egress point or at individual hosts in the system.

The message passed from the Firewall to the reasoning engine has the following structure: $message = < number, srcIPaddr, destIPaddr, portnumber, action, ruletype >$ where $number$ refers to the order of the rule in the firewall table. $srcIPaddr$ and $destIPaddr$ are the IP addresses for the source and destination of communication; *portnumber* is the TCP or UDP port number (16-bits in IPv4); *action* is one of three options: allow, deny or drop; and *ruletype* refers to the change made to the rule table: adding a new rule, modifying an existing rule or deleting an existing rule. For the purposes of our experiments, we only considered firewall rule tables composed of *allow* rules followed by a *denyall* rule. So effectively, the rule table creates a policy where allowed communication is explicitly defined and everything else not defined, is denied.

The algorithm can be divided into four parts: how to select the nodes and edges to be added, if the rule has type *add* (lines 1 to 11); how to select the nodes and edges to be deleted, if the rule has type *delete* (lines 13 to 29); checking for the resulting changes to the BN to not introduce cycles and to confirm that the resulting nodes are part of a path to the nodes representing the critical assets (lines 31 to 37); and finally, the converting the *destIPaddr:port* nodes into their corresponding *address:port:vulnerability* nodes in the BN.

When a rule has type *add* or *delete*, the algorithm checks if the source and destination addresses are new to the BN or already exist. If a node exists, then the edges shared with its parents (line 4) or its children (line 7) should be included to the set of edges to add (E_a). Also, the edge explicitly defined by the rule is included in (E_a). If a node is new, then it should be added to the set of nodes to add (V_a). A similar approach (but with opposite results) is used for case when a rule has type *delete*.

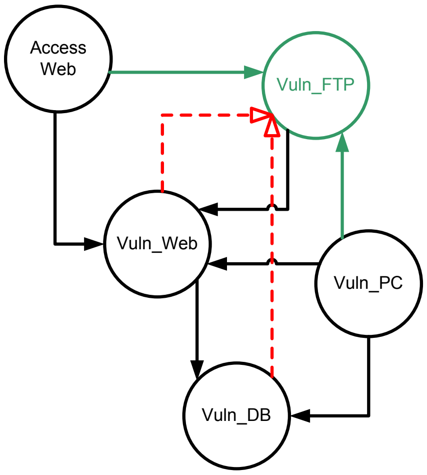
The algorithm then checks the nodes and edges in the resulting BN by running *Depth First Search* (DFS) to determine if the nodes have a path to the critical assets.

If the nodes do not, then they are pruned. DFS also checks if the addition creates any cycles and if so, the back edges are deleted. The first is an important optimization focusing the attention of DIADS to the critical assets and limiting the growth of the BN.

Finally, the algorithm transforms the nodes in the sets V_a and V_d to nodes in the BN. It does this by doing a lookup in a matrix R that maps the *host* $\times$ *port* to the vulnerability. It acquires the raw data for this from the *National Vulnerability Database* (NVD) [25], a public repository of vulnerability management data.

No.	Source	Destination	Action
1	Any	Web:80	Allow
2	Web	DB:3306	Allow
3	Web	DB:22	Allow
4	DB	Web:22	Allow
5	PC	DB:3306	Allow
6	PC	DB:22	Allow
7	Any	FTP:21	Allow
8	Any	Any	Deny

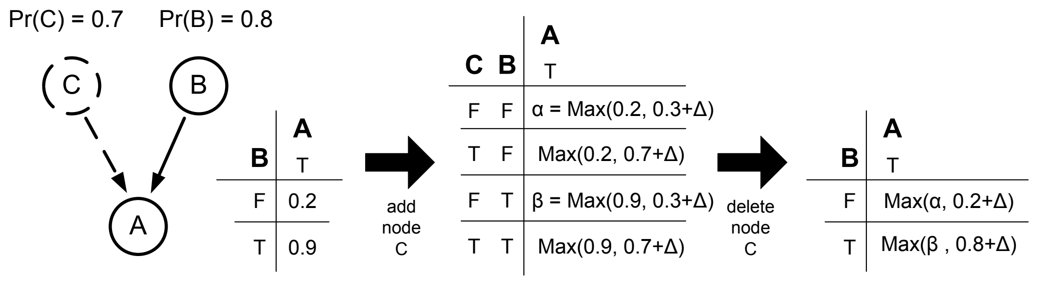
(a) Firewall rule table



(b) Bayesian network

Fig. 3.4. Impact of changes to a firewall rule. A new rule (No.7) in the firewall table changes the topology of the Bayesian network. Two of the four new edges, shown as dashed lines, will be removed by the algorithm since they lead to a cycle. A BN node is actually host $\times$ port $\times$ vulnerability, but here for simplicity, we have a single vulnerability per service (i.e. per host $\times$ port).

As an example, consider a distributed system connected to the Internet, with three computers: a web server (accessible from the Internet), a database server and a desktop computer. The database server and the desktop computer are connected to the same subnet, while the web server is connected to a separate subnet (DMZ). All computers are protected by a network-based firewall and the rule table is shown



B	A: T
F	0.2
T	0.9

C	B	A: T
F	F	α = Max(0.2, 0.3+Δ)
T	F	Max(0.2, 0.7+Δ)
F	T	β = Max(0.9, 0.3+Δ)
T	T	Max(0.9, 0.7+Δ)

B	A: T
F	Max(α, 0.2+Δ)
T	Max(β , 0.8+Δ)

Fig. 3.5. Example for algorithm 02: initialization of BN CPT. To add a new parent (C) to an existing node (A), we create the marginal probability $Pr(C)$ from its CVSS (base metric) value and use it to update the new CPT of A.

in Figure 3.4(a). A Bayesian network can be built from the table, as shown in Figure 3.4(b). The critical asset is the database server and for simplicity purposes, we have assumed one existing vulnerability per host.

If the rule `any` −− > `FTP:21 allow` is now added to the network firewall because a new FTP server has been deployed and connected to the DMZ network, the resulting Bayesian network is shown in Figure 3.4(b). A new node, `Vuln_FTP`, is added and will have five edges. Four are inbound, created from the added rule and one outbound, from rule No. 1 in the table. The inbound edges from nodes `Vuln_Web` and `Vuln_DB` are not included in the final Bayesian network as they make the graph cyclical.

3.4.3 Algorithm 2: Update of BN CPTs based on firewall changes

Algorithm 2 produces a list of CPTs for the changed nodes, i.e., nodes for which there is an increase or reduction in the number of parents of the nodes, according to the output from Algorithm 1. To update the CPT, we use the base metric value of the $CVSS$ score [54] of the node (corresponding to a vulnerability) to be added or removed and divide it by 10 to use it as its marginal probability value. Then if the resulting CPT is for an existing node, we take $max(newProb(v_i) + \Delta, oldProb(v_i))$. Figure 3.5 shows an example of how we use the formula.

In figure 3.5, first a new parent node C is added to an existing node A in the BN. We take the base metric score (7) of the vulnerability corresponding to node C and divide it by 10. Then use the formula $max(Prob(C) + \Delta, oldProb(A|\text{previous evidence}))$ to create the new CPT. In our experiments, we use $\Delta = 0.05$. Figure 3.5 also shows the CPT when node C is later removed. The base metric score of the other parent node (B) is used to update the CPT.

3.4.4 Algorithm 3: BN update of CPT based on incremental trace data

The alerts received by the reasoning engine from the individual sensors are used to update the CPTs in the Bayesian network in an incremental manner. To achieve this, this algorithm uses the set of alerts received during a window of time and the matrix R, that maps the existing vulnerabilities in the system to their corresponding hosts and ports. The output of the algorithm is the set of CPTs with the updated values.

The algorithm uses a popular and powerful model known as Noisy-OR [53] to represent each CPT. Noisy-OR allows us to specify the CPT of a node with n parents, using with $n + 1$ parameters as opposed to 2^n for binary nodes. This prevents the exponential growth experienced by the CPT of a node when the number of parents (n) is large. The Noisy-OR model assumes that effect of each parent on the CPT of the edge to the child node (v_i) is independent from that of the other parents and is sufficient to produce the effect (represented by the child node) in the absence of all other parents. An additional parent node is added to capture all other causes that were not modeled explicitly. The marginal probability of this node is $1 - p_0$. Then the CPT can be built with the following formula, where C represents a combination of the values for the parents of the child node:

$$Prob(v_i|C) = 1 - (1 - p_0) \prod_{A=parent(v_i)\in C} \left(\frac{1 - Prob(v_i|A = T, \text{Others} = \text{F})}{1 - p_0}\right)$$

3.4.5 Algorithm 4: Update choice of sensors based on runtime inference

The final algorithm of our framework is used to reconfigure the detection sensors. This includes adding and removing sensors, as well as reconfiguring existing ones. The high level objective is to reduce the uncertainty of knowing if the critical asset has been achieved or not. The algorithm works by looking at the alerts received and uses them as evidence to compute the posterior probability of each Bayesian network node that corresponds to the critical asset.

The first step of the algorithm (line 1) is to compute the posterior probability for the critical asset, given the evidence received from the currently enabled sensors in the system. If the value is larger than a threshold (determined by the system administrator), this is taken as indication that the critical asset is likely to be compromised and therefore greater certainty is needed in the determination of the security state. Therefore, the algorithm measures (lines 3 and 4) the impact of candidate sensors, which are close to the detected alerts and the critical asset. A radius can be set a priori in terms of the number of edges away from a particular node to determine the candidate set of sensors. Previous work [36] has shown that the effect of a sensor on a Bayesian network node fades beyond 2-3 hops and thus this restriction appears reasonable.

The algorithm determines a new set of detectors by using the *Fully Polynomial Time Approximation Scheme* (FPTAS) presented in [55] for the problem of determining the placement of intrusion detection sensors. The same cost bound is maintained which will prevent the algorithm from blissfully adding new sensors. This problem has been mapped to the *0-1* Knapsack problem for which a dynamic programming solution (FPTAS) exists that runs in *pseudo-polynomial* time (running time scales up as the solution approaches the optimal). The algorithm finishes by comparing the set of current detectors with the new set. The delta between the sets indicates the set of detectors to be disabled or enabled, which is output by the algorithm.

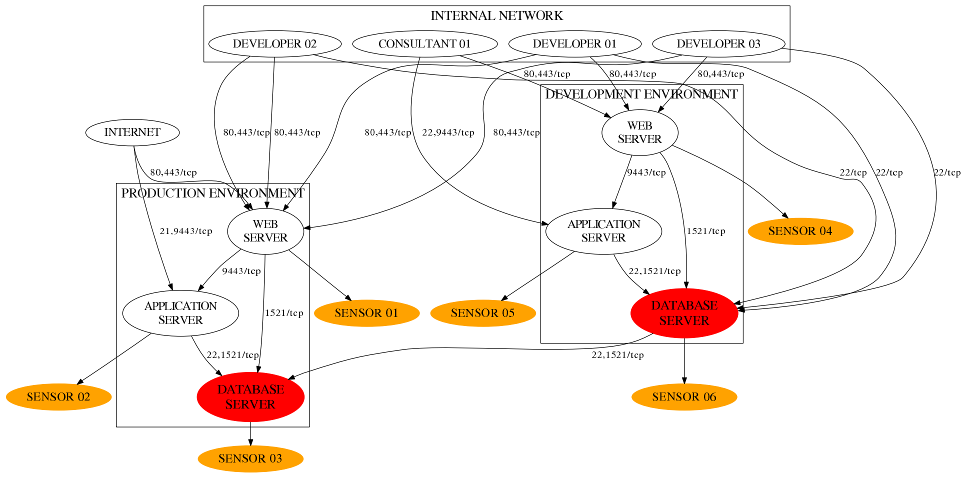


Fig. 3.6. Connectivity graph for testing scenario, showing the TCP ports enabled for communication between different hosts. The shaded nodes represent the critical asset (databases) in the protected system.

3.5 Experiments and Results

3.5.1 Experimental Setup

For our experiments, we used attacks against a real-world distributed system which is part of an NSF Center at our university and serves content and simulation tools for an engineering domain for thousands of users. The system includes fifteen hosts that include two environments, one for production and another for development of applications and staging prior to moving them to the production environment. Each environment includes a web server, an application server and a database server. A team of developers' and consultants' computers have access to subsets of both environments. Communication between all hosts is controlled by firewall rules at each host. The corresponding connectivity graph is shown in Figure 3.6.

In our experiments, the database servers are the critical assets to protect. A strong motivation to pick the databases is their role to store critical information for the organization. We created a Bayesian Network (BN) from the distributed system by first generating a list of the vulnerabilities found by the OpenVAS [56] vulnerability scanner on servers and client machines. Each vulnerability was then mapped to a node in the BN by associating it to the host and service(port) where the vulnerability was found. Finally, the nodes were connected according to the connectivity information for the distributed system. The BN had 345 nodes and 1948 edges. We then pruned the BN to only include high risk vulnerabilities, according to OpenVAS, as these ones are the primary vectors used by attackers to compromise systems. The final BN had 90 nodes and 582 edges and is presented in Appendix D.

We provide comparative results between DIADS (our algorithms presented in this chapter) and a static and heuristic-driven choice of sensors. All results are presented as *Receiver Operating Characteristics* or *ROC curves* [57]. The curve is a graphical plot of the tradeoff between true positive rate (TPR, detection rate) and the false positive rate (FPR, false alerts) for a detector. The different points in the ROC curves are generated by varying the threshold for the probability value for the BN nodes corresponding to the critical assets.

We had a total of 18 possible sensors; 3 sensors for each of the web server, application server, and database server, in both the development and the production environments. They are all generic sensors with signatures customized to detect the class of attack into which the corresponding (vulnerability) node can be categorized. For all experiments, for both baseline and DIADS, we constrain the algorithms to pick a set of 6 from 18 possible detectors.

It is important to note that DIADS' goal is to improve the performance of a set of detectors, by considering temporal information (i.e. when detectors are sending alerts about a progressing attack or when changes occur to the distributed system). For our experiments, we defined detectors with adequate but not perfect performance (in terms of TP and FP). It is not our goal to improve the performance of individual detectors.

3.5.2 Experiment 1: Dynamic Reconfiguration of Detection Sensor

The first experiment compared the performance between a dynamic reconfiguration of sensors and an static set of sensors, all close to the database servers. The static setup follows the intuitive decision of turning on all the sensors at the critical assets, in this case the database servers. To test both setups, we use an attack scenario that had the following progress: the attack started from the Internet, compromised the production web server, from where to compromise the applications server and then elevate permissions, and finally compromise the database server. Further details for all attack scenarios and the Bayesian network used in all experiments, are provided in [58].

In this experiment, a set of alerts are generated for the first three steps of the attack scenario. This set serves as evidence and is provided to the reasoning engine for DIADS to recompute the set of sensors. As shown in Figure 3.7, the dynamic reconfiguration setup outperforms the static configuration of sensors. The area under the continuous line (dynamic) is greater than the area under the dotted line (static) by 16% ($Area_{DIADS} = 0.7810$ and $Area_{baseline} = 0.6728$). This also means, the dynamic setup provides a higher detection rate at points when both setups have the same false alarm rate.

A notable point is that the difference between both setups is not large. This should be expected as the static setup is concentrated around the database servers (the critical asset and final setup in the attack scenario) while the dynamic setup is scattered around the protected system.

3.5.3 Experiment 2: Dynamism from Firewall Rules Changes

Experiment 2 tested the performance of the dynamic and static setups as changes were made to the firewall rule table of the protected system. We considered two real scenarios: (1) removing from the system a host belonging to a developer and (2) adding a direct communication path is created from a consulant's host to the database

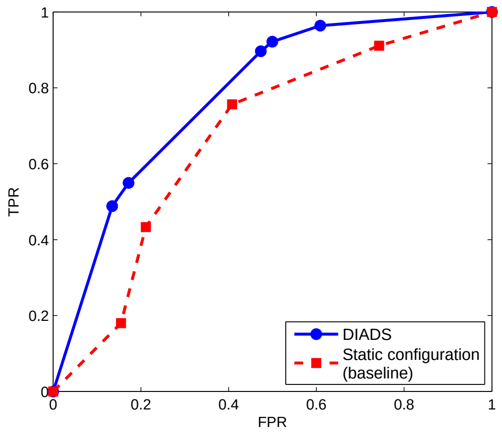
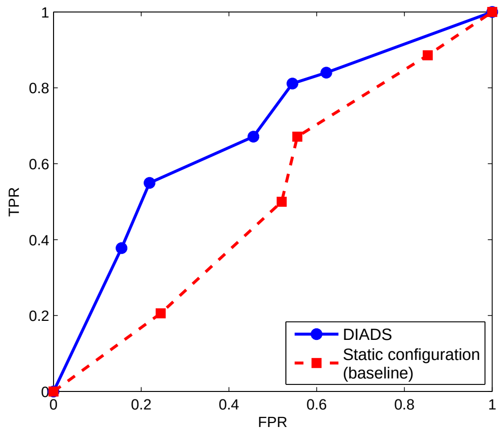


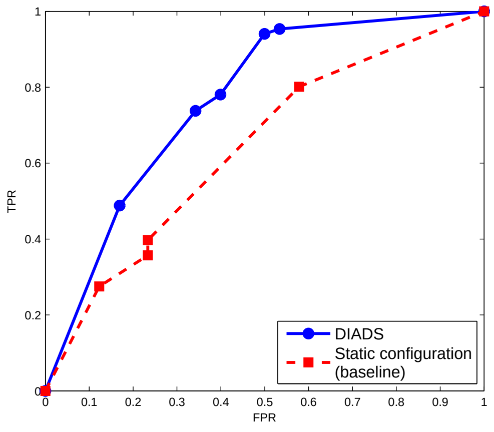
Fig. 3.7. Performance comparison between dynamic configuration of DIDS and a set of detectors monitoring only DB servers.

server, in the development environment (in this case, the consultant determined some changes to the database schema had to be tried out in the development environment prior to unveiling it on production). For the static configuration, one sensor was deployed on each host in both development and production environments.

