Approval Sheet

Title of Dissertation: Intrusion Detection: Modeling System State to Detect and Classify Aberrant Behaviors

Name of Candidate: Jeffrey L. Undercoffer
Doctor of Philosophy, 2004

Thesis and Abstract Approved: _____

Dr. John Pinkston
Professor and Chair
Department of Computer Science and Electrical Engineering

Date Approved: _____

Curriculum Vitae

Name: Jeffrey L. Undercoffer.

Permanent Address:

Degree and date to be conferred: Doctor of Philosophy, 2004.

Date of Birth:

Place of Birth:

Collegiate institutions attended:

- University of Maryland, University College, Bachelor of Science, Computer Science, 1984.
- University of Maryland, College Park, Master of Science, Software Engineering, 1999.
- University of Maryland, Baltimore County, Doctor of Philosophy, Computer Science, 2004.

Major: Computer Science.

Professional publications:

- Jeffrey Undercoffer, Lalana Kagal, Filip Perich, Anupam Joshi, and Tim Finin. "Vigil: Policy Based Access Control for Pervasive Computing Environments", under review *9th European Symposium on Research in Computer Security*, September, 2004.
- Filip Perich, Lalana Kagal, and Jeffrey Undercoffer. "Using Distributed Belief to Improve Query Processing Accuracy in Mobile A d-Hoc Networks", under review *23rd ACM SIGMOD International Conference on Management of Data*, June, 2004.

- James Parker, Jeffrey Undercoffer, John Pinkston, and Anupam Joshi. "On Intrusion Detection in Mobile Ad Hoc Networks", in *Proceedings, 23rd IEEE International Performance Computing and Communications Conference – Workshop on Information Assurance* , April 2004.

- Jeffrey Undercoffer, Anupam Joshi, Tim Finin, and John Pinkston. "A Target Centric Ontology for Intrusion Detection: Using DAML+OIL to Classify Intrusive Behaviors", in *Knowledge Engineering Review*, 2004.

- Sasikanth Avancha, Jeffery Undercoffer, Anupam Joshi and John Pinkston. "Security for Sensor Networks" *Wireless Sensor Networks*, edited by Taieb Znati, Krishna M. Sivalingam and Cauligi Raghavendra. Kluwer Academic Publishers, 2004.

- Jeffrey Undercoffer and Anupam Joshi. "Data Mining, Semantics and Intrusion Detection: What to dig for and Where to find it" In *Data Mining: Next Generation Challenges and Future Directions*, edited by Hillol Kargupta, Anupam Joshi, K. Sivakumar, and Yelena Yesha. MIT Press, 2004.

- Sasikanth Avancha, Jeffery Undercoffer, Anupam Joshi and John Pinkston. "Secure Sensor Networks for Perimeter Protection", in *International Journal of Computer and Telecommunications Networking*, 2003.

- Jeffrey Undercoffer, Andrej Cedilnik, Filip Perich, and Lalana Kagal, and Anupam Joshi. "A Secure Infrastructure For Service Discovery And Management In Pervasive Computing" in *Journal of Mobile Networks and Applications (MONET)*, 2003.

- Jeffrey Undercoffer, Filip Perich, Anupam Joshi and John Pinkston. "SHOMAR: A Secure Framework for Distributed Intrusion Detection Services" under review, *ACM Transactions on Information and System Security*, 2003.

- Jeffrey Undercoffer, Anupam Joshi, and John Pinkston. "Modeling Computer Attacks: An Ontology for Intrusion Detection" in *Proceedings of the Sixth International Symposium on Recent Advances in Intrusion Detection*, 2003.

- Hiren Shah, Jeffrey Undercoffer and Anupam Joshi. "Fuzzy Clustering for Intrusion Detection" in *Proceedings of the 12th IEEE International Conference on Fuzzy Systems*, 2003

- Lalana Kagal, Jeffrey Undercoffer, Anupam Joshi, and Tim Finin. "A Security Architecture Based on Trust Management for Pervasive Computing Systems" in *Proceedings of the Grace Hopper Celebration of Women in Computing*, 2002.

- J. Undercoffer, S. Rajavaram, H. Shah, V. Shanbhag, and A. Joshi. "Neighborhood Watch: An Intrusion Detection and Response Protocol for Mobile Ad-hoc Networks" at *UMBC Student Research Conference*, 2002.

- Jeffrey Undercoffer, Anupam Joshi, Tim Finin, and John Pinkston. "A Target-Centric Ontology for Intrusion Detection" in *18th International Joint Conference on Artificial Intelligence Workshop on Ontologies in Distributed Systems*, 2003.

- James Butler, Jeffrey Undercoffer and John Pinkston. "Hidden Processes: The Implication for Intrusion Detection" in *The Proceedings of the IEEE Systems, Man and Cybernetics Society Information Assurance Workshop*, 2003.

- Anupam Joshi and Jeffrey Undercoffer. "On Web, Semantics, and Data Mining: Intrusion Detection as a Case Study" in *Proceedings of the NSF Workshop on Next Generation Data Mining*, 2003.

- Sasikanth Avancha, Jeffery Undercoffer, Anupam Joshi and John Pinkston. "A Clustering Approach to Secure Sensor Networks" *Technical Report TR-CS-03-19*, Department of Computer Science and Electrical Engineering University of Maryland Baltimore County, 2003.

- Lalana Kagal, Jeffrey Undercoffer, Anupam Joshi, and Tim Finin. "Vigil: Providing Trust for Enhanced Security in Pervasive Systems" *Technical Report TR-CS-02-19*, Department of Computer Science and Electrical Engineering University of Maryland Baltimore County, 2002.

- Ed Perl, Jeffrey Undercoffer, and Deepinder Sidhu. "A Perspective on the Scalability of MPLS Signaling" *Technical Report TR-CS-02-07*, Department of Computer Science and Electrical Engineering University of Maryland Baltimore County, 2002.

Professional positions held:

-

- Research Assistant (Sep. 2001 - Jan. 2004).

  Department of Computer Science and Electrical Engineering University of Maryland, Baltimore County

- Solutions Director (Jan. 1999 - Dec. 2001)

  World Wide Security Practice - Unisys Corporation

- Assistant to the Special Agent in Charge. (Nov. 1977 - Jan. 1999).

  Presidential Protective Division - Unites States Secret Service

Abstract

Title of Dissertation: Intrusion Detection: Modeling System State to Detect and Classify Aberrant Behaviors

Author: Jeffrey L. Undercoffer, PhD, 2004

Thesis directed by: Dr. John Pinkston, Professor and Chair
Department of Computer Science and
Electrical Engineering

We present a dual-phase host-based intrusion detection process. We have demonstrated, through experimental validation, that our process improves the current state of intrusion detection capabilities. The first phase uses cluster analysis to compare samples of low-level operating system data to an established model of normalcy. The second phase takes instances of non-conforming data from phase-1, maps that data to instances of our *target-centric* ontology and reasons over it. The reasoning process serves two purposes: primarily it is intended to classify the anomalous data as a specific type, or class, of attack. Its secondary purpose is to provide an orthogonal test to differentiate between true and false positives.

We developed a novel metric (*self-distance*) to quantify the streams of system calls generated by a process, and we have constructed a feature set from the low-level operating system data, which is subsequently used as input to the clustering process. We experimented with different clustering algorithms (*Fuzzy c-Medoid, k-Means*, and *Principal Direction Divisive Partitioning*), distance measures (*Euclidean* and *Mahalanobis*), and the effects of *z-normalizing* the data set. Our experiments indicated that the *Fuzzy c-Mediod* algorithm using the *Mahalanobis* metric as a distance measure was the optimal performer, yielding an

F-Measure of .9822. The F-Measure is a common method for describing accuracy and is combination of *precision* and *recall*.

We experimentally demonstrated the case for migrating from taxonomic classification systems and their syntactical representation languages to ontologies and semantically rich ontology specification languages. We created a data model of the relationships that hold between the low-level data and instances of attacks and intrusions. We used the DARPA Agent Markup Language + Ontology Inference Layer to specify the data model as a ontology and the *Java Theorem Prover*, a sound and complete First Order Logic theorem prover, to reason over and classify instances data that were deemed to be anomalous in the first phase of our process. Our classification mechanism achieved an *F-Measure* of .9776.

The overall *F-Measure* of our dual-phase process was .9718. Ignoring the characteristics of the data population is a classic mistake that is made when evaluating intrusion systems. This is also referred to as the base-rate fallacy. When evaluating the posterior probability (the probability of an alarm given an intrusion) of our process we achieve a score of .998.

We also present two novel mechanisms to detect and mitigate aberrant behaviors encountered in Mobile Ad Hoc and Wireless Sensor networks. Both of these networks consist of resource constrained devices. Accordingly, we present our intrusion detection mechanisms as protocols that monitor network state rather than system state.

Intrusion Detection: Modeling System State to Detect and Classify Anomalous Behaviors

by

Jeffrey L. Undercoffer

Dissertation submitted to the Faculty of the Graduate School

of the University of Maryland in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

2004

In memory of Joseph John Undercoffer

May 5, 1976 — December 14, 2001

First and foremost I would like to thank and acknowledge my wife Nancy. Her support, patience, and understanding made this endeavor possible. I also need to acknowledge the many people who contributed to my work, I am indebted to all of you. I would like to thank John Pinkston, my adviser, mentor, and friend. To faculty members: Anupam Joshi, Tim Finin, Jacob Kogan, and Charles Nicholas. Your support and intellectual contributions were priceless. To Filip Perich, Sasikanth Avancha, Jim Parker, and Jamie Butler — much of the research presented here is the result of our collaborative efforts.

אתה חונן לאדם דאת ומלמד לאנוש בינה. חננו
מאתך דעה בינה והשכל. ברוך אתה ה׳ חונן הדאת.

TABLE OF CONTENTS

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Intrusion Detection State of the Art

"Intrusion Detection Systems have failed to provide value relative to their costs and will be obsolete by the year 2005" [39]. This indictment of *"intrusion detection state of the practice"* was handed down in June, 2003 by Gartner, Inc., a firm that specializes in research and analysis of the technology market place. The Gartner advisory states that the following issues contributed to their assessment that Intrusion Detection Systems (IDSs) are a failure:

i. False positives and false negatives.

ii. An increased burden on the Information Security organization by requiring full-time monitoring (24 hours a day, 365 days per year).

iii. A taxing incident response process.

iv. An inability to monitor traffic at transmission rates greater than 600 megabits per second.

The Gartner report concludes by recommending that enterprises redirect money earmarked for IDSs toward other barrier technologies such as network firewalls. Before addressing the Gartner report, some history and background on IDS research is in order.

Intrusion detection research has been ongoing for approximately 20 years. One of the earliest papers in this field of study is James Anderson's 1980 paper *Computer Security,*

Threat Monitoring, and Surveillance [2]. Anderson posited that computer systems must be actively monitored in order to mitigate threats against them. Seven years later, Dorothy Denning wrote *An Intrusion Detection Model* [27], providing a framework for IDSs. Denning held that evidence of malicious activity would be reflected in the audit records of the affected system.

Given the 20 plus years of research, one should think that IDSs are well advanced, detecting and blocking intruders as they attempt to cross the perimeters of our networks or detecting "insiders" as they attempt to abuse their system privileges. Unfortunately, as made clear by the Gartner recommendation, this is not the case. According to the Carnegie Mellon Software Engineering Institute's *State of the Practice of Intrusion Detection Technologies* [1], most commercial IDSs use a signature-based approach and do not provide a complete intrusion detection solution. The Carnegie Mellon report further states: *"despite substantial research and commercial investments, IDS technology is immature, and its effectiveness is limited"*. Similarly, McHugh [83] states that based upon the results of the 1998 and 1999 DARPA off-line intrusion detection evaluations [77], research systems, like their commercial counterparts, are very poor at detecting new attacks. McHugh continues, stating that *"the intrusion detection field is making little forward progress . . . none of the systems funded by DARPA have achieved major breakthroughs nor has an individual system or combination of systems approached the goals that DARPA has set for their IDS program.*

IDSs are categorized according to scope — network based or host based — and method – signature detectors or anomaly detectors. Anomaly detectors attempt to detect usage outside the bounds of established statistical norms. To do this, they create and maintain usage profiles that reflect the normal behavior of the users and processes on the system. Traditionally, anomaly detectors are comprised of a sensor that monitors aspects of system behavior and a decision process that determines if the sensed data is consistent with the predefined notion of acceptable behavior.

By contrast, signature detectors, also referred to as misuse detectors, filter usage against a given set of attack signatures. Typically, they examine TCP/IP network traffic using pattern

matching techniques to detect an actual or attempted intrusion and raise an alarm whenever the event matches a pre-defined rule or signature.

The difference between misuse and anomaly detectors is that misuse detectors define a model of "prohibited" behavior while anomaly detectors define a model of "permitted" behavior.

A network-based IDS, although run on a single host, is responsible for the entire network, or some network segment, while a host-based IDS is only responsible for the host on which it resides. A significant difference between host-based and network-based detectors is in the data set over which each type operates. A network-based detector is generally limited to the information contained in TCP/IP packets while a host-based detector potentially has access to all of the information available to the system on which it is running.

Based upon its criticisms, it appears as though the Gartner report is predicated upon the use of network-based IDSs (e.g.: "An inability to monitor traffic at transmission rates greater than 600 megabits per second") and the SANS operational model.

The SANS Institute [86, 87], an information security research, certification and education organization, recommends the use of the SNORT [98] network intrusion detection system. SANS also recommends that an enterprise's IDS be monitored 24 hours per day, 365 days per year. SNORT, because it is freely available, is the standard by which most other IDSs are evaluated. As pointed out by [24, 44] SNORT is the most widely used IDS because it is free and it generally performs as well as most commercial IDSs. Our own and other's research [66, 92] indicates that SNORT generates a significant number of false positives. In our conversations with computer security practitioners, the overwhelming majority state that network-based IDSs are the "only way" to conduct intrusion detection and that SNORT is the "only intrusion detector".

Given the shortcomings of relying solely upon network based IDSs, the Gartner evaluation of the IDS state of the art comes as no surprise. It fails, however, to attribute causality for the present condition of IDS technologies. The following list enumerates what many researchers have identified as the central issues leading to the success (or failure) of IDSs and

the research goals that we established in 2001 in response to those issues:

i. The elimination of the Single Point of Failure that is characteristic of most IDSs. A review of many academic IDSs [93] [96], as well as commercial IDSs [98], reveals that most, if not all, are one monolithic system. Those systems claiming to be distributed operate by placing sensors throughout the network and either send their alarms to a central management station or send their data to a central location for evaluation and subsequent alarm. An additional danger presented by the use of this monolithic single point of failure configuration is that the IDS may be subjected to denial of service attacks.

   - The two-phase, host-based, IDS presented in our work is only one of a distributed coalition of similar IDSs. In our proposed coalition each host-based IDS is responsible for itself and communicates using an ontology specification language with other host-based IDSs in its coalition.

ii. Data Fusion from Multiple Sensors. Bass [111] suggests using cluster analysis on input from distributed sensors as well as fusing data from heterogeneous devices. Kerschbaum et al. [63] suggest using sensors embedded in the operating system to detect attacks. Allen et al. [1] suggest integrating multiple sources and types of information.

   - Our work addresses the issue of using sensors embedded within an operating system. We record, at system call granularity, 119 low-level kernel attributes that represent process, network, and global system state. We use cluster analysis to discern between state that is representative of normal and abnormal behavior.

iii. Definition of an attack taxonomy from the systems' point of view. McHugh [83] recommends classifying attacks based upon the protocol layer and the particular protocol within the layer that is used as the vehicle for the attack. Allen et al. [1] criticize the characterization of an intrusion from the attacker's perspective, suggesting a classification scheme according to vulnerabilities as an alternative methodology. Commenting

on the Internet Engineering Task Force's emerging standard – the *Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition* (IDMEF)[21] and its ability to enable interoperability between heterogeneous IDS sensors, Kemmerer and Vigna [61] state that the IDMEF is a first step but additional effort is needed to provide a common ontology that allows IDS sensors to agree upon what they observe.

- Attack classification is a particularly salient point and needs to be underscored. A taxonomic classification implicitly defines the data model that an IDS operates over. In the case of an anomaly detector, it is a model of acceptable behavior and in the case of a misuse detector it is the model of intrusive behavior. Regardless of the detector type, meaningful taxonomic characteristics are core to the effective operation of an IDS. If they are limited to those elements that are directly observable by an IDS, then there is a greater likelihood that they can be accurately measured and evaluated. Of equal importance is a language that can effectively describe and communicate information about instances of the data model specified by the taxonomy.

  We have created a *target-centric* ontology, one that incorporates and subsumes a taxonomy. We have found that in applying ontologies to the problem of intrusion detection, the power and utility of the ontology is realized not by the simple representation of the attributes of the attack but rather because those instances may be "reasoned" over to derive additional knowledge.

iv. The development of an expressive IDS language. Doyle, et al. [31] maintain that some limitations with signature and statistical methods stem not from the methods themselves but rather from reliance on inadequately expressive languages for describing significant events and communicating information about them.

- "Language" is paramount to the effectiveness of the IDS because information regarding an attack or intrusion needs to be intelligibly conveyed, especially in

distributed environments, and acted upon. Moreover, the language must have constructs that support the notion of correlation and aggregation.

We use the DAML+OIL [54] ontology specification language as a combined event recognition, correlation, and communication language. DAML+OIL supports the correlation and linkage of data from diverse sources in a principled way. Using DAML+OIL, once data is linked, it may then be aggregated and used to determine facts that are not directly represented in any one source of data.

We have devised a method to map instances of non-conforming data (from our model of normal behavior), which are numeric values, to instances of our ontology, which are characterized as classes, properties and relationships. Our method employs a statistical analysis to perform the mapping.

v. A reduction of the high false alarm rate. Allen et al. [1] recommend the integration of data from multiple sources. The idea of using multiple sources is predicated upon the notion that during sophisticated attacks information from a single sensor is unlikely to detect suspicious activity.

- We have extended the traditional view of anomaly detection, which calls for a sensor and a decision process. Our decision process is two-stage. In addition to cluster analysis on the low-level kernel data, we perform an additional test by reasoning over instances of abnormal data. In addition to providing an orthogonal test, the reasoning stage, in conjunction with the *target-centric* ontology, correlates and aggregates seemingly disparate, but related events. Our experiments have demonstrated that our approach is able to detect multi-phased and distributed attacks.

vi. The use of real world test data in order to test IDSs adequately . The only corpus of test data available to researchers today is the data set that was synthetically produced at MIT's Lincoln Labs for use in the DARPA off-line Intrusion Detection Evaluation

[77]. The use of this data set has become a contentious issue and some researchers [80, 83] recommend against its use.

- During our testing we used traces (5 million +) of system calls that were collected from live systems. In other experiments, we have mirrored web servers and have replayed their transaction logs while simultaneously attacking the system.

Most, if not all, of the aforementioned central issues lie beyond the ability of network based misuse detectors, hence Gartner's assessment of the *"Intrusion Detection State of the Practice"*.

Gartner's recommendation that *enterprises redirect money earmarked for IDSs toward barrier technologies such as network firewalls* requires deeper thought and analysis. William Wulf, the President of the National Academy of Engineers, during Congressional testimony [121], has equated the use of barrier technologies to the "Maginot Line" [1], emphasizing that this model does not work. Recently, during his keynote address before the National Science Foundation's Cyber Trust Point meeting [122], he called into question both the soundness of current software engineering practices (stating that developers are producing software with little regard for security) and the feasibility of relying upon barrier devices to protect a network. The premise of Dr. Wulf's address was that each individual system should be responsible for its own security. We believe that assigning the responsibility for detecting malicious and aberrant behavior to a single device residing at the network gateway (viz.: Gartner's recommendation to use firewalling technologies) is nothing more then a continuation of Wulf's "virtual Maginot Line".

1.2 Thesis Statement

It is serendipitous that Dr. Wulf identified so closely with the thesis that we proposed over two years prior to his keynote address. Specifically, our thesis, motivated by the inade-

[1] The Maginot Line, named after Andre Maginot, the French Minister of War 1928 - 1932, was a series of defensive fortifications built by France along its border with Germany and Italy. It failed. In 1940, the German army bypassed the Maginot Line, entered France through a "neutral" third country and swiftly defeated her.

quacies of network based signature detection, states that host based anomaly detection using the right model will improve the current state of detection capabilities.

Like Denning, we hold that evidence of malicious activity is reflected in system level data. However, by analyzing data at a much lower level than audit data, we can profile the system's normal state and perform anomaly detection.

1.2.1 Two Phase Intrusion Detection Framework

Our intrusion detection model is two-phased. By instrumenting the Linux kernel and collecting low-level attributes in three separate data streams – process, system, and network – we created a model of the quiescent state of the system. We then used these attributes either "as is" or as components in the construction of a feature. For example, one feature is derived from sequences of system calls. We use a novel measure, *self-distance*, to characterize the streams of system calls. We then use the Kullback-Leibler metric to measure the relative entropy between the self distances of a known exemplar of a system call stream to an unknown sample that was produced by the same type of process.

During the first phase, low-level data is taken from the system under observation and compared to the model, testing for conformance. During the second phase nonconforming data is represented as an instance of the data model defined by our ontology, asserted into a knowledge base, and reasoned over using *First Order Logic*. The primary purpose of the second phase is to classify the anomalous data as belonging to a particular the type of attack or intrusion. Its secondary purpose is to perform an orthogonal test in order to reduce false positives. In our framework, once anomalous data has been classified, an alarm can be generated and the attack can be communicated to other IDSs as an instance of the ontology. The other IDSs can add this information to their knowledge bases, and if applicable, use it to deduce that a more comprehensive attack is taking place.

Our IDS methodology is built upon notions introduced by the semantic web, most notably the concept of machine understandable documents. These documents (data in our case) do not imply an artificial intelligence application, rather, they take advantage of a machine's

ability to solve a well defined problem by performing well defined operations on well defined data. By using an ontology representation language to define our data model, instances of abnormal and intrusive events are able to be reported and communicated to associated IDSs, which in turn can correlate, aggregate and evaluate them for the existence of distributed and multi-phased attacks. Figure 1.1 illustrates our distributed intrusion detection framework.

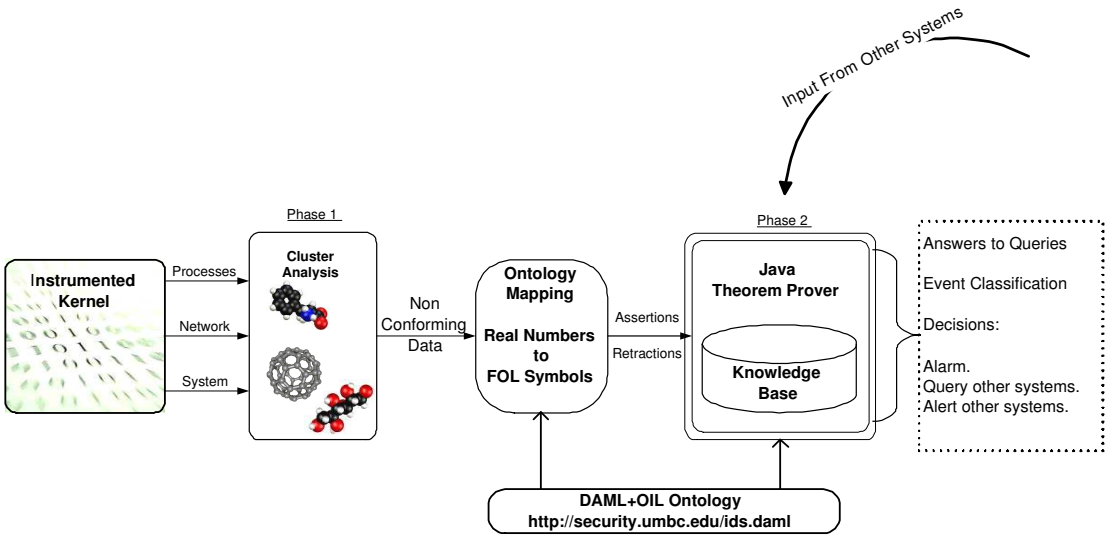


FIG. 1.1. Dual-Phase Intrusion Detection Methodology

Not long ago, the expense of collecting and processing low level kernel data was both computationally and cost prohibitive. Today, processor speed has dramatically increased while their costs have decreased and many computers are now configured as dual processor machines. This market dynamic makes our thesis practicable whereas it would not have been just a short time ago because the second processor may be dedicated to the handling of security related functions.

1.3 Dissertation Overview

This dissertation advances the field of intrusion detection research in several key areas. Its main thrust, as detailed in Chapters 2 - 7, is the use large data streams and semantic knowledge to differentiate between normal and anomalous behaviors occurring in the hosts

of distributed and interconnected systems. The difficult task of detecting and preventing intrusive behavior is exacerbated by the manner in which our computing environment has evolved and matured. In the 1950's IBM Chairman Thomas Watson opined that the world-wide demand for computers will at most be five. In the 1970's the concept of the personal computer was shunned by industry. In the 1980's and early 1990's, in the face of increasing personal computer usage, no one envisioned the Internet as we have it today. In spite of its benefits, our interconnectedness brings with it a plethora of security problems. These problems are difficult to mitigate because they were not anticipated during the evolutionary process. Consequently, our security solutions are retrofitted onto existing computer and network paradigms. We endeavor to prevent this dynamic from repeating in two emerging models — Mobile Ad Hoc Networks and Sensor Networks.

Mobile ad hoc and sensor networks are still in their nascent stage. Although ancillary to the main body of this work, Chapters 8 and 9 lay the foundation for their secure internetworking. In both types of networks, we build security and intrusion detection mechanisms into the communications protocols used by the mobile ad hoc devices and the sensors. We anticipate that, as mobile ad hoc and sensor networks mature, the will be amalgamated with the prevailing distributed and interconnected systems. We leave the task of defining ontologies that incorporate our secure communications protocols those researchers who will focus on that merger.

The following is a chapter by chapter synopsis of our work.

Chapter 1 is this introduction where we provide an overview of our work. We have stated the key issues facing researchers in the field of intrusion detection and we have presented industry's assessment of the current *intrusion detection state of the practice*.

Chapter 2 addresses a worst case scenario for intrusion detectors — *hidden processes*. A maliciously hidden processes is undetectable to a network-based IDS and might undermine a host-based IDS. In this chapter we show how to detect and mitigate their effects. We state methods that will ensure the integrity of the *system call table* and the *interrupt descriptor table*. We state how we extended our assurance process in order to instrument the Linux

kernel and capture the streams of low-level kernel data. We used this data to build our model and used subsequent data samples to test the system for conformance to the model.

Chapter 3 presents our novel *self distance* measurement that we used to characterize streams of system calls. Our measure quantifies the degree of intrinsic regularity within a sequence of system calls. We detail our experiments, comparing the performance of our novel measurement to those used in other research that also rely upon system calls to differentiate between normal and abnormal behavior.

Chapter 4 details the feature sets that we constructed from low-level kernel data. It presents our use of *Principal Component Analysis* to gain insight regarding the degree of information that individual features convey.

Chapter 5 details how we constructed the model of normal behavior. We have experimented with the *Mahalanobis Metric* [17] and *Euclidean Distance* to measure distances between the feature vectors. We also experimented with *Fuzzy* [70], *Principal Direction Divisive Partitioning* [10], and *K-Means* [36] clustering algorithms as well as the effects of *z-normalization* [42] on the data set.

Chapter 6 details our empirical analysis of over 4,000 types of attacks and intrusions. We determined the means of attack most frequently employed (i.e.: as manifested at and experienced by the victim), their most likely consequences, the system component (network protocol stack, process, or system) most often targeted, and the most frequently employed means of effecting an attack. We also identified the location from whence attacks most frequently originate. This study is the foundation of our *target-centric* ontology.

Chapter 7 details our *target-centric ontology* and presents the results of our experiments. We mapped the data that failed to conform to our model of normalcy as instances of the data model specified by our ontology. We classified them as being representative of a specific type of an attack or intrusion by asserting them into a knowledge base and reasoning over them using First Order Logic. Our experiments include correlating and aggregating data from disparate events, where we were able to infer that a more comprehensive attack has occurred.

Chapter 8 presents our intrusion detection mechanisms for mobile ad hoc networks. Our method relies upon packet snooping to detect aberrant behavior. Our extensions, which are applicable to several mobile ad hoc routing protocols, offer two response mechanisms: passive, which unilaterally determine if a node is intrusive; and active, which collaboratively determines if a node is intrusive. We have implemented our extensions using the GloMoSim simulator and have detailed their efficacy under a variety of operational conditions.

Chapter 9 details our intrusion detection and security protocol for wireless sensor networks. We state why security mechanisms that are designed for mobile ad hoc environments are inadequate or not appropriate for sensor networks. Our protocol facilitates the detection and elimination of network nodes displaying aberrant behavior. We implement our protocol using SensorSim and present our experimental results.

Chapter 10 offers the conclusion to this dissertation, provides a synopsis of our accomplishments, and lays the groundwork for future research.

Chapter 2

Securing and Instrumenting the Linux Kernel

According to many security practitioners, inherent operating system insecurities obviate any trust being placed in the data that they produce. Consequently, they adhere to the notion that network-based IDSs are the only viable option. Network-based IDSs, unfortunately, are only able to operate over the information contained in TCP/IP packets, consequently they are not able to detect as many types of attacks and intrusions as their host based counterparts [120].

This chapter introduces a novel "worse case" class of operating system compromise — the *hidden process* — and presents ways to detect and mitigate their effects. Moreover, hidden processes are a class of intrusions that are undetectable by network-based IDSs. We present two methods to detect hidden processes in the Linux environment. The first method requires modification to the operating system kernel. The other method employees a loadable kernel module to add functionality to the operating system and we have extended this method to instrument the kernel in order to provide low-level data to our host-based IDS.