For the first firewall change where a developer's host was removed, we tested both setups using an attack scenario starting from another developer's host. This represents the increasingly common client-side attacks. The attack starts as the developer visits a malicious website that installs some malware on the host. Then permissions are elevated thanks to another existing vulnerability in the developer's host. Then a vulnerability in the database server (production) is exploited and finally another vulnerability is used to access the data in the database. For the second firewall change where a direct communication path is created, we used a different attack scenario. The attack starts from another developer's host that also visits a malicious website and malware is installed in the host. Then a vulnerability in the web server (development) is exploited, after which the application server and finally the database server, all part of the development environment, are compromised.



(a) Removing a host

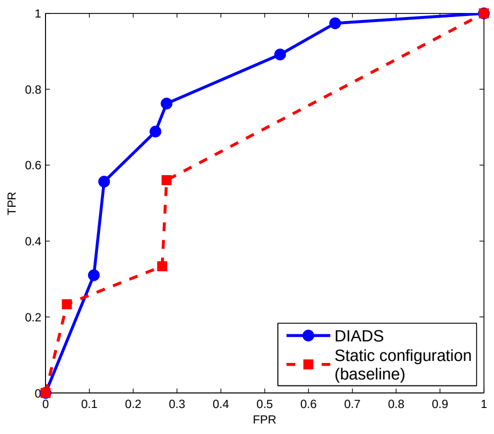


(b) Opening ports to DB server

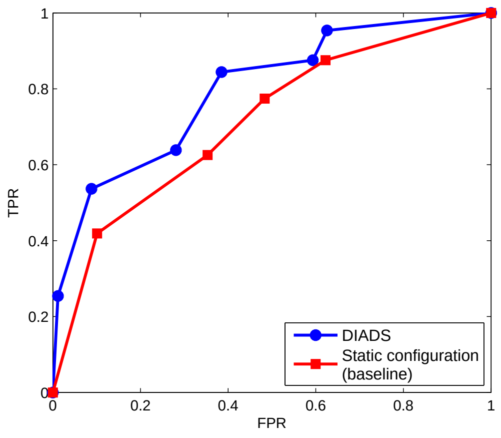
Fig. 3.8. Impact on topology changes. (a) Removing a host (developer) from network. (b) Allowing direct access between the consultant box and the DB development server.

For DIADS, the BN was modified based on the firewall rule changes and the dynamic programming algorithm picked the set of detectors after receiving the alerts at the start of the attack - the starting point being the same as in the static case.

Results from this experiment are shown in Figures 3.8(a) and 3.8(b). The dynamic reconfiguration setup performs better under both attack scenarios than the static configuration. The area under the curve is greater by 32.7% ($Area_{DIADS} = 0.6809$ and $Area_{baseline} = 0.5132$) in the scenario when a host is removed and 20% ($Area_{DIADS} = 0.7659$ and $Area_{baseline} = 0.6383$) in the scenario when a direct access is set up between a consultant box and the DB development server. We consider the most interesting result to be in Figure 3.8(b), where both setups show similar performance at the start. Both lines in the ROC curve have similar slopes, which is expected as the dynamic and static setups share 4 out of the 6 initial sensors. But as the alerts from the first three attack steps are provided to the reasoning engine in the dynamic setup, three sensors are reconfigured. This is the cause of the difference in performance, as shown in the ROC cuve.



(a) Attack from the Internet



(b) Attack from the internal network

Fig. 3.9. Comparison between our dynamic technique and a static setup for two attacks scenarios. The dynamic reconfiguration technique allows to reconfigure the detection sensors as alerts from the initial steps of the attacks are received.

3.5.4 Experiment 3: Dynamism with Attack Spreading

The goal of this experiment was to see if DIADS can reconfigure sensors on the fly as an attack spreads through the protected system. We used two different attack scenarios: (1) one starting from the Internet and (2) another starting from the internal network, a developer's host. An attack starting from the internal network usually requires less steps to reach the critical asset than attacks starting from the Internet. The static configuration picks sensors as in the earlier experiment 2 (one for each host).

The results are presented in Figures 3.9(a) and 3.9(b) for the two attack scenarios. In the attack starting from the Internet, the static setup shows a lower false alerts rate than the dynamic setup. But as evidence is provided, the ROC curve shows that the dynamic setup improves its performance. The curve shows the importance of taking into account the alerts from the initial stages of the attack to improve the performance of detection system. The improvement over the static setup, in terms of the area under the curve, is 23% ($Area_{DIADS} = 0.7845$ and $Area_{baseline} = 0.6366$). During the experiments, 4 of the 6 original sensors are replaced by the reasoning engine.

For the attack starting from internal network, the ROC curve in Figure 3.9(b) shows a similar performance between both setups. Three of the six sensors selected for the static setup are on the attack path and are quite accurate. Therefore, though DIADS outperforms the static setup, the advantage is not very large (11% where $Area_{DIADS} = 0.7964$ and $Area_{baseline} = 0.7128$). This experiment shows promise that inferencing in BN can be done fast enough relative to the speed of attacks. Of course, further experimentation is needed with a variety of attacks (and attack speeds).

3.6 Conclusions and Future Work

Current attacks to distributed systems involve multiple steps, with the ultimate goal of compromising a critical asset such as a database where important data is stored for an organization. In this chapter, we presented a distributed intrusion detection system called DIADS that picks and places sensors in a protected system, decreasing the uncertainty inherent in estimating the security state of the critical assets in the system. DIADS has the ability to evolve when changes are made to the topology of the protected system and with further evidence coming in the form of alers while the deployed system is operational.

Future work will include experimenting further with the size of the Bayesian network. We consider we made reasonable assumptions when pruning the Bayesian network, such as only including high risk vulnerabilities as nodes. Still, as the size of the CPTs for the nodes in the Bayesian network grows exponentially in terms of the number of nodes' parents, we would like to answer the question of whether inferencing can be done fast enough. Another area to explore is the impact of evasion techniques or attacks directly targeted against DIADS. If an attacker has complete knowledge of our model, she might launch attacks to falsely cause reconfiguration of our sensors away from the attack paths.

4. WEBCRAWLING TO GENERALIZE SQL INJECTION SIGNATURES

4.1 Introduction

Network intrusion detection systems (NIDS) are an important and necessary component in the security strategy of many organizations. These systems continuously inspect network traffic to detect malicious activity and when this happens, send alerts to system administrators. One type of NIDS, called *misuse-based detector*, uses signatures of attacks to inspect the traffic and flag the malicious activity. But a potential problem faced by these signature-based NIDS is that as new attacks are created and as new kinds of benign traffic are observed, the signatures need to be continuously updated. The current approach to creation of the signatures is manual. Consequently, keeping them updated is a Herculean task that involves tedious work by many security experts at organizations that provide the NIDS software. A big drawback of the signature-based schemes that has been pointed out by many researchers and practitioners [59], [60] is that due to their relatively static nature, they miss zero-day attacks. These are attacks that target hitherto unknown vulnerabilities and consequently, no signature exists for such attacks. Our goal in this work is to automatically generate signatures by performing data mining on attack samples. Further, we aim to create generalized signatures; "generalized" implies the signatures will be able to match some zero-day attacks as well, not just the attack samples that it has been trained on.

We look at the rulesets of three popular misuse-based detectors—Snort, Bro, and ModSec. From this, we observe the reflection of the ad hoc manual nature of the signature creation (and update) process. We observe that many rulesets include signatures that are too specific, many are disabled by default, and several show clear overlaps. For example, 70% of the almost 20,000 signatures included in the ruleset for Snort [30] (snapshot 2920) are disabled by default. The ruleset file `sql.rules` includes different sets of signatures that could be merged as one. For example, signatures with identifiers 19439 and 19440 have the same regular expression, except for the last character and hence could easily be merged. Further, we found multiple examples of signatures using very simple regular expressions, which increases the risk of generating false positive (FP) alerts. For example, several rules in Snort use the regex `.+UNION\s+SELECT` which includes the SQL statements `UNION` and `SELECT` to detect SQL injection attacks but is also popular in queries from web applications to databases.

In this chapter, we propose a solution for the automatic creation of generalized signatures represented as regular expressions, by applying a sequence of two data mining techniques to a corpus of attack samples. The goal is to make the process less manual (and tedious) and target detection of zero-day attacks. We call our solution *pSigene* (pronounced `‘‘psy-gene’’`), which stands for **p**robabilistic **Si**gnature **gene**ration. *pSigene* follows a four-step process. In the first step, it crawls multiple public cybersecurity portals to collect attack samples. In the second step, it extracts a rich set of features from the attack samples. In the third step, it applies a specialized *clustering* technique to the attack sample (training) data collected in step 2. The clustering technique also gives the distinctive features for each cluster. In the last step a generalized signature is created for each cluster, using *logistic regression* modeling, which is trained both on attack sample data (from step 2) and some benign traffic data. Logistic regression gives a probabilistic classifier — given a new sample, it can tell with a probability value what is the likelihood of the sample belonging to any given cluster.

There exists a variety of cybersecurity portals from which attack samples can be gleaned (step one of the process outlined above), including SecurityFocus [24], the Open Source Vulnerability Database [61], the Exploit Database [62] and PacketStorm Security [63]. It is crucial to collect a diverse and comprehensive set of attack samples

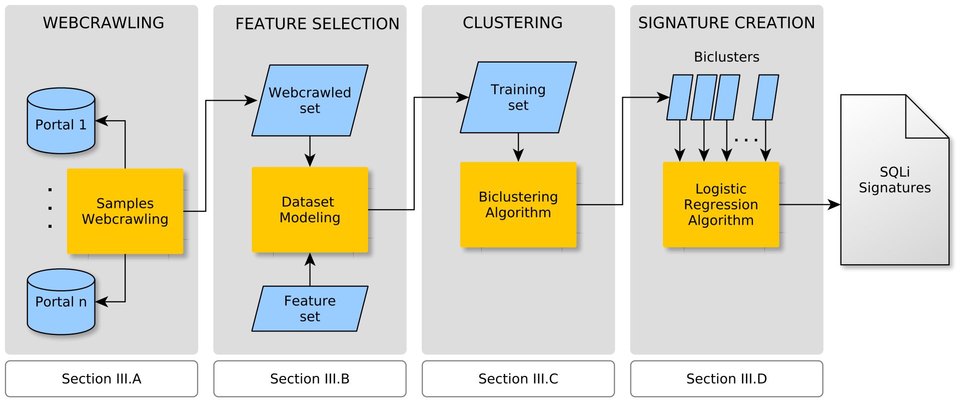


Fig. 4.1. Components of the pSigene architecture, a webcrawl-based creation process for SQLi attack signatures. For each component, there is a reference to the section providing further details. It is shown below each component.

for training the algorithm, which will create the clusters. Regular expressions (regexs) are a structural notation for describing similar strings. Regexs are a powerful tool used to define languages, per the automata theory definition. Current NIDS have incorporated the usage of regexs to generalize signatures, so variations of attacks can be detected. We adopt the use of regexs for our generalized signatures.

Most of the efforts to date to automate the signature creation process has been related to malware activity [64], [65], such as for worms and botnets. This landscape is different from ours in that we target attack steps that have a small "distance" from legitimate activity. Consider for example, SQL injection attacks, which we use to demonstrate and evaluate *pSigene*. A small variation in the `where` clause, such as a tautology "1 == 1", followed by a comment demarcation symbol ";", can cause a legitimate-looking SQL query to become an attack sample. Second, we consider activities where the feature set of each sample is very rich. For example, we first started with 477 features for SQL injection attacks, corresponding to various keywords, symbols *and their relative placements*. The rich feature set poses challenges and constraints on the kinds of machine learning techniques that can be used.

We demonstrate our solution specifically with SQL injection attacks (shortened as SQLi attacks). Although there exist elegant preventive solutions to solve this problem, like parameterizing SQL statements [66] and escaping special SQL characters [67], in practice it seems elusive to completely implement such solutions. SQLi attacks have been very dominant in the last couple of years, being used in high-profile cases such as intrusions to large technology organizations [68], government agencies [69], and software companies [70], [71]. Signatures to improve detection mechanisms, such as what *pSigene* delivers, are necessary as they complement prevention mechanisms.

pSigene effectively suggests the number of signatures necessary to detect the attacks while helping to reduce the size of each signature, in terms of the number of features necessary to define each signature. In our experiments, *pSigene* collected a set of 30,000 attack samples from which we extracted a set of 159 features and created nine signatures, all but one of which required 14 or fewer features. For testing, we used the popular SQL injection tool called SQLmap [72]. The experimental results showed that our signature set was able to detect 86.53% of all attacks while only generating 0.037% of false positives. This is a higher true positive rate for SQLi than Snort (79.55%) and Bro (73.23%), which use manually created and progressively refined signatures. Bro had no false positive while Snort had about 5X false positives compared to *pSigene*. ModSecurity however performed better than *pSigene* with a true positive rate of 96.07%, and a false positive rate only slightly worse (0.0515% compared to our 0.037%). *pSigene* allows for tuning the relative true positive and false positive rates by varying the threshold for the probabilities that are given by the logistic regression process.

The contributions of this work are:

1. An automatic approach to generate and update signatures for misuse-based detectors.

2. A framework to generalize existing signatures. The detection of new variations of attacks is achieved by using regular expressions for the generalized signatures.

3. We rigorously benchmark our solution with a large set of attack samples and compare our performance to popular misuse-based IDS-es. Our evaluation also brings out the impact of practical use case whereby periodically new attack samples are fed into our algorithm and consequently the signatures can be progressively, and automatically, updated.

The remainder of this chapter is structured as follows: Section 4.2 presents the threat model used along with the different components for the proposed framework. Section 4.3 describes the dataset used to evaluate *pSigene*, the evaluation results along with a comparison to existing open-source rulesets. We also determine the performance implications of using our approach. A discussion follows in Section 4.4 about the usage scenarios and limitations of our approach. Then we give an overview of previous approaches to automatically generating signatures and detecting attacks through interactions between web services and databases. We end with some conclusions and future work.

4.2 Framework Design

The goal of *pSigene* is to generate generalized signatures from traces of attack samples and non-malicious network traffic. As shown in figure 4.1, the generation of the signatures involves four phases. First multiple public cybersecurity portals on the Internet are crawled to collect attack samples. In the second step, a set of features is extracted from the attack samples. The third step calls for the sample set to be grouped using a *clustering* technique. This step also gives the features that distinguish each cluster. In the final step, a generalized signature is created for each cluster, using *logistic regression* modeling. The process allows to create signatures that represent a set of similar attacks, reducing the number of rules handled by an NIDS. Additionally, by starting from samples of real attacks, we reduce the chances for the resulting signatures to flag non-malicious traffic as malicious.

To develop our system, we consider the prevalent class of SQL injection (SQLi) attacks. They have been a very relevant and popular attack vector for the last few years, as it targets databases (indirectly) available on the Internet. To consider SQLi attacks, our threat model assumes attacks against custom-developed web applications, connected to a database (commonly known as three-tier system). The profile of the attackers includes highly motivated individuals that might have an economical incentive to compromise the three-tier system. An attacker starts by having a publicly accessible description of the system and then browses the web application, looking for forms where she can provide user input and then this input can serve as parameters for an SQL statement.

4.2.1 Webcrawling for Attack Samples

The first phase is to crawl the web to collect attack samples that later are used to generate the generalized signatures. The objective is to take advantage of the multitude of public web sources available that provide attack samples. This approach looks to proactively collect samples from multiples web sources, which is the opposite of a more common strategy to use honeypots to collect attack samples.

We chose to proactively collect samples because we are targeting slow moving attacks (such as SQLi), they present a greater diversity than typically handled by honeypots, the distance between legitimate requests and malicious requests is often quite small, and above all, for a purely logistical reason — to speed up the data collection. Our approach was facilitated in practice by the wide availability of well-maintained data sources of SQLi attack samples, some of which provide APIs to enable automated sample collection. A practical point here is that what we see during the web crawling is the entire HTTP request payload and we extract the SQL query from it by leaving out the HTTP address, the port, and the path (typically a "?" indicates the start of the query string).

Table 4.1: Examples of SQLi Vulnerabilities published in July 2012.

VULNERABILITY	**CVE ID**
Joomla 1.5.x RSGallery 2.3.20 component	CVE-2012-3554
Drupal 6.x-4.2 Addressbook module	CVE-2012-2306
Moodle 2.0.x mod/feedback/complete.php 2.0.10	CVE-2012-3395
RTG 0.7.4 and RTG2 0.9.2 95/view/rtg.php	CVE-2012-3881

To collect the attack samples, we crawled different cybersecurity portals between April and June of 2012. Each portal or site is a public repository of computer security tools, exploits, and security advisories, where security professionals and hackers share examples of different attacks. Examples of cybersecurity portals include Security Focus [24], the Exploit Database [62], PacketStorm Security [63], and the Open Source Vulnerability Database [61]. This last site also provides its own search API, making it easy for security practicioners and researchers to automate the collection process of data on those sites. For sites that do not provide such capability, one can use the APIs provided by search engines, such as Google custom search API [73].

There are also open forums and mailing lists where users share attack samples. In our experiments, we collected over 30,000 SQLi attack samples from a few sources and used these as our dataset to generate the generalized signatures during the experiments.

It is important for our signature generalization approach to work effectively that the sample collection be as comprehensive as possible. As one heuristic-based check for this, we manually inspected the high and medium risk SQL injection vulnerabilities published during the month of July 2012 in the National Vulnerability Database [25] for web applications using the MySQL database — approximately 30 in number.

In each case, we found examples of SQLi attacks in our dataset that could be launched against each of the web applications reviewed. Table 4.1 includes some examples of the SQLi vulnerabilities published in July 2012.

Once the attack samples are collected, the samples need to be standardized in preparation for the data analysis. We use a set of 5 transformations, including uppercase $\rightarrow$ lowercase, URL encoding $\rightarrow$ ascii characters, and unicode $\rightarrow$ ascii characters.

We found such standardization was necessary since the data came from a variety of sites and even within each site, there is a plethora of contributors. Thankfully, the standardization process was easily automated using only 5 rules.

4.2.2 Feature Selection

We characterize each sample using a set of features, which will be used as input for the clustering algorithm. To create the set of features, we use three sources that are domain-specific for the SQLi attack scenario: (1) SQL reserved words [74], (2) SQLi signatures from the Bro [75] and Snort [30] NIDS and the ModSecurity web application firewall (WAF) [76], and (3) SQLi reference documents [77], [78]. A summary of the feature sources is presented in Table 4.2.

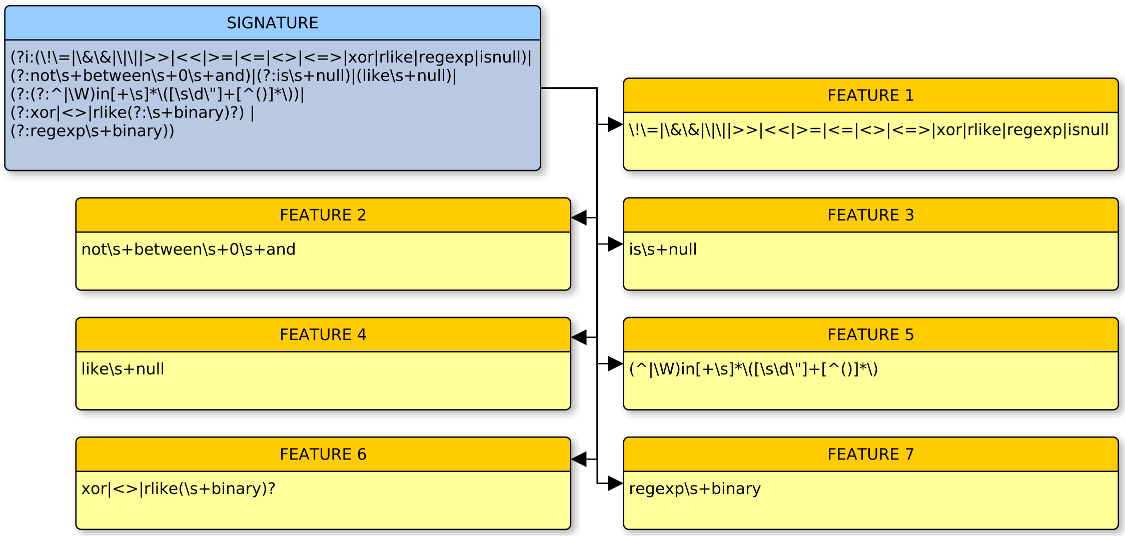
The SQL reserved words are used as features since they represent identifiers or functions, necessary to create SQL queries like in SQLi attacks. Examples of reserved words used to create the feature set for SQLi attacks include `SELECT`, `DELETE`, `CURRENT_USER`, and `VARCHAR`. In this chapter, we limited the feature set to only include the reserved words for the MySQL database management system, thus excluding special-purpose keywords used in Microsoft SQL and other non MySQL databases.

We also looked at existing signatures for features since the signatures are the result of a usually long optimization process, so it could be assume that these signatures have components (strings inside a signature) that can be used as features to help identify attacks. We did not use a whole signature as a single feature, but rather divided the signature into logical components and each component then was used as

Table 4.2
Sources of SQLi features.

FEATURE SOURCE	EXAMPLES	DESCRIPTION
MySQL Reserved Words	`create` `insert` `delete`	Words are reserved in MySQL and require special treatment for use as identifiers (table and column names) or built-in functions.
NIDS/WAF Signatures	`in\s*?\(+\s*?select \)?;` `[^a-zA-Z&]+=`	SQLi signatures from popular open-source detection systems are deconstructed into its components, such as the regular expression groups found in each signature.
SQLi Reference Documents	`\' ORDER BY [0-9]-- -` `/\*/` `\"`	Common strings found in SQLi attacks, shared by subject matter experts.

a feature. To divide each signature, we looked at all the regular expressions found inside round brackets or parentheses. The regex engine considers the expression inside the parentheses as a single group. In this case, we use the group as a single feature.



```
SIGNATURE
(?i:(\!\=|\&\&|\|\||>>|<<|>=|<=|<>|<=>|xor|rlike|regexp|isnull)|
(?:not\s+between\s+0\s+and)|(?:is\s+null)|(like\s+null)|
(?:(?:^|\W)in[+\s]*\([\s\d\"]+[^()]*\))|
(?:xor|<>|rlike(?:\s+binary)?) |
(?:regexp\s+binary))

FEATURE 1
\!\=|\&\&|\|\||>>|<<|>=|<=|<>|<=>|xor|rlike|regexp|isnull

FEATURE 2
not\s+between\s+0\s+and

FEATURE 3
is\s+null

FEATURE 4
like\s+null

FEATURE 5
(^|\W)in[+\s]*\([\s\d\"]+[^()]*\)

FEATURE 6
xor|<>|rlike(\s+binary)?

FEATURE 7
regexp\s+binary
```

Fig. 4.2. Example of the creation of SQLi features from decomposing existing rules. A ModSec signature (left blue box) was broken down into 7 features. Features 6 and 7 were not included in the final feature set as they were replaced by simpler features or are for queries to non-MySQL databases.

Another mechanism used to split a signature was to look at the alternation operator "|" found in the signature. We used this mechanism at our own discretion and in case where we found evidence from other feature sources, that dividing the signature with the alternation operator could prove beneficial.

Figure 4.2 shows an example of a signature taken from the ModSecurity Core Rule Set (CRS) and the corresponding features generated to represent the SQLi attack samples in our experiments. The original signature is a regular expression with seven case insensitive groups, for which we proceeded create the corresponding seven features. Features 6 and 7 were not included in our final feature set as they were replaced by shorter regular expressions.

The SQLi reference documents also provide logical components that are found in SQLi attack samples. Although there is no formal classification of SQLi attacks, these documents provide a layout of different types of attacks found in web applications.

The documents also helped to determine when and how to divide the signatures and which features to select if two or more overlap.

The resulting feature set consisted of a series of regular expressions representing (1) relevant characters, (2) SQL tokens, and (3) popular strings found in SQLi attacks. The group of strings allowed for our system to also consider the relative position of SQL tokens among them, when creating the features. As an example, feature `=[-0-9%]+` only considers a number if it is preceeded by the `=` character. Additionally, our chosen clustering algorithm can handle some redundant features, i.e., some features that do not help to discriminate between malicious and non-malicious SQL queries. This is because the clustering technique that we use will not output such features for any of the attack clusters and thus, they will not appear in any signature.

All features included in the set were of numeric type, each one measuring the number of times a feature was found in an attack sample. For example, in the SQLi attack sample `?artist=0+div+1+union#foo*/*bar select#foo 1,2,current_user` features `*` and `union.+select` would return values 2 and 1 respectively. The resulting feature set used in the experiments had 159 entries, after removing those features that were not found in any of the samples used in the training phase of the system. We originally had 477 features, which were pruned down to 159 through the simple rule we just mentioned. The removed features also corresponded to cases for attacks to non-MySQL databases (not considered in our experiments) or because of multiple features looking for similar SQLi strings (overlapping features).

70 (out of 159) entries in the resulting feature set performed as binary features. That is, the value for each of these features was either one (confirming the existence of the corresponding SQL token or string in a sample) or a zero (non existence) in each of the attack samples.

The process of creating the feature set might at first blush seem intensely manual. But in our experience, the process was automatable for the most part. Both the fragmentation of the existing signatures and the reserved words (rows 1 and 2 in Table 4.2) could be automated since they follow from unambiguous rules. In the case of analyzing the reference documents, this was partially automated and served more to validate features created with the other sources. Additionally, we believe that the feature space was exhausted so the creation of the feature set should be considered a one-time task, for one kind of attack (such as SQLi).

We also considered using only binary features, i.e., the binary flag whether a feature is present or absent in a sample, rather than its count. However, this did not produce good results. When using a clustering algorithm on the samples represented by the binary features, only a large number of very small clusters were produced. This effectively meant that a large fraction of training samples were not covered, an undesirable situation.

Each attack sample that provides the input to the clustering algorithm later used is characterized by its values for the 159 features. The resulting data is organized in a matrix where the samples are the rows of the matrix and the features are the columns. The size of the matrix was then 30,000 by 159 and can be classified as sparse because 85% of its cells were populated with zeroes. About 6% of its cell values were ones. Visual inspection of the matrix revealed that any one feature was zero in most samples and non-zero in the few remaining samples. Different features exhibited this property in different samples.

4.2.3 Creating Clusters for Similar Attack Samples

We use the biclustering [79] technique to analyze our matrix, which is popularly used in gene expression data analysis. The objective of this technique is to identify blocks in the sample dataset built by a subset of features to characterize a subset of samples. Given a set of m rows and n columns (i.e., an $m \times n$ matrix), the biclustering algorithm generates biclusters - a subset of rows which exhibit similar behavior across a subset of columns. To achieve this, the biclustering technique first clusters the rows (samples) of the matrix and then clusters the columns (features) of the row-clustered data. Biclustering has found wide success in analyzing gene expression data, in which

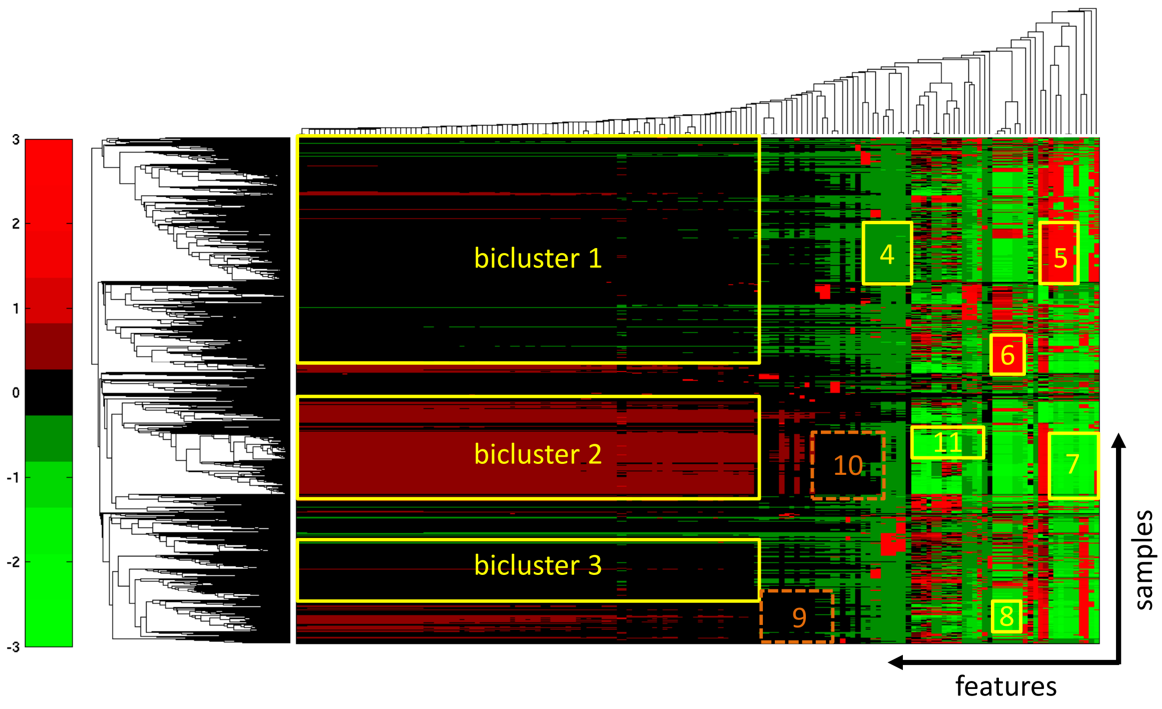


Fig. 4.3. Heat map with two dendrograms of the matrix data representing the samples dataset. The 30,000 attack samples are the rows and the 159 features are the columns. The heat map also shows the seven biclusters selected to create the signatures.

a subset of genes induces a similar linear ordering along a subset of conditions (e.g., different patients, different tissues, or varying cellular environments) [80].

To formalize the concept of bicluster, a sample set D is given as a $|N| \times |F|$ matrix where N is the set of samples and F is the set of features. The elements d_{ij} of the matrix indicate the relationship between sample i and feature j. Then, a bicluster B_{RC} is a block that includes a subset of the rows $R \subseteq N$ and a subset of columns $C \subseteq F$, sharing one or more similarity properties.

The objective is to identify subsets of attack samples which share similar values for a subset of features. Each subset of samples (cluster) may use different sets of features. We want to create a signature for each bicluster and the biclustering technique allows using different features for different biclusters. This enables us to create compact and distinctive signatures for the wide variety of SQLi attacks. The biclusters are nonoverlapping (i.e., no two biclusters have spatial overlap) and nonexclusive (i.e., two biclusters may use overlapping set of features) (Figure 4.3). The heatmap shows 11 clusters that are formed, by visual analysis of the color patterns. A contiguous region with one color pattern constitutes one cluster. Note that not all features are used in the cluster formation; thus, there are some gaps for the feature dimension when you consider all the clusters. Note also that not all samples are covered in a cluster, indicating that some attack samples are considered so different that they do not fit within any cluster. This may indicate that our training set has some noise in it. Being able to deal with some noise in a training set is an important property for any machine learning algorithm and we are heartened to see that that is the case with *pSigene*.

We use a simple approach to achieve the biclustering technique, performing a two-way hierarchical agglomerative clustering (HAC) algorithm, using the Unweighted Pair Group Method with Arithmetic Mean (UPGMA). The way biclustering worked is first it did a clustering of the samples and then within each cluster, it clustered by the features. Thus, it identified what were the discriminating features for each cluster. The UPGMA algorithm produces a hierarchical tree, usually presented as a dendrogram, from which clusters can be created. It works in a bottom-up (agglomerative) approach by first partitioning the sample set of size N into N clusters, each one containing a single sample. Then, the Euclidean pairwise distance is calculated among the initial, single sample clusters in order to merge the two closest ones. After the first round of paired clusters finishes, UPGMA is used to recursively merge the clusters. At each step, the nearest two clusters are combined into a higher-level cluster. The distance between any two clusters A and B is taken to be the average of all distances between pairs of objects ”x” in A and ”y” in B, that is, the mean distance between elements of each cluster. This biclustering process is repeated until a single cluster containing all the samples is formed. Note that this is just the termination point from biclustering; its results will guide us to pick the multiple clusters as we explain below.

Next, the results from applying the biclustering technique are presented as a heat map in Figure 4.3. On each axis, the corresponding dendrograms are also shown. The heat map shows the graphical representation of the reordering of the matrix $|N| \times |F|$ into a set of bi-clusters. Each bicluster is represented as an area of similar color as the heat map simultaneously exposes the hierarchical cluster structure of both rows and columns, as explained in [81]. Each column in the matrix is standardized as follows: the statistical mean and standard deviation of the values is computed. The mean is then subtracted from each value and the result divided by the standard deviation. As a value is closer to the mean, it is shown with the black color in the heat map. The highest and the lowest values are shown in red and green, respectively. Figure 4.3 also shows the dendrograms produced by the HAC algorithm for both rows (sample set) and columns (feature set). The dendrogram consists of many inverted `U`-shaped lines that connect different clusters in a hierarchical tree. The height of each `U` represents the distance between the two clusters being connected.

We tested different pair-wise distance metrics and linkage criteria for the HAC algorithm on the sample set. We selected the Euclidean distance metric and the UPGMA method as they helped produce clean bi-clusters, represented by boxes of (mostly or completely) a single color in the heat map 4.3. To validate the accuracy of the HAC algorithm, we also calculated the cophenetic correlation coefficient for each dendrogram. The cophenetic correlation for a cluster tree is defined as the linear correlation coefficient between the cophenetic distances obtained from the tree, and the original distances (or dissimilarities) used to construct the tree. Thus, it is a measure of how faithfully the tree represents the dissimilarities among observations. The cophenetic distance between two observations is represented in a dendrogram by the height of the link at which those two observations are first joined. That height is the distance between the two subclusters that are merged by that link. In our experiments, we found the cophenetic correlation coefficient value of 0.92, a promisingly high number. Ultimately, the above-mentioned explorations of the design space required visual inspection of multiple heatmaps rather than the alternative: use of multiple security experts and an almost *zen master*-like grasp of regular expressions.