2.1 Introduction and Background

The most significant and far reaching consequence of a computer attack is the intruder gaining *root* (administrative) access to the targeted computer [28]. This type of access opens the system to continued misuse and exploitation. According to the CERT/CC advisories, root access is the most frequent consequence of an attack. Criscuolo [20] and Toxin [112] have

observed that once an attacker has gained unauthorized administrative access to a computer, he will often install a *Rootkit* as a means of continued access to the targeted system.

Rootkits, which first appeared in 1993, are a collection of tools and *trojaned* replacements of core system utilities. These core utilities include binaries (such as *top, ps, ls, du* and *netstat*) which are used to manage a system and ensure that it is operating properly. Rootkits replace these core system utilities with modified versions in order to hide the presence of the intruder and his tools. A network-based IDS is unable to detect the presence of a "trojaned" core system binary, whereas, as we later demonstrate, our proposed host-based system does.

We have tried to anticipate future manifestations of attack and intrusions, consequently we developed a method to hide a process. We posited that rather than employing trojaned binaries, which interface between the administrator and the operating system, an attacker might directly hide the tool from the operating system. To explore the feasibility of hiding a process on a system without making detectable modifications to the system call table or system utilities, we have written our own program that hides a process on the Windows (NT, 2000 and XP) operating system and we modified an existing program [22] to hide processes on the Linux operating system.

2.2 Hidden Processes

In the following subsections we detail the steps that our program takes to hide processes on both the Windows (NT, 2000 and XP) and Linux operating systems. Due to Window's prevalence, its subversion affords the greatest profitability to the attacker. Linux is also an excellent target since it is widely deployed and is used to provide numerous network infrastructure services.

2.2.1 Windows

When queried about the processes currently running on the system, the Windows operating system presents a list of active processes that is obtained by traversing a doubly linked list referenced in the *EPROCESS* structure (process descriptor) of each process. Specifically,

a process' EPROCESS structure contains a *LIST-ENTRY* structure that has members *FLINK* and *BLINK*. FLINK and BLINK are pointers to the processes that are forward and behind the current process descriptor. Figure 2.1 illustrates the EPROCESS block of the Windows kernel.

To hide a process in Windows we located the Kernel's Processor Control Block (KPRCB), which is unique and located at the address `0xffdff120`. We followed the *CurrentThread* pointer to the *ETHREAD* block. From the ETHREAD block we followed the pointer from the *KTHREAD* structure to the EPROCESS block of the current process. We then traversed the doubly linked list of EPROCESS blocks until we located the process that we wish to hide. Once located, we change the FLINK and BLINK pointer values of the forward and rearward EPROCESS blocks to point around the process to be hidden. Referring to Figure 2.1, the BLINK contained in the forward EPROCESS block is set to the value of the BLINK contained in the EPROCESS block of the process to hide and the FLINK of the rearward process is set to the value of the FLINK contained in the EPROCESS block of the process that we are hiding.

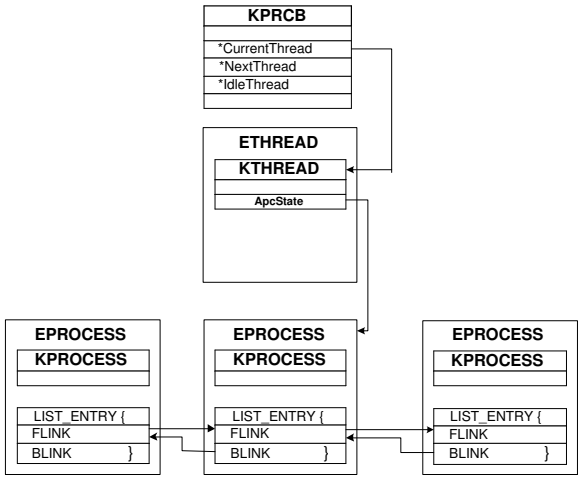


FIG. 2.1. Windows Kernel Data Structures

Intuitively, one would think that hiding a process by removing its process descriptor from the doubly linked list of EPROCESS blocks would prevent the process from being

allocated a time slot in which to execute. We have observed that this is not the case. The Windows scheduling algorithm is highly complex, is done at thread granularity, is priority based, and is preemptive. Accordingly, a thread is scheduled to run for a *quantum* of time, which is the length of time before Windows interrupts the thread to check for other threads of the same or higher priority or to reduce the priority level of the current thread. A process may have multiple threads of execution and each thread is represented by an ETHREAD structure that contains pointers to its siblings. Although we have been unable to precisely determine why "un-linking" a process' EPROCESS block from the doubly linked list does not adversely affect execution of the process, we strongly suspect that the Windows scheduler references those threads from some other linked list, not the EPROCESS block. It should be noted that the data structures employed by the various Windows operating systems are not publicly documented.

We have implemented the task of hiding a process by writing a device driver (.sys), which is similar to a LKM in Linux, and by writing a Dynamically Loadable Library (DLL) that provides an interface to the device driver. To hide a process, we load the device driver and invoke it by passing the name and the unique *PID* (Process Identifier) of the process to hide. This procedure hides the process and unloads the device driver. We minimize the chance that an intrusion detection system would notice our activity because it takes only a few milliseconds to hide a process and unload the device driver . If this process were to be executed as a system booted, and before any IDS software is executed, it would be virtually impossible to detect.

As detailed above, we did not make any changes to the system call table. We used Microsoft's *Windbg* in an attempt to locate the hidden process by running the Windbg as master on one machine and slaving the machine with the hidden process to it via a serial connection. Windbg could not detect the presence of the hidden process.

2.2.2 Linux

We located two attack tools that reportedly mask Linux processes from the operating system, specifically *Adore* [103] and *Phantasmagoria* [22]. The *Adore* tool would not run at all, and we needed to make run-time modifications to the Linux *Scheduler* in order for *Phantasmagoria* to run. With one exception, the procedure for hiding a process on the Linux operating system is similar to the one used for Windows.

The Linux operating system's process descriptor is a data structure of type *task_struct* (the layout of the data structure is included as Appendix C). Unlike Windows, in Linux there is a strong one to one correspondence between a process and its process descriptor. In Linux the individual process descriptors contain pointers that make up the *run queue* of runnable processes.

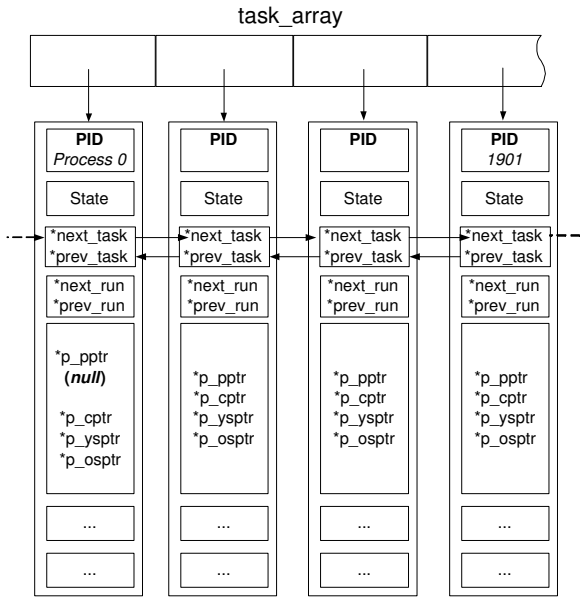


FIG. 2.2. Linux Process Descriptors

As depicted in Figure 2.2, the *task_array* contains a pointer to each process descriptor. The first entry in the task_array is to *process 0*, which does not have a parent process and is synonymous with *init_task* or *swapper*. Process 0 is the first process started by the Linux kernel and is at the head of the doubly linked list that is referenced by *next_task* and *prev_task*

pointers of each process descriptor. The run queue, a data structure that points to those processes whose state is *runnable* is also maintained via a linked list formed by the *next_run* and *previous_run* pointers of the process descriptor. Finally, each process descriptor contains pointers to its parent, sibling, and child processes.

To hide a process, we unlinked the process descriptor from the *task_array*, removed any referencing links from corresponding parent, sibling and child process descriptors, and just as in Windows we reset the *next_task* and *prev_task* links of any referencing process descriptors to point around the subject process descriptor. To maintain a reference to the hidden process we set the parent pointer (*p_pptr*) of Process 0's process descriptor to point to the hidden process. Moreover, we were able use *p_pptr* of Process 0 as the root of a list of hidden processes in the event we wanted to hide additional processes.

Linux, unlike Windows, relies exclusively upon the pointers contained within the process descriptor for scheduling CPU time to a process. Under normal execution, the Linux scheduler, at the start of each *epoch*, traverses the doubly linked list of process descriptors, assigning each process *quantums* or CPU time slices. We found that when we hid the process we also needed to perform a run-time modification to the scheduler so that it would also traverse the list pointed to by Process 0's *p_pptr* and the hidden process to the run queue. Figure 2.3 illustrates the concept of removing PID 1901 from the normal list and hiding it by making it the parent of Process 0.

2.3 Detecting Hidden Processes

In the following section we present three methods for detecting hidden processes. The one method for Windows employs an existing tool. Of the two methods used for Linux, one requires modification to the operating system and the other adds operating system functionality via a loadable kernel module (LKM). As will be described, we extended the LKM to instrument the Linux kernel and export data streams containing low-level kernel attributes. We also use the LKM to offer reasonable assurances regarding the integrity of the system call table and the interrupt descriptor table. Intruders have historically modified the system call

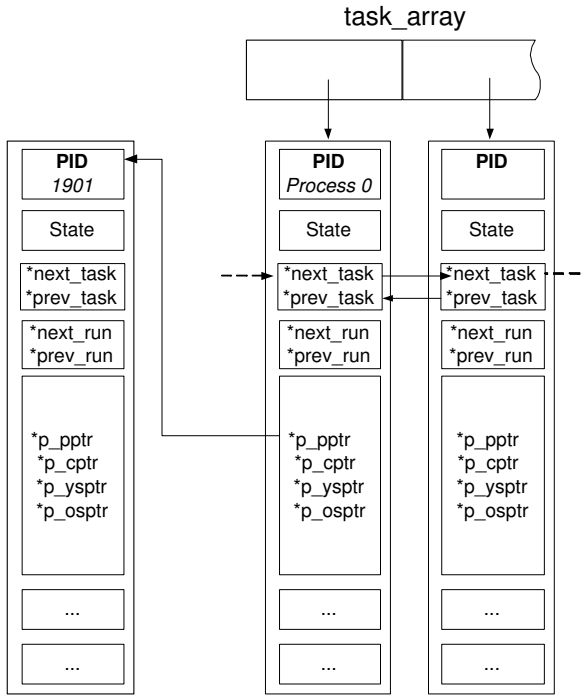


FIG. 2.3. A Hidden Linux Process Descriptor

table giving rise to the notion held by many security practitioners that there are no operating systems that can be trusted.

2.3.1 Detecting Hidden Processes in Windows

Hunt and Brubacher introduce *Detours* [51], a library for intercepting arbitrary Win32 binary functions, as a means of instrumenting and extending application and operating system functionality in the Windows environment. Detours replaces the first few instructions of a *target function* with an unconditional jump to a user provided *detour function*. The instructions from the target (original) function are preserved in a *trampoline function*. Detours are inserted at execution time, where the code of the target function is modified in memory – not on disk. Figure 2.4 shows the temporal ordering of the detour, trampoline and target functions.

In Windows, context switching is carried out by a call to *SwapContext* which is a func-

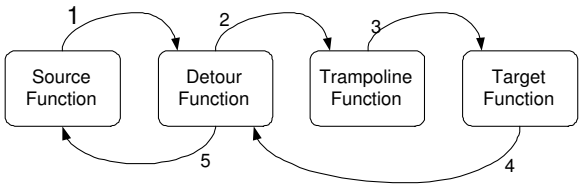


FIG. 2.4. Temporal Ordering of a Detour Function

tion in *ntoskrnl.exe*. The first seven bytes of SwapContext are:

```
OR CL, CL

MOV byte ptr ES:[ESI+2D], 02
```

When SwapContext is called, the value contained in the *EDI* register is a pointer to the next thread that is to be swapped in and the value contained in the *ESI* register is a pointer to the current thread, which is to be swapped out. In our detection method, we replace the seven bytes of SwapContext with a five byte unconditional jump to our Detour Function. The Detour Function verifies that the KTHREAD of the thread to be swapped in (referenced by the EDI register) points to an EPROCESS block that is appropriately linked to the doubly linked list of EPROCESS blocks.

2.3.2 Detecting Hidden Processes in Linux

We have devised two methods that will detect hidden process in the Linux environment. The first method is straightforward and requires modification to the kernel's scheduler function. Accordingly, we are able to detect hidden processes by modifying the scheduler so that when a process' state is set to *Task_Running* and it is placed in the run queue by setting the *next_run* and *prev_run* pointers of its process descriptor, the doubly linked list of *next_task* and *prev_task* pointers is traversed to ensure that they are correctly linked. This takes place once per *epoch* and the system overhead is n^2 traversals where n is the number of active processes.

Our second method, although more complex, does not require modification to the kernel. It also serves as a segue into our method of instrumenting the Linux kernel. The following treatment of our method and LKM assumes the Intel i686 architecture.

The interrupt descriptor table (IDT) is a table of interrupt vectors where each vector points to an interrupt handler. The IDT is comprised of 256 8-byte pointers and resides in the first 2k of addressable memory. With one exception, all of the interrupts are hardware interrupts. The exception is interrupt `x80` (decimal 128), which is a software interrupt that calls the system call handler. On the i686 architecture, a system call number is placed in the `EAX` register and interrupt `x80` is invoked. The address of the system call handler is retrieved from the IDT, and control is passed to that address.

Kernel system calls are the lowest level of system functionality. System calls provide read and write access to file systems, access to network connections, time of day functions, and invoke process execution. In order to hide files, directories, processes, or network connections without modifying any system binaries, an attacker will need to modify the function addresses in the kernel system call table (i.e.: *sys_call_table*) so that they point to spurious functions.

In order to ensure the integrity of the system, we maintain a copy of the addresses of the handler offsets from the interrupt vector and the kernel function calls. These copies are periodically checked to ensure that the IDT and the system call table have not been modified.

In order to mitigate the effects of hidden processes, we intercept all calls to the system call handler. Prior to copying the addresses of the interrupt handlers contained in the IDT, we save the address of the handler that is stored at address `x80` and replace it with the address of our own kernel function. Consequently, whenever a process makes a kernel system call, we intercept that process and traverse the doubly linked list of file descriptors to ensure that the process is properly linked and not “hidden”.

The computational overhead imposed by the LKM method increases linearly with the number of active processes.

2.4 Instrumenting the Linux Kernel

We have adapted our LKM to serve as a sensor embedded in the operating system. We use it to monitor system state at the process, network, and "global" system levels.

When loaded, our LKM replaces the address of the call handler with the address of our LKM. (The Linux source file for the call handler functions is `entry.S`). Whenever a process makes a system call, instead of jumping to the call handler, control is passed to our LKM, where we save the process' state information. We now have immediate access to that process' process descriptor (the Linux *task_struct*). All of the available information about a process can be learned by accessing the additional data structures pointed to by the pointers in the process descriptor. This information includes: the process' memory regions, shared libraries, register values, memory faults, and page faults. Our LKM extracts selected data and exports it. Once the data has been exported, the process' register vales are restored, the stack pointer is set to the appropriate value, and we then pass control to the system call handler. When the system call handler finishes, control is returned directly to the calling process.

The low-level kernel data can now be used to build a model of normal system state. Once we have constructed the model, we compared subsequent instances of the low-level kernel to the model to test for anomalous behavior that is indicative of attacks and intrusions.

2.5 Chapter Conclusions

In this chapter we have demonstrated how to detect hidden processes, as well as changes to the system call and interrupt descriptor tables (one of the most common methods of subverting a system). These safeguards afford a host-based intrusion detection system some assurances that the host has not been compromised.

Our assurance function operates within the operating system and has been extended to output streams of low-level kernel data that are specific to the processes that are running on the system.

To offer an additional level of assurance, Arbaugh, et. al [3] have specified the *AEGIS*

bootstrap protocol which systematically loads the BIOS, the operating system, and all device drivers from a trusted source. Although *AEGIS* is only concerned with loading trusted software, not testing for modification while it is in process, our procedure ensures that binaries that are resident in memory have not been modified and remain consistent with the image that was loaded. *AEGIS* in concert with our methods, which are able to detect changes to loaded binaries, affords a reasonable degree of assurance in the integrity of an operating system.

Chapter 3

Measuring System Calls in Terms of Self-Distance

Considerable research effort has been dedicated to using system call sequences to detect anomalous behavior. As will be described, Stephanie Forrest et al. have shown that characterizing sequences of system calls holds promise. There are also several analyses and critiques of Forrest's work and of the concept of using system calls to characterize system behavior in general. This chapter synopsizes that research and its analyses and presents our novel system call measurement. Our novel measurement quantifies the intrinsic regularity and density of a system call that is observed in contiguous system calls. We refer to it as the system call's *self-distance*.

3.1 Introduction and Background

Forrest et al. [37, 116] have developed an intrusion detection methodology modeled after the way the human immune system distinguishes between harmful and benign antibodies. Immunologists describe the immune system's dilemma as one of distinguishing "self" from dangerous "other" (or "nonself") in order to eliminate the dangerous "other". In their immunological intrusion detection model, traces of system calls are used to distinguish between "self" and "other" at the system level.

They use a collection of system call traces from privileged processes to define an empirical model of the program's normal behavior – "self". They then take subsequent traces of system calls and compare those traces to the empirical model to determine if the subsequent

trace conforms to the model. If any portion of the subsequent trace fails to conform to the model it is declared to be "other" and therefore intrusive.

Their empirical model is defined in terms of short *n-grams* of system calls. To classify a trace, a fixed-size window of size *w* (here $w = 6$) is slid over the trace. As the window slides across the sequence, and for each system call thereafter, the preceding system call is recorded at different positions within the window, numbered from 0 to $w - 1$. This results in a model that is comprised of a lexicon of all possible sequential sequences of size *w*. To test whether a process is normal or intrusive, system call sequences (*n-grams* of size 6) from the process under observation are compared to the empirical model that defines the program's profile.

This method was experimentally implemented as *stide* (sequence time-delay embedding) and *t-stide* (threshold sequence time-delay embedding). The difference between *stide* and *t-stide* is that *t-stide* determines the relative frequency of each *n-gram* in the sequence. If the frequency is under a user specified threshold (typically $.001$), it is declared to be "rare" and is not included in the process' profile.

In [101], Forrest et al. extended their previous work, calling it *pH*, short for process homeostasis. Homeostasis is a term for the biological process of maintaining a normal internal environment. *pH* monitors every executing process on a computer at the system-call level and responds to anomalies by either delaying the process by some amount of time *t* or by terminating the process.

The *pH* prototype was implemented as a patch for the Linux kernel, where the modified kernel is capable of monitoring every executed system call. The training of pH was conducted in real-time where a process' profile is established after some set number of system calls, wherein the system "believes" it has been adequately trained. In testing the prototype they found that some processes, such as *X-server*, can be perturbed by normal user actions, consequently freezing the system. They also found that processes that make a large number of system calls in a short period of time prematurely acquire their "normal" profile.

Tan et al. [107–110] have extensively analyzed and tested *stide* and *t-stide*, hereafter collectively referred to as *stide*. They found that *stide* was blind to *foreign sequences*, which

they define as two consecutive sequences of size w that are found in the normal profile, such that if the window size were expanded to $2w$ the aggregated sequence would not be included in the normal profile. They refer to this phenomena as a "blind spot" and have demonstrated that the size of the blind spot increases with the size of w. Their analysis indicates that attacks can be hidden within the blind spot. They also show that attacks can be aggregated so that a lesser attack masks a more egregious attack.

Wagner and Soto [115] have constructed a theoretical framework for a mimicry attack. They define a mimicry attack as one whose sequence of system calls mimics a normal sequence of system calls. They implemented their framework by first building a normal profile of system call sequences for the `wuftpd` server (w = 6). They then crafted a "theoretical" buffer overflow attack comprised of sequences of system calls that matched the normal profile. This theoretical framework could not be implemented, however. Specifically, they could not code a buffer overflow attack and have it compile to the requisite sequence of system calls needed to execute their mimicry attack.

Eskin et al. [33] have extended Forrest et al.'s work by incorporating dynamic window sizes into the process. The premise of their extension is that there exists intrinsic regularity in the sequences of system calls produced by a running process. They determine the regularity of a sequence of system calls by measuring the conditional entropy under varying window sizes. They then choose the window size wherein the sequence of system calls exhibits the least amount of entropy.

3.2 Defining Self Distance

Eskin et al. point out that the greater the regularity within a sequence of system calls, the better the predictive value of the sequence when compared to subsequent sequences. Our *self-distance* metric takes advantage of both the regularity and the density of the specific system call types with a sequence of system calls.

We have constructed a feature that uses the Kullback-Leibler dissimilarity measure between the probability distributions of the *self distances* between a baseline exemplar and

those of an unknown sample. Both the baseline exemplar and the sample are from a process of the same type (e.g.: sendmail, apache, login, etc). We weight Kullback-Leibler dissimilarity measure with the information gain value *I* of the intrinsic regularity of a sequence of system calls of the baseline. The following explanation provides the theory behind our metric.

Information Gain. Information gain is a measure of the intrinsic periodicity and density of a particular system call within a sequence of system calls. Consider the following:

Suppose we have a hypothetical system call, a, and a hypothetical sequence of unit length. Furthermore suppose that we have four (4) occurrences of system call a, where a is the first and last system call in the sequence and the intervening occurrences of a are evenly spaced in the sequence. Figure 3.1 shows our hypothetical system call a, as having *self-distances* of $\frac{1}{3}$, $\frac{1}{3}$, and $\frac{1}{3}$ within the sequence. We use Equation 3.1 to measure the degree of regularity conferred by the *self-distances* of system call a to the sequence.

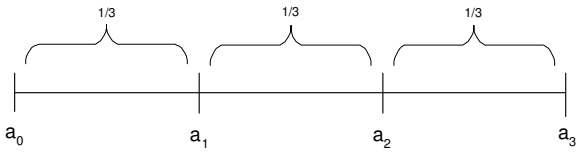


FIG. 3.1. Uniformly Distributed Occurrences of System Call a Within the Sequence

$$I = -\sum x_i log x_i \tag{3.1}$$

where x is a unique system call number and i is number of intervening system calls between x and the previous occurrence of x in a stream of system calls

Therefore, the information gain, I, of system call a as depicted in Figure 3.1 is:

$$I_a = -(\tfrac{1}{3}\, log\tfrac{1}{3} \;+\; \tfrac{1}{3}\, log\tfrac{1}{3} \;+\; \tfrac{1}{3}\, log\tfrac{1}{3}) = 3(-\tfrac{1}{3}\, log\tfrac{1}{3}) \;= -log\tfrac{1}{3} \;=\; log3 \;\approx\; .477$$

Now consider the case of the hypothetical system call b, illustrated in Figure 3.2. We have four (4) occurrences of b of varied spacing within the sequence.

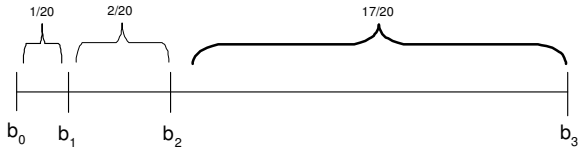


FIG. 3.2. Varied Occurrences of System Call b Within the Sequence

The application of Equation 3.1 to the sequence illustrated in Figure 3.2 results in the information gain of:

$$I_b = -(\tfrac{1}{20}\, log \tfrac{1}{20} \;+\; \tfrac{2}{20}\, log \tfrac{2}{20} \;+\; \tfrac{17}{20}\, log \tfrac{17}{20}) = -(-.665 \;-\; .1 \;-.0599) \;\approx\; .2249$$

As illustrated, the more regular the periodicity and the greater density of the system call within the sequence, the greater the system call's information gain.

Definitions:

C is the set (alphabet) of all system calls.

CallTrace. By $CallTrace$ we mean a map from $\{1,2, \ldots, T\}$ into C, where T is the temporally greatest system call and where each c in $CallTrace$ originated from the same Process ID (PID).

CallProfile. By $CallProfile$ we mean the inverse images of a $CallTrace$ for each $c \in C$.

SelfDistance. By *SelfDistance* we mean the sequence of differences d (defined by Equation 3.2) of consecutive elements in a $CallProfile\ x$, for $x \in C$. (i.e. if $CallProfile(x) = k_1, k_2, \ldots, k_M$ (where M is the number of occurrences of call x) then the self distance set for

call x is d_j where d_j is defined by Equation 3.2.

$$d_j = d(k_j) \ - \ d(k_{j-1}); \text{ for } j = 2, ..., M \tag{3.2}$$

I is the information gain derived from the periodicity and density of a system call c in a $CallTrace$ and is formally specified in Equation 3.3.

$$I(c) = - \sum_{d \in SelfDistance_c} \frac{d}{N} log \frac{d}{N}; \\ where \ N = \mid CallTrace \mid \tag{3.3}$$

Collection. By $Collection$ we mean a set of $CallTrace$s where each $CallTrace$ in the $Collection$ originated from a process of the same type (i.e.: they all came from *emacs* or *sshd*, etc).

Constructing the Baseline. We construct a baseline for a particular type of process by taking a $Collection$ and generating and aggregating all of the corresponding *CallProfile*s and *SelfDistance*s for each $CallTrace \in Collection$. We have a function $f(c, d)$ that produces the number of calls c at distance d for each system call in the $Collection$. Likewise, the information gain I for each system call c is calculated over the entire $Collection$.

Constructing a Sample S for Comparison to the Baseline B. A sample is constructed from another $\overline{CallTrace}$, where $\overline{CallTrace}$ is a subsequent CallTrace derived from the same type of process (i.e.: *sendmail, named, login, etc.*) as was the baseline. Likewise, the $\overline{CallProfile}$, $\overline{SelfDistance}$ and the distribution $\overline{f(c, d)}$ are also calculated.

Using the Kullback-Leibler Dissimilarity Metric to Compare Distributions S and B. The *Kullback-Leibler* dissimilarity metric is a measure of the relative entropy between two

discrete distributions that have probability functions B_x and S_x, respectively. It is defined by Equation 3.4:

$$\mathcal{D}(B_x \mid\mid S_x) \; := \; \sum_x S_x \, log \, \frac{S_x}{B_x} \tag{3.4}$$

and it has the following properties:

i. $\mathcal{D}(B_x \mid\mid S_x) \; \neq \mathcal{D}(S_x \mid\mid B_x)$; it is asymmetric

ii. $\mathcal{D}(B_x \mid\mid S_x) \; \geq 0$.

iii. $\mathcal{D}(B_x \mid\mid S_x) = \infty \; \forall \; (\; B_i \; = 0 \; \wedge \; S_i \; \neq \; 0)$

iv. $\mathcal{D}(B_x \mid\mid S_x) = 0 \; \forall \; (\; B_i \neq 0 \; \wedge \; S_i \; = \; 0)$

Equation 3.5 shows how we use the Kullback-Leibler dissimilarity measure to calculate the self distance metric of the Sample, S, in respect to the Baseline, B.

$$\mathcal{SD}_{BS} \; = \; \sum_c I_c \left(\sum_d \; S_{cd} \, log \, \frac{S_{cd}}{B_{cd}} \right) \tag{3.5}$$

When comparing the sample S to the baseline B with the Kullback-Leibler dissimilarity measure there are two potential conditions that result in a score of infinity. If this occurs, the following heuristics are applied:

$$\begin{aligned} -(S_{cd} \; lg \; S_{cd}) \quad &: \quad d \in S_{cd} \wedge d \notin B_{cd} \\ 40 \quad &: \quad c \in S_{cd} \wedge c \notin B_{cd} \end{aligned}$$

3.3 Experimental Design

To test the merits of our *self-distance* metric, we acquired $Collections$ of $CallTraces$ produced by the `lpr, named, xlock, login, ps, inetd, ftpd`, and `sendmail` processes under normal operation. 80% of each process' $Collection$ was used to create a baseline exemplar for that process. The remaining 20% of the process' $Collection$ was used to create "unknown" $CallTraces$ of that process. We also produced our own data by creating

a $Collection$ from Apache Server version 1.3.27, which mirrored the *International Federation for Information Processing* and *Music Machines* Web sites. Traffic was generated by replaying the Web sites' transaction logs against the web server.

To establish a baseline dissimilarity score, we applied Equations 3.2 and 3.4 to each known $CallTrace \in Collection$, and recorded the maximum score(Max_{Score}). We then constructed "unknown" samples, S, as previously defined, and applied Equation 3.4 to measure the dissimilarity between the baseline and each of the unknown samples. If the unknown sample's score was $> Max_{Score}$, the sample was declared to be anomalous and counted as a false positive.

We also acquired $Collections$ of $CallTraces$ of those same processes, but while they were under attack. The $CallTraces$ were processed as previously described and we created "unknown" samples corresponding to each $CallTrace \in Collection$. Using Equation 3.4, we compared the unknown sample to the baseline, producing a dissimilarity score. If the resulting score was greater than Max_{Score}, that $CallTrace$ was considered to be a true positive and represented an attack. If the score was less than or equal to Max_{Score}, then the result was counted as a false negative.

3.3.1 Data Sets

We used the *stide*, *t-stide*, and *pH* data sets from the University of New Mexico. These data sets consist of streams of system calls from processes under normal operation and those same processes while under attack. To generate attack $CallTraces$ for our Apache Server data set, we attacked it with with a buffer overflow attack and a resource consumption attack, collecting the $CallTraces$ during the attacks. The following provides an overview of each process and the attacks that were used against them:

A. `sendmail`. Sendmail is a Unix-based implementation of the *Simple Mail Transfer Protocol* (SMTP) used for transmitting and delivering e-mail. When a sendmail server receives e-mail, it attempts to deliver the mail to the intended recipient. If it cannot deliver the message, it queues it for later delivery.