4.2.4 Creation of Generalized Signatures

From each bicluster b_j, we create a signature Sig_{b_j} which characterizes the samples in that bicluster, plus is more generalized. Specifically, in our solution, a signature Sig_{b_j} is a logistic regression model built to predict whether an SQL query is an attack similar to the samples in cluster b_j.

Logistic regression is a very popular classification method since the output values for the hypothesis function, lay in the range between 0 and 1. These values are interpreted as the estimated probability that a sample belongs to a class. In our scenario, we use logistic regression to compute the probability that an HTTP request, as seen by the IDS, includes an SQLi attack.

Each bi-cluster b_j is defined by a set of features F_j and a set of samples S_j. We then create the corresponding signature of a bicluster by using the features as the variables in the hypothesis function and training this function with the samples from the bicluster, as well as normal traffic. Now, *pSigene* needs to come up with a signature for cluster b_j using the features F_j. For this it calculates the parameters Θ_j (which is a vector of individual parameter values), using the labeled data of attack samples from cluster b_j as well as benign network traffic data. The intuition behind the calculation of Θ_j is that it should minimize the errors in the labeled training set. We give the mathematical formulation behind this computation later in the section. An example signature created by this method (for cluster 6 in this case) is given for Θ_6^T at the end of this section.

Having calculated Θ_j, let us see how *pSigene* would work during the operational phase (the test phase). When a sample is available to *pSigene*, to determine if it belongs to attack class j, it calculates the value of the hypothesis function:

$$h_\theta(F_j) = g(\Theta^T F_j)$$

We use for g the sigmoid function which is defined as:

$$g(z) = \frac{1}{1 + e^{-z}}$$

This gives a value between 0 and 1 and is interpreted as the probability that the test sample belongs to attack class j.

To find the optimal parameters Θ of the regression model we use the Preconditioned Conjugate Gradients (PCG) method [82], with the cost function in logistic regression as:

$$J(\Theta) = \frac{1}{m}\sum_{i=1}^{m}[-y^{(i)}log(h_\Theta({F_j}^{(i)})) - (1 - y^{(i)})log(1 - h_\Theta({F_j}^{(i)}))] + \frac{\lambda}{2m}\sum_{j=1}^{n}{\Theta_j}^2$$

The intuition behind this formulation is that the first term is the cost due to mislabeling a sample in the training set - the two sub-terms are respectively for mislabeling an attack sample in the training set and a benign sample in the training set. The second summation term of the cost function is a regularization term based on the ridge-regression (L_2-norm) method, with λ as the regularization parameter, m is the number of samples in a bicluster b_j, and n is the number of parameters Θ, excluding the intercept term (Θ_0). The regularization helps to avoid over-fitting of the model, especially for a sparse matrix as in our case, where a small fraction of the features are influencing the predictions but the identities of those influential features is unknown. We tested λ for values between 0.01 and 100, and for each value ran a test to determine the true positive and false positive rates (TPR, FPR). We chose a value of 50 for all our experiments, as this value provided a nice tradeoff between fitting the samples well and keeping the parameters Θ relatively small.

The gradient of the cost is a vector Θ where the k^{th} feature is defined as follows:

Table 4.3
Features included in Signature 6.

FEATURE NUMBER	FEATURE (Regular Expression)
25	`=`
37	`=[-0-9\%]*`
53	`<=>|r?like|sounds\s+like|regex`
36	`([^a-zA-Z&]+)?&|exists`
28	`[\?&][^\s\t\x00-\x37\\|]+?`
32	`\)?;`

$$\frac{\partial J(\Theta)}{\partial \Theta_k} = \begin{cases} \frac{1}{m}\sum_{i=1}^{m}\left(h_\Theta(F_j^{(i)}) - y^{(i)}\right)F_{jk}^{(i)} & \text{for } k = 0 \\ \frac{1}{m}\left[\sum_{i=1}^{m}\left(h_\Theta(F_j^{(i)}) - y^{(i)}\right)F_{jk}^{(i)} + \lambda\Theta_k\right] & \text{for } k \geq 1 \end{cases}$$

As an example of how we used logistic regression in the SQL injection attack scenario, consider a bicluster b_6 obtained after running the biclustering algorithm. This bicluster has a set S_6 of $2,741$ samples and a set F_6 of the features listed in Table 4.3.

After training with the set S_6 of $2,741$ samples (attack class) and one day of non-malicious traffic (other class), we compute the parameters Θ_6 of the generalized signature for bicluster S_6:

$$\begin{aligned}\Theta_6^T = -3.761054 &+ 0.262131 f_{6,25} + 0.262131 f_{6,37} \\ &+ 0.261463 f_{6,53} + 0.261584 f_{6,36} \\ &- 0.117270 f_{6,28} + 0.708324 f_{6,32}\end{aligned}$$

Table 4.4 shows the probability values for three samples, when using signature 6. The attack sample produces a probability value of 0.9926 while the two benign samples have probability values of 0.0694 and 0.1928.

Table 4.4
Probability Values produced by Signature 6

SAMPLE	TYPE	PROB. VALUE
`?option=com\_simplefaq\&amp; task=answer\&amp;Itemid=9999 \&amp;catid=9999\&amp;aid=-1 +union+select+1,concat\_ws(0x3, username,password,email),3,4, 5,6,7,8,9,10,11,12,13,14,15, 16,17,18,19, 20+from+jos\_u sers`	attack	0.9926
`/mod/resource/view.php?id=21154`	benign	0.0694
`/blocks/mle/dwn/index.php? vendor=Samsung\&device=X830`	benign	0.1928

4.3 Evaluation

We evaluated *pSigene* and the signatures in three other IDSes by using SQL attack samples and benign web traffic. In the first experiment, we determined the TPR and FPR by using two network traces collected from real systems (we describe the details of the traces in Section 4.3.2.

The second experiment involved incrementing the number of attack samples given during the learning step of the signature creation phase to see if there is improvement. Finally, the third experiment determined performance impact of matching the signatures generated by *pSigene* when the signatures were integrated with the Bro IDS. We also compared to the performance to the three other three signature sets used in experiment 1.

4.3.1 SQLi Signature Sets

We analyzed four different sets of SQLi signatures, taken from popular open-source NIDS (Snort and Bro) and a web application firewall (ModSec). A summary of the different signatures used in the evaluation is presented in Table 4.5. The fact that some of the SQLi rules are disabled by default in some of the IDSes may indicate the perception that there exists overlaps between rules. The high usage of regex is because it holds the promise that a regex will be able to match a wide set of attacks. This observation motivated us to build on regex's in choosing the features in *pSigene*. A description of each signature set follows:

Bro

It is a network analysis framework that can be used as a signature-based IDS. It comes with a set of signatures to perform low-level pattern matching. We analyzed the 6 SQLi rules present on Bro v2.0 [75] to detect SQLi attacks. All six of the rules make extensive usage of regular expressions.

Table 4.5
Comparison between different SQLi rulesets.

Rules Distribution	Version	Number SQLi rules	SQLi rules Enabled	Usage of Regex
Bro	2.0	6	100%	100%
Snort Rules	2920	79	61%	82%
Emerging Threats	7098	4231	0%	99%
ModSecurity	2.2.4	34	100%	100%

Snort

Snort is an open source network IDS that performs packet-level analysis to detect attacks and comes with its own ruleset, which gets updated periodically. We downloaded and analyzed version 2920 of the ruleset [83]. It included 79 SQLi-related rules, 82% of which use regular expressions.

Emerging Threats

An open source project that publishes detection rulesets for two IDS, Snort and Suricata [84]. The ruleset is updated daily. We analyzed version 7098 of the ruleset, which includes over 4,200 SQLi-related rules. 99% of those rules use simple regular expressions. For the purposes of our experiments, we merged this signature set with Snort's signature set.

ModSec

ModSecurity (shortened as "ModSec") is a web application firewall (WAF) used to protect Apache web servers from attacks such as SQLi. The OWASP ModSecurity Core Rule Set (CRS) project is an open-source initiative to provide the signatures used by ModSecurity to detect attacks to web applications. We analyzed CRS version 2.2.4, which includes 22 SQLi-related rules. All of the rules make use of regular expressions.

A number of differences exist between the different NIDS where these signatures sets are used. First, Snort and Bro use a deterministic approach to handle the signatures. In other words, these systems produce an alert only if all the requisites defined in a signature are met. In contrast, ModSecurity takes a probabilistic approach and uses a scoring scheme where signatures are weighted and can contribute to determine the level of anomaly for a particular trace.

A second difference between the NIDS involves the regular expression engine that each one uses. Snort and ModSecurity use the Perl-compatible regular expressions (PCRE) library, while Bro's regex engine is POSIX compliant. PCRE is a more sophisticated library and has a richer syntax and set of supported features (such as, backtracking over a string).

A third difference is the composition of the SQLi signatures analyzed and used in the experiments. The signatures found in Bro and ModSecurity made extensive use of regexs, while Snort's signatures include much simpler regexs. Of the 6 signatures found in Bro, the average length of the regular expressions was 247.7 characters (max: 429, min: 27). Meanwhile, regular expressions in ModSecurity had an average length of 390.2 characters (max: 2917, min: 28) and in Snort were 27.1 (max: 40.1, min: 0).

All these differences impact the detection capabilities of these systems. For all the experiments presented in this chapter, we considered only those signatures that used or included regular expressions. We did this to allow for a fairer comparison between *pSigene* and the other IDSes, since *pSigene* uses regular expressions for its features.

4.3.2 SQLi Test Datasets

We used two test datasets to evaluate the performance of the different signature sets. The test dataset used to compute FPR corresponds to a 1-week network trace at a university institution. We captured all HTTP traffic to the main web servers at the university, including the institutional web servers, the registration and payment servers, and the web interface for the mailing servers. The network trace amounts to 4.53 GB and included over 1.4 million HTTP GET requests. Although no ground truth existed for this trace, we ran it through all three signature sets and manually reviewed the alerts generated. All alerts were false positives; therefore we concluded no malicious attack was included in the trace.

The other testing dataset was used to compute the TPR of all signature sets. We generated this testing dataset by running SQLmap [72], a popular SQL injection scanning tool, against a vulnerable web application running Apache Tomcat and MySQL database. SQLmap was launched against the application which contained 136 vulnerabilities, triggering the scanning tool to generate over 7200 attack samples. To collect this testing dataset, we set up an isolated network which only had the test traffic and thus the traces were not contaminated with other traffic.

4.3.3 Implementation in Bro

To run our experiments, we integrated the signatures generated by *pSigene* into the Bro NIDS and then instructed Bro to use only our signatures and not its own. Additionally, we coded a function `count_all` that accepted as input two parameters, a regular expression and a string, and returned the number of times the regular expression was found in the string. Bro is an extensible IDS in that it allows one to plug in user scripts, which we did. Plus, it allows one to develop core functions and integrate them with Bro after recompiling it. The user scripts can then reach back and call these core functions. We also used this feature to integrate our new function `count_all`. This function was used in *pSigene* to count the number of occurrences corresponding to the features. *pSigene* is invoked by Bro from its upper policy layer, which is analogous to where Bro's own SQLi signatures sit.

4.3.4 Experiment 1: Accuracy and Precision Comparison

We performed the evaluation separately with 7 signatures (corresponding to 7 clusters) and with 9 signatures (corresponding to 9 clusters). As shown in Figure 4.3, we considered 11 biclusters to produce two signature sets, one with seven signatures and another with nine. The set of seven signatures showed a group of rules that obtained a higher TPR than Bro and Snort, while also producing a very low FPR. The results from the set of nine signatures allowed determining how much the TPR can be improved while also measuring the increase in the FPR. From the heatmap, we visually identified 11 biclusters as this permitted to include a large percentage of all the samples in the original training set, while giving reasonably homogeneous colored areas. From the experiments, we determined that biclusters 9 and 10, as labeled in Figure 4.3, gave poor signatures and so we dropped them from the evaluation. Their bad detection rate can be traced to two main causes: (1) almost all the samples included in each of bi-clusters 9 and 10 have the selected features with values of zero (a small percentage had values of 1); and (2) each bi-cluster has a small number of features. Note the black color in the heatmap in Figure 4.3 tells us that the normalized value of the selected features is zero; in addition, the mean value is also close to zero. These two reasons make these signatures incapable of discriminating between normal and malicious traffic.

The result is shown in Table 4.6. Our signatures had higher detection rate (86.53% for 9 signatures and 82.72% for 7 signatures) than Snort (79.55%) and Bro (73.23%) , but lower than ModSecurity (96.07%). Both our signature sets had the lowest FPRs, only behind Bros signature set (which did not raise a single false positive). Although the other FPRs were very low, one should not be deceived by these numbers. A FPR of 0.174%, as recorded for Snort, represents over $2,463$ false alarms generated over the one week traffic, while ModSecs TPR represents over 730 false alarms. In comparison, our sets produced 523 false alarms in the case of nine signatures and 226 in the case of seven signatures.

Table 4.6
Accuracy Comparison between different SQLi rulesets.

RULES	TPR (%)	FPR (%)
Bro	73.23	0.0000
Snort - Emerging Threats	79.55	0.1742
ModSecurity	96.07	0.0515
Generalized Signatures (9)	86.53	0.037
Generalized Signatures (7)	82.72	0.016

ModSecurity achieved the highest TPR of all signatures sets at 96%. We had suspected this to be a difficult result to improve. The ModSec set has been developed and tested over several years by a team of security developers, and is part of a popular open-source WAF. This situation has fostered the development of a robust signature set and our results confirmed it. Further, the focused nature of these rulesets for web application detection also explains its good detection rate.

Accuracy and Precision of Individual Signatures

We wanted to drill deeper into the overall accuracy and precision result of *pSigene* to see what the contribution from each of the signatures is. For this, we plotted the ROC curves for each of the 9 signatures for the entire test data. The result is shown in Figure 4.4. To generate the ROC curve for a given signature, we ran *pSigene* with only that signature enabled and we varied the probability threshold for the output of logistic regression. In the ROC curve, the point (0, 1) corresponds to the ideal case

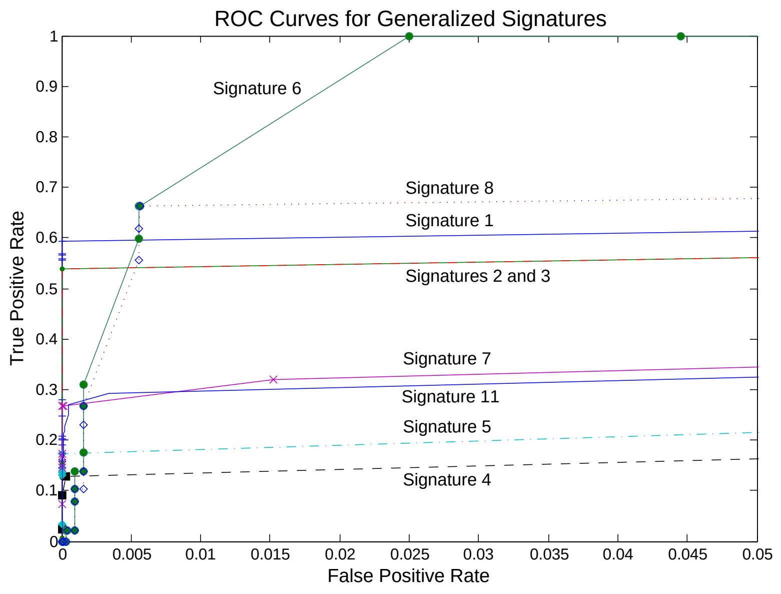


Fig. 4.4. ROC curves for each of the signatures generated for the generalized set. The plot shows different performance for each signature, suggesting that each one can be tuned separately which can improve the overall detection rate of the set.

and the greater is the area under the curve, the better the performance is. Note that in this plot, the FPR only goes till 0.05, not till 1. This is because the maximum value of FPR for the systems under test does not grow beyond 0.05.

The first observation is that there is wide variability in the quality of the signatures. Signature 6 performs well while signature 4 lags. Second, some signatures are quite insensitive to the threshold settings — signatures 1, 2, 3, and 8 — since their detection rates go up only slightly. Third, signature 6 will produce false positives faster than signatures 1 and 8. From a ROC curve like this and with an idea of a desired TPR and FPR, a security administrator can visually, and approximately, decide which signatures to enable or disable. We believe this is a useful and practical visualization for signatures of misuse-based IDSes in general, not just for *pSigene*.

Coverage of Individual Signatures

Another aspect of the clusters and the corresponding signatures is how many samples does each cover and how many features are used in each cluster's signature. This result is shown in Table 4.7. There is quite a large range of cluster sizes, as is of the number of features output by biclustering. The largest cluster has 44% of the samples while the smallest has 5.5%. Three clusters use 57% of the total number of features (90 out of 159). However, an interesting, and *not* a priori obvious, observation is that logistic regression does significant amount of pruning of features for these three clusters. Thus, logistic regression downplays the role of some features in classifying a sample as being malicious or benign. For example, for cluster 3, logistic regression throws out 88% of the features, for cluster 2 86% of the features, and for cluster 1 63% of the features. We hypothesize that this large amount of filtering by logistic regression is due to two causes. First, the reduction of the feature set from 477 to 159 is a manual process and there still remain overlaps between some of them. Second, biclustering provides a clustering but it may be clustering noisy data, while logistic regression pays attention to the quality of the data itself.

Nevertheless, biclustering is a crucial step and needs to precede logistic regression. Biclustering creates some order out of the chaos of the large amount of samples and large set of features, by identifying the samples which are similar and by identifying a superset of features according to which they are similar.

4.3.5 Experiment 2: Incremental Learning

In this experiment, we first incremented the number of attack samples while learning the Θ parameters in logistic regression to create the signatures. We progressively added some attack samples from the test dataset into the training dataset - we experimented with 20% and 40% of the test dataset being included in the training. This reflects the real world scenario where fresh attack samples will be fed to *pSigene* and *pSigene* will do incremental training with these new samples. Thus, over time,

Table 4.7
Details of signatures for each cluster created by *pSigene*.

BICLUSTER NUMBER	NUMBER OF SAMPLES	NUMBER OF FEATURES FROM BICLUSTERING	NUMBER OF FEATURES IN SIGNATURE
Bicluster 1	13272	90	33
Bicluster 2	5477	90	13
Bicluster 3	2629	90	11
Bicluster 4	6947	12	8
Bicluster 5	4245	8	5
Bicluster 6	2741	6	6
Bicluster 7	3928	10	5
Bicluster 8	1676	8	6
Bicluster 11	1671	15	14

pSigene will be able to detect more and more of the attacks as it operates for longer periods and gets incrementally trained. Note that the incremental training is also an automatic process and therefore, we are spared the tedium of manually updating prior signatures. How exactly incremental training is done is described in Section

When adding 20% of the SQLmap dataset, we obtained a TPR = 89.13% and a FPR = 0.039%. After augmenting the training dataset with 40% of the samples from the SQLmap set, the TPR increased to 91.15% while the FPR also increased to 0.044%. In both cases (20% and 40%), we used sets of 10 signatures.

From the results, the TPR showed an increment of a bit over 2%, for each round of the experiment. This can be explained as we first randomized the SQLmap set and then divided it into 20% parts. So one can hypothesize that *pSigene* is seeing some similar attack samples in the test phase.

The FPR also increased slightly from the 20% to 40% experiment. Such behavior highlights a limitation of our approach. By only adding more samples for the malicious class in the training phase, we should be improving the TPR and this is reflected in our results. But this does not necessarily reduce the FPR because we are not adding more samples from the non-malicious class and so our ability to model normal traffic is not improving. In fact, in the extreme, if the training data has a great imbalance of a large amount of malicious labeled samples and a small amount of benign samples, the FPR is expected to go up.

4.3.6 Experiment 3: Performance Evaluation

In this section, we report the overhead of *pSigene* signatures against Bro and ModSec signatures that are implemented as `*.bro` scripts in the Bro NIDS. Specifically, we measured the average processing-time per HTTP request for each signature in SQLmap dataset. The minimum, average, and maximum processing times across the signatures of the three systems are presented in Figure 4.5. We observe that on average, *pSigene* gives a slowdown of $17X$ and $11X$ against Modsec and Bro signatures respectively. The increased processing-time in *pSigene* is majorly attributed to the `count_all` function call, which counts the number of regex matches for each HTTP request string. We observe from the data that the signatures with a large number of invocations of `count_all` take a disproportionately large fraction of the total processing time. Given that we run these measurements on a relatively resource-starved machine (700 MHz (CPU), 512 MB (RAM)) and still the worst case processing time was less than 2 ms, we would expect that signature matching in *pSigene* will not become a bottleneck. Importantly, the signature matching is completely parallelizable - each parallel thread can match one signature and this functionality is inbuilt in Bro (Bro's cluster mode). But we do not have this obvious performance optimization implemented yet.

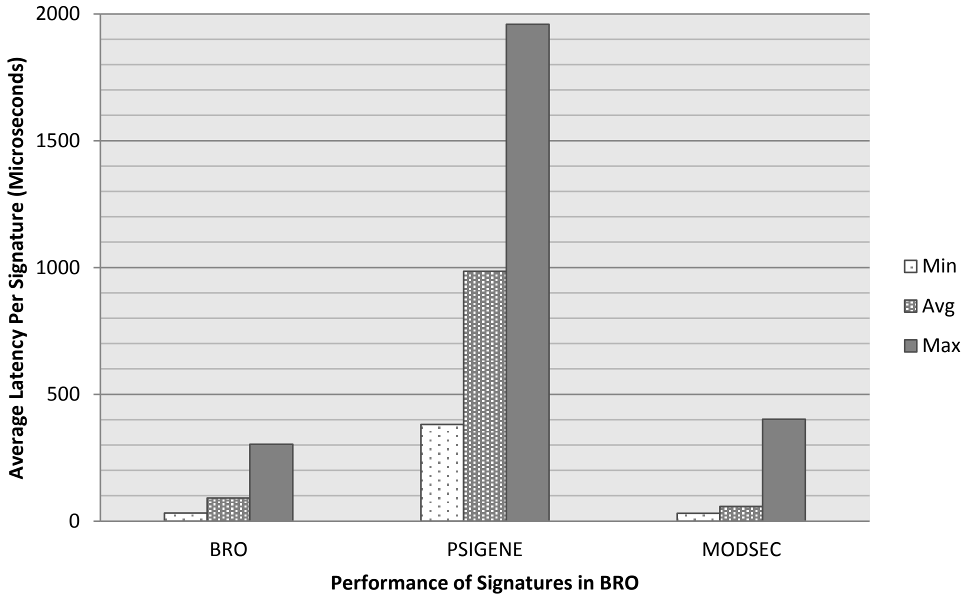


Fig. 4.5. Minimum, Average, and maximum processing time across all signatures from Bro, *pSigene*, and ModSec sets.

4.4 Discussion

An important issue to consider in *pSigene*, is how to adapt it after it is deployed, i.e., improve its signatures as new attack samples are collected. We presented results from an experiment on incremental learning and its benefits. In this section, we define an approach so *pSigene* can systematically update its signature set.

Given new attack samples, we can sequentially (i.e., incrementally) update our existing bi-clusters and also sequentially update our logistic regression models, instead of retraining both of them using the entire data set. These sequential updates enable us to deal with massive amounts of dynamic attack data, and update the whole detection system in a computationally efficient way. A typical usage scenario that we envisage is that new attack samples will be fed into *pSigene* daily and we would retrain the system to come up with new signatures. Thus, the signatures can evolve and the resultant detection performance can improve. We emulate such a usage scenario in our experiments in Section 4.3.5.

Specifically, given new attack samples, we will examine how similar a new sample is, to the existing bi-clusters. We can use the same Euclidean distance metric that we use in the initial training. If it is similar to certain (one or several) existing bi-clusters up to a threshold, we add this sample into these existing bi-cluster and adjust the parameters of these bi-clusters – such as the mean parameters – accordingly. If the new sample is not similar to any existing bi-cluster, we will list it as a candidate for a new signature. Given multiple such candidates we will run our bi-clustering algorithm to generate new signatures. With increasingly more new samples, we can repeat this process to incrementally adjust bi-clusters and obtain new signatures.

Given new samples, we can use Bayesian online learning to sequentially update the classifier. First, if new features are added because of emerging biclusters based on new samples, we can first expand the classifier accordingly with zero weights for new signatures before applying online learning updates. Then, we will update the classifier based on the new samples. In a Bayesian framework, this goal can be naturally achieved via the update of the posterior distribution of the classifier. To conduct the needed computation, we can either resort to sequential Monte Carlo methods [85], such as particle filtering, or deterministic approximation such as virtual vector machine [86].

Combining the above strategies, we can efficiently update the signatures and the classifiers as *pSigene* runs in an operational deployment over time.

Another aspect that this work throws light on is the importance of good training data for creating the clusters and subsequently the signatures. It is imperative that the training data be representative of the kinds of attacks that will be seen in operation, though they do not need to be identical. How far apart can the attacks in training and test be? This is a perennial question that is asked of machine learning algorithms in all different contexts. This is also a question that does not have a completely satisfactory answer. The partial answer is that the features should be chosen to be rich enough that they are likely to capture important characteristics of the zero-day attacks. Thus, signatures based on such features will likely be able to match some of the zero-day attacks. The feature selection process needs to be repeated for each kind of attack, but not for each attack sample. This makes this process more feasible in practice. In contrast, manual signature update is a process that needs to be done for each attack sample and is therefore not as scalable. Of course, in practice, a signature update is done in a batch mode after a certain number of attack samples have been collected.

4.5 Related Work

The work presented in this chapter is related to three areas of intrusion detection: automatic signature creation, signature generalization, and the interaction between web applications and databases. We discuss how previous work in these three areas relates to our research.

An interesting and important work on automatic signature creation is by Yegneswaran, et al [64]. The authors presented a framework based on machine learning algorithms to produce attack signatures. It aims to create generalized signatures that represent a set of attack vectors, as the framework clusters similar attacks detected. A key point by the authors and which we agree with is that the framework requires protocol knowledge in order to produce effective signatures and such insight impacts the resulting detection mechanism. Knowing the syntax, semantics and behavior of a protocol allows to produce accurate signatures. When this information is not considered, there is higher probability to false alarms and false negatives. Distinct from our work, they take a passive approach as HTTP and NetBIOS-based malware traffic is collected from honeynets, whereas we proactively collect from the web samples for SQL-based attacks. Additionally, our framework is agnostic to transport- and network-level information, which is important for their framework. Finally, we rely heavily on regular expressions, looking to produce rich, optimized regex signatures. Their approach to regexs is somehow limited as it only uses simple metacharacters such as *, +, and ? to express clusters of signatures.

Previous work on signature generalization also includes [87], [88], [65], and [89]. In [87], several conditionals and parameters in SNORT rules are modified, using a similar approach to classic rule learning operators such as generalization and specialization. They analyze each signature separately to generalize it, while our approach uses clusters of attack samples to then create generalized signatures. In [88], a system called Polygraph generates signatures that consist of multiple disjoint substrings. In doing so, Polygraph leverages our insight that for a real-world attack to function properly, multiple invariant substrings must often be present in the payload. In other words, every attack sample includes invariant components that help to identify it. In our approach, we look to cluster samples so it helps identify the invariants that can appropriately represent the attacks. Similarly, [65] applies pattern-matching techniques and protocol conformance checks on multiple levels in the protocol hierarchy to network traffic captured at a honeypot system, to produce worm signatures. [89] extends this idea to detect zero-day polymorphic worms on high-speed networks. In both cases, the goal is to detect worms at the network layer while our general approach considers protocol information and suited for other types of attacks.

Robertson et al. [90] present an anomaly generalization technique to automatically translate suspicious requests to a web server into anomaly signatures. This approach is complementary to ours and uses heuristics-based techniques to infer web-based attacks. For the class of SQL injection attacks, the technique performs a simple scan for common SQL language keywords and syntactic elements. This results in basic signatures to detect SQLi attacks, but no details were provided on the performance of these signatures. Other papers that present similar anomaly-based intrusion detection techniques for SQLi attacks include [91] and [92].

The interaction between web applications and databases to improve the detection rate of attacks against these resources has been covered in [93], [94], [95], and [96]. [93] et al. present a novel approach for automatically detecting potential server-side vulnerabilities of this kind in legacy web applications through blackbox analysis. [94] proposes a serially composed system with a web-based anomaly detection system, a reverse HTTP proxy, and a database anomaly detection system to increase the detection rate of web-based attacks. In [95], a system that automates repair from intrusions in web applications is presented. It works by continuously recording activity in the application and constructs a global dependency graph from this logged information to retroactively patch vulnerabilities by rolling back parts of the system to an earlier checkpoint. Finally, [96] looks at the problem of scarce training data for anomaly-based intrusion detection systems. By applying clustering techniques to determine similar clusters between sets of HTTP requests made to different components of web applications, the undertrained profiles of the applications can be enhanced with similar well-trained profiles.

4.6 Conclusions and Future Work

In this work, we presented a system called *pSigene*, for the automatic generation and update of intrusion signatures. The system benefits from mining the vast amount of public data available on attacks. We tested our architecture for the prevalent class of SQLi attacks and found our signatures to perform very well, compared to existing signature sets, which have been created manually and with a tremendous amount of security expertise and progressive refinement over the period of multiple years.

Our framework allows one to generalize existing signatures and the detection of new variations of attacks (i.e., some kinds of zero-day attacks) is achieved by using regular expressions for the generalized signatures. We also rigorously benchmarked our solution with a large set of attack samples and compare our performance to popular misuse-based IDS-es. The evaluation also brings out the impact of a practical use case whereby periodically new attack samples are fed into our algorithm and consequently the signatures can be progressively, and automatically, updated. In contrast, to improve the other signature sets requires the manual inspection and testing of the signatures, which could overwhelm a system administrator with limited resources.

Future work will include the implementation of the incremental update operation, as described earlier. This task has some open design choices in terms of the machine learning technique to use and empirical evidence is needed to guide our choice. We will also improve the online performance of the signature matching process. This will be done first by simply parallelizing the process and next by optimizing the code path within Bro through which our signature matching occurs.

5. FUTURE WORK

5.1 Implementation of DIADS

The objective is to perform an evaluation of different types of detection sensors to determine the parameters needed to create the conditional probability tables (CPT) for the Bayesian network model. Using the popular Bro [43] and Snort [23] intrusion detection sensors, we evaluate them to determine their performance when detecting attacks against different components (web, application, and database) of a distributed system. To evaluate the sensors, several types of performance measurements are considered [97], [98].

Additionally, the DIADS [99] framework should be implemented and tested to determine its operational performance in a real environment. For this, the Bro sensor can be used as the baseline detection engine to develop the rest of the DIADS framework, specially the reasoning engine. A specific type of attack (or two) can be used to evaluate the framework. Candidates include those attacks associated to high False Positive rates (FPR), when detected by a single intrusion detection sensor. Examples include SQL injection attacks [?]. The objective would then be to determine the improvement (in terms of the FPR) provided by DIADS, compared to a single detector.

5.2 Determining Confidence Levels from Intrusion Alerts to Configure Detection Sensors

The DIADS framework currently considers the alerts as absolute values to determine the configuration setup of a set of detectors. The Bayesian model defines the performance of each detector, by its corresponding conditional probability table

(CPT). Still, this definition does not allow the DIADS framework to evaluate the on-going performance of a detector. Whenever a detector sends an alert to the reasoning engine, currently we dont determine how accurate or correct the alert is.

A method to fix the lack of evaluation on the quality of an alert is to compute confidence intervals for each detector, based on the success experienced of past alerts. Such intervals indicate the reliability of the alert send by a detector. For the DIADS framework, confidence intervals will act as a triggering mechanism for algorithm 4, which reconfigures DIADS given the newly received evidence (alerts). The original IDES model [100] and later work [101] have stated the need to consider the confidence interval for an alert when determining how should an intrusion detection system react. Those works do not provide a mechanism on how could the confidence intervals be used.

5.3 Incremental Deployment of Intrusion Detectors in a Dynamic Distributed System

The current DIADS framework does not consider the current setup of detection sensors to determine the new configuration of the sensors. When new alerts are received, DIADS considers all possible choices based on a finite radius set around the node in the Bayesian model. The node corresponds to the vulnerability associated to the alert received. For example, DIADS would not consider the current setup of the detection sensors when determining how to reconfigure the detection system, in light of new evidence.

In this work, the DIADS framework would be modified to perform incremental configuration of the intrusion detection sensors. This should improve the reconfiguration time taken by the framework and potentially help to detect multi-stage attacks faster than under the current framework.

LIST OF REFERENCES

LIST OF REFERENCES

[1] B. Foo, Y.-S. Wu, Y.-C. Mao, S. Bagchi, and E. H. Spafford, "Adepts: Adaptive intrusion response using attack graphs in an e-commerce environment," in *2005 International Conference on Dependable Systems and Networks (DSN 2005), 28 June - 1 July 2005, Yokohama, Japan, Proceedings*, pp. 508–517, IEEE Computer Society, 2005.

[2] K. Ingols, R. Lippmann, and K. Piwowarski, "Practical attack graph generation for network defense," in *ACSAC '06: Computer Security Applications Conference, 22nd Annual*, pp. 121–130, December 2006.

[3] X. Ou, W. F. Boyer, and M. A. McQueen, "A scalable approach to attack graph generation," in *CCS '06: Proceedings of the 13th ACM conference on Computer and communications security*, (New York, NY, USA), pp. 336–345, ACM, 2006.

[4] S. Jha, O. Sheyner, and J. M. Wing, "Two formal analys s of attack graphs," in *15th IEEE Computer Security Foundations Workshop (CSFW-15 2002), 24-26 June 2002, Cape Breton, Nova Scotia, Canada*, pp. 49–63, IEEE Computer Society, 2002.

[5] V. Mehta, C. Bartzis, H. Zhu, E. M. Clarke, and J. M. Wing, "Ranking attack graphs," in *Recent Advances in Intrusion Detection, 9th International Symposium, RAID 2006, Hamburg, Germany, September 20-22, 2006, Proceedings*, vol. 4219 of *Lecture Notes in Computer Science*, pp. 127–144, Springer, 2006.

[6] C. Krugel, D. Mutz, W. K. Robertson, and F. Valeur, "Bayesian event classification for intrusion detection," in *19th Annual Computer Security Applications Conference (ACSAC 2003), 8-12 December 2003, Las Vegas, NV, USA*, pp. 14–23, IEEE Computer Society, 2003.

[7] N. B. Amor, S. Benferhat, and Z. Elouedi, "Naive bayes vs decision trees in intrusion detection systems," in *Proceedings of the 2004 ACM Symposium on Applied Computing (SAC), Nicosia, Cyprus, March 14-17, 2004*, pp. 420–424, ACM, 2004.