(a) `sscp`. The sunsendmailcp (sscp) attack script uses a special command line option to cause sendmail to append an email message to a file. By using this script on a file such as /.rhosts, a local user may obtain root access.

(b) `decode`. In older sendmail installations, the alias database contains an entry called "decode", which resolves to the `uudecode` binary, a Unix program that converts a binary file that was encoded into ASCII characters for transmission back into its original form and name. `uudecode` respects absolute filenames, so if a file "bar.uu" says that the original file is "/home/foo/.rhosts" then when uudecode is given "bar.uu", it will attempt to create foo's .rhosts file. Sendmail will generally run uudecode as a semi-privileged process so that email sent to decode cannot overwrite files on the system. However, if the target file is world-writable, the decode alias entry allows these files to be modified by a remote user.

(c) `loops`. This attack consists of a local forwarding loop. It occurs in sendmail when a set of ∼./forward file forms a logical circle.

(d) `syslogd`. The syslogd attack uses the syslog interface to overflow a buffer in sendmail. A message is sent to the victim machine, causing it to log a very long, specially created error message. The log entry overflows a buffer in sendmail, replacing part of sendmail's running image with the attacker's machine code.

B. `lpr`. The lpr utility adds a job to the print queue by copying the specified file into its spooling directory. Strictly speaking, the file is only added to the print queue, the `lpd` print daemon handles the task of sending the file to the actual print device.

(a) `lprcp`. The `lprcp` attack uses `lpr` to replace the contents of an arbitrary file. This attack exploits the fact that older versions of `lpr` use only 1,000 different names for printer queue files, and do not remove the old queue files before reusing them. The attack produces 1,001 unique PIDS which results in 1,001 $CallTraces$. In the first trace, `lpr` places a symbolic link to the victim file in

the queue. The middle traces advance `lpr`'s counter, until on the last trace, the victim file can be overwritten with the attacker's own file.

C. `named`. The `named` daemon is the name server and routing daemon. Specifically, it is a Domain Name System (DNS) server, which is part of the BIND distribution.

   (a) `Inverse Query Buffer Overrun`. BIND 4.9 releases prior to BIND 4.9.7 and BIND 8 releases prior to 8.1.2 do not properly perform a bounds check when calling `memcpy()` when responding to an inverse query request. An improperly or maliciously formatted inverse query on a TCP stream can crash the server or it may allow an attacker to gain root privileges.

D. `wuftpd`. Ftpd is the Internet File Transfer Protocol server process. The server uses the TCP protocol and typically listens to ports 20 and 21 (control and data). `wuftpd` is a replacement ftp daemon for Unix systems..

   (a) `misconfiguration vulnerability`. The wuftpd binary was mis-configured when it was compiled, consequently allowing users SITE EXEC (execute a program) access to /bin, consequently permitting root privileges.

E. `login`. The `login` utility is used when signing onto a system. It can also be used to switch from one user to another at any time.

   (a) `trojan login`. The trojaned `login`, which allows an intruder to login through a "backdoor", has been substituted for the legitimate binary. This allows the attacker unfettered access to the system.

F. `ps`. The `ps` utility provides a snapshot of the current processes running on a system. It uses the `/proc` file system to acquire information regarding each process.

   (a) `trojan ps`. The trojaned `ps` has been substituted for the legitimate binary. This trojaned binary hides the attacker's activities from the system administrator.

G. `inetd`. The `inetd` program is started as a foreground process, initiates a daemon process to run in the background, and then exits. The daemon process initiates child processes that perform a fixed set of initialization steps before executing some other program.

   (a) `DoS`. This denial-of-service attack consumes network connection resources. The intrusive $CallTrace$ includes a startup process, a daemon process, and several child processes. Only the daemon process is expected to show any deviation from normal behavior.

H. `xlock`. The `xlock` program allows the user to "lock" their X terminal.

   (a) `buffer overflow`. A buffer overflow condition exists in some implementations of `xlock`. It is possible to attain unauthorized access by calling a vulnerable version of `xlock` that has the setuid or setgid bits set.

I. `Apache Server`. The Apache HTTP Server is an open source HTTP (Web) server for the Unix and Windows operating systems.

   (a) `chunking vulnerability`. The HTTP 1.1 specification allows for "chunked" encoding, where a message is transfered in a series of chunks. Certain versions of the Apache Server are vulnerable to a chunking vulnerability that causes a buffer overflow in a `memcpy()` function when re-assembling those chunks.

   (b) `DoS`. Memory leaks occur when memory is allocated and never released. Certain versions of Apache Server are vulnerable to remotely induced memory leaks. This remotely induced memory leak in Apache Servers constitutes a Denial of Service attack because it consumes all of the computer's memory and causes it to freeze.

Table 3.1 details each $Collection$. The table contains the number of traces, the number of unique PIDS, and the number of system calls collected per process and per attack. Table 3.2 presents the similar details regarding our own self generated $Collection$.

Data Set	Normal Data				Intrusion Data			
	Program Name	No. of Traces	No. of PIDS	No. of Syscalls	Attack Name	No. of Traces	No. of PIDS	No. of Syscalls
1	UNM sendmail	1	147	1,571,583	smcp decode loops	3 2 5	3 10 10	1,119 3,067 2,569
2					syslogd	4		6,504
2	UNM lpr 1	1	1,234	553,336	lprcp	1	1,001	164,232
3	MIT lpr	1	2,766	467,464	lprcp	1	1,001	165,248
4	UNM lpr 2		10	2,938	lprcp	1	1,001	164,231
5	named	1	28	9,230,572	buf over #1 buf over #2	3 2	3 2	969 831
6	wuftpd	2	7	8,603	mis. conf.	1	5	1,363
7	login	1	24	8,906	trojan #1 trojan #2	1 1	5 8	2,054 2,083
8	ps	1	19	12,307	trojan #1 trojan #2	1 1	11 15	2,463 4,505
9	inetd	3	541		DoS	31	8371	
10	xlock (synthetic)	71	71	339,177	buf over #1 buf over #2	1 1	1 1	489 460

Table 3.1. University of New Mexico Data Sets

Data Set	Normal Data				Intrusion Data			
	Program Name	No. of Traces	No. of PIDS	No. of Syscalls	Attack Name	No. of Traces	No. of PIDS	No. of Syscalls
11	Apache Server	10	10	102,920	chunking DoS	10 10	10 10	1,320 860

Table 3.2. Our Generated Apache Server Data Set

3.3.2 Results and Discussion

Our *self distance* metric correctly classified all of the normal $CallTraces$. When processing the intrusive $CallTraces$ we failed to correctly classify all of the `decode` and `loops` attacks that were made against `sendmail`. Although we caught some of the `decode` and `loops` attacks, we had 80% and 10% false negatives respectively.

As we stated in the discussion of `sendmail` and the attacks made against it, the `decode` attack overwrites a protected file. Because we are only examining system calls we could not differentiate between a write to a non-protected and a write to a protected file. Our results are detailed in Table 3.3.

Normal Data			Attack Data		
Program Name	$Score_{Max}$	% False Positives	Attack Name	No. Attacks in $CallTraces$	% False Negatives
UNM Sendmail	.019385	0	smcp decode loops syslogd	3 10 10 4	0 80 10 0
UNM lpr 1	.028535	0	lprcp	1	0
MIT lpr	.049562	0	lprcp	1	0
UNM lpr 2	.029079	0	lprcp	1	0
named	.030134	0	buf ov 1 buf ov 2	3 3	0 0
wuftpd	.027593	0		5	0
login	.042293	0	troj 1 troj 2	5 8	0 0
ps	.014905	0	troj 1 troj 2	11 15	0 0
inetd	.005634	0	DoS	1	0
xlock	.062143	0	buf ov 1 buf ov 2	1 1	0 0
Apache Server	.039102	0	buf ov DoS	1 1	0 0

Table 3.3. Results

3.4 Chapter Conclusions

Our results (both true and false positive rates) surpass those achieved by Forrest et al., confirming that the Kullback-Leibler dissimilarity measure, weighted by information gain of the *self-distance* distributions between a unknown sample and a baseline exemplar, is a suitable measure to distinguish between processes that are under attack and those that are not.

Although metrics derived from streams of system calls prove to be indicative of a process' behavior there may be many attacks and intrusions that are immune from detection by this method alone. Additional metrics will be required to detect their presence. The next chapter details the feature vectors (which include our *self-distance* measure) that we have constructed from low-level kernel data that supply those other metrics.

Chapter 4

Feature Set Construction from Low-Level Kernel Data

In their 1990 paper *IDES: A Progress Report* [78] Lunt et al. note the existence of low-level system data and suggest that it be used for intrusion detection. We are unaware of any other researchers comprehensively making use of low-level kernel data. This chapter details the feature sets that we constructed from low-level kernel data.

We collected low-level kernel data at the process, network, and system levels. We used this data to create feature vectors of 34 attributes at the process level, 67 attributes at the network level, and 18 attributes at the system level. Each vector is representative of system state at the time it was taken. Process data is sampled once per system call while network and system data is sampled at intervals of .5 seconds each.

As will be detailed in Chapter 5, the feature vectors derived from each sample were used as input in the creation of our model of normal behavior. Once the models were established, we created additional feature vectors derived from subsequent data samples and tested them for conformance to the model.

We used *Principal Component Analysis* (PCA) [58] to analyze the data sets (made up of the feature vectors) that represent our model. PCA gives us insight regarding the features that are the most meaningful in the model of normal behavior.

4.1 Process Level Data

We collected several low-level attributes from each process of interest (e.g: network attached processes, core system binaries, etc.) and used those attributes to construct the feature sets. Each low-level attribute consists of numeric values. Some were memory addresses that remained static and, when calculating the mean and standard deviation of that feature within the feature set, resulted in a standard deviation of 0. In those cases, the corresponding feature became a binary value that serves as a flag to indicate that some value other than the norm was encountered. Other values, such as return addresses from system calls were limited to a fixed set of addresses. Again, we used a binary value as that feature to indicate that some other address was observed. Finally, in cases such as heap size, which is dynamic, the actual size was used as the feature because the amount of dynamically allocated memory and its pattern of change have an *a priori* predictive value.

time The time that the sample was taken. This is recorded as type long time_t and is used to synchronize events that occur in other processes, the network model, system model, or on other hosts.

self distance metric The Kullback-Leibler dissimilarity measure of the difference between the *self-distance* probability distributions of the model and the sample. This value is assigned to blocks of system calls. For example, the Apache Web server executes 68 system calls when processing a single HTTP get request. Processes such as netstat, ls, and top generate varying numbers of system calls depending upon the number of connections, number of files, or the number of processes. Accordingly, the sequence lengths are varied to match the observed number of system calls generated by each process for one "atomic" operation.

nice The nice value ranges from 19 (nicest) to -19 (not nice to others). It quantifies the scheduling priority that the process requests for itself.

unknown calling address This is a binary value (0-1) to indicate that the calling address

(from a system call) was not previously encountered in the model.

calling address outside of code segment This is a binary value (1) that indicates that the calling address of a system call is outside of the code segment.

return address outside of code segment This is a binary value (1) that indicates that the return address of a system call is outside of the code segment.

unknown return address This is a binary value (0-1) to indicate that the return address (from a system call) was not previously encountered in the model.

code size This value is the actual size of the process, calculated by (mm->end_code - mm->start_code). We should expect that it will not deviate from instance to instance of the process.

library size This value is the actual library size of the libraries that are linked to the the process at run time. We should expect that it not deviate from instance to instance of the process.

current stack size This is a value that indicates the current stack size. During normal execution, the stack should grow and shrink. Rapid and sustained growth may be an indicator of a problem.

virtual memory size The size of the virtual memory owned by the process. This is calculated by traversing the AVL tree[1] pointed to by the first vm_area_struct and summing each vm_end - vm_start.

number of memory regions owned The kernel attempts to merge regions when a new one is allocated. Ownership of several regions could indicate a resource consumption attack.

[1] An AVL tree is another balanced binary search tree. Named after their inventors, Adelson-Velskii and Landis, they were the first dynamically balanced trees to be proposed. Like red-black trees, they are not perfectly balanced, but pairs of sub-trees differ in height by at most 1, maintaining an O(log n) search time. Addition and deletion operations also take O(log n) time.

resident set size limit This value reflects changes of the limit to the process' resident set size.

resident set size A value reflecting the change in number of pages that the process has in real memory.

locked virtual memory A value reflecting changes in the amount of virtual memory that is locked by the process. As with all of the memory measurements, we expect to see allocation and deallocation of resources that are commensurate with the changes to the other attributes.

xds The xds register is a pointer to the user's data segment. This feature contains a binary value (0-1) that indicates that the value has changed to one that was not encountered in the model.

system time The amount of time that the process has spent in s_time.

user time The amount of time that the process has spent in u_time.

open files A value indicating the number of files that are opened by the process. We expect this value to remain fairly constant.

child processes The number of processes forked by the process. A large number of children is indicative that there may be some type of resource consumption attack in progress.

minor faults A value representing the change in the number of minor faults the process has made, those which have not required loading a memory page from disk.

child minor faults A value representing the change in the number of minor faults that the process and its children have made, those which have not required loading a memory page from disk.

major faults A value representing the change in the number of major faults the process has made, those which have required loading a memory page from disk.

child major faults A value representing the change in the number of major faults that the process and its children have made, those which have required loading a memory page from disk.

page swaps A value representing the change in page swaps made by the process.

child page swaps A value representing the change in the number of page swaps made by the process' children.

links A value representing a change in the number of symbolic links held by a process. Some attacks are carried out by causing long chains of recursive links.

files open The number of files that are opened by a process. This number should fluctuate within a given range during the life of the process.

files locked The number of files that are locked by a process. We expect that the range of values will remain fairly constant.

address limit.seg The value of the highest memory address to be checked. This may be changed by set_fs, where the fs value determines whether or not argument validity checking should be performed.

cpu time A value representing a change in the cpu time alloted to the process.

changed UID A binary value (1) indicating that the UID value has changed.

changed GID A binary value (1) indicating that the GID value has changed.

changed SUID A binary value (1) indicating that the SUID value has changed.

4.2 Network Level Data

We collected 67 attributes from the network protocol stack, to include the number of active connections, half open connections, and the number of ports that have a process listening on them. Except for the connection information, the data is cumulative information defined

by the Management Information Base (MIB) for network management of TCP/IP-based Internets [100]. We used this data to produce a feature set containing statistical information that reflects the amount of change within each time interval. The feature set follows:

time The time that the sample was taken. This is recorded as type long time_t and is used to synchronize events that occur in other processes, the network model, system model, or on other hosts.

tcp_established This number provides the number of tcp connections including connections of type: "ESTABLISHED", "SYN_SENT", "FIN_WAIT1", "FIN_WAIT2", "TIME_WAIT","CLOSE", "CLOSE_WAIT", "LAST_ACK", and "CLOSING".

tcp_syn_recv This provides the number of connections that are in the state of "SYN_RECV"; the number of half open connections. A high number is indicative of a syn flood attack.

tcp_listen This provides the number of processes that are listening. Depending on the system, it should remain static.

Ip_Forwarding The indicator of whether this host is acting as an IP gateway in respect to the forwarding of datagrams received by, but not addressed to this host.

Ip_DefaultTTL The default value inserted into the Time_To_Live field of the IP header of datagrams originated at this host, whenever a TTL value is not supplied by the transport layer protocol.

Ip_InReceives The number of input datagrams received from interfaces, including those received in error, during the interval defined by the observation period. This value is indicative of the rate.

Ip_InHdrErrors The number of input datagrams that were discarded due to errors in their IP headers, including bad checksums, version number mismatch, other format errors, time_to_live exceeded, errors discovered in processing their IP options, etc, during the interval defined by the observation period.

Ip_InAddrErrors The number of input datagrams discarded because the IP address in their IP header's destination field was not a valid address to be received at this host. This count includes invalid addresses (e.g., 0.0.0.0) and addresses of unsupported Classes (e.g., Class E).

Ip_ForwDatagram The number of input datagrams for which this host was not their final IP destination, as a result of which an attempt was made to find a route to forward them to that final destination. In entities which do not act as IP Gateways, this counter will include only those packets which were Source_Routed via this host, and the Source_Route option processing was successful.

Ip_InUnknownProtos The number of locally_addressed datagrams received successfully but discarded because of an unknown or unsupported protocol.

Ip_InDiscards The number of input IP datagrams for which no problems were encountered to prevent their continued processing, but which were discarded (e.g., for lack of buffer space). Note that this counter does not include any datagrams discarded while awaiting reassembly.

Ip_InDelivers The total number of input datagrams successfully delivered to IP user_protocols (including ICMP).

Ip_OutRequests The total number of IP datagrams which local IP user_protocols (including ICMP) supplied to IP in requests for transmission. Note that this counter does not include any datagrams counted in ipForwDatagrams.

Ip_OutDiscards The number of output IP datagrams for which no problem was encountered to prevent their transmission to their destination, but which were discarded (e.g., for lack of buffer space). Note that this counter would include datagrams counted in ipForwDatagrams if any such packets met this (discretionary) discard criterion.

Ip_OutNoRoutes The number of IP datagrams discarded because no route could be found to transmit them to their destination. Note that this counter includes any packets counted

in ipForwDatagrams which meet this 'no_route' criterion. Note that this includes any datagrams which a host cannot route because all of its default gateways are down.

Ip_ReasmTimeout The maximum number of seconds that received fragments are held while they are awaiting reassembly at this host.

Ip_ReasmReqds The number of IP fragments received that need to be reassembled at this host.

Ip_ReasmOKs The number of IP datagrams successfully reassembled.

Ip_ReasmFails The number of failures detected by the IP reassembly algorithm (for whatever reason: timed out, errors, etc). Note that this is not necessarily a count of discarded IP fragments since some algorithms (notably the algorithm in RFC 815 [16]) can lose track of the number of fragments by combining them as they are received.

Ip_FragOKs The number of IP datagrams that have been successfully fragmented at this host.

Ip_FragFails The number of IP datagrams that have been discarded because they needed to be fragmented at this host but could not be (e.g.: because the DF flag was set).

Ip_FragCreates The number of IP datagram fragments that have been generated as a result of fragmentation at this host.

Icmp_InMsgs The total number of ICMP messages which the host received. Note that this counter includes all those counted by icmpInErrors.

Icmp_InErrors The number of ICMP messages which the host received but determined as having ICMP_specific errors (bad ICMP checksums, bad length, etc.).

Icmp_InDestUnreachs The number of ICMP Destination Unreachable messages received.

Icmp_InTimeExcds The number of ICMP Time Exceeded messages received during the interval defined by the observation period.

Icmp_InParmProbs The number of ICMP Parameter Problem messages received during the interval defined by the observation period.

Icmp_InSrcQuenchs The number of ICMP Source Quench messages received during the interval defined by the observation period.

Icmp_InRedirects The number of ICMP Redirect messages received during the interval defined by the observation period.

Icmp_InEchos The number of ICMP Echo (request) messages received during the interval defined by the observation period.

Icmp_InEchoReps The number of ICMP Echo Reply messages received during the interval defined by the observation period.

Icmp_InTimestamps The number of ICMP Time stamp (request) messages received during the interval defined by the observation period.

Icmp_InTimestampReps The number of ICMP Time stamp Reply messages received during the interval defined by the observation period.

Icmp_InAddrMasks The number of ICMP Address Mask Reply messages received during the interval defined by the observation period.

Icmp_InAddrMaskReps The number of ICMP Address Mask Request messages received during the interval defined by the observation period.

Icmp_OutMsgs The total number of ICMP messages which this host attempted to send during the interval defined by the observation period. Note that this counter includes all those counted by icmpOutErrors.

Icmp_OutErrors The number of ICMP messages which this host did not send due to problems discovered within ICMP, such as a lack of buffers. This value should not include

errors discovered outside the ICMP layer, such as the inability of IP to route the resultant datagram.

Icmp_OutDestUnreachs The number of ICMP Destination Unreachable messages sent during the interval defined by the observation period.

Icmp_OutTimeExcds The number of ICMP Time Exceeded messages sent during the interval defined by the observation period.

Icmp_OutParmProbs The number of ICMP Parameter Problem messages sent during the interval defined by the observation period.

Icmp_OutSrcQuenchs The number of ICMP Source Quench messages sent during the interval defined by the observation period.

Icmp_OutRedirects The number of ICMP Redirect messages sent. For a host, this object will always be zero, since hosts do not send redirects during the interval defined by the observation period.

Icmp_OutEchos The number of ICMP Echo (request) messages sent during the interval defined by the observation period.

Icmp_OutEchoReps The number of ICMP Echo Reply messages sent during the interval defined by the observation period.

Icmp_OutTimestamps The number of ICMP Time stamp (request) messages sent during the interval defined by the observation period.

Icmp_OutTimestampReps The number of ICMP Time stamp Reply messages sent during the interval defined by the observation period.

Icmp_OutAddrMasks The number of ICMP Address Mask Request messages sent during the interval defined by the observation period.

Icmp_OutAddrMaskReps The number of ICMP Address Mask Reply messages sent during the interval defined by the observation period.

Tcp_RtoAlgorithm The algorithm used to determine the timeout value used for retransmitting unacknowledged octets during the interval defined by the observation period.

Tcp_RtoMin The maximum value permitted by a TCP implementation for the retransmission timeout, measured in milliseconds. More refined semantics for objects of this type depend upon the algorithm used to determine the retransmission timeout. In particular, when the timeout algorithm is rsre(3), an object of this type has the semantics of the UBOUND quantity described in RFC 793 [25].

Tcp_RtoMax The minimum value permitted by a TCP implementation for the retransmission timeout, measured in milliseconds. More refined semantics for objects of this type depend upon the algorithm used to determine the retransmission timeout. In particular, when the timeout algorithm is rsre(3), an object of this type has the semantics of the LBOUND quantity described in RFC 793[25].

Tcp_MaxConn The limit on the total number of TCP connections the host can support. In entities where the maximum number of connections is dynamic, this object should contain the value 1.

Tcp_ActiveOpens The number of times TCP connections have made a direct transition to the SYN_SENT state from the CLOSED state during the interval defined by the observation period.

Tcp_PassiveOpens The number of times TCP connections have made a direct transition to the SYN_RCVD state from the LISTEN state during the interval defined by the observation period.

Tcp_AttemptFails The number of times TCP connections have made a direct transition to the CLOSED state from either the SYN_SENT state or the SYN_RCVD state, plus the

number of times TCP connections have made a direct transition to the LISTEN state from the SYN_RCVD state during the interval defined by the observation period.

Tcp_EstabResets The number of times TCP connections have made a direct transition to the CLOSED state from either the ESTABLISHED state or the CLOSE_WAIT state during the interval defined by the observation period.

Tcp_CurrEstab The number of TCP connections for which the current state is either ESTABLISHED or CLOSE_ WAIT.

Tcp_InSegs The total number of segments received, including those received in error. This count includes segments received on currently established connections during the interval defined by the observation period.

Tcp_OutSegs The total number of segments sent, including those on current connections but excluding those containing only retransmitted octets during the interval defined by the observation period.

Tcp_RetransSegs The total number of segments retransmitted, that is, the number of TCP segments transmitted containing one or more previously transmitted octets during the interval defined by the observation period.

Tcp_InErrs The total number of segments received in error during the interval (e.g., bad TCP checksums).

Tcp_OutRsts The number of TCP segments sent containing the RST flag during the interval defined by the observation period.

Udp_InDatagrams The total number of UDP datagrams delivered to UDP users during the interval defined by the observation period.

Udp_NoPorts The total number of received UDP datagrams for which there was no application at the destination port during the interval defined by the observation period.

Udp_InErrors The number of received UDP datagrams that could not be delivered for reasons other than the lack of an application at the destination port during the interval defined by the observation period.

Udp_OutDatagrams The total number of UDP datagrams sent from this host during the interval defined by the observation period.

4.3 Global System Level Data

We collected measurable system data at .5 second intervals. Measurable system data includes: Memory Usage (by user, process, and time of day) CPU Load (by user, process, and time of day), Number of Concurrent Users, Number of Processes, and Disk Usage (file reads, writes, and page faults).

time The time that the sample was taken. This is recorded as type long time_t and is used to synchronize events that occur in other processes, the network model, system model, or on other hosts.

per_mem_used The amount of memory used, expressed as a percentage of available memory.

per_swap_used The amount of swap memory used, expressed as a percentage of available memory

cpu_one_minute The average cpu load for the previous one minute.

cpu_five_minute The average cpu load for the previous five minutes.

cpu_ten_minute The average cpu load for the previous ten minutes.

current_procs The number of processes currently running on the system.

count_users The number of users currently logged into the system. If a user is logged in twice, he will be counted twice.

time_user_mode A value representing the change in the amount of time that the CPU spent in user mode since the last sample was taken.

time_user_mode_low A value representing the change in the amount of time that the CPU spent in low priority user mode since the last sample was taken.

time_sys_mode A value representing the change in the amount of time that the CPU spent in system mode since the last sample was taken.

time_idle A value representing the change in the amount of time that the CPU was idle since the last sample was taken.

pages_in A value representing the change in the number of pages that were paged into memory from disk since the last sample was taken.

pages_out A value representing the change in the number of pages that were paged out of memory from disk since the last sample was taken.

swap_in A value representing the change in the number of swap pages that were brought into memory since the last sample was taken.

swap_out A value representing the change in the number of swap pages that were brought out of memory since the last sample was taken.

context_switch A value representing the change in the number of context switches that have occurred since the last sample was taken.

count_procs A value representing the change in the number of forks since the last sample was taken.

4.4 Discussion

We used *Principal Component Analysis* (PCA) [58] to discover the most significant features (given our data from a correctly functioning system) of the feature sets that we constructed at the process, network and system levels. PCA, also referred to as eigen-analysis, is

mainly used to reduce the dimensionality of a data set while retaining as much information as possible. Accordingly, the first principal component is the linear combination of features accounting for the greatest amount of variation, the second principal component accounts for the second largest amount of variation and is independent to the first principal component, etc.

PCA is performed on the variance-covariance matrix of our feature set. The variance-covariance matrix (hereafter referred to as the covariance matrix) enables us to measure distance in a manner that is invariant to linear transformations of the data. The main diagonal of a covariance matrix contains the variance of each feature. The remaining entries in the matrix contain the covariances between the two opposing (row versus column) features. Given n sets of variates denoted $X_1, \ldots, X_n$, the covariance $cov(x_i, x_j)$ is defined by Equation 4.1

$$V_{ij} = covariance(x_i, x_j) \equiv \langle (x_i - \mu_i)((x_j - \mu_j) \rangle \tag{4.1}$$

where μ_i and μ_j are the means of x_i and x_j. The matrix V_{ij} of the quantities V_{ij} = $cov(x_i, x_j)$ is called the variance-covariance is matrix where $i = j$ defines the variance.

We used *Singular Value Decomposition* (SVD) [117] to compute the eigenvalues and their corresponding eigenvectors of the covariance matrix for each of the baseline data sets, sorting the results in descending order by eigenvalue. When using PCA to reduce the dimensionality of a data set, the general practice is to take those (eigenvalue, eigenvector) pairs that account for 85% of the sum of the eigenvalues. The basic idea is that "noise", or inconsequential features, will be removed.

Accordingly, we took the (eigenvalue, eigenvector) pairs that account for 85% of the sum of the eigenvalues to measure the representation, R, of each feature f that was included in the dominant eigenvalues. This operation was carried out according to equation 4.2, where n is the number of eigenvectors in the top 85% and m is the size of the eigenvector.

$$R(f_r) = \sum_{j=1}^{n} \frac{f_{rj}^2}{\sum_{i=1;i\neq r}^{m} f_{ij}^2} \tag{4.2}$$

Table 4.1 shows the application of Equation 4.2 at the process level. It lists the principal features of the process state baselines and the degree (on a scale of 0 to 1) to which they are represented in the eigenvectors that correspond to the top 85% eigenvalues. Seven (7) eigenvalues accounted for the top 85%.

Feature	**Quantity**
Self-Distance metric	.999940
Change in CPU user time	.999213
Change in CPU system time	.999213
Change in the stack size	.998674
Change in the number of locks	.991897
Change in the number of minor faults	.556307
Change in the number of major faults	.499332

Table 4.1. Principal Features at the Process Level

Table 4.2 shows the application of Equation 4.2 at the network level. It lists the principal features of the network state baseline and the degree (on a scale of 0 to 1) to which they are represented in the eigenvectors that correspond to the top 85% eigenvalues. Five (5) eigenvalues accounted for the top 85%.

Feature	**Quantity**
UDP Change in NO Port	.498949
UDP Change in Out Datagrams	.498835
TCP Change in Current Established	.492124
ICMP Change in In Echos	.492124
IP Change in In Delivers	.356346
IP Change in IP Ins	.353587

Table 4.2. Principal Features at the Network Level

Table 4.3 shows the application of Equation 4.2 at the system level. It lists the principal features of the system state baseline and the degree to which they are represented in the eigenvectors that correspond to the top 85% eigenvalues. Eight (8) eigenvalues accounted for the top 85%.

Feature	Quantity
Change in Pages in	.985320
Number of Current Processes	.931737
Percentage of memory used	.855775
Change in current processes	.811692
Change in context switches	.617284
Change in time the CPU spent in user mode	.581675
Change in time the CPU spent in system mode	.550951
Change in time the CPU spent in idle mode	.533352

Table 4.3. Principal Features at the System Level

4.5 Chapter Conclusions

This chapter lists the features that we derive from the low-level kernel data. Eigen analysis of the feature set indicates that the *self-distance* metric is the dominant feature in the feature set of a running process. One might think, based upon the results of the eigen analysis, that only the dominant features need to be monitored. The analysis was conducted on data sets that represented normal process state, many of the features have no variance (e.g.: return addresses that were not previously encountered), consequently are not in the dominant set of features. This analysis informed us that our models must be constructed in a manner that highlights subsequent instances wherein the feature varies significantly from its mean.