[8] A. Valdes and K. Skinner, "Adaptive, model-based monitoring for cyber attack detection," in *Recent Advances in Intrusion Detection, Third International Workshop, RAID 2000, Toulouse, France, October 2-4, 2000, Proceedings*, vol. 1907 of *Lecture Notes in Computer Science*, pp. 80–92, Springer, 2000.

[9] D. Jones, C. Davis, M. Turnquist, and L. Nozick, "Physical security and vulnerability modeling for infrastructure facilities," in *Sandia National Laboratories*, 2005.

[10] F. Anjum, D. Subhadrabandhu, S. Sarkar, and R. Shetty, "On optimal placement of intrusion detection modules in sensor networks," *Broadband Networks, International Conference on*, vol. 0, pp. 690–699, 2004.

[11] T. Berger-Wolf, W. Hart, and J. Saia, "Discrete sensor placement problems in distribution networks," *Mathematical and Computer Modelling*, vol. 42, no. 13, pp. 1385 – 1396, 2005.

[12] S. Ray, D. Starobinski, A. Trachtenberg, and R. Ungrangsi, "Robust location detection with sensor networks," *IEEE Journal on Selected Areas in Communications*, vol. 22, no. 6, pp. 1016–1025, 2004.

[13] A. Krause, C. Guestrin, A. Gupta, and J. M. Kleinberg, "Near-optimal sensor placements: maximizing information while minimizing communication cost," in *Proceedings of the Fifth International Conference on Information Processing in Sensor Networks, IPSN 2006, Nashville, Tennessee, USA, April 19-21, 2006*, pp. 2–10, ACM, 2006.

[14] P. Ning, Y. Cui, and D. S. Reeves, "Constructing attack scenarios through correlation of intrusion alerts," in *Proceedings of the 9th ACM Conference on Computer and Communications Security*, pp. 245–254, ACM, 2002.

[15] C. Peikari and A. Chuvakin, *Security Warrior.* O'Reilly Media, 2004.

[16] S. Axelsson, "The base-rate fallacy and the difficulty of intrusion detection," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 3, pp. 186–205, 2000.

[17] M. D. (Ed.), "Design of an intrusion-tolerant intrusion detection system," tech. rep., Maftia Project, 2002.

[18] A. A. Cárdenas, J. S. Baras, and K. Seamon, "A framework for the evaluation of intrusion detection systems," in *2006 IEEE Symposium on Security and Privacy (S&P 2006), 21-24 May 2006, Berkeley, California, USA*, pp. 63–77, IEEE Computer Society, 2006.

[19] G. Gu, P. Fogla, D. Dagon, W. Lee, and B. Skorić, "Measuring intrusion detection capability: an information-theoretic approach," in *ASIACCS '06: Proceedings of the 2006 ACM Symposium on Information, computer and communications security*, (New York, NY, USA), pp. 90–101, ACM, 2006.

[20] F. V. Jensen and T. D. Nielsen, *Bayesian networks and decision graphs.* Springer, second ed., 2007.

[21] R. Lippmann, D. Fried, I. Graf, J. Haines, K. Kendall, D. McClung, D. Weber, S. Webster, D. Wyschogrod, R. Cunningham, and M. Zissman, "Evaluating intrusion detection systems: the 1998 darpa off-line intrusion detection evaluation," in *DARPA Information Survivability Conference and Exposition, 2000. DISCEX '00. Proceedings*, vol. 2, pp. 12 –26 vol.2, 2000.

[22] "Iptables firewall." `http://www.netfilter.org/projects/iptables/`, 2008.

[23] M. Roesch, "Snort: Lightweight intrusion detection for networks," in *Proceedings of the 13th Conference on Systems Administration (LISA-99)*, pp. 229–238, USENIX, 1999.

[24] "Bugtraq vulnerability database." `http://www.securityfocus.com/vulnerabilities`, 2008.

[25] NIST, "National vulnerability database." http://nvd.nist.gov/nvd.cfm, 2008.

[26] D. R. Kuhn, T. Walsh, and S. Fires, "Nist special publication 800-58 security considerations for voice over ip systems," 2005.

[27] K. Murphy, "Bayes net toolbox for matlab." http://www.cs.ubc.ca/~murphyk/Software, March 2006.

[28] G. Modelo-Howard, "Addendum to determining placement of intrusion detectors for a distributed application through bayesian network modeling." http://cobweb.ecn.purdue.edu/dcsl/publications/detectors-location_addendum.pdf.

[29] J. F. Lemmer and D. E. Gossink, "Recursive noisy or - a rule for estimating complex probabilistic interactions," *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, vol. 34, no. 6, pp. 2252–2261, 2004.

[30] "Snort ids." http://www.snort.org, 2008.

[31] S. L. Lauritzen and D. J. Spiegelhalter, "Local computations with probabilities on graphical structures and their application to expert systems," *Journal of the Royal Statistical Society. Series B (Methodological)*, vol. 50, no. 2, pp. 157–224, 1988.

[32] R. A. Baeza-Yates and B. A. Ribeiro-Neto, *Modern Information Retrieval.* ACM Press / Addison-Wesley, 1999.

[33] O. H. Ibarra and C. E. Kim, "Fast approximation algorithms for the knapsack and sum of subset problems," *J. ACM*, vol. 22, no. 4, pp. 463–468, 1975.

[34] H. Kellerer, U. Pferschy, and D. Pisinger, *Knapsack Problems.* Springer, 1 ed., October 2004.

[35] V. V. Vazirani, *Approximation Algorithms.* Springer, March 2004.

[36] G. Modelo-Howard, S. Bagchi, and G. Lebanon, "Determining placement of intrusion detectors for a distributed application through bayesian network modeling," in *RAID '08: Proceedings of the 11th international symposium on Recent Advances in Intrusion Detection*, (Berlin, Heidelberg), pp. 271–290, Springer-Verlag, 2008.

[37] B. Acohido, "Hackers breach heartland payment credit card system," *USA Today*, January 2009.

[38] "Health information privacy: Breaches affecting 500 or more individuals." http://www.hhs.gov/ocr/privacy/hipaa/administrative/breachnotificationrule/postedbreaches.html.

[39] S. R. Snapp, J. Brentano, G. V. Dias, T. L. Goan, L. T. Heberlein, C. lin Ho, K. N. Levitt, B. Mukherjee, S. E. Smaha, T. Grance, D. M. Teal, and D. Mansur, "Dids (distributed intrusion detection system) - motivation, architecture, and an early prototype," in *In Proceedings of the 14th National Computer Security Conference*, pp. 167–176, 1991.

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

[41] G. Vigna and R. A. Kemmerer, "Netstat: A network-based intrusion detection system," *Journal of Computer Security*, vol. 7, pp. 37–71, 1999.

[42] E. H. Spafford and D. Zamboni, "Intrusion detection using autonomous agents," *Computer Networks*, vol. 34, no. 4, pp. 547–570, 2000.

[43] V. Paxson, "Bro: a system for detecting network intruders in real-time," *Computer Networks*, vol. 31, no. 23-24, pp. 2435–2463, 1999.

[44] C. Kreibich and R. Sommer, "Policy-controlled event management for distributed intrusion detection," in *In Proceedings of the 4th International Workshop on Distributed EventBased Systems (DEBS05*, 2005.

[45] S. Noel, E. Robertson, and S. Jajodia, "Correlating intrusion events and building attack scenarios through attack graph distances," in *20th Annual Computer Security Applications Conference (ACSAC 2004), 6-10 December 2004, Tucson, AZ, USA*, pp. 350–359, IEEE Computer Society, 2004.

[46] E. Nowicka and M. Zawada, "Modeling temporal properties of multi-event attack signatures in interval temporal logic," 2006.

[47] J. Wing, *Information Assurance: Dependability and Security in Networked Systems*, ch. Scenario graphs applied to network security. Morgan Kaufmann, 2007.

[48] M. Frigault, L. Wang, A. Singhal, and S. Jajodia, "Measuring network security using dynamic bayesian network," in *QoP '08: Proceedings of the 4th ACM workshop on Quality of protection*, (New York, NY, USA), pp. 23–30, ACM, 2008.

[49] P. Ning and D. Xu, "Learning attack strategies from intrusion alerts," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, pp. 200–209, ACM, 2003.

[50] P. Ning, D. Xu, C. G. Healey, and R. S. Amant, "Building attack scenarios through integration of complementary alert correlation method," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2004, San Diego, California, USA*, The Internet Society, 2004.

[51] A. Valdes and K. Skinner, "Probabilistic alert correlation," in *Recent Advances in Intrusion Detection, 4th International Symposium, RAID 2001 Davis, CA, USA, October 10-12, 2001, Proceedings*, vol. 2212 of *Lecture Notes in Computer Science*, pp. 54–68, Springer, 2001.

[52] X. Qin and W. Lee, "Statistical causality analysis of infosec alert data," in *In Proceedings of The 6th International Symposium on Recent Advances in Intrusion Detection (RAID 2003*, pp. 73–93, 2003.

[53] J. Pearl, *Probabilistic Reasoning in Intelligent Systems: Networks of Plausible Inference.* Morgan Kaufmann Publishers Inc., 1988.

[54] F. of Incident Response and S. Teams, "Common vulnerability scoring system (cvss)." `http://www.first.org/cvss`.

[55] G. L. Gaspar Modelo-Howard, Saurabh Bagchi, "Approximation algorithms for determining placement of intrusion detectors," tech. rep., Purdue University, 2 2011.

[56] OpenVAS, "The open vulnerability assessment system." `http://www.openvas.org`.

[57] J. Swets, "The relative operating characteristic in psychology," *Science*, vol. 182, pp. 990–1000, December 1973.

[58] G. Modelo-Howard, "Addendum: Secure configuration of intrusion detection sensors." `http://sites.google.com/site/securecomm11msa/`.

[59] C. Kruegel, *Intrusion Detection and Correlation: Challenges and Solutions.* Santa Clara, CA, USA: Springer-Verlag TELOS, 2004.

[60] R. Di Pietro and L. V. Mancini, *Intrusion Detection Systems.* Springer Publishing Company, Incorporated, 1 ed., 2008.

[61] "Open source vulnerability database." `http://www.osvdb.org`, 2012.

[62] O. Security, "Exploit database," 2012.

[63] "Packetstorm security portal." `http://packetstormsecurity.org`, 2012.

[64] V. Yegneswaran, J. T. Giffin, P. Barford, and S. Jha, "An architecture for generating semantics-aware signatures," in *Proceedings of the 14th conference on USENIX Security Symposium - Volume 14*, (Berkeley, CA, USA), pp. 97–112, USENIX Association, 2005.

[65] C. Kreibich and J. Crowcroft, "Honeycomb: creating intrusion detection signatures using honeypots," *SIGCOMM Comput. Commun. Rev.*, vol. 34, pp. 51–56, jan 2004.

[66] *Automatically Preparing Safe SQL Queries*, Jan 2010.

[67] R. Kaushik and R. Ramamurthy, "Efficient auditing for complex sql queries," in *Proceedings of the 2011 international conference on Management of data*, SIGMOD '11, (New York, NY, USA), pp. 697–708, ACM, 2011.

[68] D. Goodin, "New sony hack exposes more consumer passwords," June 3, 2011.

[69] "Royal navy website attacked by romanian hacker." `http://www.bbc.co.uk/news/technology-11711478`, November 8, 2010.

[70] A. Moscaritolo, "Oracle's mysql.com hacked via sql injection," March 28, 2011.

[71] "Adobe hacker says he used sql injection to grab database of 150,000 user accounts," November 2012.

[72] B. Damele and M. Stampar, "Sqlmap," July 2012.

[73] Google, "Google custom search api." https://developers.google.com/custom-search, 2012.

[74] "Mysql 5.5 reference manual, rev. 31755," August 2012.

[75] V. Paxson, R. Sommer, S. Hall, C. Kreibich, J. Barlow, G. Clark, G. Maier, J. Siwek, A. Slagell, D. Thayer, and M. Vallentin, "The bro network security monitor." http://www.bro-ids.org, 2012.

[76] "Modsecurity core rule set," January 2012.

[77] R. Salgado, "Websec sql injection pocket reference," 2011.

[78] J. Clarke, *SQL Injection Attacks and Defense.* Syngress Publishing, 1st ed., 2009.

[79] J. M. Kraus, G. Palm, F. Schwenker, and H. A. Kestler, *Semi-Supervised Clustering in Functional Genomics*, pp. 243–271. Wiley-VCH Verlag GmbH & Co. KGaA, 2009.

[80] A. Prelić, S. Bleuler, P. Zimmermann, A. Wille, P. Bühlmann, W. Gruissem, L. Hennig, L. Thiele, and E. Zitzler, "A systematic comparison and evaluation of biclustering methods for gene expression data," *Bioinformatics*, vol. 22, no. 9, pp. 1122–1129, 2006.

[81] M. B. Eisen, P. T. Spellman, P. O. Brown, and D. Botstein, "Cluster analysis and display of genome-wide expression patterns," *Proc Natl Acad Sci U S A*, vol. 95, pp. 14863–14868, Dec 1998.

[82] S. Eisenstat, "Efficient implementation of a class of preconditioned conjugate gradient methods," *SIAM Journal on Scientific and Statistical Computing*, vol. 2, no. 1, pp. 1–4, 1981.

[83] "Snort rules." http://www.snort.org/snort-rules, 2012.

[84] "Suricata intrusion detection and prevention engine." http://www.openinfosecfoundation.org, 2012.

[85] J. Liu, *Monte Carlo strategies in scientific computing.* Springer Verlag, 2008.

[86] T. P. Minka, R. Xiang, and Y. A. Qi, "Virtual vector machine for bayesian online classification," in *Proceedings of the Twenty-Fifth Conference on Uncertainty in Artificial Intelligence*, UAI '09, (Arlington, Virginia, United States), pp. 411–418, AUAI Press, 2009.

[87] U. Aickelin, J. Twycross, and T. Hesketh-Roberts, "Rule generalisation in intrusion detection systems using snort," *CoRR*, vol. abs/0803.2973, 2008.

[88] J. Newsome, B. Karp, and D. Song, "Polygraph: automatically generating signatures for polymorphic worms," in *Security and Privacy, 2005 IEEE Symposium on*, pp. 226 – 241, may 2005.

[89] Z. Li, M. Sanghi, Y. Chen, M.-Y. Kao, and B. Chavez, "Hamsa: fast signature generation for zero-day polymorphic worms with provable attack resilience," in *Security and Privacy, 2006 IEEE Symposium on*, pp. 15 pp. –47, may 2006.

[90] W. Robertson, G. Vigna, C. Kruegel, and R. Kemmerer, "Using Generalization and Characterization Techniques in the Anomaly-based Detection of Web Attacks," in *Proceeding of the Network and Distributed System Security Symposium (NDSS)*, (San Diego, CA), February 2006.

[91] A. Kamra, E. Bertino, and G. Lebanon, "Mechanisms for database intrusion detection and response," in *Proceedings of the 2nd SIGMOD PhD workshop on Innovative database research*, IDAR '08, (New York, NY, USA), pp. 31–36, ACM, 2008.

[92] S. Y. Lee, W. L. Low, and P. Y. Wong, "Learning fingerprints for a database intrusion detection system," in *Proceedings of the 7th European Symposium on Research in Computer Security*, ESORICS '02, (London, UK, UK), pp. 264–280, Springer-Verlag, 2002.

[93] P. Bisht, T. Hinrichs, N. Skrupsky, R. Bobrowicz, and V. N. Venkatakrishnan, "Notamper: automatic blackbox detection of parameter tampering opportunities in web applications," in *Proceedings of the 17th ACM conference on Computer and communications security*, CCS '10, (New York, NY, USA), pp. 607–618, ACM, 2010.

[94] G. Vigna, F. Valeur, D. Balzarotti, W. Robertson, C. Kruegel, and E. Kirda, "Reducing Errors in the Anomaly-based Detection of Web-Based Attacks through the Combined Analysis of Web Requests and SQL Queries," *Journal of Computer Security*, vol. 17, no. 3, 2009.

[95] R. Chandra, T. Kim, M. Shah, N. Narula, and N. Zeldovich, "Intrusion recovery for database-backed web applications," in *Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles*, SOSP '11, (New York, NY, USA), pp. 101–114, ACM, 2011.

[96] W. Robertson, F. Maggi, C. Kruegel, and G. Vigna, "Effective anomaly detection with scarce training data," in *Proceedings of the Network and Distributed System Security Symposium, NDSS 2010, San Diego, California, USA, 28th February - 3th March 2010*, The Internet Society, 2010.

[97] P. Mell, V. Hu, R. Lippmann, J. Haines, and M. Zissman, "An overview of issues in testing intrusion detection systems," tech. rep., NIST, 6 2003.

[98] C. Gates, C. Taylor, and M. Bishop, "Dependable security: Testing network intrusion detection systems," in *Third Workshop on Hot Topics in System Dependability (HotDep'07)*, 2007.

[99] G. Modelo-Howard, J. Sweval, and S. Bagchi, "Secure configuration of intrusion detection sensors for changing enterprise systems," in *SecureComm '11*, 2011.

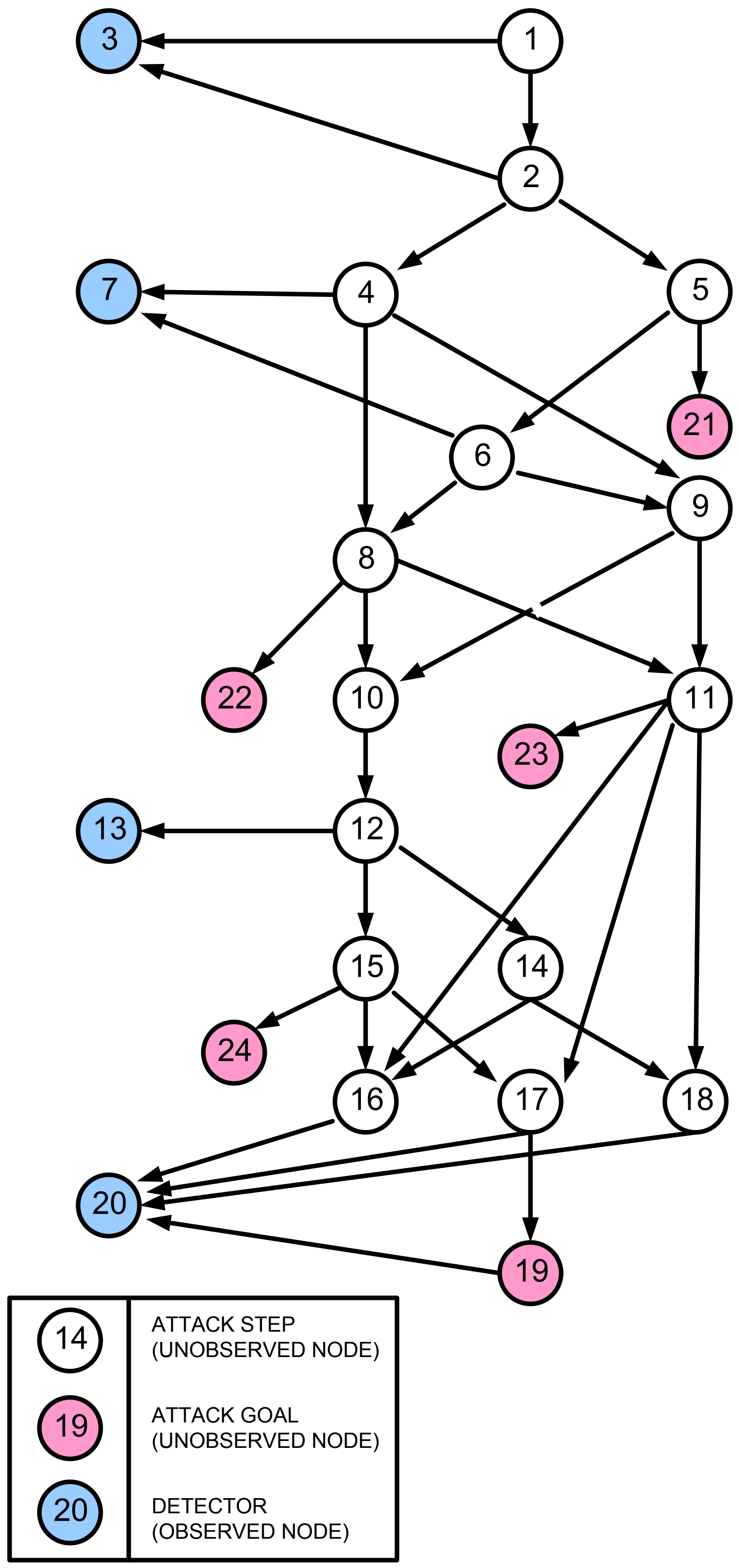
[100] D. E. Denning, "An intrusion-detection model," *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING*, vol. 13, no. 2, pp. 222–232, 1987.

[101] L. Peck, "Countering ambiguity in exterior intrusion detection," in *Security Technology, 2008. ICCST 2008. 42nd Annual IEEE International Carnahan Conference on*, pp. 89 –93, oct. 2008.

APPENDICES

A. DESCRIPTION OF E-COMMERCE BAYESIAN NETWORK

We provide a description of the Bayesian network built for the e-commerce system used in the experiments. It includes a description of each node in the Bayesian network, for the observed and unobserved nodes, as well as the corresponding probability values (shown as tables) associated with each node.



DESCRIPTION OF NODES

NODE 1: Attack step - ping or traceroute to web servers

NODE 2: Attack step - run portscanner on web servers

NODE 3: Detector alert – IPTables

NODE 4: Attack step - exploit ssldump vuln. on web server

NODE 5: Attack step - access web server admin site

NODE 6: Attack step - Brute force admin pwd

NODE 7: Detector alert – Snort

NODE 8: Attack step - Copy hacker tool to web server by using tftp

NODE 9: Attack step - Install vuln scanner on web server

NODE 10: Attack step - Run portscanner on internal network

NODE 11: Attack step - Install sniffer to capture pwds

NODE 12: Attack step - Exploit rpc.statd service on app controller

NODE 13: Detector alert – Libsafe

NODE 14: Attack step - Exploit remote vuln. on MySQL server

NODE 15: Attack step - Brute force root pwd on app controller

NODE 16: Attack step - Run SQLplus to execute queries on tables

NODE 17: Attack step - Connect to MySQL server with admin account

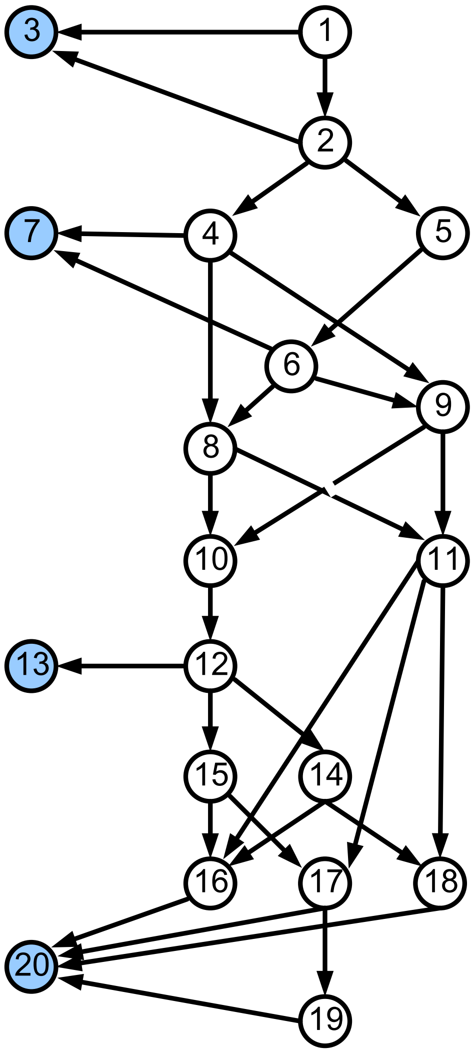
NODE 18: Attack step - Read customer data table

NODE 19: Attack step - Copy customer credit card list

a

NODE 20: Detector alert - Database IDS (Application Security DbProtect)

Fig. A.1. Bayesian network for the e-commerce system with corresponding description of the nodes. Each node is either an attack step or a detector.



	X1
F	0.99998
T	0.00002

	X2	
X1	F	T
F	0.7	0.3
T	0.3	0.7

		X3	
X2	X1	F	T
F	F	0.9	0.1
T	F	0.6	0.4
F	T	0.4	0.6
T	T	0.2	0.8

	X4	
X2	F	T
F	0.3	0.7
T	0.1	0.9

	X5	
X2	F	T
F	0.3	0.7
T	0.1	0.9

	X6	
X5	F	T
F	0.9	0.1
T	0.7	0.3

		X7	
X6	X4	F	T
F	F	0.95	0.05
T	F	0.7	0.3
F	T	0.3	0.7
T	T	0.15	0.85

		X8	
X6	X4	F	T
F	F	0.3	0.7
T	F	0.2	0.8
F	T	0.2	0.8
T	T	0.1	0.9

		X9	
X6	X4	F	T
F	F	0.99	0.01
T	F	0.3	0.7
F	T	0.3	0.7
T	T	0.01	0.99

		X10	
X9	X8	F	T
F	F	0.99	0.01
T	F	0.1	0.9
F	T	0.1	0.9
T	T	0.01	0.99

		X11	
X9	X8	F	T
F	F	0.99	0.01
T	F	0.1	0.9
F	T	0.1	0.9
T	T	0.01	0.99

	X12	
X10	F	T
F	0.2	0.8
T	0.1	0.9

	X13	
X12	F	T
F	0.9	0.1
T	0.1	0.9

	X14	
X12	F	T
F	0.2	0.8
T	0.1	0.9

	X15	
X12	F	T
F	0.2	0.8
T	0.1	0.9

				X20	
X19	X18	X17	X16	F	T
F	F	F	F	0.99	0.01
T	F	F	F	0.4	0.6
F	T	F	F	0.35	0.65
T	T	F	F	0.3	0.7
F	F	T	F	0.35	0.65
T	F	T	F	0.3	0.7
F	T	T	F	0.3	0.7
T	T	T	F	0.15	0.85
F	F	F	T	0.35	0.65
T	F	F	T	0.3	0.7
F	T	F	T	0.3	0.7
T	T	F	T	0.15	0.85
F	F	T	T	0.3	0.7
T	F	T	T	0.15	0.85
F	T	T	T	0.15	0.85
T	T	T	T	0.01	0.99

			X16	
X15	X14	X11	F	T
F	F	F	0.9	0.1
T	F	F	0.8	0.2
F	T	F	0.7	0.3
T	T	F	0.3	0.7
F	F	T	0.7	0.3
T	F	T	0.3	0.7
F	T	T	0.3	0.7
T	T	T	0.2	0.8

		X17	
X15	X11	F	T
F	F	0.95	0.05
T	F	0.7	0.3
F	T	0.8	0.2
T	T	0.2	0.8

		X18	
X14	X11	F	T
F	F	0.9	0.1
T	F	0.6	0.4
F	T	0.4	0.6
T	T	0.2	0.8

	X19	
X17	F	T
F	0.9	0.1
T	0.1	0.9

Fig. A.2. Bayesian network for the e-commerce system with the conditional probabilities values used for the experiments.

B. CALCULATION OF APPROXIMATION RATIO FOR GREEDY ALGORITHM

We provide the calculation of the approximation ratio for a *Greedy algorithm* of the 0-1 knapsack problem (KP), for the bounded case (when there is a limited number of items from which to pick and put in the knapsack). The proofs for the calculation of the approximation ratio are adapted from [34] and [35]. We include the proofs in this paper to make the previous proofs more accessible to a systems security audience and to show the thinking process that went on behind our search for FPTAS after having designed the Greedy solution.

KP can be formally defined as the following: given an *instance* with item set N, consisting of n items x_i, each with a profit p_i and weight w_i. The knapsack has a capacity value c. The objective is to select a subset of N such that the total profit of the selected items ($\Sigma_{i=1}^{n} p_i x_i$) is maximized subject to the corresponding total weight not exceeding the knapsack capacity ($\Sigma_{i=1}^{n} w_i x_i \leq c$). The optimal solution value is denoted by z^{OPT}.

The idea of the Greedy algorithm with a solution value z^{G} is to start with an empty knapsack, sort the items in decreasing order according to its profit to weight ratio $\frac{p_i}{w_i}$ and go through the sorted items, adding every item into the knapsack while its capacity is not overwhelmed. The final step is making a comparison between the given solution and the highest profit value of any item. The larger of the two is finally taken and is denoted by z^{mG}. This final step can be considered a modification of the original Greedy algorithm found in literature [33], but necessary to guarantee an approximation ratio of $\frac{1}{2}$ to the optimal solution.

A linear programming relaxation (LKP) is made to compute the approximation ratio, omitting the integer constraint of KP and optimizing instead over all nonnegative real values. Naturally, the optimal solution value z^{LKP} of the relaxed problem is at least as large as the original value z^{OPT} because the set of feasible solutions for the original KP is a subset of the feasible solutions for the relaxed problem. The Greedy algorithm for LKP packs the items in decreasing order of profit-to-weight ratio, similar to the original Greedy algorithm, but with one difference. When adding an item s to the knapsack would cause the capacity c to overflow for the first time, only an appropriate fractional part of the item is used. Item s is referred as the split item, its corresponding profit as p_s and weight as w_s. The split solution, not including the split item, is defined by a profit $\hat{p}$ and weight $\hat{w}$. Therefore, the optimal solution value of LKP is defined as

$$z^{LKP} = \Sigma_{i=1}^{s-1} p_i + (c - \Sigma_{i=1}^{s-1} w_i)\frac{p_s}{w_s}. \tag{B.1}$$

The value z^{LKP} is an upper bound on the optimal solution for KP. A tighter upper bound U^{LP} for z^{OPT} can be obtained by using the floor of z^{LKP}, i.e. $U_{LP} := \lfloor z^{LKP} \rfloor$, since all data are integers. Then we get the following bounds on z^{LP}:

$$\hat{p} \le z^{OPT} \le U_{LKP} \le z^{LKP} \le \Sigma_{i=1}^{s} p_i = \hat{p} + p_s = z^G + p_s. \tag{B.2}$$

Another consequence of these considerations is the following fact:

$$z^{OPT} - z^G \le z^{OPT} - \hat{p} \le p_{max}, \tag{B.3}$$

where p_{max} denotes the largest profit of any item in the set N.

The Greedy algorithm has an approximation ratio of $\frac{1}{2}$ and this bound is tight. As proof, we know from (3) that: $z^{OPT} \le z^G + p_{max} \le z^{mG} + z^{mG} = 2z^{mG}$

The tightness of the bound can be shown by the following example. Item 1 is given by $w_1 = 1$, $p_1 = 2$, and $b_1 = 1$ (number of item 1 available). Item 2 is given by $w_2 = p_2 = M$ and $b_2 = 2$. The knapsack capacity is $c = 2M$. The Greedy algorithm would pack item 1 first and then an item 2, reaching a solution value of $2 + M$ while the optimal solution would pack items 2 and would reach a value of $2M$. Choosing M large enough, the ration between the approximate and optimal solution value can be arbitrarily close to $\frac{1}{2}$.

C. ALGORITHMS FOR DIADS

The DIADS framework presented in chapter 3 is composed of four algorithms:

BN-STRUCTURE-UPDATE The structure of the Bayesian Network (BN) is updated, using the changes made to the firewall rule table. The algorithm produces a list of nodes and edges that should be added or deleted from the BN and is presented later in this appendix (algorithm C.1).

BN-CPT-UPDATE The conditional probability tables (CPT) of the BN are updated, using the changes made to the firewall rule table. The algorithm produces a lists of CPTs for the changed nodes in the BN, i.e., nodes for which there is an increase or deduction in the number of parents and according to the output from the BN-STRUCTURE-UPDATE algorithm. We present the BN-CPT-INITIALIZATION below (algorithm C.2).

CPT-UPDATE-NOISY-OR The alerts received by the reasoning engine from different detection sensors are used to update the CPTs in the BN, in an incremental manner. This algorithm uses a popular and powerful model known as Noisy-OR **??** that represents the core of the algorithm.

SENSOR-RECONFIGURATION This algorithm is used to reconfigure the detection sensors. This includes adding and removing sensors, as well as reconfiguring existing ones. The algorithm is presented below as C.3.