In the next chapter we will detail how we have used these features to construct models of selected processes, the network protocol stack, and the system in general. We will detail our experiments and their results. In Chapter 6, we momentarily digress to report on our empirical analysis of over 4,000 attacks and intrusions. This analysis served as the foundation for our ontology, which is presented in Chapter 7.

Chapter 5

Modeling System Behavior

Host-based anomaly intrusion detectors compare current system state to a model of "permitted" behavior, testing for non-conformance to the model. This chapter details how we constructed our model of normal system state. We used cluster analysis to model the system and have experimented with *Fuzzy c-Medoid* [70], *Principal Direction Divisive Partitioning* [10], and *K-Means* clustering [36] to produce clusters that model normal behavior. Likewise, we have experimented with the *Mahalanobis Metric* [17] and the *Euclidean Distance* to measure distances between the feature vectors in order to cluster them. We have also experimented with the effects of z-normalization [42] on the data set.

To determine which combination of clustering method, distance measure, and normalization technique was optimal, we experimented by taking known samples of benign and anomalous data and tested them for conformance to each of the models. We evaluated each model by using *Precision, Recall* and *F-Measure* [73] as a performance metric.

In addition to creating models of normalcy for processes of interest, the network, and the global system, we also created a mean vector and standard deviation vector for the feature sets that were used to create the model. We used these mean and standard deviation vectors to map instances of non-conforming data, which are numeric values, to instances of our ontology, which are represented as classes, relationships, and properties.

5.1 Model Creation

A vector representing state (process, network, or system) at a particular instance of time is a vector with n entries $x = (x_1, \ldots, x_n)^T$. This vector carries information describing the feature set that we have constructed and is taken from the host under observation. Assuming that X is a set of vectors representing the baseline state of the system, our goal is to partition X into clusters, so that clusters $\pi_1, \ldots, \pi_k$ contain vectors $x \in X$ corresponding to normal behavior.

As stated in the introduction to this chapter, we used three different clustering algorithms, two different distance measures, and experimented with the effects of z-normalizing the data set. The following explains these measures.

Z-Normalization Z-Normalization is used to standardize the parameters recorded in a vector to zero mean and unit variance. For each set of vectors X, each parameter $x_i[j]$ represented in a vector has a mean α_j given in Equation 5.1 and a standard deviation σ_j given in Equation 5.2.

$$\alpha_j = \frac{\sum_{i=1}^{n} x_j[i]}{n} \tag{5.1}$$

$$\sigma_j = \frac{\sqrt{\sum_{i=1}^{n} (x_j[i] - \alpha_j)^2}}{n-1} \tag{5.2}$$

Using the the mean α and standard deviation σ, Equation 5.3 gives the algorithm for z-normalizing each vector $x \in X$.

$$Z(x) \; = \; \frac{x[j] - \alpha_j}{\sigma_j} \tag{5.3}$$

Z-normalization scales a data set so that all parameters are equally weighted. This equal weighting mitigates the condition wherein the parameters with large values dominate those with small values. For example, in a two parameter vector, if one parameter

has a range 500 - 1000 and the other parameter has a range of 2 - 10, the first parameter will dominate even though small changes in the second parameter may be more significant. This disparity also comes into play when conducting *Principal Component Analysis* (PCA) [58]. When doing PCA on a data set, the parameters with the greatest variances will dominate the linear combinations of the eigenvectors, potentially masking the discovery of the principal components.

Principal Direction Divisive Partitioning The Principal Direction Divisive Partitioning (PDDP) clustering algorithms starts with a root cluster (i.e.: the initial data set) and recursively splits it into pairs of children clusters until the specified number of clusters have been created.

PDDP does not use a distance measure, rather, it uses *SVD* to calculate the (eigenvalues, eigenvectors) of the cluster. It then uses the eigenvector that corresponds to the largest eigenvalue as the principal direction of the mean centroid of the cluster and projects each of the vectors in that cluster as a data point onto the mean vector. Two new clusters are formed by bifurcating the mean vector as illustrated in Figure 5.1.

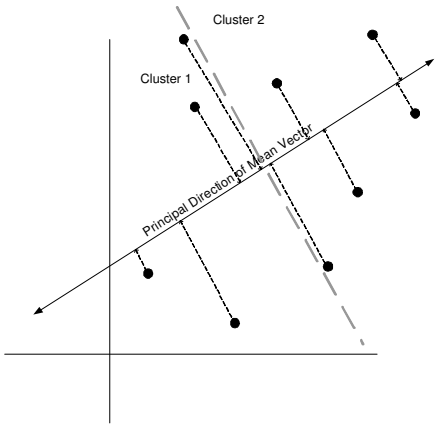


FIG. 5.1. Illustration of Principal Direction Divisive Partitioning

Fuzzy c Medoid Clustering The Fuzzy c-Medoids (FCMdd) algorithm takes a matrix containing the pair wise distances between the vectors as input. FCMdd attempts to min-

imize the intra-cluster (Equation 5.4) distance while maximizing the inter-cluster distance (Equation 5.5).

$$\text{intra-cluster}_k = \frac{\Sigma_{i \in k} \Sigma_{j \in k, i \neq j} D(i,j)}{|k| \times (|k| - 1)} \tag{5.4}$$

$$\text{inter-cluster}_{k1,k2} = \frac{\Sigma_{i \in k1} \Sigma_{j \in k2} D(i,j)}{|k1| \times |k2|} \tag{5.5}$$

The FCMdd algorithm is in the category of alternating cluster estimation paradigms, and is not guaranteed to find the global minimum. Consequently, we repeated this process several times to increase the reliability of our results. FCMdd reportedly finds the optimal number of clusters for the data set. The strategy to is to over-specify the initial number of clusters and allow FCMdd to merge the extra clusters until "optimality" is reached.

The FCMdd assigns feature vectors to a cluster in a *fuzzy* or "possibilistic" manner, for example a vector could be assigned to Cluster X with a degree of possibility of .8 and simultaneously be assigned to Cluster Y with a degree of possibility of .75. Once the cluster assignments are made, the FCMdd algorithm attempts to produce the optimal number of clusters as follows:

i. Form a fully connected graph where each cluster serves as a node.

ii. Using the *Jaccard* index, given in Equation 5.6, and the inter-cluster distance r, given in Equation 5.5, between the medoids of the clusters, weight the edge between the clusters according to equation 5.7.

$$Index_{Jaccard}(X,Y) = \frac{|X \cap Y|}{|X \cup Y|} \tag{5.6}$$

$$W(X,Y) = r(\mathbf{x},\mathbf{y})\,(1 - Index_{Jaccard}(X,Y)) \tag{5.7}$$

iii. Once the edges between each cluster have been weighted, calculate the *Minimum Spanning Tree* of the graph. As each edge is cut, the two clusters that were connected are merged.

k-Means Clustering We use the traditional k-means clustering algorithm, which is a greedy algorithm for clustering n objects into k clusters. There are two problems that are inherent to *k-Means Clustering* algorithms. The first is determining the initial partition and the second is determining the optimal number of clusters.

In our implementation, we set the number of clusters to be the same number as was created by the FCMdd algorithm. We then selected the feature vectors that were farthest apart to serve as the initial centroids.

Euclidean Distance Given two vectors, x_i and x_j, with n parameters, the Euclidean distance between them is given in Equation 5.8.

$$d(x_i, x_j) \;=\; \sqrt{(x_{i,1} - x_{j,1})^2 \;+ (x_{i,2} - x_{j,2})^2 \;+ \ldots \;(x_{i,n} - x_{j,n})^2} \tag{5.8}$$

Mahalanobis Distance The Mahalanobis metric uses the inverse variance-covariance matrix of the set of feature vectors, essentially weighting the difference between two vectors by a linear combination of the original data set. The Mahalanobis metric is given in Equation 5.9, where $\mathcal{C}^{-1}$ is the inverted variance/covariance matrix, and x_i, x_j are the feature vectors for which the distance is calculated.

$$d(x_i, x_j) \;=\; \sqrt{(x_i - x_j)^T \, \mathcal{C}^{-1} \, (x_i - x_j)} \tag{5.9}$$

By using the variance/covariance matrix, the Mahalanobis distance takes the data set's variability into account. Rather than treating all values equally, it weights the differences by the range of variability of the features. The Mahalanobis metric is calculated in units of standard deviation from the feature's mean. Therefore, the clusters form an ellipse that is elongated in the direction of the mean vector.

The benefit of using the Mahalanobis metric is that it scales the coordinate axis and corrects for correlation between the different features. This is advantageous because it mitigates the limitations of the *Euclidean* distance that were presented during our discussion of z-normalization. These advantages, however, are not without additional

cost. The covariance matrix and its inverse can be hard to determine accurately and the memory and time requirements grow quadratically rather than linearly with the number of features. As stated in Equation 5.9, $\mathcal{C}^{-1}$ is the inverse variance/covariance matrix of our data set. In those instances where the variance/covariance matrix is singular, we need to construct a pseudo inverse as follows:

Without loss of generality, the *Singular Value Decomposition* of $\mathcal{C}$ is of the form:

$$C = U\Lambda V^T \tag{5.10}$$

where U and V are square matrices with orthogonal columns such that:

$$U^T U = V^T V = \mathcal{I} \tag{5.11}$$

and Λ is an n x n matrix with real diagonals λ_i such that $\lambda_1 \geq \lambda_2 \geq \ldots \geq \lambda_n$. We can, therefore, rewrite our variance/covariance matrix as:

$$\mathcal{C} = \lambda_1 V_1 U_1^T + \lambda_2 V_2 U_2^T + \ldots + \lambda_n V_n U_n^T \tag{5.12}$$

This also allows us to rewrite the inverse variance/covariance matrix as:

$$\mathcal{C}^{-1} = \frac{1}{\lambda_1} U_1^T V_1 + \frac{1}{\lambda_2} U_2^T V_2 + \ldots + \frac{1}{\lambda_n} U_n^T V_n \tag{5.13}$$

The pseudo inverse is constructed by applying Equation 5.13 $\forall \lambda_i > 0$.

During our experiments, we observed that the Euclidean distance between the vectors $x \in X$ differed significantly from the vectors $y \in Y$ where Y = z-normal(X). The vectors marking the diameter of the set also varied.

This was not the case with the Mahalanobis distance. The vectors x and y, where Y = z-normalized(X), resulted in identical distances and diameter of the data set. This phenomena is explained by the proof in Figure 5.2.

Let $x_1, \ldots, x_p$ $x_i \in \mathbb{R}^n$ represent a set of vectors X
where each parameter has a mean of 0.
The variance/covariance matrix $\mathcal{C}_x = X^T X$
The Mahalanobis distance $d(x_i, x_j) = (x_i - x_j)\ C_x^{-1}\ (x_i - x_j)^T$
Let $d_1, \ldots, d_n$ be a set of constants such that $d_i = \frac{1}{\sigma_{x_i}}$

$$\mathbf{D} = \begin{bmatrix} d_1 & 0 & \ldots & 0 \\ 0 & d_2 & & \vdots \\ \vdots & \ldots & \ddots & \vdots \\ 0 & & \ldots & d_n \end{bmatrix}$$

If $y_i = Dx_i$ and $D^{-1}y_i = x_i$ then $Y = DX$. (z-normal(X)).
$D^{-1}Y = X$ because D is symmetric and
$Y^T = X^T D^T = X^T D$
The covariance matrix $\mathcal{C}_y = Y^T Y = DX^T XD = DC_x D$.
The inverse covariance matrix $\mathcal{C}_y^{-1} = D^{-1}\mathcal{C}_x^{-1}D^{-1}$
The Mahalanobis distance $d(y_i, y_j) = (y_i - y_j)\ C_y^{-1}\ (y_i - y_j)^T$

$\therefore d(y_i, y_j) = (y_i - y_j)\ C_y^{-1}\ (y_i - y_j)^T$
$= (x_i - x_j)D^T D^{-1} C_x^{-1} D^{-1} D (x_i - x_j)^T$
$= (x_i - x_j)\ C_x^{-1}\ (x_i - x_j)^T$

FIG. 5.2. Proof of Equality of the Mahalanobis Distance Between Unnormalized Data and Z-normalized Data when C is Invertible

5.2 Experiments

We experimented by using all possible combinations of clustering algorithms, distance functions, and z-normalization. Once we created a model, additional sets, W, of vectors w, were taken from the system while it was operating under normal conditions. We measured the distance of each vector $w \in W$ from the clusters $\pi_1, \ldots, \pi_k$ to determine membership in one of the clusters. Vectors that were determined to not have membership in one of the clusters were considered to be false negatives.

Similarly, additional set(s) Y, of vectors y, were taken from the same system while it was under attack. We measured the distance of each vector $y \in Y$ from the clusters $\pi_1, \ldots, \pi_k$ to ensure that they do not have membership in the clusters. Those vectors that were determined to have membership in one of the clusters were considered to be false positives.

Clustering Algorithm	Distance Function	Data Type
FCMdd	Mahalanobis	Plain
	Euclidean	Plain
		Z-Normalized
k-Means	Mahalanobis	Plain
	Euclidean	Plain
		Z-Normalized
PDDP		Plain
		Z-Normalized

Table 5.1. Combinations of Clustering Algorithms, Distance Measures, & Normalization Technique

We ran four (4) rounds of experiments for each of the eight (8) combinations listed in Table 5.1. The first round consisted of five (5) attacks directed against network connected processes. For this round we used different versions of the the *Apache* HTTP server running on an installation of Red Hat Linux. To exercise the HTTP Server, we mirrored two web sites and replayed their usage logs against them. During this round we collected process data. The attacks, their consequences, the HTTP server version, and Linux kernel version are listed in Table 5.2.

Attack Number	Attack	Consequence	HTTP Server Version	Kernel Version
1	Buffer Overflow	Denial of Service	1.3.27	2.4.7-10
2	Resource Consumption (memory leak)	Denial of Service	2.0.39	2.4.7-10
3	Long Slashes	Exposes Directory	1.3.12	2.4.7-10
4	Buffer Overflow	User to Root	1.3.23	2.4.7-10
5	MIME Flood	Denial of Service	1.3.1	2.4.7-10

Table 5.2. Attacks and Consequences: Network Connected Processes

The second round of experiments was carried out by replacing four (4) core system binaries (utilities typically used for system administration) with "trojaned" versions. The trojaned versions were altered to hide the presence of files, processes, users, and network

connections. Table 5.3 lists those system binaries, their function, and the kernel version used during the experiment.

Attack Number	Binary	Consequence	Kernel Version
6	ls	hides specified file	2.4.7-10
7	netstat	hides specified connections	2.4.7-10
8	ps	hides specified processes	2.4.7-10
9	top	hides processes and recalculates CPU load	2.4.7-10

Table 5.3. Attacks and Consequences: Trojaned Binaries

The third round of attack consisted of five (5) different attacks that were directed against the network protocol stack. Most attacks against the protocol stack typically result in a denial of service (DoS). Some attacks resulted in a degradation of network resources, some resulted in a degradation of processing ability, and some froze the machine. The attacks, their action, and the Linux kernel version are listed in Table 5.4.

Attack Number	Attack	Consequence	Kernel Version
10	tcp portscan	scans for open ports	2.4.20-8
11	syn flood 1	1/2 open a connection	2.4.20-8
12	ping of death	large icmp messages	2.4.20-8
13	ip frag	transmit overlapping ip fragments	2.4.20-8
14	syn flood 2	floods target with illformed Syn segments	2.4.20.8

Table 5.4. Attacks and Consequences: Network Protocol Stack

The final round of attacks, although an attack against a network connected process and a locally effected DoS attack, were used to model and test global system behavior. Accordingly, they consisted of two (2) memory consumption attacks. Those attacks, their actions, and the Linux kernel version are listed in Table 5.5.

In summation, we ran five (5) rounds of experiments consisting of 5, 4, 5, and 2 sets of attacks for the eight (8) combinations of clustering algorithms, distance measures, and data conditioning technique. This totaled 128 different trials.

Attack Number	Attack	Consequence	Kernel Version
15	local DoS	memory consumption	2.4.20-8
16	remote DoS	mime flood against Apache 1.3.27	2.4.20-8

Table 5.5. Attacks and Consequences: System Resources

5.2.1 Methodology

For each trial, we had 8,000 samples of normal data, 2,000 sample of "unknown" normal data, and 2,000 samples of data taken while the host was under attack. Once the model of normalcy was constructed, the "unknown" normal data was compared to it for inclusion and the "attack" data was compared to it for exclusion. Since we were doing anomaly detection, we tested for inclusion to the model — not exclusion. Therefore, we consider a false negative to exist whenever the normal data is deemed to not fit the model. Likewise, we consider a false positive to exist whenever attack data is deemed to fit the model. Our goal was to minimize false positives and false negatives. Table 5.6 illustrates the confusion matrix for our classification scheme. Although are ultimate goal was to detect intrusions, at this first stage we were classifying unknown data in order to determine conformance to a model of normalcy. Therefore, *a false negative is normal data that is wrongly classified as abnormal. Similarly, a false positive is aberrant data that is incorrectly classified as conforming to the model of normal behavior.*

Actual Classification	Predicted Classification	
	Anomalous	Normal
Anomalous	True Negative	False Positive
Normal	False Negative	True Positive

Table 5.6. Confusion Matrix for Actual and Predicted Classifications

We used *precision, recall,* and *F-Measure* [73], as metrics to measure each model's performance. Table 5.7 defines these metrics.

Precision	=	$\frac{TruePositives}{(TruePositives + FalsePositives)}$
Recall	=	$\frac{TruePositives}{(TruePositives + FalseNegatives)}$
F-Measure	=	$\frac{(1+\beta^2)* Recall* Precision}{\beta^2* Recall+ Precision}$

Table 5.7. *Precision, Recall,* and *F-Measure*; β Corresponds to the Relative Importance of *Precision* vs. *Recall* and is Usually Set to 1

5.2.2 Fuzzy c Medoid Clustering

During initialization we set the initial cluster number to 24 and selected candidate medoids such that the initial medoid was the one most central to the data set and each subsequent candidate medoid was selected so that it was most dissimilar to the previously selected medoids. The FCMdd is a possibilistic algorithm, that is each data point is determined to belong to a cluster with some degree of possibility. The degree of possibility can be set and is referred to as the "fuzzifier". We selected a value of 1.125.

Because the Mahalanobis distance of unconditioned data and the z-normalized data resulted in the same dissimilarity measure (viz.: the Proof in Figure 5.2), we only experimented with unconditioned data using the Mahalanobis distance metric.

Mahalanobis Distance

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	7999	0.079	0.847	0.004	0.000	0.997
		1	0.000				
2	2	75	0.005	0.696	0.042	0.000	0.978
		7925	0.138				
3	2	7999	0.033	0.936	0.007	0.000	0.996
		1	0.000				
4	2	6	0.089	0.420	0.007	0.000	0.996
		7994	0.146				
5	2	7999	0.051	0.925	0.000	0.000	1.000
		1	0.000				
6	3	3	0.077	(1-2)0.039	0.000	0.000	1.000
		5	0.002	(2-3)0.196			
		7992	0.240	(1-3)0.184			
7	2	1189	0.076	0.126	0.000	0.000	1.000
		6811	0.099				
8	3	76	0.026	(1-2)0.952	0.000	0.000	1.000
		6	0.026	(2-3)0.590			
		7918	0.108	(1-3)0.363			
9	3	364	0.100	(1-2)0.544	0.000	0.000	1.000
		1029	0.063	(2-3)0.232			
		6607	0.068	(1-3)0.316			
10	2	1	0.000	0.708	0.009	0.624	0.757
		7999	0.026				
11	2	1	0.000	0.708	0.000	0.000	1.000
		7999	0.026				
12	2	1	0.000	0.708	0.000	0.000	1.000
		7999	0.026				
13	2	1	0.000	0.708	0.000	0.000	1.000
		7999	0.026				
14	2	1	0.000	0.708	0.000	0.000	1.000
		7999	0.026				
15	3	1	0.000	(1-2)0.693	0.003	0.000	0.998
		1	0.000	(2-3)0.724			
		7998	0.032	(1-3)0.693			
16	3	1	0.000	(1-2)0.693	0.003	0.000	0.998
		1	0.000	(2-3)0.724			
		7998	0.032	(1-3)0.693			
						AVERAGE	**0.982249**

Table 5.8. Performance of the FCMdd Clustering Algorithm using the Mahalanobis Distance

Euclidean Distance: Unconditioned Data

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	650	0.002	0.985	0.000	0.258	0.885
		7350	0.015				
2	2	1083	0.005	0.696	0.073	0.000	0.962
		6917	0.138				
3	2	22	0.010	1.000	0.000	0.000	1.000
		7978	0.019				
4	2	70	0.001	0.999	0.000	0.905	0.688
		7930	0.003				
5	2	11	0.011	1.000	0.070	0.013	0.957
		7989	0.020				
6	2	6879	0.060	0.940	0.000	0.000	1.000
		1121	0.024				
7	2	7516	0.060	0.942	0.000	0.000	1.000
		484	0.022				
8	3	19	0.29	(1-2)0.974	0.000	0.015	0.992
		4128	0.001	(2-3)0.642			
		3799	0.001	(1-3)0.666			
9	3	162	0.066	(1-2)0.946	0.001	0.054	0.973
		717	0.002	(2-3)0.579			
		7121	0.002	(1-3)0.728			
10	2	31	0.230	0.569	0.053	1.000	0.642
		7969	0.142				
11	2	31	0.230	0.569	0.053	0.063	0.942
		7969	0.142				
12	2	31	0.230	0.569	0.053	0.011	0.967
		7969	0.142				
13	2	31	0.230	0.569	0.053	0.008	0.968
		7969	0.142				
14	2	31	0.230	0.569	0.053	0.009	0.968
		7969	0.142				
15	2	1	0.000	0.979	0.000	0.000	1.000
		7999	0.021				
16	2	1	0.000	0.979	0.000	0.001	0.999
		7999	0.021				
						AVERAGE	**0.932687**

Table 5.9. Performance of the FCMdd Clustering Algorithm on Unconditioned Data using Euclidean Distance

Euclidean Distance: Z-Normalized Data

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	7999	0.028	0.976	0.002	0.000	0.999
		1	0.000				
2	2	7999	0.032	0.986	0.000	0.000	1.000
		1	0.000				
3	2	7999	0.023	0.928	0.000	0.000	1.000
		1	0.000				
4	2	7999	0.026	0.881	0.022	0.017	0.980
		1	0.000				
5	2	7999	0.025	0.911	0.070	0.013	0.957
		1	0.000				
6	3	5	0.095	(1-2)0.985	0.001	0.000	0.999
		5	0.069	(2-3)0.606			
		7990	0.050	(1-3)0.780			
7	3	15	0.026	(1-2)0.999	0.000	0.000	1.000
		12	0.038	(2-3)0.671			
		7973	0.078	(1-3)0.749			
8	2	7981	0.036	0.475	0.037	0.000	0.981
		19	0.108				
9	3	6	0.071	(1-2)0.984	0.000	0.000	1.000
		16	0.067	(2-3)0.824			
		7978	0.058	(1-3)0.546			
10	2	3	0.472	0.787	0.004	0.632	0.757
		7997	0.045				
11	2	3	0.472	0.787	0.004	0.000	0.997
		7997	0.045				
12	2	3	0.472	0.787	0.004	0.000	0.997
		7997	0.045				
13	2	3	0.472	0.787	0.004	0.000	0.997
		7997	0.045				
14	2	3	0.472	0.787	0.004	0.000	0.997
		7997	0.045				
15	3	1	0.000	(1-2)1.000	0.000	0.000	1.000
		1	0.000	(2-3)0.743			
		7998	0.026	(1-3)0.660			
16	3	1	0.000	(1-2)1.000	0.000	0.000	1.00
		1	0.000	(2-3)0.743			
		7998	0.026	(1-3)0.660			
						AVERAGE	**0.978812**

Table 5.10. Performance of the FCMdd Clustering Algorithm on Z-Normalized Data using Euclidean Distance

5.2.3 k-Means Clustering

Mahalanobis Distance

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	5654	0.044	0.131	0.081	0.000	0.957
		2346	0.097				
2	2	1945	0.108	0.266	0.142	0.000	0.923
		6055	0.094				
3	2	1719	0.025	0.043	0.142	0.000	0.923
		6281	0.031				
4	2	2809	0.050	0.221	0.031	0.000	0.984
		5191	0.120				
5	2	1892	0.049	0.083	0.000	0.000	1.000
		6108	0.041				
6	2	2942	0.065	0.102	0.000	0.000	1.000
		5058	0.068				
7	2	368	0.116	0.158	0.000	0.000	1.000
		7632	0.092				
8	2	1166	0.080	0.246	0.000	0.000	1.000
		6834	0.080				
9	2	1034	0.063	0.248	0.000	0.000	1.000
		6096	0.090				
10	2	6377	0.027	0.032	0.018	0.829	0.698
		1623	0.018				
11	2	6377	0.027	0.032	0.009	0.000	0.995
		1623	0.018				
12	2	6377	0.027	0.032	0.009	0.000	0.995
		1623	0.018				
13	2	6377	0.027	0.032	0.009	0.000	0.995
		1623	0.018				
14	2	6377	0.027	0.032	0.009	0.000	0.995
		1623	0.018				
15	2	41	0.031	0.119	0.003	0.000	0.998
		7959	0.031				
16	2	41	0.031	0.119	0.003	0.000	0.998
		7959	0.031				
						AVERAGE	**0.966312**

Table 5.11. Performance of the k-Means Clustering Algorithm using the Mahalanobis Distance

Euclidean Distance: Unconditioned Data

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	650	0.026	0.040	0.009	0.000	0.995
		7350	0.014				
2	2	3249	0.001	0.204	0.171	0.074	0.871
		4751	0.318				
3	2	6238	0.008	0.059	0.418	0.368	0.702
		1762	0.003				
4	2	7930	0.020	0.999	0.155	0.084	0.876
		70	0.011				
5	2	6199	0.005	0.058	0.0800	0.013	0.951
		1801	.0.003				
6	2	1121	0.024	0.940	0.141	0.000	0.924
		6879	0.060				
7	2	4601	0.002	0.200	0.150	0.000	0.918
		3399	0.210				
8	2	4182	0.001	0.646	0.001	0.000	0.999
		3818	0.007				
9	2	7283	0.034	0.584	0.046	0.000	0.976
		717	0.002				
10	2	1517	0.113	0.356	0.061	1.000	0.638
		6483	0.054				
11	2	1517	0.113	0.398	0.061	0.052	0.943
		6483	0.054				
12	2	1517	0.113	0.398	0.061	0.011	0.963
		6483	0.054				
13	2	1517	0.113	0.398	0.061	0.018	0.959
		6483	0.054				
14	2	1517	0.113	0.398	0.061	0.009	0.964
		6483	0.054				
15	2	4207	0.007	0.030	0.002	0.000	0.999
		3793	0.027				
16	2	4207	0.007	0.030	0.002	0.002	0.998
		3793	0.027				
						AVERAGE	**0.921187**

Table 5.12. Performance of the k-Means Clustering Algorithm on Unconditioned Data using Euclidean Distance

Euclidean Distance: Z-Normalized Data

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	6479	0.028	0.976	0.002	0.000	0.998
		1521	.034				
2	2	1083	0.022	0.872	0.000	0.000	1.000
		6917	0.029				
3	2	402	0.054	0.007	0.086	0.000	0.955
		7598	0.021				
4	2	7999	0.026	0.881	0.094	0.000	0.950
		1	0.000				
5	2	1668	0.018	0.034	0.131	0.000	0.929
		6332	0.027				
6	2	6859	0.041	0.076	0.000	0.000	1.000
		1141	0.065				
7	2	4578	0.052	0.125	0.000	0.000	1.000
		3422	0.095				
8	2	3818	0.035	0.060	0.000	0.000	1.00
		4182	0.026				
9	2	2447	0.064	0.073	0.000	0.000	1.000
		5553	0.066				
10	2	1917	0.039	0.074	0.017	0.991	0.661
		6083	0.029				
11	2	1917	0.039	0.074	0.017	0.029	0.977
		6083	0.029				
12	2	1917	0.039	0.074	0.017	0.000	0.991
		6083	0.029				
13	2	1917	0.039	0.074	0.017	0.000	0.991
		6083	0.029				
14	2	1917	0.039	0.074	0.017	0.000	0.991
		6083	0.029				
15	2	1333	0.023	0.038	0.034	0.000	0.982
		6667	0.026				
16	2	1333	0.023	0.038	0.034	0.000	0.982
		6667	0.026				
						AVERAGE	**0.962937**

Table 5.13. Performance of the k-Means Clustering Algorithm on Z-Normalized Data using Euclidean Distance

5.2.4 Principal Direction Divisive Partitioning

Unconditioned Data

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	7350	0.015	0.985	0.105	0.258	0.831
		650	0.002				
2	2	1083	0.059	0.925	0.171	0.073	0.871
		6917	0.001				
3	2	7978	0.019	1.00	0.863	.162	0.210
		22	0.010				
4	2	7930	0.003	0.999	0.169	0.035	0.890
		70	0.001				
5	2	7989	0.020	1.000	0.028	0.008	0.981
		11	0.011				
6	2	1121	0.030	0.082	.999	0.062	0.001
		6879	0.042				
7	2	4601	0.050	0.126	0.642	0.000	0.527
		3399	0.100				
8	2	3818	0.035	0.060	0.999	0.000	0.001
		4812	0.026				
9	2	717	0.002	0.584	0 .107	0.000	0.943
		7283	0.034				
10	2	5150	0.018	0.296	0.046	1.000.	0.645
		2490	0.166				
11	2	5150	0.018	0.296	0.046	0.853	0.679
		2490	0.166				
12	2	5150	0.018	0.296	0.046	0.011	0.970
		2490	0.166				
13	2	5150	0.018	0.296	0.046	1.000	0.645
		2490	0.166				
14	2	5150	0.018	0.296	0.046	0.009	0.971
		2490	0.166				
15	2	6904	0.028	0.030	0.016	0.000	0.991
		1096	0.023				
16	2	6904	0.028	0.030	0.016	0.002	0.990
		1096	0.023				
						AVERAGE	**0.696625**

Table 5.14. Performance of the PDDP Clustering Algorithm on Unconditioned Data

Z-Normalized Data

Attack Number	Clusters	Cluster Size	Intra Cluster Distance	Inter Cluster Distance	False Negatives	False Positives	*F-Measure*
1	2	6786	0.020	0.530	0.004	0.000	0.997
		1214	0.068				
2	2	4837	0.027	0.035	0.000	0.000	1.00
		3163	0.035				
3	2	5229	0.027	0.022	0.967	0.000	0.063
		2771	0.019				
4	2	2130	0.041	0.035	0.061	0.000	0.996
		5870	0.023				
5	2	4674	0.020	0.034	0.000	0.067	0.967
		3326	0.033				
6	2	6004	0.043	0.066	0.000	0.000	1.000
		1996	0.071				
7	2	3150	0.073	0.107	0.000	0.000	1.000
		4850	0.078				
8	2	3818	0.035	0.060	.999	0.000	0.001
		4182	0.026				
9	2	1276	0.134	0.130	0.028	0.000	0.985
		6724	0.043				
10	2	6032	0.026	0.077	0.016	0.981	0.663
		1968	0.049				
11	2	6032	0.026	0.077	0.016	0.029	0.977
		1968	0.049				
12	2	6032	0.026	0.077	0.016	0.000	0.991
		1968	0.049				
13	2	6032	0.026	0.077	0.016	0.000	0.991
		1968	0.049				
14	2	6032	0.026	0.077	0.016	0.000	0.991
		1968	0.049				
15	2	1000	0.023	0.030	0.000	0.000	1.000
		7000	0.028				
16	2	1000	0.023	0.030	0.000	0.011	0.994
		7000	0.028				
						AVERAGE	**0.854331**

Table 5.15. Performance of the PDDP Clustering Algorithm on Z-Normal Data

5.2.5 Results

We used both z-normalized and unconditioned data with the PDDP, k-Means, and FCMdd clustering algorithms. Z-normalization improved the performance of each clustering algorithm when compared to unconditioned data. The *F-Measure*s of the PDDP algorithm increased from .696626 to .854331; k-Means went from .921187 to .962937; the FCMdd went from .932687 to .978812. These results are using the Euclidean distance for the FCMdd and k-Means algorithms (PDDP does not use a distance metric).

Comparing the performance between the k-Means and FCMdd clustering algorithms when using the Mahalanobis metric, the FCMdd algorithm was the better of the two scoring .982249, to the k-Means' score of .966312.

The FCMdd using the Mahalanobis metric was the optimal performer with a score of .982249, followed by the FCMdd using the Euclidean distance on z-normalized data with a score of .978812. The k-Means algorithm using the Mahalanobis metric ranked third with a score of .966312.

Figure 5.3 graphs the performance of the FCMdd clustering algorithm when analyzing attack data. Figure 5.4 graphs the performance of the k-Means clustering algorithm. Figure 5.5 graphs the performance of the PDDP clustering algorithm. Figure 5.6 is a graph comparing the performance of the best of each clustering algorithm.

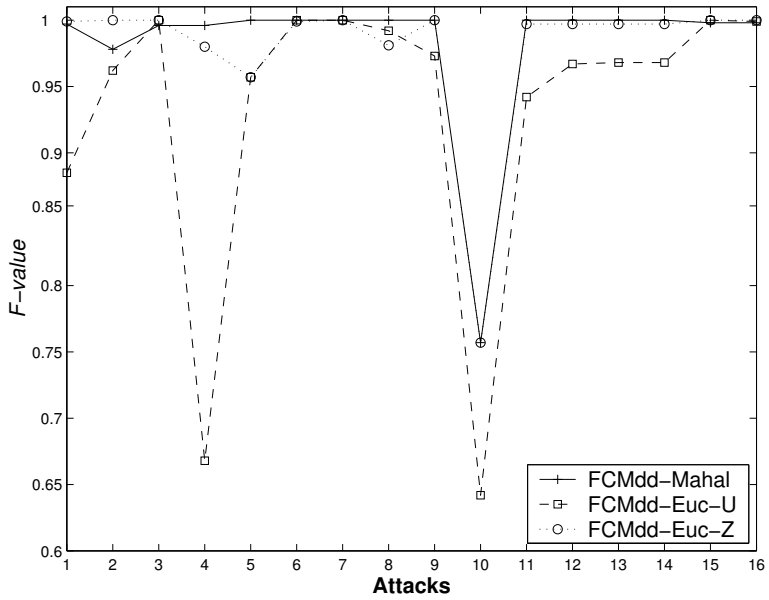


FIG. 5.3. Performance Comparison of the FCMdd Clustering Algorithm: Mahalanobis Distance, Euclidean Distance with Un-normalized Data, and Euclidean Distance with Z-normalized Data

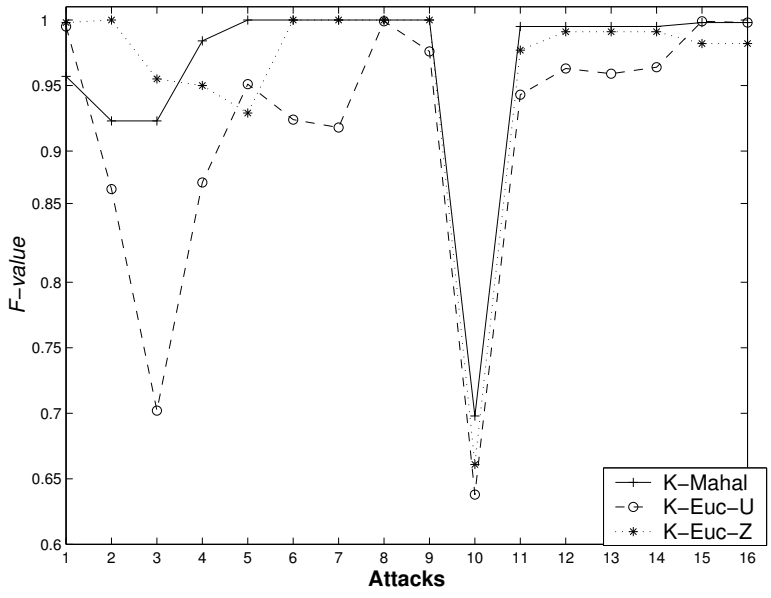


FIG. 5.4. Performance Comparison of the k-Means Clustering Algorithm: Mahalanobis Distance, Euclidean Distance with Un-normalized Data, and Euclidean Distance with Z-normalized Data

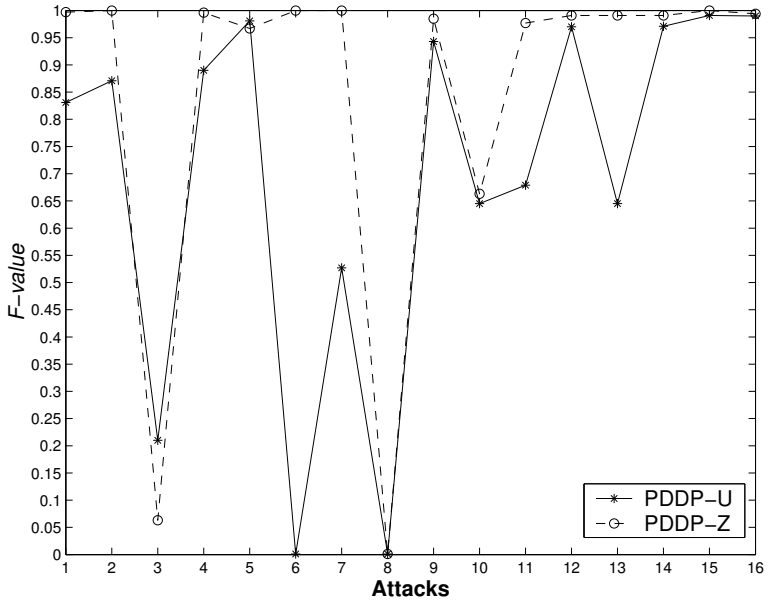


FIG. 5.5. Performance Comparison of the PDDP Clustering Algorithm: Un-normalized Data and Z-normalized Data

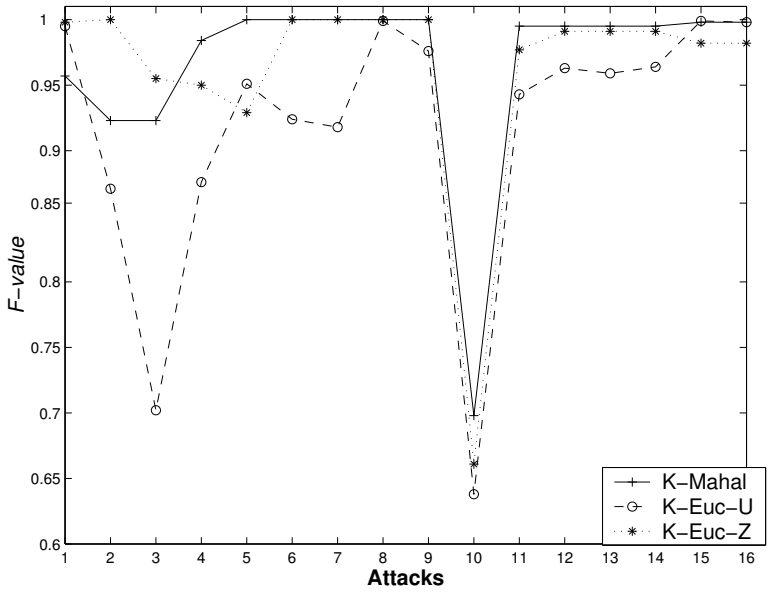


FIG. 5.6. Performance Comparison of the Best of each Clustering Algorithm: FCMdd with Mahalanobis Distance, k-Means with Mahalanobis, and PDDP with Euclidean Distance of Z-normal Data

5.3 Chapter Conclusions

Our results clearly indicate that the FCMdd clustering algorithm, using the Mahalanobis metric, provided the optimal model of the normal state system. We used this model for the remainder of our work.

This concludes the first of our two stage process where our goal was to test subsequent feature vectors that were taken when the system was operating in a quiescent state for conformance to the model and feature vectors that were taken when the system was under attack for non-conformance to the model.

Our model was not able to detect the majority of the stealthy port scans hence they were counted as false positives. This is not to say, however, that the port scan went undetected. We scanned the targeted system for approximately 35 minutes (long enough to take 2,000 samples from the network interface at .5 second intervals). We did detect scanning activity during 37.6% of that time. Although our reasoning process was only able to confirm 50% for these, reducing the detection rate to 18.8% for this type of activity, we were able to confirm that the system was scanned.

The false negatives (good data deemed bad) and true negatives (bad data deemed bad) are passed from the first to the second phase of our process, where we used them in subsequent experiments to evaluate our ontology and reasoning process.

In Chapter 6 which follows, we stepped back and took a macro view of security by analyzing over 4,000 types of attacks and intrusions. Our analysis provided the foundation for our ontology, which is detailed in Chapter 7.

Chapter 6

Target Centric Attack Classification: An Empirical Analysis

Allen et al. [1] and McHugh [83] assert that the characterization of intrusive behavior has typically been from the attacker's point of view, each suggesting that alternative taxonomies need to be developed. Allen et al. also state that intrusion detection is an immature discipline and has yet to establish a commonly accepted classification framework.

We have reviewed the existing taxonomies for intrusion detection and have found them to be inadequate for our purposes. To remedy this situation, we have conducted an empirical analysis of over 4,000 computer attacks and their corresponding attack strategies and have developed a *target-centric* taxonomy.

6.1 The Goal of Taxonomic Classification

A *taxonomy* may be broadly defined as a classification system with a systematic arrangement into groups or categories according to established criteria. Glass and Vessey [41] contend that taxonomies provide a set of unifying constructs so that the area of interest can be systemically described and aspects of relevance may be interpreted. The overarching goal of any taxonomy, therefore, is to supply some predictive value during the analysis of an unknown specimen.

According to Simpson [106] classifications may be created either *a priori* or *a poste-*

riori. An *a priori* classification is created non-empirically whereas an *a posteriori* classification is created by empirical evidence derived from some data set. Simpson defines a taxonomic character as a feature, attribute or characteristic that is divisible into at least two contrasting states and used for constructing classifications. He further states that taxonomic characters should be observable solely from the object in question.

To be effective, taxonomies for intrusion detection must provide criteria that are limited to the attributes and properties of an attack as they are experienced by the target. Failing to do so will not only prevent an IDS from effectively classifying and categorizing the attack, but will also impede the IDS from recognizing the intrusive event.

6.2 Existing Classification Schemes

Howard [50] provides a *"Complete Computer and Network Attack Taxonomy"*, that include the categories *Attackers, Tools, Access, Results and Objectives*. Howard classifies attackers as *Hackers, Spies, Terrorists* and *Teenagers* with objectives that include *Political Gain* and *Financial Gain*. However accurate he may be regarding an attacker, his tools and his motives, these characteristics are not discernible by analyzing an instance of the intrusive event. Specifically, an IDS does not have the means to know whether an attacker is a terrorist or a teenager or if the attacker's objective is financial gain or curiosity.

During the 1998 and 1999 DARPA Off Line Intrusion Detection System Evaluations [48, 77], Weber [118] and Kendall [62] provided taxonomies classified by *Initial Privilege Level, Method of Transition to a New Privilege Level* and *New Privilege Level*. Kendall includes *Social Engineering* [1] in the *Method of Transition* category. Since detecting off-line human interaction is beyond the scope of an IDS, that specific taxonomic character is not discernible by objectively observing the attack. Weber's taxonomy defines the category *Consequence*, with sub-categories of *Denial of Service*, *Remote to Local*, *User to Root* and

[1] Social engineering is a term that describes a non-technical kind of intrusion that relies heavily on human interaction and often involves tricking other people into breaking normal security procedures. A social engineer runs what used to be called a "con game".

Probe. We have incorporated these classifications into our work.

Lindqvist and Jonsson [75] state that they *"focus on the external observations of attacks and breaches which the system owner can make"*, consequently, they create a taxonomy in terms of *intrusion techniques* and *intrusion results*. Their categories of intrusion techniques are: *Bypassing Intended Controls, Active Misuse of Resources* and *Passive Misuse of Resources* and their categories of intrusion results are: *Exposure, Denial of Service and Erroneous Output*. They provide two examples of passive misuse of resources – *"automated searching using a personal tool"* and *"automated searching using a publicly available tool"*. These taxonomic characters are not discernible by objective observation of an attack because knowledge of an attack tool's origin is beyond the scope of an IDS.

In their *"Taxonomy of Security Faults"*, which defines a classification scheme for security faults in the Unix operating system, Aslam et. al [4] group vulnerabilities according to *Emergent Faults, Environment Faults, Coding Faults* and *Other Faults*. They define *Coding Faults* as faults introduced during software development that include errors in programming logic, missing or incorrect requirements, and design errors. Alsam's work largely rests upon Landwehr et al.'s *"A Taxonomy of Computer Security Flaws with Examples"* [72].

Landwehr et al. developed a taxonomy that was meant to be used during the software development process to enhance application security. Their taxonomy is categorized according to genesis (how), time of introduction (when) and location (where). In their work, the authors use the term *"flaw"* as a synonym for *"software bug"*. The presupposition of their paper is that software errors produce incorrect results that cause security failures. Although Landwehr et al.'s taxonomy is not directly mappable to an IDS, it does underscore the notion that many potential faults and vulnerabilities are intrinsic to the software development process.

In a recent paper, McHugh et al. [84] characterize two competing attack perspectives — the target's view and the attacker's view. They state that these views focus on the following manifestations:

Victim View

- What happened?
- Who and what is affected?
- Who is the intruder?
- Where and when did the intrusion originate?
- How and why did the intrusion happen?

Attacker View

- What is my objective?
- What vulnerabilities exist in the target system?
- What damage or other consequences are likely?
- What exploit scripts or other attack tools are available?
- What is my risk of exposure?

These perspectives each require two different sets of observables and measureables. If an IDS were to attempt to incorporate the "Attacker View", its rule set, or programmatic logic, will lack sufficient information to answer these interrogatories.

We advocate using the victim's view in IDS research. Accordingly, we focus on the effectors (data) of an attack, observing that an attack consists of:

- Input directed to a system component (how and why did the intrusion happen).
- The input was received from an attached tty or via the network (where and when did the intrusion originate).
- The input served as an agent of change, causing an aberrant condition (how did the intrusion happen).

- The aberrant condition resulted in some unintended consequence (who and what is affected).

The intermediate goal of our analysis, therefore, is to to classify an attack according to: the system component to which the input was directed; the causal effects of the input; and the consequences of the attack. The final result of our analysis is to identify the observable properties of an attack (i.e. in terms of the features and attributes that we detailed in Chapter 4) and the relationships that hold between them during an attack.

6.3 Empirical Analysis

Our analysis used data contained in the *CERT/CC Advisories* maintained by the "Computer Emergency Response Team/Coordination Center" of Carnegie Mellon University's Software Engineering Institute and the *"Internet Catalog of Assailable Technologies"* (ICAT) maintained by the National Institute of Standards and Technology. Both provide a catalog of known computer attacks, vulnerabilities and exploits.

CERT obtains its data from reports of adverse computer incidents. After a forensic examination of the adverse incident, and providing that the incident has wide spread impact, CERT posts an advisory. We used 280 of the 286 advisories that CERT has issued since its inception in 1985. The six that we did not use pertained to attacks against peripheral devices. The format of a CERT advisory is shown in Appendix B.

ICAT is a collection derived from multiple sources, to include but not limited to: CERT, Internet Security Systems (ISS), Bugtraq, Microsoft and Security Focus. Currently, it contains 4,160 entries and is classified according to severity, loss type, vulnerability type, exposed system component, etc. Its format is shown in Appendix A. The ICAT Meta-base classification scheme is not mutually exclusive. For example, the ICAT Meta-base lists the exposed component of the *Land* [2] attack as both the network protocol stack and the operating system as well as stating that multiple vulnerabilities are responsible for enabling

[2]The *Land* attack is an IP Denial of Service Attack where a SYN packet in which the source address and port are the same as the destination address and port.

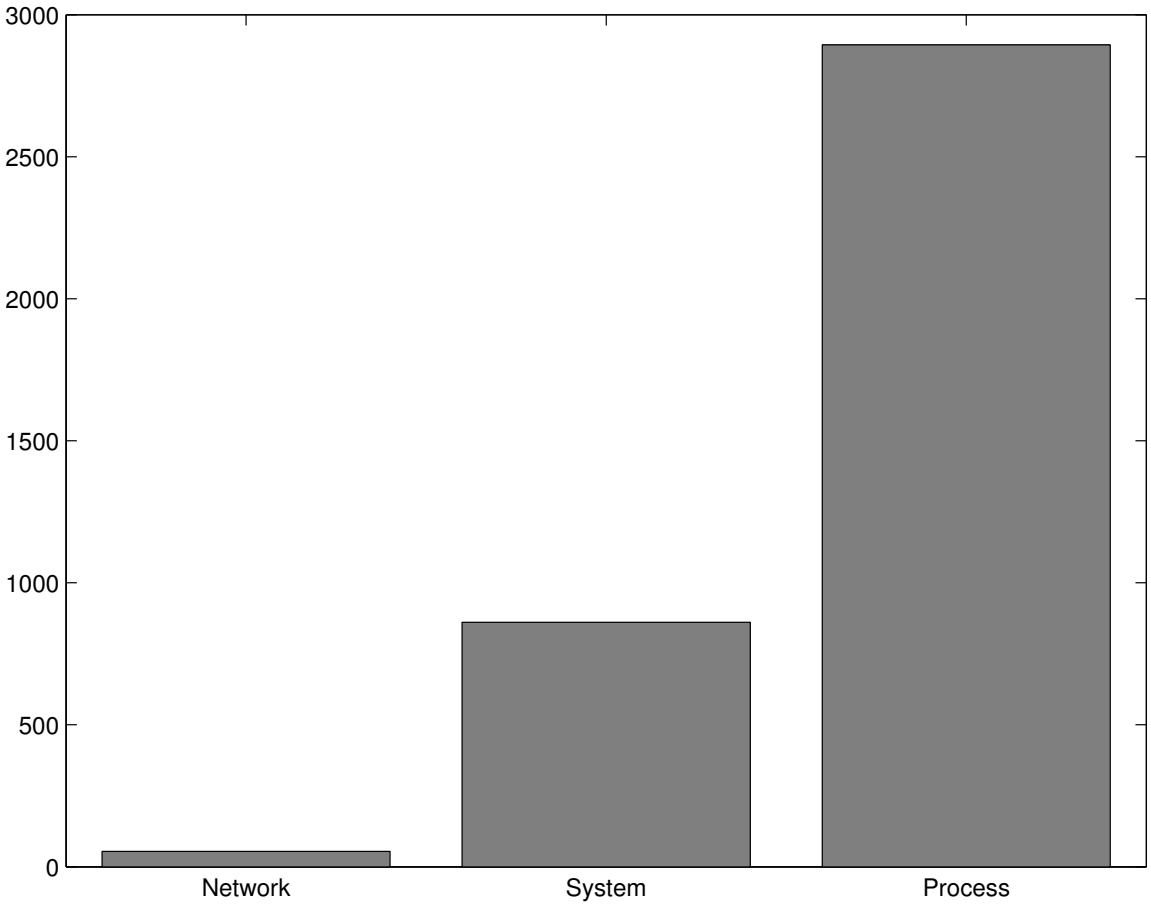
the Land attack: “Input Validation Error”, “Buffer Overflow”, “Boundary Overflow” and an “Exceptional Condition Handling Error”. CERT provides a more accurate description, stating that the Land attack is directed against the network protocol stack and results in an input validation error. We reclassified many of the ICAT Meta-base entries to ensure that each subcategory was mutually exclusive and non-ambiguous. This yielded 3,809 entries that were used for our analysis.

We examined both the CERT and ICAT data to ensure thoroughness and completeness, and we compared the results of each to test for continuity between the two data sets.

We begin our analysis by plotting the incidence of attacks according to the system component (process, system or network) that was targeted. We then plot the *means, consequences* and *location* of attack against each of the system components. The definitions of means, consequence, and location will be given as this chapter progresses.

6.3.1 Incidence of Attack Against System Components

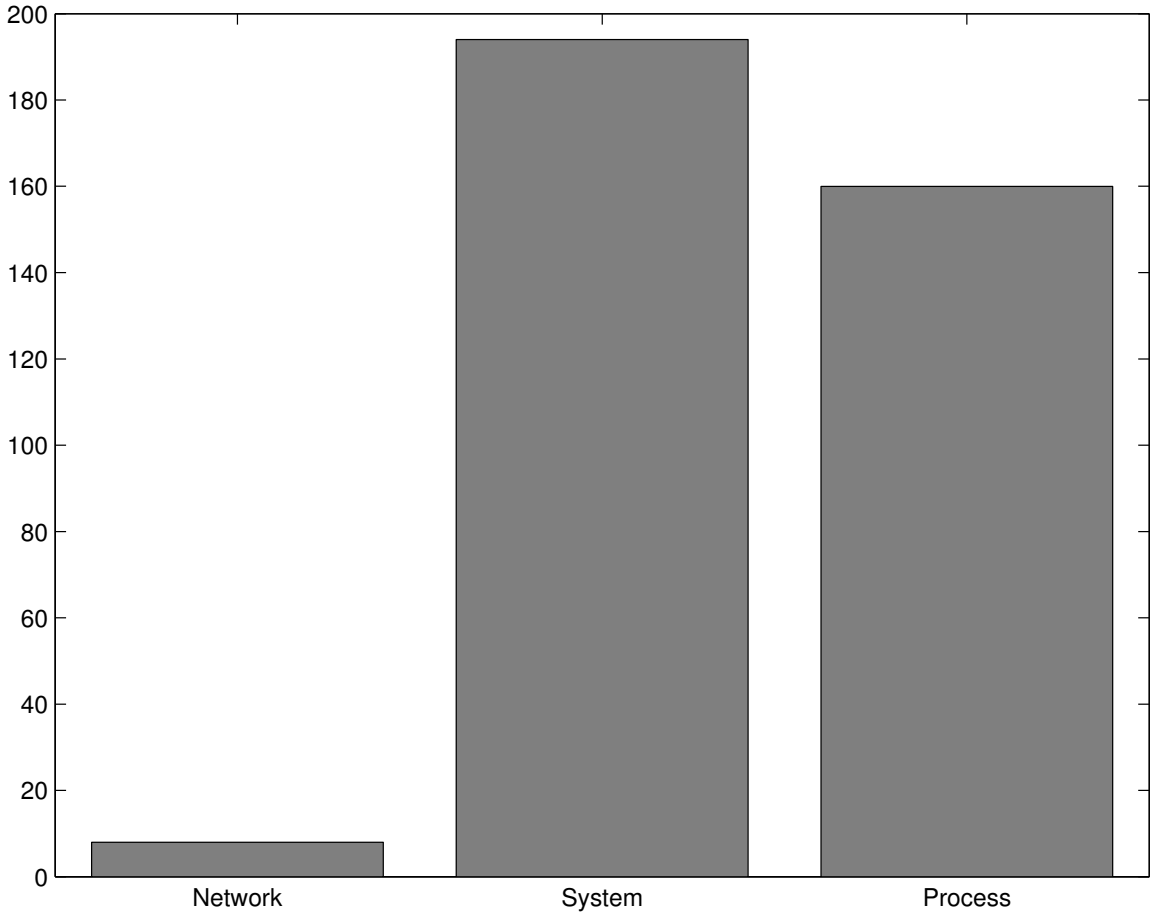
Figure 6.1 illustrates the distribution of the system components targeted according to the ICAT Meta-base. The distribution of attacks comparison indicates that processes are the most frequently targeted system component. According to the ICAT data, the processes that are most frequently targeted run at the *root* level, and include web servers, mail servers and core system binaries (e.g.: login, inetd, etc.).



Component	Number of Attacks
Network	54
System	861
Process	2894

FIG. 6.1. ICAT: System Component Most Frequently Targeted

Likewise, Figure 6.2 illustrates the distribution of system components targeted as counted by the CERT advisories. According to the CERT data, the "system" includes core system binaries such as login, inetd, and xinetd. This is in contrast to ICAT, where they are counted as "stand-alone" processes. The CERT data indicates that the system is the most often attacked, followed closely by processes.



Component	Number of Attacks
Network	8
System	194
Process	160

FIG. 6.2. CERT: System Component Most Frequently Targeted

6.3.2 Means of Attack

We define *Means of Attack* as the immediate system reaction to input. Typically, the input exploits a software flaw or it is not type checked to verify conformance to an anticipated format. The following categories, compiled from [14, 15, 19, 85], describe the type of input that are reportedly used as a means of attack.

i. Input Validation Error. An input validation error exists if input is received by a software component and it is not properly bounded or checked. This class is further sub-divided as:

   (a) Buffer Overflow. A buffer overflow results from an overflow of a static-sized data structure. Typically, this is a temporary data structure that is located on the process' stack.

   (b) Boundary Condition Error. This type of error occurs when a process attempts to read or write beyond a valid address boundary ,or a system resource is exhausted. An error occurs when the condition is not caught.

   (c) Malformed Input. A process accepts syntactically incorrect input, extraneous input, or the process lacks the ability to handle field-value correlation errors.

ii. Logic Exploit. Logic exploits occur when race conditions or unanticipated states are induced. Logic exploits are further categorized as follows:

   (a) Exception Condition. An error resulting from the failure to catch and handle a run time error. Examples include failing to catch erroneous results generated by an arithmetic operation that requires positive input (e.g.: log base 2), attempting to divide by zero, or an operation that results in infinity.

   (b) Race Condition. An error occurring during a timing window between two operations. They usually result from the use of a shared variable that is not locked.

(c) Serialization Error. An error that results from the improper serialization of operations. For example, a process expects A to occur before B, but instead B occurs before A and this is not caught.

(d) Atomicity Error. An error occurring when a partially-modified data structure is used by another process. For example, reading or writing pipe data is atomic if the size of the data is less than the size of the pipe. If the data is larger than the size of the pipe, an error occurs if it is not caught and handled.