Algorithm C.1 BN-STRUCTURE-UPDATE $(message, A)$

Input: message $m = (number, srcIPaddr, destIPaddr, portnumber, action, ruletype)$. This input represents an addition, change, or deletion of a firewall rule; Adjacency matrix representation of Bayesian network $BNet = (V, E)$ consists of a $|V|x|V|$ matrix $A = (a_{ij})$ such that $a_{ij} = 1$ if $(i, j) \in E$ otherwise $a_{ij} = 0$

Output: V_a = set of nodes to add, V_d = set of nodes to delete, E_a = set of edges to add, E_d = set of edges to delete

```
 1: //case when a rule is added
 2: if $ruletype$ = add then
 3:    if $srcIPaddr : *$ in $A$ then
 4:       add all $(parents(srcIPaddr : *), srcIPaddr : *)$ to $E_a$
 5:    end if
 6:    if $destIPaddr : port$ in $A$ then
 7:       add all $(destIPaddr : port, children(destIPaddr : port))$ to $E_a$
 8:    else
 9:       add $E_a \leftarrow (srcIPaddr : *, destIPaddr : port)$
10:    end if
11: end if
12: // case when a rule is deleted
13: if $ruletype$ = delete then
14:    add $E_d \leftarrow (srcIPaddr : *, destIPaddr : port)$
15:    if $srcIPaddr : *$ in $A$ then
16:       if not$parents(srcIPaddr : *)$ then
17:          add $V_d \leftarrow srcIPaddr : *$
18:       else
19:          add all $(parents(srcIPaddr : *), srcIPaddr : *)$ to $E_d$
20:       end if
21:    end if
```

```
22:     if $destIPaddr : port$ in $A$ then
23:         if not$children(destIPaddr : port)$ then
24:             add $V_d \leftarrow destIPaddr : port$
25:         else
26:             add all $(destIPaddr : port, children(destIPaddr : port))$ to $E_d$
27:         end if
28:     end if
29: end if
30: // check if new edge creates a path to the end goal and if node creates a cycle
31: for all $address : port \in V \cup V_a$ do
32:     run DFS from $address : port$
33:     if not$(address : port \rightarrow V_{CA})$ then
34:         remove $address : port$ from $V_a$
35:     end if
36:     add $backedges$ to $E_d$
37: end for
38: // convert address:port node to address:port:vulnerability node
39: for all $address : port \in V_a$ do
40:     if $vulnerability(address : port) \in NVD$ then
41:         update $address : port$ to $address : port : vulnerability(v_i)$ in $V_a$ and $E_a$
42:     else
43:         remove $address : port$ from $V_a$
44:     end if
45: end for
46: for all $address : port \in V_d$ do
47:     search BNET and replace for corresponding $address : port : vulnerablity(v_i)$
48: end for
49: return $V_a, V_d, E_a, E_d$
```

Algorithm C.2 BN-CPT-UPDATE (V_a, V_d, E_a, E_d)

Input: V_a = set of nodes to add, V_d = set of nodes to delete, E_a = set of edges to add, E_d = set of edges to delete

Output: S_{CPT} = set of CPTs to update

```
1: for all $v_i \in V_a$ do
2:     new $Prob(v_i) = CVSS(v_i)/10$
3:     add each $outedge(v_i) \in E_a$
4:     for all $children(v_i)$ do
5:         update CPT using $max(newProb(v_i) + \Delta, oldProb(v_i))$
6:     end for
7: end for
8: for all $(v_i, v_j) \in E_a$ do
9:     new $Prob(v_i) = CVSS(v_i)/10$
10:    add each $(v_i, v_j) \in E_a$
11:    for all $children(v_i)$ do
12:        update CPT using $max(newProb(v_i) + \Delta, oldProb(v_i))$
13:    end for
14: end for
15: for all $v_i \in V_d$ do
16:    new $Prob(v_i) = CVSS(v_i)/10$
17:    remove all $inedge(v_i)$ and $outedge(v_i)$
18:    for all $children(v_i)$ do
19:        update CPT using $max(newProb(v_i) + \Delta, oldProb(v_i))$
20:    end for
21: end for
22: for all $(v_i, v_j) \in E_d$ do
23:    new $Prob(v_i) = CVSS(v_i)/10$
24:    remove all $(v_i, v_j) \in E_d$
25:    for all $v_j$ do
26:        update CPT using $max(newProb(v_i) + \Delta, oldProb(v_i))$
27:    end for
28: end for
```

Algorithm C.3 SENSOR-RECONFIGURATION $(E, Detectors_{existing})$

```
Input: $E = evidence$, represented by set of alerts received; $Detectors_{existing}$ = set of
    detectors currently enabled
Output: set of nodes to enable/disable. Nodes correspond to $< address, port, vulnerability >$ tuple so can be mapped to a detection sensor
1: compute $a = Prob(critical\ asset\ |E)$
2: if $a > threshold$ then
3:    Create set of candidate sensors close to $E$ and $critical\ asset$
4:    Run $FPTAS(BN)$
5: end if
6: $Detectors_{disable} = |Detectors_{existing} - Detectors_{FPTAS}|$
    $Detectors_{FPTAS}, Detectors_{disable}$
```

D. BAYESIAN NETWORK USED FOR DIADS EXPERIMENTS

Below we show the Bayesian Network (BN) used for the experiments presented in chapter 3. The BN was created from a real-world distributed system which is part of an NSF Center at Purdue University. The system includes fifteen hosts that include two environments, one for production and another for development of applications and staging, prior to moving them to the production environment. Each environment includes a web server, an application server, and a database server. A team of developers' and consultants' computer have access to subsets of both environments.

To create the BN, we first generated a list of vulnerabilities found in the distributed system with the OpenVAS [56] vulnerability scanner. Each vulnerability was then mapped to a node in the BN, by associating it to the host and service (port) were the vulnerability was found. The nodes were connected according to the connectivity information for the distributed system. The resulting BN had 345 nodes and 1948 edges. We then pruned the BN to only include high risk vulnerabilities, according to the OpenVAS tool, as these ones are the primary vectors used by attackers to compromise sustems. The final BN, shown below, has 90 nodes and 582 edges.

The use the following color code to identify the computers and servers of the distributed system: The six light blue boxes in the top left corner of the diagram correspond to the developers' computers. The three purple boxes (top right corner) are for the consultants' computers. The two orange boxes (bottom left corner) are the web and application servers for the production environment. To their right, the red box corresponds to the database server in the same environment. The final three boxes (bottom right corner) are the three servers in the development environment: database (red), web (yellow), and application (yellow).

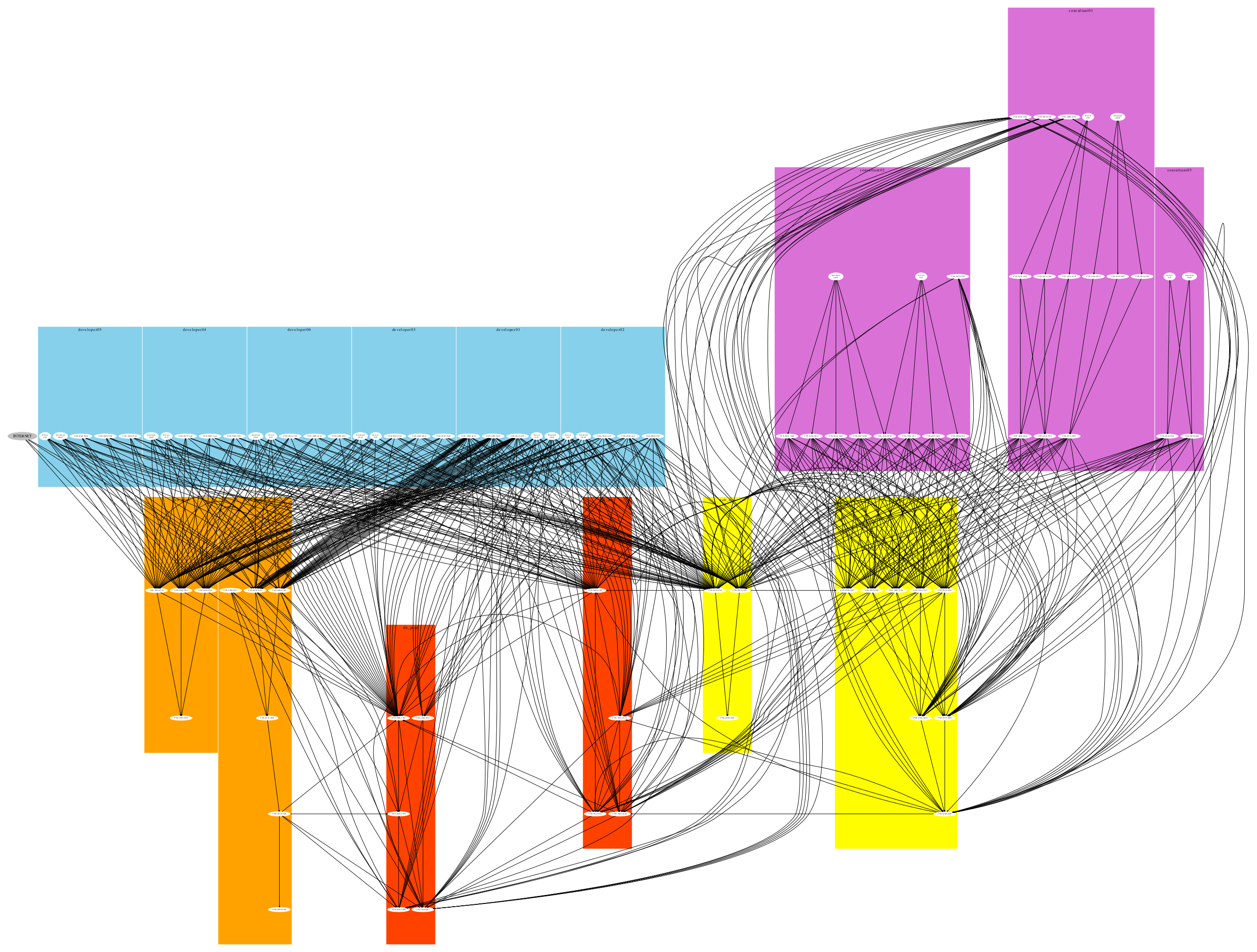


Fig. D.1. Bayesian Network created from an NSF Center at Purdue University and used for experiments presented in chapter 3.

E. SET OF SIGNATURES GENERATED WITH PSIGENE

From each bicluster b_j, we create a signature Sig_{b_j} which characterizes the samples in that bicluster, plus is more generalized. Specifically, in our solution, a signature Sig_{b_j} is a logistic regression model built to predict whether an SQL query is an attack similar to the samples in cluster b_j. In other words, the signature is the hypothesis (sigmoid) function produced with logistic regression:

$$Sig_{b_j}(\Theta_j) = \frac{1}{1 + e^{-\Theta_j^T}}$$

Nine signatures were created for our experiments. For each of these signatures, we present below the features used and the corresponding coefficients Θ_j computed from the logistic regression phase:

Table E.1: Coefficients and Features for Signature 1.

COEFFICIENT	**FEATURE (Regular Expression)**
-2.892839	
-0.133652	`insert`
0.007340	`[\"’‘]\s*?(x?or)\s*?[\"’‘]?\d`
-0.008639	`^[\W\d]+\s*?desc`
-0.204895	`drop`
0.019437	`length`
-0.003320	`delete`
0.019437	`\bselect\b.{0,40}\bascii\b`
-0.001505	`exec`
-0.134897	`[\"’‘]|?[\w-]{3,}[^\w\s.,]+[\"’‘]`

COEFFICIENT	FEATURE (Regular Expression)
-0.146769	[\"’‘]\s*?(and)\s[^\d]+[\w-]+.*?\d
0.213769	in\s*?\(+\s*?select
0.154269	benchmark\((.*?)\,(.*?)\)
-0.001133	create
0.069142	x?x?or[\s(]+\w+[\s)]*?[!=+]+[\s\d]*?[\"’‘=()]
0.199486	;\s*?select\s*?[\[(]?\w{2,}
0.212993	^[\W\d]+\s*?select
1.093432	[’"]
1.107889	[\"’‘]\s*?|{
-0.156529	[\"’‘]\s*?[^\w\s]?=\s*?[\"’‘]
-0.156529	[\"’‘]\W*?[+=]+\W*?[\"’‘]
0.296613	\!\=|\&\&|\\|\\||>>|<<|>=|<=|<>|<=>|xor|rlike|regexp|isnull
0.108078	\\|\\|\s*?\w+\(
0.227742	\blike\W*?char\W*?\(
-0.005946	[\"’‘]\s*?[^\w\s?]+\s*?[^\w\s]+\s*?[\"’‘]
0.209669	\(\s*?select\s*?\w+\s*?\(
0.334135	[\s(]load_file\s*?\(
-0.011009	,.*[\da-f\"’‘][\"’‘]\Z
-0.262865	!=|<=|>=|<>|<|>|\^|is\s+not|not\s+like|not\s+regexp
0.346328	--[^-]*?-
0.101613	^["’‘;]
0.117828	[\"’‘].*?*\s*?\d
0.769650	db_name\W*\(
0.754294	--[\s\r\n\v\f]

Table E.2: Coefficients and Features for Signature 2.

COEFFICIENT	**FEATURE (Regular Expression)**
-3.367189	
0.036476	`insert`
0.021010	`^[\W\d]+\s*?desc`
-0.145124	`drop`
0.008877	`delete`
0.004067	`exec`
0.003068	`create`
0.278587	`[’"]`
0.280161	`[\"’‘????????]\s*?|{`
-0.063421	`[’;]--`
-0.063421	`[\"’‘????????]\s*?--`
0.001034	`\!\=|\&\&|\\|\\||>>|<<|>=|<=|<>|<=>|xor|rlike|regexp|isnull`
-0.022594	`!=|<=|>=|<>|<|>|\^|is\s+not|not\s+like|not\s+regexp`
0.006042	`--[^-]*?-`

Table E.3: Coefficients and Features for Signature 3.

COEFFICIENT	**FEATURE (Regular Expression)**
-4.561008	
-0.017283	`insert`
-0.004940	`^[\W\d]+\s*?desc`
-0.155173	`drop`
-0.001858	`delete`
-0.000827	`exec`
-0.000620	`create`

COEFFICIENT	FEATURE (Regular Expression)
1.816763	`[’"]`
1.872849	`[\"’‘]\s*?|{`
-0.000207	`\!\=|\&\&|\\|\\||>>|<<|>=|<=|<>|<=>|xor|rlike|regexp|isnull`
-0.263687	`!=|<=|>=|<>|<|>|\^|is\s+not|not\s+like|not\s+regexp`
-0.001239	`--[^-]*?-`

Table E.4: Coefficients and Features for Signature 4.

COEFFICIENT	FEATURE (Regular Expression)
-3.623943	
-0.046207	`([\s’\"‘\(\)]*)([\d\w]+)([\s’\"‘\(\)]*)` `(=|<=>|r?like|sounds\s+like|regexp)[\s’\"‘\(\)]*\2`
0.350203	`@`
0.019025	`coalesce\s*?\(|@@\w+\s*?[^\w\s])`
1.774084	`char`
2.559842	`ch(a)?r\s*?\(\s*?\d)`
0.019470	`information_schema`
0.019460	`\btable_name\b`
0.019460	`\Wtable_name\W`

Table E.5: Coefficients and Features for Signature 5.

COEFFICIENT	FEATURE (Regular Expression)
-8.431682	
2.627309	`\(`
0.815482	`(current_)?user\s*?\([^\)]*?`

COEFFICIENT	FEATURE (Regular Expression)
0.758978	select.*?\w?user\(
1.097862	database\W*\(
1.097862	(current_)?database\s*?\([^\)]*?

Table E.6: Coefficients and Features for Signature 6.

COEFFICIENT	FEATURE (Regular Expression)
-3.761054	
0.262131	=
0.262131	=[-0-9\%]*
0.261463	<=>|r?like|sounds\s+like|regex
0.261584	([^a-zA-Z&]+)?&|exists
-0.117270	[\?&][^\s\t\x00-\x37\\|]+?
0.708324	\)?;

Table E.7: Coefficients and Features for Signature 7.

COEFFICIENT	FEATURE (Regular Expression)
-4.670291	
3.322430	\(
0.015823	(current_)?user\s*?\([^\)]*?
0.806333	(x?x?\s+)\s*?\w+\(
0.002432	\bselect\b.{0,40}\busers?\b
3.601496	version

Table E.8: Coefficients and Features for Signature 8.

COEFFICIENT	**FEATURE (Regular Expression)**
-5.397672	
0.377096	=
0.377096	=[\-0-9%]*
0.410321	=|<=>|r?like|sounds\s+like|regex
0.377096	([^a-zA-Z&]+)?=|[eE][xX][iI][sS][tT][sS]
0.172626	[\?&][^\s\t\x00-\x37\\|]+?
0.148376	\)?;

Table E.9: Coefficients and Features for Signature 9.

COEFFICIENT	**FEATURE (Regular Expression)**
-4.738639	
0.019820	[\"’][\s\d]*?[^\w\s]+\W*?\d\W*?.*?[\"’\d]
0.019820	[()*<>%+-][\w-]+[^\w\s]+[\"’][^,]
-0.006550	cast
-0.001040	[\"’;]+$
-0.000174	^[\W\d]+\s*?union

Table E.10: Coefficients and Features for Signature 10.

COEFFICIENT	**FEATURE (Regular Expression)**
-4.197985	
0.039358	(\/*.*?*\/)+?
0.039358	*/

COEFFICIENT	FEATURE (Regular Expression)
-0.825733	([\s’\"\(\)]*)([\d\w]+)([\s’\"\(\)]*) (=|<=>|r?like|sounds\s+like|regexp)[\s’\"\(\)]*
-0.003813	@
-0.018053	char
-0.000582	ch(a)?r\s*?\(\s*?\d

Table E.11: Coefficients and Features for Signature 11.

COEFFICIENT	FEATURE (Regular Expression)
-5.912046	
0.044724	(length of string)
-0.096772	[\s’\"\(\)]*
-1.086292	([\s\t\x00-\x37]|\/*.*?*\/|\)?;)+.*?
-0.333336	[A-Za-z]{1}
0.381274	[\s\t\r\n\v\f]{1}
0.381274	" " (space)
-0.435454	[0-9]{1}
0.061053	[\s\t\x00-\x37]
1.397264	[!"#$%&’()*+,./:;<=>?@˜_‘{|}˜-]{1}
0.987805	([\d\w]+)
-0.035547	and
-0.107282	([ˆa-zA-Z&]+)=
-0.000218	([\s\t\x00-\x37]|\/*.*?*\/|\)?;)+ ([xX]?[oO][rR]|[nN]?[aA][nN][dD])
-0.066444	[\[(]+[a-zA-Z&]{2,}?

VITA

VITA

Gaspar Modelo-Howard received his B.Sc. degree in Electrical and Electronics Engineering from Technological University of Panama (Panama) in 1996. He obtained his M.Sc. degree in Information Security from Royal Holloway College, University of London (United Kingdom) in 1999. Gaspar joined Purdue University in the Fall of 2006 as a Ph.D. student in Computer Engineering. He worked as a research assistant for Prof. Saurabh Bagchi and Prof. Guy Lebanon. His research interests include systems security, intrusion detection, and machine learning. He has worked as a security engineer for more than 10 years at the Panama Canal Authority and Purdue University. During his Ph.D. studies, he spent the Summer of 2009 at Hewlett-Packard Labs in Princeton, New Jersey developing a rule-based policy compliance checking software for a payment card industry standard. Additionally, his research project was selected for three years to represent Purdue University in the Northrop Grumman Cyber-Security Consortium, an industry-academia partnership set out to advance research in cyber-security.