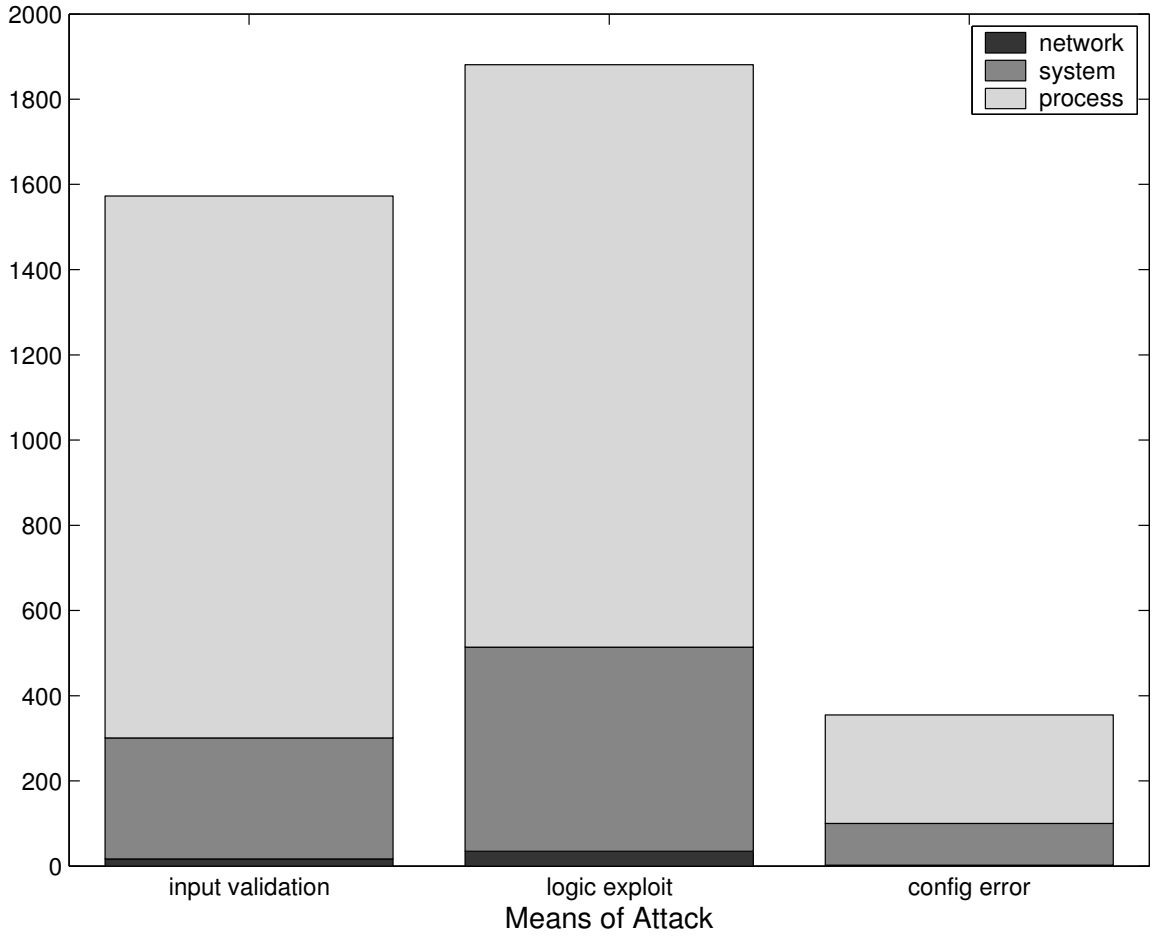
iii. Configuration Error. A configuration error results when user controllable settings in a system are set such that the system is vulnerable. This vulnerability is not due to how the system was designed but on how the end user configures and uses the system. These may be further categorized as follows:

(a) An error that results from a system utility being installed with incorrect parameters.

(b) An error that occurs when a system utility is installed in the wrong location.

(c) An error that occurs when access permissions allow a system utility to violate a security policy.

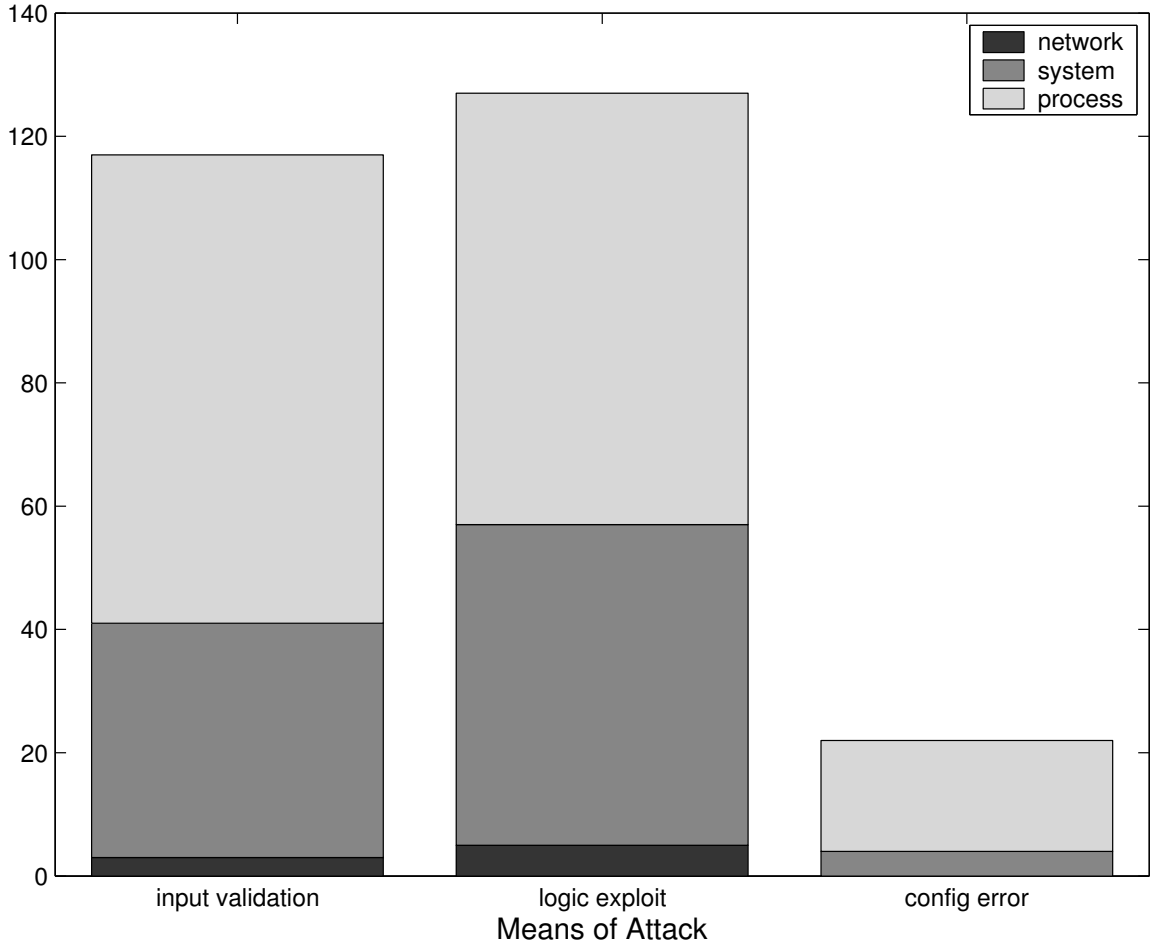
Figure 6.3 illustrates the means of attack against each of the system components as reflected in the ICAT data. The ICAT shows that logic exploits are the principal means of attack. The data also shows that processes are overwhelmingly the target of the attack. Logic exploits account for 50.37% of the attack strategies overall and 33.76% of the attack strategies directed against processes. Input validation errors are a close second and account for 31.42% of the attack strategies directed against processes.



Component	Input Valid	Exploit	Config
Network	17	35	2
System	284	479	98
Process	1272	1367	255

FIG. 6.3. ICAT: Means of Attack

Figure 6.4 illustrates the means of attack according to CERT's data. Like the ICAT data, CERT shows that the primary means of attack is also logic exploits and that input validation errors are a close second.



Component	Input Valid	Exploit	Config
Network	3	5	0
System	38	52	4
Process	74	68	18

FIG. 6.4. CERT: Means of Attack

6.3.3 Consequences of Attack

The consequence is the final result of an attack. For example, a *Land* attack directed against the network protocol stack induces an input validation error, which, until the protocol stack was made more robust, resulted in a *Denial of Service*. Another example is a *Buffer Overflow* attack directed against an HTTP server that results in the attacker gaining *Unauthorized Root Access*. We have categorized the consequences of attack as "denial of service", "unauthorized user access", "unauthorized root access", "loss of confidentiality" and "probe". These categories are defined as follows:

i. Denial of Service. The attack results in the system being placed into an unstable state or all of the system resources being consumed by meaningless functions.

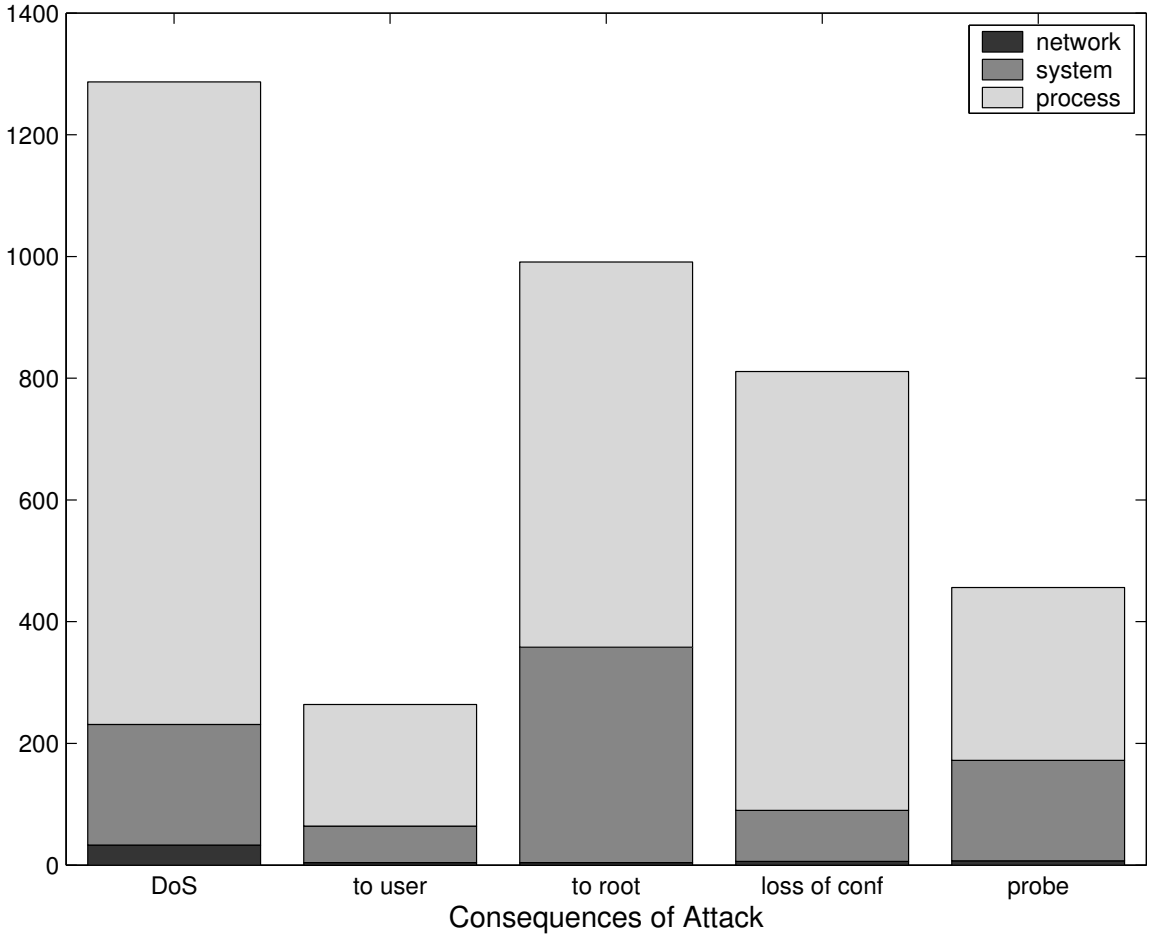
ii. Unauthorized User Access. The attack results in the attacker having access to services on the target system at a privilege level that is equivalent to an ordinary user.

iii. Unauthorized Root Access. The attack results in the attacker being granted privileged access to the system, consequently having complete control of the system.

iv. Loss of Confidentiality. The attack results in an information leak from the system. This does not include the loss of information as a result of unauthorized root or user access.

v. Probe. This type of an attack results in the disclosure of the system's profile.

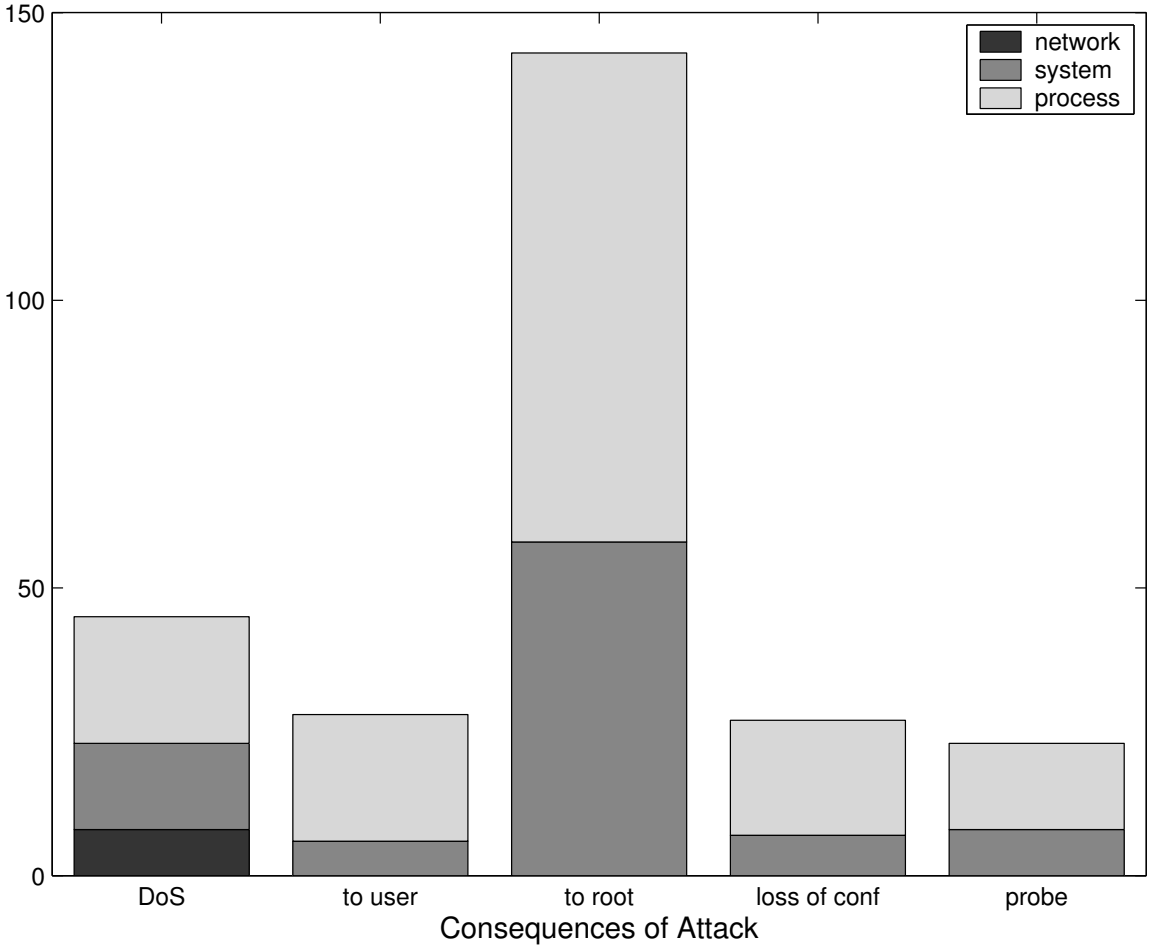
Figure 6.5 illustrates the consequences of the attack according to the ICAT data. This data shows that denial of service is the most likely consequence of an attack, followed by unauthorized root (superuser) access.



Component	DoS	User	Root	Conf	Probe
Network	33	4	4	6	7
System	198	60	354	84	165
Process	1056	200	633	721	284

FIG. 6.5. ICAT: Consequence of Attack

Figure 6.6 illustrates the consequences of attack according to the CERT data. The CERT data shows unauthorized root access as the most likely consequence. The disagreement between the two data sets is attributable to the selection process of each. The ICAT data set contains information regarding all types of attacks, whereas CERT only publishes a security alert for attacks that are widespread and of serious consequence.

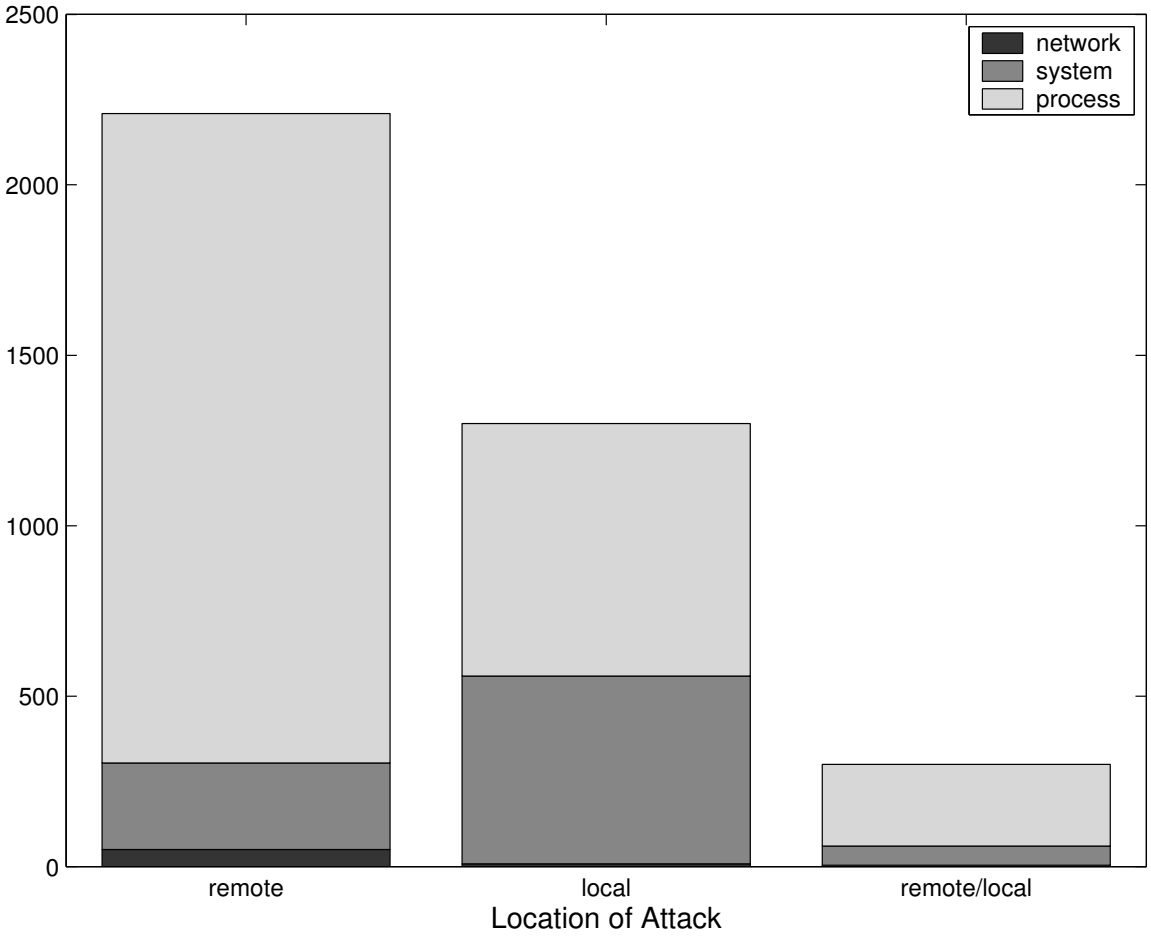


Component	DoS	User	Root	Conf	Probe
Network	8	0	0	0	0
System	15	6	56	7	8
Process	22	22	83	20	15

FIG. 6.6. CERT: Consequence of Attack

6.3.4 Location of Attack

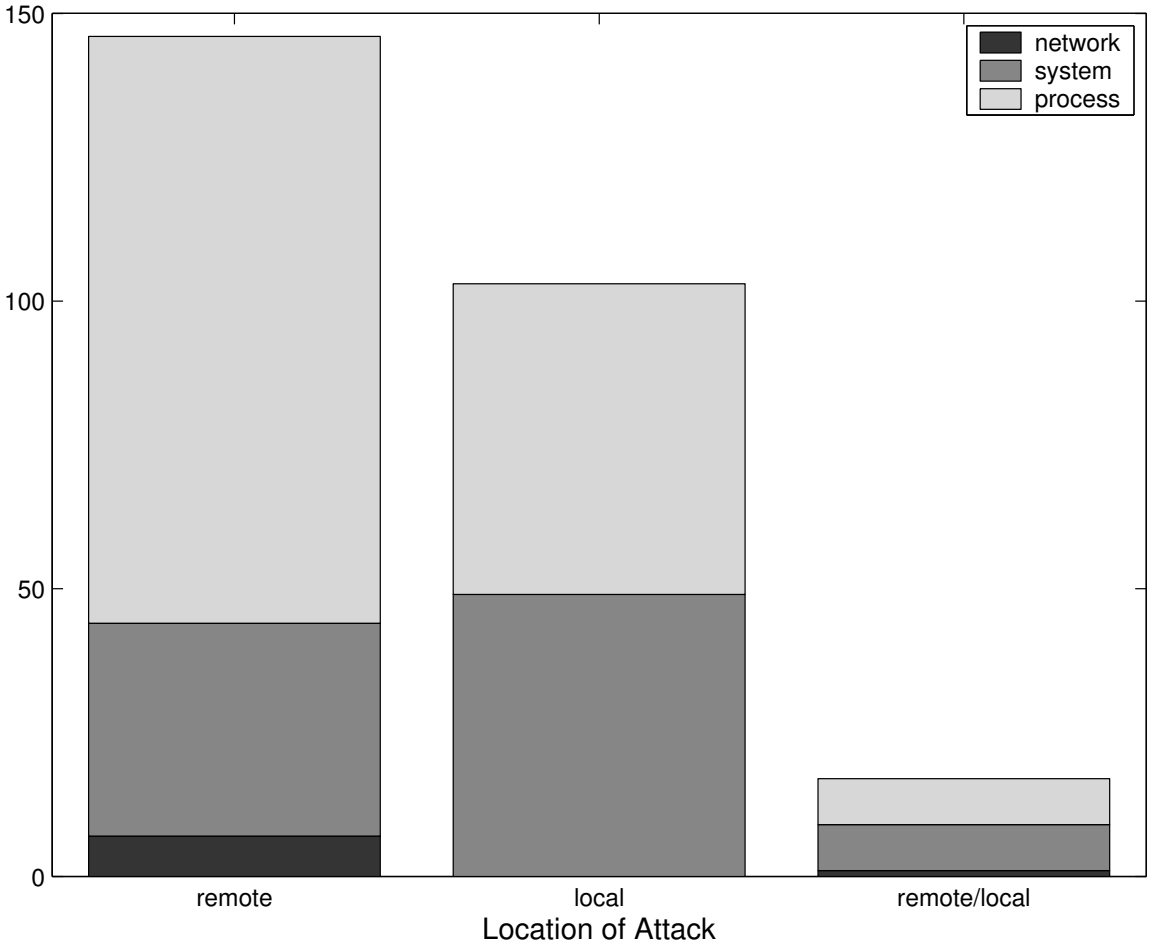
Attacks are possible because the attacker has access to the target, either directly from a terminal or via a network connection. Figure 6.7 illustrates the distribution of locations of attack according to the ICAT data. The data indicates that nearly twice as many attacks are carried out via a network connection. The data also shows that most attacks that target the operating system occur locally.



Component	Remote	Local	Remote/Local
Network	50	8	4
System	254	551	56
Process	1905	741	240

FIG. 6.7. ICAT: Location of Attack

Figure 6.8 illustrates the distribution of locations of attack according to the CERT data. Both CERT and ICAT agree that most attacks are carried out remotely and that attacks targeting the operating system are usually conducted from a local terminal.



Component	Remote	Local	Remote/Local
Network	7	0	1
System	36	48	8
Process	101	53	8

FIG. 6.8. CERT: Location of Attack

6.4 Results

Our analysis reveals that exploiting a software flaw in a network connected process is the most common means of an attack. According to the CERT data, *root* access is the most common consequence of an exploited vulnerability, while the ICAT data shows that the most frequent consequence is a denial of service. As previously stated, this discrepancy (root access vs. denial of service) is most likely the result of CERT issuing advisories only in cases where the vulnerabilities have great and widespread consequence, with root access being the gravest of consequences.

Both the ICAT and CERT data show that attacks against the system are the second most common. They are effected by a logic exploit, resulting in the attacker gaining root access, and are carried out locally. This means that the attacker has physical access to the machine – an "insider".

Recall Gartner's recommendation that enterprises redirect money earmarked for intrusion detection to barrier technologies. Although network attached processes are the most frequent targets of an attack, the data shows that attacks against the system are the most consequential and they occur locally via a terminal not via the network.

6.5 Chapter Conclusions

Our analysis indicated that the overwhelming majority of attacks are carried out by directing malformed input to a network attached process to exploit a software vulnerability. It also indicated that the most consequential attacks are effected by "insiders" who carry out an attack via an attached terminal. This implies that Network Based ID will not detect the most serious attacks.

We have classified attacks according to *Targeted Component, Means, Consequence,* and *Location of Attacker*. We have also defined subclasses for the Means, Consequence and Location categories. These class and subclasses will be incorporated into the *Target Centric Ontology* defined in Chapter 7.

Although we will present the case for migrating from taxonomies to ontologies, our intent is not to criticize the use of taxonomies. To the contrary, they have served their purpose well, particularly in identifying and classifying the characteristics of computer attacks and intrusions. We do, however, advocate leveraging their work by building upon existing taxonomies and transitioning to ontologies. We feel that this is necessary and warranted because, according to Staab and Maedche [102], taxonomies do not contain the necessary *meta-knowledge* required to convey modeling primitives such as concepts, relations and axioms that are required to make sense of and operate on specific objects. Ontologies do.

Chapter 7

A Target-Centric Ontology for Intrusion Detection

We detail the second phase of our intrusion detection process in this chapter. During the first phase we modeled the quiescent state of the system and tested subsequent samples of kernel data (process, system, and network) for conformance to the model. The purpose of the second phase is to take the nonconforming data from the first phase as input and classify it according to the type of intrusion or attack that it represents. The second phase also provides an orthogonal test meant to reduce the number of false alarms.

We define our data model of intrusive behaviors as an ontology, a term borrowed from philosophy. Ontologies provide formal specifications of the concepts and relationships that exist between entities within a domain of discourse, and are intended to facilitate knowledge sharing.

We present the benefits of transitioning from taxonomies to ontologies and ontology specification languages by comparing and contrasting *XML*, the syntactic language employed by the *Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition* (IDMEF)[21], which is an Internet Engineering Task Force emerging standard, to *DAML+OIL* [1] [54], an ontology specification language. Commenting on the IDMEF and its ability to enable interoperability between heterogeneous

[1] DAML+OIL is being replaced by the Web Ontology Language (OWL) [82], a W3C standard.

IDSs, Kemmerer and Vigna [61] state that the IDMEF is only a first step, however, additional effort is needed to provide a common ontology that allows IDSs to agree on what they observe.

We also demonstrate an ontology specification language's ability to simultaneously serve as an attack recognition, attack reporting and attack correlation language — much needed functionality in the realm of IDSs. We use our ontology in conjunction with the *Java Theorem Prover* (JTP) [35], a sound and complete *First Order Logic* (*FOL*) theorem prover, to reason over and classify the anomalous instances that failed to fit the model of quiescent behavior presented in Chapter 5.

7.1 Background

A central component of an IDS is the taxonomy employed to characterize and classify an attack or intrusion, and a language that describes instances of that taxonomy. The language is paramount to the effectiveness of the IDS because information regarding an attack needs to be intelligibly conveyed, especially in distributed environments, and acted upon. Several taxonomies have been proposed by the research community. Some include a descriptive language, but most do not. Likewise, several attack languages have been proposed, but most are not grounded in any particular taxonomy, hence their associated classification schemes are *ad hoc* and localized. The inherent problem with this approach is threefold:

i. Most attack and signature languages are particular to specific domains, environments and systems, consequently, they are not extensible, are incommunicable between non-homogeneous systems, and their semantics are vague and usually lack grounding in any formal system of logic.

ii. In order for a software system to operate over instances of a data model characterized by a taxonomy, the data model must be encoded within the software system. Any changes or updates to the data model necessitate a change to the software system.

iii. Taxonomies only provide schemata for classification. They do not contain the necessary and sufficient constructs needed to enable a software system to reason over an instance of the taxonomy. (Table 7.1 lists sufficient constructs.)

To mitigate the effects of theses problems, we advocate transitioning from taxonomies to ontologies. Ontologies, unlike taxonomies, provide powerful constructs that include machine interpretable definitions of the concepts within a domain and the relations between them. These constructs enable software systems to share a common understanding of the information at issue, in turn empowering them with a greater analytical capability. Gruber [45] defines an ontology as an explicit specification of a conceptualization. We view them as a formal specification of the concepts and relationships that can exist between entities within a domain of discourse. As we later illustrate, ontologies may be specified as a graph, a set on *n-triples* or by a semantic language such as the *Resource Description Framework Schema* (RDFS) [12] or (DAML+OIL) [54].

Semantic languages differ greatly from syntactic languages. Both types of languages employ tags that define a grammar, however, semantic languages have additional tags that support the language's semantic properties. Ontology representation languages are a type of semantic language and may be mapped into first-order relational sentences and a set of first-order logic axioms. This mapping restricts the allowable interpretations of the non-logical symbols (i.e., relations, functions, and constants), enabling instances of the ontology to be operated over using theorem provers and other reasoning systems. The ontology, in combination with a logic system, therefore, constitutes *knowledge representation*.

7.1.1 Intrusion Detection Languages

There are several *attack languages* proposed in the literature. These languages are often categorized as *Event, Response, Reporting, Correlation*, and *Recognition* Languages [31, 32]. We concentrate on correlation, reporting and recognition languages because an ontology representation language is able to simultaneously serve as all three.

A. P-BEST

The P-BEST Toolset [76] (Production-Based Expert System Toolset) is a correlation language that is used to specify the inference formula for reasoning and acting upon facts asserted into its fact base and from facts derived from external events.

The P-BEST toolset consists of a rule translator, a library of runtime routines and a set of garbage collection routines. To use P-BEST, rules and facts are written in the P-BEST *production rule specification language* by the end-user. The rules are then translated into a *C* language expert system program. The resulting expert system program may then be compiled into a stand-alone executable or a set of library routines that may then be linked into some other framework.

According to [31], the P-BEST language lacks concepts that are specific to event recognition and consists solely of a formalism for expressing probabilistic and linguistic rules.

B. STATL

STATL [32] is an extensible transition-based attack detection language specifically designed to support intrusion detection. The language allows one to describe computer penetrations as sequences of actions that an attacker performs in order to compromise a computer system. In STATL, scenarios are *attacker centric*. A STATL description of an attack scenario can be used by an intrusion detection system to analyze a stream of events and detect possible ongoing intrusions. This language provides constructs to represent an attack as a composition of states and transitions. The constructs are similar to those used in programming languages, describing conditional, sequential and iterative events. STATL, however, lacks the necessary constructs for combining sub-events into larger events.

C. LogWeaver

LogWeaver [43] is a log auditing tool that takes a system log as input and processes it

according to a signature (rule) file. The signature file defines the type of events that are to be monitored and reported on. LogWeaver is able to match regular expressions and make correlations between events provided that they are executed by the same user. LogWeaver employs logic that is based upon *model checking* [99].

LogWeaver does not include signatures, but rather defines a syntax and grammar for the end-user to use when writing signatures. Once written, a signature compiles into an *automaton*, which is a graph like structure composed of states and transitions between them.

D. CISL

The Common Intrusion Detection Framework (CIDF) [59] started as a DARPA initiative in 1998. CIDF was an effort to develop protocols and application programming interfaces to give IDS research projects the ability to share information and resources and to enable IDS component reuse by multiple systems. The CIDF framework is comprised of components which exchange data in the form of a *GIDO* (generalized intrusion detection object) which are represented in a standard format. This standard format is specified in the Common Intrusion Specification Language (CISL) [34], a reporting language. The CIDF effort lost inertia and many of its developers now work on the IETF's IDMEF, infusing IDMEF with some of CISL's concepts and notions.

According to [30] CISL provides a reasonably rich vocabulary for conveying the structure of concrete instances of a set of events involving networked computers. It does not provide a vocabulary for describing classes of such events and it lacks the facilities for representing ambiguity and nonexistence or negation.

E. IDMEF

The Internet Engineering Task Force's proposed *Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition* [21] is a profound effort to establish an industry wide data model which

defines computer intrusions. It defines a data model that is representative of data exported by an IDS. It also defines data formats and exchange procedures for inter/intra IDS exchanges. The data model is defined in an XML *Document Type Definition* and implemented in the Extensible Markup Language (XML) [114]. XML is a syntactic language.

Additionally, the IDMEF mandates a hierarchal configuration of three IDS components, *sensors, analyzers, and managers*. Sensors are located at the bottommost level of the hierarchy. Sensors output data to analyzers, which in turn report up to a manager, located at the topmost level of the hierarchy.

In the next section we present the case for migrating from taxonomies to ontologies. We compare and contrast the syntactic language XML, to the semantic language DAML+OIL, because the IDMEF data model, encoded in XML, is an emerging IETF standard.

7.2 Syntax versus Semantics

The IDMEF's principal shortcoming is its use of XML, which is limited to a syntactic representation of the data model. This is not an indictment of XML, which in fact serves its designer's intentions. This limitation, however, requires that each individual IDS interpret and implement the data model programmaticaly. This shortcoming may be mitigated by using an ontology representation language such as DAML+OIL.

Semantic languages like RDF-S and DAML+OIL, are descriptive logic languages that are grounded in both model-theoretic[2] and axiomatic semantics[3] and having been designed specifically for the Internet, they are able to:

i. Model the attributes and characteristics of a domain $\mathcal{D}$ using a language $\mathcal{L}$.

ii. Decouple the data model from the underlying system of computational logic.

[2]model-theoretic semantics is the process of constructing mathematical models of logical consequence and establishing when the model satisfies a formula

[3]axiomatic semantics is the process of defining a language using axioms and proof rules

Feature	**Description**	**DAML +OIL**	**XML**
bounded lists	Uses a first/rest structure to represent unordered bounded lists, with nil representing the end of the list.	Yes	No
cardinality constraints	minCardinality and maxCardinality	Yes	Yes
class expressions	Wherever a Class is referenced allows an expression involving *unionOf, disjointUnionOf,intersectionOf* or *complementOf*.	Yes	No
data types	e.g: numerical, temporal and string data types	Yes	Yes
defined classes	Allows new classes to be defined based on property values or other restrictions of an existing class.	Yes	No
enumerations	Allows specification of a restricted set of values for a given attribute to include *oneOf*	Yes	No
equivalence	Supports *equivalentTo* for classes, properties, and instances to support reasoning across ontologies and knowledge bases	Yes	No
extensibility	Allows new properties to used with existing classes.	Yes	No
formal semantics	Semantics have been expressed in both model-theoretic and axiomatic forms.	Yes	No
inheritance	Fully supports *subClassOf* and *subPropertyOf*	Yes	No
inference	Has constructs such as *TransitiveProperty, UnambiguousProperty, inverseOf*, and *disjointWith* for reasoning engines.	Yes	No
local restrictions	Allows restrictions to be associated with a Class/Property pairs.	Yes	No
qualified constraints	Allows expressions such as "all children of X are of type Y".	Yes	No
reification	Provides a standard mechanism for recording data sources, timestamps, etc., without intruding on the data model.	Yes	No

Table 7.1. Language Feature Comparison: DAML+OIL versus XML

iii. Report the existence of an instance of the domain (model) in a manner that is "comprehensible" by any entity that possesses the specific ontology.

iv. Aggregate and store specific instances of the domain in a knowledge base to enable the conclusion that some larger and more comprehensive instance of the ontology exists.

Table 7.1 provides a feature by feature comparison between DAML+OIL and XML. We use this comparison to highlight the differences between the constructs and functionality of the two language types.

7.3 From Taxonomies to Ontologies: *The case for ontologies*

According to Davis et al. [23], knowledge representation is a surrogate or substitute for an object under study. In turn, the surrogate enables an entity, such as a software system, to reason about the object. Knowledge representation is also a set of *ontological* commitments specifying the terms that describe the essence of the object, in other words, *meta-data* or data about data describing their relationships. According to Welty et al. [119], an ontology, at its deepest level, subsumes a taxonomy. Similarly, Noy and McGuinness [88] state that the process of developing an ontology includes arranging classes in a taxonomic hierarchy.

Ontologies, unlike taxonomies, provide powerful constructs that include machine interpretable definitions of the concepts within a specific domain and the relations between them. Ontologies not only provide IDSs with the ability to share a common understanding of the information at issue but also further enable IDSs with improved capacity to reason over and analyze instances of data representing an intrusion. Moreover, within an ontology, characteristics such as cardinality, range and exclusion may be specified and the notions of inheritance and multiple inheritance are supported. We are not dispensing with taxonomies. To the contrary, we are leveraging the classification schemes that they provide in the creation of our ontology.

The relationship among objects in a data model defined by an ontology may be highly complex. An ontology may be represented, for example, as a set of statements in an ontology specification language, a set of *n-triples* (Subject, Predicate and Object) or as a *Resource Description Framework* (RDF) graph. The precise definition of the RDF data model is defined as:

i. A set called *Resources*.

   (a) A subset of Resources called *Properties*

ii. A set called *Literals*.

iii. A set called *Statements*, where each element is a triple of the form: $\{$*sub, pred, obj* $\}$.

Where *pred* is a member of Properties, *sub* is a member of Resources, and *obj* is either a member of Resources or Literals.

Figure 7.1 shows the basic graphical representation of the RDF data model.

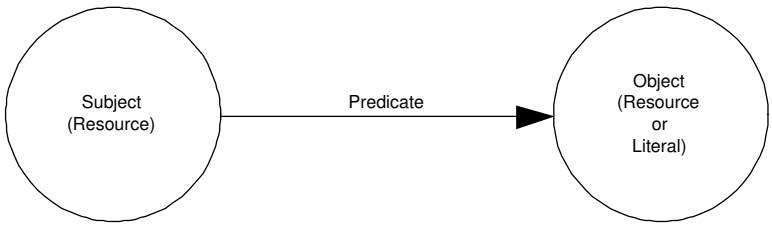


FIG. 7.1. Graph of the RDF Data Model

In applying ontologies to the problem of intrusion detection, the power and utility of the ontology is not realized by the simple representation of the attributes of the attack. Instead, the power and utility of the ontology is realized by the fact that we can express the relationships between collected data and use those attributes and their constructs, relationships, and saliency to deduce new and additional knowledge about attacks and intrusions. Moreover, specifying an ontological representation of the data model defining an intrusion decouples it from the logic of the IDS. This decoupling of the data model enables heterogeneous IDSs to share data without a prior agreement as to the semantics of the data. To effect this sharing, an instance of the ontology is shared between IDSs in the form of a set of DAML+OIL (or RDF) statements. If the recipient does not understand some aspect of the data, it obtains the ontology in order to interpret and use the data as intended by its originator.

The following two examples illustrate the benefits of ontologies. The first example is trivial and is used to show the differences between an ontology and a taxonomy. The second example, which is based on our experimentation, illustrates the deductive power of an ontology.

The class “family” (abbreviated for our purposes) is specified as a taxonomy using an XML Document Type Definition (DTD) and as an ontology using DAML+OIL. The DTD is depicted in Figure 7.2, and an instance of the DTD is illustrated in Figure 7.3.

```
<?xml version="1.0"?>
<!ELEMENT name (#PCDATA)>
<!ELEMENT son (#PCDATA)>
<!ELEMENT daughter (#PCDATA)>
<!ELEMENT sex (#PCDATA)>
<!ELEMENT mother (#PCDATA)>
<!ELEMENT person (name, son*, daughter*, sex?, mother?)>
<!ELEMENT family (person+)>
```

FIG. 7.2. DTD specification of a family

```
<family>
  <person>
     <name>Bob</name>
      <son>Jake</son>
      <daughter>Jane</daughter>
      <daughter>June</daughter>
      <sex>male</sex>
    </person>
  <person>
     <name>Bea</name>
  </person>
  <person>
     <name>Jake</name>
     <mother>Bea</mother>
  </person>
  <person>
       <name>Jane</name>
       <mother>Bea</mother>
  </person>
  <person>
        <name>June</name>
        <mother>Bea</mother>
  </person>
</family>
```

FIG. 7.3. Instance of the DTD specified Family

A rule-based expert system that operates over the DTD and the XML annotated instance could not answer the question "list all females and their female children". In order to correctly answer (the answer is Bea and her children Jane and June), domain knowledge regarding the nature of familial relationships is required. That required additional knowledge is not and cannot be encoded in a DTD.

Alternatively, the DAML specified "Family" depicted in Figure 7.4 imparts the requisite domain knowledge, specifically, that daughters and mothers are restricted to female persons and that there is a parent-child relationship between mothers and daughters. Given the ontology of Figure 7.4 and the data in Figure 7.3, it follows that since Jane and June are Bob's

daughters, they are female, and that Jane and June's mother is Bea, and because a mother is a female person, we may infer that Bea is a female with female children — namely Jane and June.

```
<daml:Ontology rdf:about= "" >
  <daml:versionInfo>$ Id: Family.daml $</daml:versionInfo>
     <rdfs:comment>
          An example ontology
     </rdfs:comment>
</daml:Ontology>

<daml:Class rdf:about="Family#Family" rdfs:label="Family">
</daml:Class>

<daml:Class rdf:about="Family#Person" rdfs:label="person">
</daml:Class>

<daml:Class rdf:about="Family#mother" rdfs:label="mother">
     <rdfs:subClassOf>
        <daml:Class rdf:about="Family#person">
     </rdfs:subClassOf>
     <rdfs:subClassOf>
        <daml:Restriction daml:sex="Family#female"/>
     </rdfs:subClassOf>
     <rdfs:subClassOf>
        <daml:Class rdf:about="Family#Family">
     </rdfs:subClassOf>
</daml:Class>

<daml:ObjectProperty rdf:about="Family#mother_of" rdfs:label="mother_of">
     <rdfs:domain rdfs:about="Family#mother">
     <rdfs:range rdf:"resource&daughter"/>
</daml: ObjectProperty>

<daml:ObjectProperty rdf:about="Family#name rdfs:label=name>
     <rdfs:domain rdfs:about="Family#person >
     <rdfs:range rdf:resource&Literal"/>
</daml: ObjectProperty>

<daml:ObjectProperty rdf:about="Family#sex rdfs:label=sex>
     <rdfs:domain rdfs:about="Family#person >
     <rdfs:range rdf:resource&Family#male"/>
     <rdfs:range rdf:resource&Family#female"/>
</daml: ObjectProperty>

<daml:Class rdf:about="Family#Daughter" rdfs:label="daughter">
     <rdfs:subClassOf>
     <daml:Class rdf:about="Family#child">
     </rdfs:subClassOf>
     <rdfs:subClassOf>
         <daml:Restriction daml:sex="Family#female"/>
     </rdfs:subClassOf>
     <rdfs:subClassOf>
         <daml:Class rdf:about="Family#Family">
     </rdfs:subClassOf>
</daml:Class>
```

FIG. 7.4. DAML+OIL Specified Family

The following example is more complex than the "family" and allows us to demonstrate how our ontology, in concert with a reasoning system, correlates and aggregates events. *JTP*, which is referenced during the example, will be discussed in the next subsection.

The *Mitnick* attack is multi-phased; consisting of a Denial of Service (DoS) attack, TCP sequence number prediction and IP spoofing. When this attack first occurred in 1994, a SynFlood attack was used to effect the DoS, however, any DoS attack would have sufficed. The following, which is illustrated in Figure 7.5, details the sequencing of the Mitnick attack. In the attack, $\mathcal{H}_2$ is the ultimate target and $\mathcal{H}_1$ has a trust relationship with $\mathcal{H}_2$.

1. The attacker initiates a SynFlood attack against $\mathcal{H}_1$ to prevent $\mathcal{H}_1$ from responding to $\mathcal{H}_2$.

2. The attacker sends multiple TCP *RST* packets to the target, $\mathcal{H}_2$, in order to predict the values of TCP sequence numbers generated by $\mathcal{H}_2$.

3. The attacker then pretends to be $\mathcal{H}_1$ by spoofing $\mathcal{H}_1$'s IP address, sending a TCP *SYN* packet to $\mathcal{H}_2$. The *SYN* packet is the first step of the 3-way TCP handshake required to establish a TCP session between $\mathcal{H}_1$ and $\mathcal{H}_2$.

4. $\mathcal{H}_2$ responds by sending a TCP *SYN/ACK* packet to $\mathcal{H}_1$, however, $\mathcal{H}_1$ does not see this packet because its input queue is full due to the excessive number of half open connections caused by the SynFlood attack. If it did receive this packet, it would send a TCP *RST* message to $\mathcal{H}_2$ causing it to abort the TCP 3-way handshake. The attacker does not see this packet either.

5. Using a TCP sequence number of $\mathcal{H}_2$ that he was able to calculate in step 2, the attacker sends a TCP *ACK* packet to $\mathcal{H}_2$. This completes the 3-way TCP handshake.

6. $\mathcal{H}_2$ now believes that it has a TCP session with $\mathcal{H}_1$ and the attacker has a one way session with his target, $\mathcal{H}_2$, that he can use to issue commands to it.

Our ontology declares that the Mitnick class consists of independent instances of:

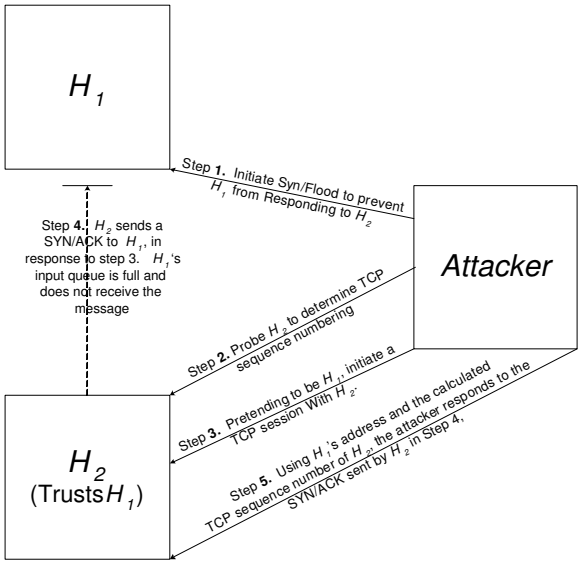


FIG. 7.5. Illustration of the Mitnick Attack

i. A denial of service (DoS) attack against a host $\mathcal{H}_1$, beginning at some time T. Note, we need only specify DoS and do not need to enumerate all of its possible types.

ii. The receipt of a TCP probe by $\mathcal{H}_2$. Again we do not need to enumerate all possible types of TCP probes.

iii. The "seeming" established TCP connection between $\mathcal{H}_1$ and $\mathcal{H}_2$.

Therefore, if we have concurrent instances of these three events, a sound and complete reasoning system that supports our ontology specification language will be able to correlate and aggregate them into a single more comprehensive event.

Figure 7.6 illustrates our DAML+OIL specification of a System, TCP Connection, SynFlood attack, and an RstProbe. The SynFlood is a subclass of a DoS and the RstProbe is a subclass of a Probe. The TCP connection, which is itself a property, has the additional property "inverseOf", meaning that if x is connected to y, then y is connected to x. We also declared the property "experiencing", which ranges over the class "Consequence", meaning that it can take on those values of that class and its subclasses, and is in the domain of the class System. In plain English, we are saying that a System may experience a SynFlood

```
<daml:Class rdf:ID="SynFlood">
  <rdfs:subClassOf rdf:resource="#DoS" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstb"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpSynRec"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:ID="RstProbe">
  <rdfs:subClassOf rdf:resource="#Probe" />
    <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpOutMsg"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstabRst"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
     <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpOutRst"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:ObjectProperty rdf:ID="connectedTo">
  <rdf:type rdf:resource=
            "http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#System"/>
  <daml:inverseOf rdf:resource="#connectedTo" />
</daml:ObjectProperty>
```

FIG. 7.6. DAML+OIL Specification: SynFlood, RstProbe, and TCP Connection Classes

attack, a Probe, or anything else that is defined as a "Consequence".

The Mitnick attack has two victims, one of which is a stepping stone. We defined the class *SystemUnderMitnickAttack* as being comprised of both victims — *SystemUnderDoSAttack* and *SystemUnderProbeAttack*, and the relationship that holds between them (i.e., the property "connectedTo").

The DAML+OIL property "intersectionOf" enables the aggregation of classes. We used this property to correlate and aggregate the other classes that make up the Mitnick attack into the class *SystemUnderMitnickAttack*. It consists of the intersection of a host x that is experiencing a DoS attack, a host y that is experiencing a Probe, and a TCP connection

between hosts x and y. Figure 7.7 illustrates the DAML+OIL specification of the Mitnick attack.

```
<daml:Class rdf:ID="SystemUnderDoSAttack">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#DoS"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:ID="SystemUnderProbeAttack">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#Probe"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:ID="SystemUnderMitnickAttack">
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Class>
      <daml:intersectionOf rdf:parseType="daml:collection">
        <daml:Class rdf:about="#SystemUnderDoSAttack"/>
        <daml:Restriction>
          <daml:onProperty rdf:resource="#connectedTo"/>
          <daml:hasClass rdf:resource="#SystemUnderProbeAttack"/>
        </daml:Restriction>
      </daml:intersectionOf>
    </daml:Class>
    <daml:Class>
      <daml:intersectionOf rdf:parseType="daml:collection">
        <daml:Class rdf:about="#SystemUnderProbeAttack"/>
        <daml:Restriction>
          <daml:onProperty rdf:resource="#connectedTo"/>
          <daml:hasClass rdf:resource="#SystemUnderDoSAttack"/>
        </daml:Restriction>
      </daml:intersectionOf>
    </daml:Class>
  </daml:unionOf>
</daml:Class>
```

FIG. 7.7. DAML+OIL Specification: Mitnick Attack

To test our DAML+OIL specification's ability to recognize the Mitnick attack, we generated a SynFlood attack against $\mathcal{H}_1$ (`68.54.120.173`) and a RST Probe against $\mathcal{H}_2$ (`130.85.93.64`). We also generated a spoofed IP connection from $\mathcal{H}_2$ to $\mathcal{H}_1$. We collected the corresponding kernel data, mapped it to *FOL* symbols and asserted them into *Java Theorem Prover*'s (JTP) knowledge base (KB) as facts. These instances are illustrated in Figure 7.8.

```
<System rdf:ID="sys_130.85.93.64">
</System>

<System rdf:ID="sys_68.54.102.173">
</System>

<rdf:Description rdf:about="#sys_68.54.102.173">
  <connectedTo rdf:resource="#sys_130.85.93.64" />
</rdf:Description>

<rdf:Description rdf:about="#sys_68.54.102.173">
  <hasNetwork rdf:resource="#sys_68.54.102.173_net1" />
</rdf:Description>

<Network  rdf:ID="#sys_68.54.102.173_net1">
  <icmpOutMsg><Rate_WA_Normal /></icmpOutMsg>
  <tcpEstabRst><Rate_WA_Normal /></tcpEstabRst>
  <tcpOutRst><Rate_WA_Normal /></tcpOutRst>
</Network>

<rdf:Description rdf:about="#sys_130.85.93.64">
  <hasNetwork rdf:resource="#sys_130.85.93.64_net2" />
</rdf:Description>

<Network rdf:ID="#sys_130.85.93.64_net2">
  <tcpSynRec><Rate_WA_Normal /></tcpSynRec>
  <tcpEstb><Amount_WA_Normal /></tcpEstb>
</Network>
```

FIG. 7.8. DAML+OIL Instances: System, Connection, SynFlood, and RstProbe

The instances illustrated in Figure 7.8 consist of a single connection from `130.85.93.64` to `68.54.120.173`, a SynFlood attack, and a RstProbe. The process of asserting our ontology, as rules, and the instances, or facts, into our KB resulted in the reasoner entailing additional acts; specifically that the system `68.54.102.173` experienced a DoS, system `130.85.93.64` experienced a Probe (because of support for inheritance), and that system `68.54.120.173` is also connected to system `130.85.93.64` (because of the "inverseOf" property of "connectedTo".

Queries to the KB take the form `(predicate, subject object)` where any one of "predicate", "subject", or "object" may be preceded by a question mark "?", indicating that it is an unbound variable. The query asks the KB if there are any "triples" that match the specified variables and if there are, the KB responds with the bindings that complete the triple.

We queried the KB for the existence of a Mitnick attack by asking:

```
(rdf:type ?host IDS:SystemUnderMitnickAttack)
```

JTP responded with:

```
Bindings 1:
   ?host = |http://security.umbc.edu/IDS#|::|sys_68.54.102.173|
Bindings 2:
   ?host = |http://http://security.umbc.edu/IDS#|::|sys_130.85.93.64|
```

Our DAML+OIL specification of the Mitnick attack includes both of the victims. To determine the relationship that held between the two victims of the Mitnick attack (i.e., which was the primary target and which was the stepping stone) we queried the KB as follows:

```
(rdf:type?host IDS:SystemUnderDoSAttack)
```

and

```
(rdf:type?host IDS:SystemUnderProbeAttack}}
```

JTP responded with:

```
Bindings 1:
?host = |http://security.umbc.edu/IDS#|::|sys_68.54.102.173|
```

and

```
Bindings 1:
?host = |http://security.umbc.edu/IDS#|::|sys_130.85.93.64|
```

indicating that the *SystemUnderProbeAttack* was the primary target and the *SystemUnderProbeAttack* was the secondary target.

Although we only asserted a single instance of a network connection into the KB, the reasoner created two instances of the connection class. This is because the `connectedTo`

property of the ontology (see Figure 7.6) states that an inverse relationship holds whenever we have a network connection. We queried our KB for all of the connections that it was aware of by asking:

```
(IDS:connectedTo ?host1 ?host2)
```

JTP responds with:

```
Bindings 1:
   ?host2 = |http://security.umbc.edu/IDS#|::|sys_130.85.93.64|
   ?host1 = |http://security.umbc.edu/IDS#|::|sys_68.54.102.173|}}
Bindings 2:
   ?host2 = |http://security.umbc.edu/IDS#|::|sys_68.54.102.173|}}
   ?host1 = |http://security.umbc.edu/IDS#|::|sys_130.85.93.64|}}
```

The previous two examples illustrated that way that a reasoning system uses facts annotated in a semantic language to deduce additional facts. The next section provides the reader with background on the types of reasoning systems that are available.

7.4 Reasoning Systems

There are two type of reasoning systems — backward-chaining and forward-chaining. Backward-chaining reasoners process queries and return proofs for the answers, while forward-chaining reasoners process assertions substantiated by proofs and draw conclusions. In a forward-chaining system, once a fact is asserted the system entails all possible inferences and they persist until they are retracted. Although it takes more time to load the KB, the benefits are realized by the short response time required to answer a query. By contrast, backward-chaining reasoning systems, sometimes called goal-directed inference engines, do not entail knowledge from asserted facts until the system is queried. This type of logic system loads its rule set quickly, however queries take more time.

During our initial research, we prototyped the logic portion of our system using Drexel's *DAMLJessKB* [67] reasoning system, an extension to the *Java Expert System Shell* (JESS)

[38]. JESS is a Java implementation of the *C Language Integrated Production System* (CLIPS) [40]. We found that although *DAMLJessKB* was sound, it was not complete. Soundness means that everything provable is true and completeness means that everything true is provable. *DAMLJessKB* was not complete because it did not fully implement the rules of DAML, hence it could not entail all of the additional knowledge that was inferable.

In addition to *DAMLJessKB*, Stanford's *Java Theorem Prover* (JTP), Ian Horrock's *Fast Classification of Terminologies* (FaCT) [49], and the *Renamed ABox and Concept Expression Reasoner* (RACER) [46] also implement DAML+OIL's rules. Both *DAMLJessKB* and *JTP* are forward chaining reasoning systems. We chose *JTP* because of its Java API.

Like *DAMLJessKB*, *JTP* can be embedded in a Java program and it employs an object oriented modular architecture of general purpose reasoning components. Upon initialization, we parse the DAML+OIL statements representing the ontology into *triples* and assert them into the reasoner's knowledge base (KB) as rules. Additional information marked up as instances of the ontology are also parsed and asserted into the KB as facts. The KB can now be queried regarding the nature of the facts. As stated, queries take the form: `(?predicate ?subject ?object)`. The reasoner responds with the existential bindings to the unbound (`?x`) query variables.

7.5 Defining the Target-Centric Ontology

We have abstracted our ontology into three layers corresponding to *Input, Means*, and *Consequence* categories introduced in Chapter 6. As will be detailed, the abstractions provided by our layered ontology architecture ensure that our approach will work across heterogeneous systems.

The *Lower* Ontology corresponds to the *Input* category. At this layer, the feature vectors that failed to conform to the model are used to instantiate classes that represent the different types of attacks. The *Lower* Ontology is only meaningful to IDSs that also use our 118 low-level kernel attributes as metrics.

The *Middle* Ontology associates the classes produced by the *Lower* Ontology with run-

ning processes, active network connections, and systems. For example it will take a class that represents a buffer overflow attack and associate it with a process. The *Middle* Ontology corresponds to the *Means* and is meaningful to all other IDSs.

The *Upper* Ontology corresponds to the *Consequence* category. At this layer, classes that were defined in the *Middle* Ontology are associated with the affected systems. As we detail each layer of the ontology we step through an example buffer overflow attack to illustrate the reasoning process.

Before we can properly address the ontology, we need to explain the process of mapping instances of the feature vectors that represent non-conforming kernel level data to *FOL* symbols so that they may be asserted into our KB and reasoned over.

7.5.1 Mapping Nonconforming Data to the Ontology

There are three types of feature vectors detailed in Chapter 4: process vectors that contain 34 elements, network vectors that contain 66 elements, and system vectors that contain 18 elements. *FOL* reasons over logical symbols, and cannot process the real numbers that our feature vectors are comprised of. Consequently, before we can reason over the anomalous feature vectors, we need to map them to the symbols employed by our ontology.

We employed clusters of feature vectors to model the system's normal state. In addition to the clusters, we have maintained vectors containing the mean and standard deviation of each feature in the exemplar data sets. The mean is calculated according to Equation 5.1 and the standard deviation is calculated according to Equation 5.2.

Our feature set characterizes the low-level kernel data as rates, quantities, or Boolean values. On input, we map each feature to a class that is the symbolic representation of its type of measure. The corresponding *FOL* symbols are specified by three quantifying classes: *Rate, Quantity*, and *BoolValue*. Binary features are mapped to the classes "true" or "false". The remaining features are mapped to a quantifying class based upon the number of standard deviations (σ) that the feature is from its mean (α). Table 7.2 presents these mappings and Figure 7.9 illustrates the DAML+OIL specification of the quantifying classes.

Class Type	**Value of Feature** f
x_WB_normal	$f < -2\sigma$
x_B_normal	$-2\sigma \leq f < -\sigma$
x_normal	$-\sigma \leq f \leq \sigma$
x_A_normal	$\sigma < f \leq 2\sigma$
x_WA_normal	$f > 2\sigma$
Amount_Inf	$f \geq 40$

Table 7.2. Mapping from Feature Values to *FOL* Symbols. x denotes either "Rate" or "Amount"

```
<daml:Class rdf:ID="Rate">
  <daml:oneOf rdf:parseType="daml:collection">
    <Rate rdf:ID="Rate_WB_Normal"/>
    <Rate rdf:ID="Rate_B_Normal"/>
    <Rate rdf:ID="Rate_Normal"/>
    <Rate rdf:ID="Rate_A_Normal"/>
    <Rate rdf:ID="Rate_WA_Normal"/>
  </daml:oneOf>
</daml:Class>

<daml:Class rdf:ID="Amount">
  <daml:oneOf rdf:parseType="daml:collection">
    <Amount rdf:ID="Amount_WB_Normal"/>
    <Amount rdf:ID="Amount_B_Normal"/>
    <Amount rdf:ID="Amount_Normal"/>
    <Amount rdf:ID="Amount_A_Normal"/>
    <Amount rdf:ID="Amount_WA_Normal"/>
    <Amount rdf:ID="Amount_Inf"/>
  </daml:oneOf>
</daml:Class>

<daml:Class rdf:ID="BoolValue">
  <daml:oneOf rdf:parseType="daml:collection">
    <BoolValue rdf:ID="True"/>
    <BoolValue rdf:ID="False"/>
  </daml:oneOf>
</daml:Class>
```

FIG. 7.9. DAML+OIL Specification: Quantifying Classes

7.5.2 Lower Ontology

The Lower ontology, illustrated in Figure 7.10, is comprised of classes that are specified as collections of attributes with restricted values. The attributes are restricted to the values that we recorded when we conducted our experimental attacks.

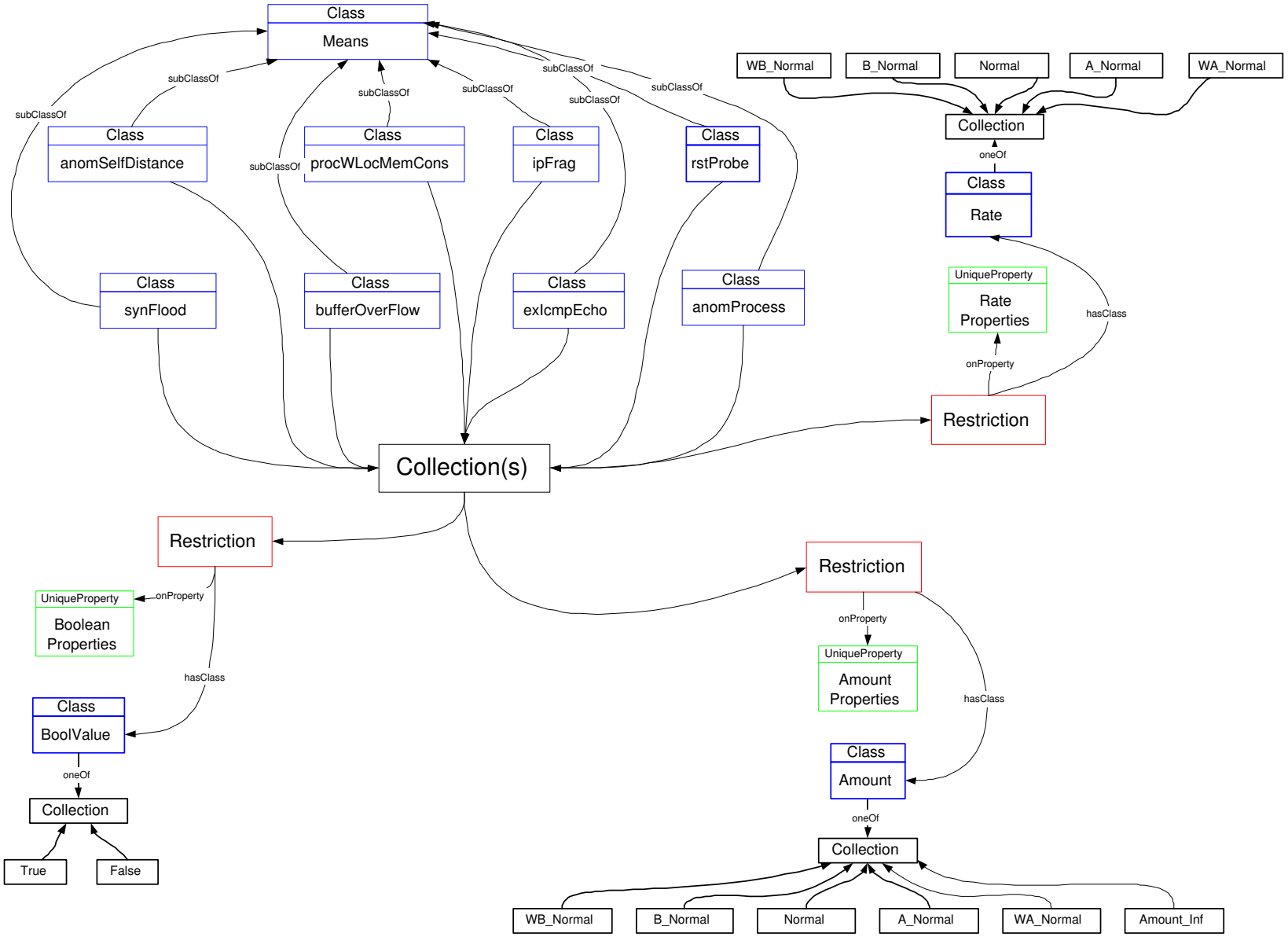


FIG. 7.10. Lower Ontology: Each Class is Specified by Combinations of the *Rate, Amount*, and *BoolValue* Classes

The following list enumerates the classes within the Lower ontology. Each of these classes represent the *Means* of an attack and are comprised of combinations of the low-level kernel attributes that have been mapped to *FOL* symbols.

a). *anomSelfDist*. This class consists of a single property — the self-distance metric. If it is instantiated, it is aggregated into other classes in the Middle ontology.

b). *anomProcess*. This class was used to detect trojaned binaries. We observed that variations in the self-distance, unkRetAdd, and codeSize properties were the most important indicators. The codeSize property served to differentiate a trojaned process from a buffer overflow.

c). *bufferOverFlow*. Buffer overflows remain the most preventable but most lethal vulnerability in a process. They are caused by programming errors and they may result in DoS or a root shell.

d). *procWMemoryConsum*. This type of attack is a class of DoS. It may be remotely effected by exploiting a vulnerable network attached process, or locally by recursively forking new processes or allocating huge blocks of memory. During the local DoS attack, at the system level we observed that memUsed, swapUsed, cpuOne, numProcs, conSwitch, rateProc and numUsers properties greatly varied.

e). *sysWMemConsum*. This class is the same as the above, however, from the system's perspective. It is comprised of the kernel data that is sampled at the "global" system level,

f). *synProbe*. Probes are used by attackers to reconnoiter a remote network or system. The simplest probe consists of sending a TCP SYN to an IP address and Port. If the remote system responds with a SYN/ACK, the attacker knows that there is a service running at the address and port number.

g). *rstProbe*. A rstProbe is a type of port scan. An attacher wanting to probe a system will send one of the following: FIN, FIN/URG/PUSH, URG, URG/PUSH, URG/FIN,

PUSH, PUSH/FIN or NULL Flags. On a Unix machine, each will elicit a RST/ACK on a closed port, and no response from an opened port. A Windows machines will reply with RST/ACK. When this occurs, the icmpOutMsg, icmpOutEchoResp, tcpEstabRst, and tcpOutRst properties greatly exceed their mean.

h). *synFlood*. The SynFlood attack is a type of DoS. The attacker sends thousands of spurious TCP EST requests (the first part of the TCP 3-way handshake) to the target. In response, the target will send a TCP SYN/ACK and reserve buffer space for the new connection. When this occurs, memory consumption increases and the tcpSynRec property exceeds its norm for several cycles.

i). *exIcmpEchoReq*. The ping of death is a type of DoS. It is crude, but remains effective at slowing down a system's network. The attacker, using a spoofed IP address, sends thousands of large (65,000) byte ICMP echo requests to the target. The target responds by sending an ICMP echo reply of the same size. During this attack, the rates of the ipInRecvs and icmpInEcho properties increase.

j). *iPFrag*. Also known as the Land attack. To carry out this attack, IP packet fragments with overlapping Fragment Offset fields are sent to the target. Most implementations of TCP/IP have had this vulnerability removed. Although this attack will no longer crash a machine, it will, however, degrade network performance. During this attack, the rate of the ipInOutReq exceeds its norm.

k). *netAnomPackets*. We observed that the ipReasmOks increased during the remote DoS attacks directed against network connected processes.

l). *exIpPacketSize*. This class is instantiated whenever the network is subjected to a sustained flow of large IP packets. For example, the ping of death attack uses large (65,000 byte) icmp echo requests. Due to the large packet size, the icmp echo requests (pings) are fragmented as they traverse the network. The target of the attack is forced to defragment the incoming packets.

m). *tcpPortScan1 & tcpPortScan2*. There are several types of TCP port scans, each type use different combinations of TCP Flags. Almost all of the variations are covered by these two classes. The middle ontology accepts either type of instance when the corresponding class is created at that layer..

n). *trojan ps*. ps is a system binary that lists all running processes, process owners, process ID (PIDS), etc. The trojaned version of ps hides specific processes.

o). *trojan netstat*.netstat is a system binary that lists network connections, routing tables, interface statistics, masquerade connections, and multi-cast memberships. The trojaned version of netstat hides specific network connections.

We use a buffer overflow attack to illustrate our ontology and the reasoning process. Figure 7.11 illustrates the DAML+OIL specification of the *bufferOverFlow* class, which is defined as a subclass of both the class *Means* and the class *InputValidErr*. If an anomalous feature vector that has been mapped to the *FOL* symbols used by our ontology matches the restrictions imposed by this class, an instance of a the *bufferOverFlow* class will be instantiated. The instantiation is not associated with a specific process or system and therefore remains "anonymous".

At this point in the reasoning process, we only have an anonymous instances of the classes *BufferOverFlow*, *InputValidErr* and *Means*. Additional associations will be made in the *Middle* Ontology where more meaningful classes will be created.

7.5.3 Middle Ontology

Figure 7.12 illustrates a subset of the Middle Ontology. The figure shows all of the bases classes, which include the class *System* and its properties *netReceived, experiencing, connectedTo*, and *hasProcess*. The figure also depicts the classes *processUnderBufferOverFlow* and *processUnderInpValErr* and their relationships to the other classes and properties. As we continue with the buffer overflow example, which follows the illustration, it will be

```
<daml:Class rdf:ID="BufferOverFlow">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#lockedVM"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#vmCodeSize"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#totVmSize"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>
```

FIG. 7.11. DAML+OIL Specification: Buffer Overflow Class

helpful to the reader to refer to the figure as we explain the classes and their properties and relationships.

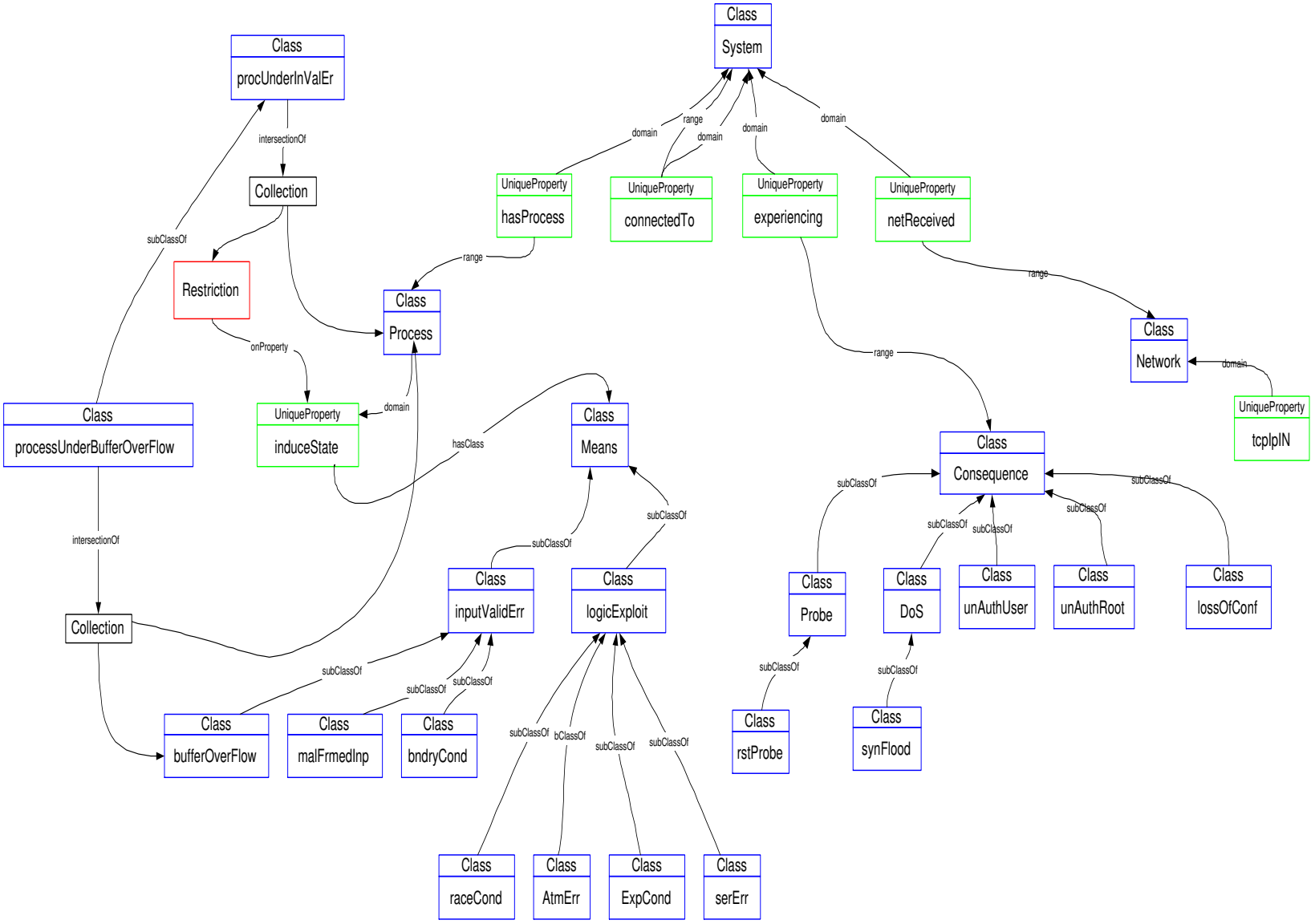


FIG. 7.12. Middle Ontology: Base, *processUnderBufferOverFlow* and *ProcessUnderInpValErr* Classes

Figure 7.13 shows the DAML+OIL specification of the *ProcessUnderInputValidErr* class and its subclass, *ProcessUnderBufferOverFlow*. The *ProcessUnderBufferOverFlow* class is instantiated when we have an instance of the *BufferOverFlow* class and we have an instance of the *Process* class that can be associated with that instance of the *BufferOverFlow* class.

We have also defined the class *ProcessUnderInputValidErr* as the intersection of a *Process* with the property *hasInducedState* that consists (has the value) of the class *InValidError*. Because the class *BufferOverFlow* is a subclass of the class *InputValidErr*, the instantiations will chain and the *ProcessUnderInputValidErr* will also be instantiated.

```
<daml:Class rdf:ID="ProcessUnderInputValidErr">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasinducedstate"/>
      <daml:hasClass rdf:resource="#InValidError"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:ID="ProcessUnderBufferOverFlow">
  <rdfs:subClassOf rdf:resource="#ProcessUnderInputValidErr" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#BufferOverFlow"/>
  </daml:intersectionOf>
</daml:Class>
```

FIG. 7.13. DAML+OIL Specification: *ProcessUnderInputValidErr* and *ProcessUnderBufferOverFlow* Classes

To recap, the reasoner has deduced that the anomalous feature vector matched the class *BufferOverFlow* and that the feature vector belonged to a specific instance of the class *Process*, therefore the reasoner created the classes *ProcessUnderBufferOverFlow*, *ProcessUnderInputValidErr* and *InputValidErr*.

The Middle Ontology includes all of the aggregates that are formed from its base classes (i.e., System, Consequence, etc.) and the classes that are instantiated in the *Lower* Ontology. The range of values for the *hasProcess* property are instances of the class *Process*. The class *Process* has just one property, *hasInducedState*. The range of values for *hasInducedState* are

taken from the Lower Ontology, which are restricted to members of the class *Means* and its subclasses.

The range of values for the property *connectedTo* are limited to classes representing other systems. The range of the property *experiencing* is limited to instances of the class *Consequence* and its subclasses. Finally, the range of the property *netReceived* is limited to instances of the class network, where the network class has the 66 properties defined in Chapter 4.

In addition to the classes that we have just discussed, the *Middle* Ontology has the following classes, all of which are instantiated in a similar manner.

- *ProcessUnderMemoryExploit*. This class is instantiated when a process allocates so much memory that it degrades system performance to the point of near exhaustion. It is the intersection of a process and an instance of anomalous memory metrics.

- *ProcessUnderTrojan*. This class is instantiated at the intersection of a system having a process and the process' profile being out of character with its model.

- *ProcessUnderBufferOverFlow*. Typically, a buffer overflow causes a process to crash. If the buffer overflow was successful, the attacker will be given a shell that runs in place of the process or as the process' child. If it was unsuccessful, the process will crash and terminate, in effect becoming a DoS. This class will be instantiated when the former occurs and the process fails to conform to the process' model.

- *ProcessUnderExploit*. This class is instantiated when a process is found to exhibit behavior defined by any of its subclasses (e.g.: atomicity error, etc).

- *ProcessUnderInputValidErr*. This class is instantiated when a process is found to exhibit behavior defined by any of its subclasses (e.g.: buffer overflow, malformed input, etc).

- *NetworkWithAnom*. This class is instantiated when a system network has received network traffic that fails to fit the model of normalcy. It is used as a precursor to

network attacks and probes.

- *NetworkUnderRSTProbe*. This class is instantiated when a system's network interface has received traffic that matches the profile of a RstProbe.

- *NetworkUnderSYNProbe*. This class is instantiated when a system's network interface has received traffic that matches the profile of a SYN Probe.

- *NetworkUnderExIpPacketSize*. This class is instantiated whenever the system's network interface has received excessively large IP packets.

- *NetworkUnderSynProbe*. This class is instantiated whenever the system's network interface has been subjected to a Syn Probe.

- *NetworkUnderTcpPortScan*. This class is instantiated whenever the system's network interface has been subjected to a TCP portscan. This class merges all of the different types of tcp port scans that are identified in the lower ontology.

7.5.4 Upper Ontology

The classes defined in the *Upper* Ontology are meaningful to other IDSs. They provide information that a specific system is under a specific type of attack. The classes at this level convey information that corresponds to the *Consequence* category defined in Chapter 6. Figure 7.14 illustrates a subset of the *Upper* Ontology.

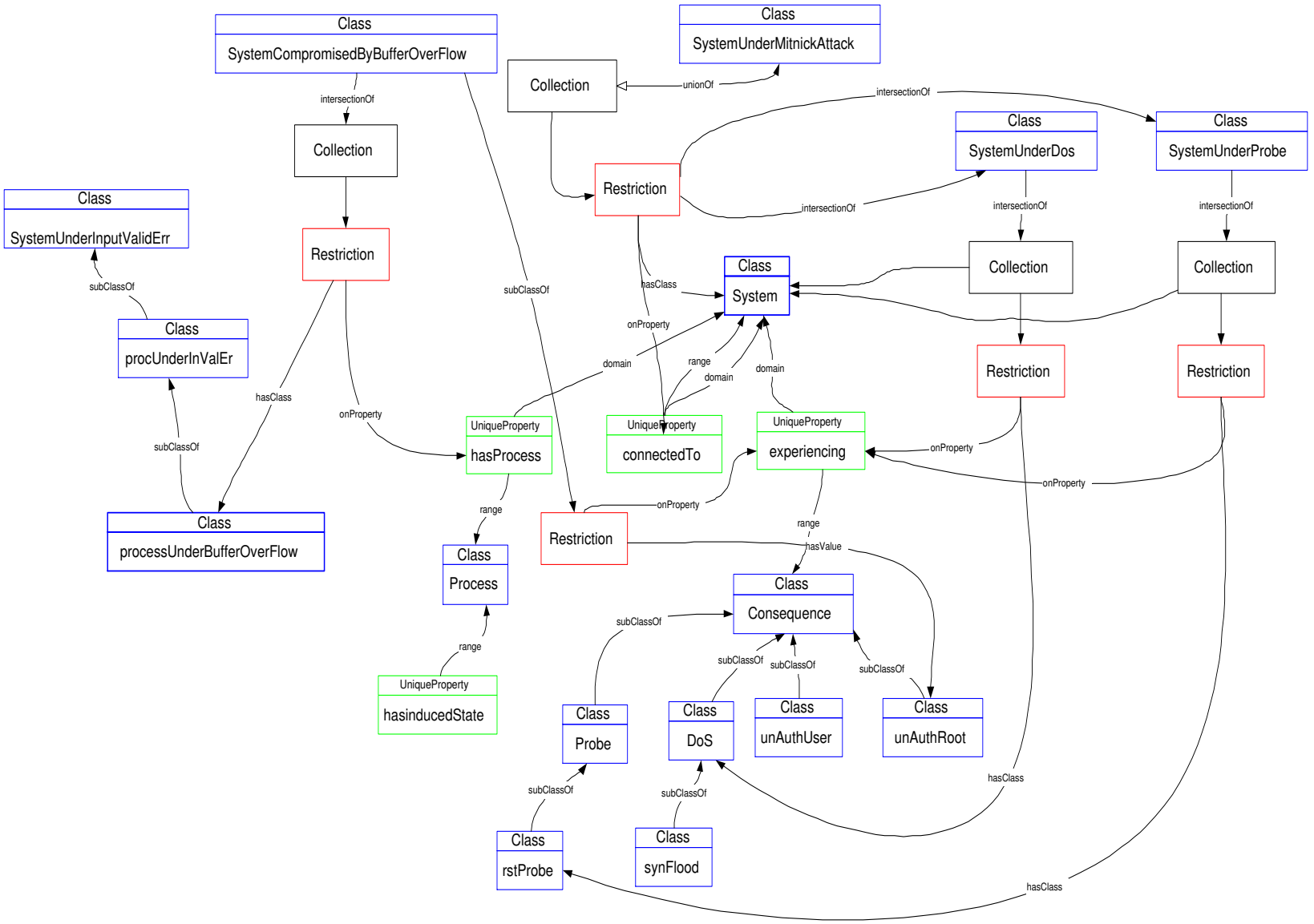


FIG. 7.14. Upper Ontology: Showing the Classes *SystemUnderBufferOverFlowAttack, SystemUnderDOSAttack, SystemUnderProbe, SystemUnderMitnickAttack*, and *SystemUnderInputValErr*

Continuing with the buffer overflow example, Figure 7.15 illustrates the DAML+OIL specification of the *SystemCompromisedByBufferOverFlow* class. It is instantiated whenever the *hasProcess* property of the *System* class has a value that is an instance of the *ProcessUnderBufferOverFlow*. It also states that the *System*'s *experienced* property be set to the *UnAuthRoot* class.

```
<daml:Class rdf:ID="SystemCompromisedByBufferOverFlow">
  <rdfs:subClassOf rdf:resource="#SystemUnderInputValidErr" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass><unAuthRoot /></daml:hasClass>
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasProcess"/>
      <daml:hasClass rdf:resource="#ProcessUnderBufferOverFlow"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>
```

FIG. 7.15. DAML+OIL Specification: *SystemCompromizedByBufferOverFlow* Class

In addition to the *SystemCompromizedByBufferOverFlow*, the *Upper* Ontology specifies the classes that identify a specific system as being subjected to a specific attack. Included in the *Upper* ontology are five *Meta* classes that identify the consequences experienced by a particular system. The classes of the *Upper* ontology follow:

a). *SystemUnderDoS*. Whenever a system experiences any type of denial of service an instance of this class will be created and bound to the system. This is one of the *Meta* classes.

b). *SystemUnderProbe*. This class is instantiated whenever the system has experienced a Probe or any of its subclasses. For example, whenever the class rstProbe is created an instance of its superclass is also created. This is one of the *Meta* classes.

c). *SystemUnderUnAuthRoot*. This class indicates that the system has experienced an unauthorized privilege escalation to the Root level. It is created whenever there is

an instance of an attack that results in the attacker gaining root access. This is one of the *Meta* classes.

d). *SystemUnderUnAuthUser*. This class indicates that the system has experienced an unauthorized privilege escalation to the User level. It is created whenever there is an instance of an attack that results in the attacker gaining user access. This class is one of the *Meta* classes and like all the *Meta* classes it is instantiated in addition to the other descriptive class. Consequently, if the KB was queried about the class that a particular system has membership in, the *Meta* classes, providing they apply, would be also returned in response to the query.

e). *SystemUnderLossOfConf*. This class is instantiated when the system has experienced a loss of confidentiality that was not the result of privilege escalation. This is one of the *Meta* classes.

f). *SystemUnderRstProbe*. This is a subclass of the *SystemUnderProbe* class.

g). *SystemUnderExploit*. This class is a superclass for attacks that induce logic exploits (e.g.: race conditions, atomicity errors). It is comprised of properties that are common to all of its subclasses.

h). *SystemUnderInputValidErr*. This class is a superclass for attacks that induce input validation errors (e.g.: buffer overflows). It is comprised of properties that are common to all of its subclasses.

i). *SystemUnderMitnickAttack*. This class is created whenever the circumstances detailed in Figure 7.7 (Mitnick Attack) exist.

j). *SystemUnderMemConsAttack*. This class is a dual of the class *SystemUnderDoSAttack*. It will also be created if a DoS consists of an attack that allocates an inordinate amount of memory.

k). *SystemCompromisedByBufferOverFlow*. This class is created at the intersection of a Process falling victim to a buffer overflow attack and a System having ownership of that process. This class creates an instance of the *unAuthRoot* class as a property of the targeted system.

l). *SystemCompromisedByTrojan*. This class states that a system has been compromised. It is created if the class *ProcessReplacedByTrojan* has been instantiated and it is used to signal the systems state.

m). *SystemUnderMemConsAttack*. This class is instantiated whenever the system falls victim to an attack that causes it to consume memory. It is created as a union of all of the different types of memory attacks.

n). *SystemUnderSynFloodAttack*. This class is created whenever the system is the victim of a Syn Flood attack. Instantiation of this class assures that the affected system's "experiencing" property will be set to the *DoS* class.

o). *SystemWithAnomProcess*. This is a general class that is created whenever the system owns a process that has anomalous self distance measures. Some of the buffer overflow attacks and trojaned binaries exhibited non-specific anomalies before they fully manifested their distinguishing characteristics. This class catches the early stages of those attacks.

p). *SystemWithAnomNetwork*. This is a general class that is created whenever a network interface has been subjected to anomalies that may be the early stage of an attack or provide some supporting evidence for a process that is under attack.

q). *SystemUnderTcpPortScan*. As the name implies, this class is created when a System is experiencing a TCP port scan. The creation of this class ensures that the affected system's "experiencing" property will consist of the *Probe* class.

r). *SystemUnderExIpPacketSizeAttack*. This class is created whenever the network protocol stack of the system is experiencing a stream of excessively large IP packets.

The complete DAML+OIL specification of our ontology is found in Appendix E.

7.6 Experiments

We used our ontology, in conjunction with *JTP*, to reason over the feature vectors that were classified as true negatives and false negatives during the experiments detailed in Chapter 5. We defined true negatives as those feature vectors that did not fit the model of normal behavior and were correctly classified as such. False negatives are the normal feature vectors that were incorrectly classified as not fitting the model. Our goal is twofold. Although our model performed very well in differentiating between normal and abnormal, it did not identify the nature or type of attack. Therefore our primary goal is to classify the true negatives according to the type of attack or intrusion that they represent. Our secondary goal is to provide an orthogonal test to reduce the misclassification rate.

Our ontology specifies the models of attack and we are testing for conformance to the model. Therefore during this stage of our process we defined a false positive as an instance of normal data that was misclassified as being intrusive. Likewise, we defined a false negative as an instance of abnormal data that was classified as not belonging to any of the classes that are used to represent an attack. We are also concerned with mis-characterized attacks. For example, a mis-characterization occurs when a DoS attack is erroneously identified as belonging to some other class of attack. We did not count a mis-characterization as a false negative because it was recognized as an attack.

7.6.1 Data

Chapter 5 detailed the experiments that we conducted in order to determine an effective means of modeling the quiescent system state. We found that the FCMdd Clustering algorithm using the Mahalanobis metric resulted in the highest *F-Measure*. Table 7.3 details the number of feature vectors that did not fit the model of normalcy and were forwarded to the reasoning process (Phase-2) for classification.

We mapped the values from each feature vector to the *FOL* symbols (the classes *Rate,*

Reference Number Number	Attack Type	Means	Normal	Malicious
1	unAuthRoot	Buffer Overflow	8	2,000
2	DoS	Logic Exploit	84	2,000
3	Loss of Conf.	Logic Exploit	14	2,000
4	User to Root	Buffer Overflow	14	2,000
5	DoS	Malformed Input	0	2,000
6	Trojan ls		0	2,000
7	Trojan netstat		0	2,000
8	Trojan ps		0	2,000
9	Trojan top		0	2,000
10	TCP Portscan	Syn Scan	18	752
11	Syn Flood	1/2 Open Conn.	0	2,000
12	Ping of Death	Large ICMP ER	0	2,000
13	IP Frags	Overlap. IP Frag	0	2,000
14	Syn Flood	Syn Scan	0	2,000
15	Local DOS	Excessive Forks	6	2,000
16	DoS	Malformed Input	6	2,000

Table 7.3. Data Set Resulting from FCMdd Clustering using Mahalanobis Distance

Amount, and *BoolValue*) specified in Table 7.2. We asserted those instances into the KB and queried it about the class type of the assertion. For example, the attack annotated as Reference Number 2 is a denial of service attack that sends malformed input to the Apache web server. This particular attack exploited a logic error that caused the Web server to allocate memory without ever deallocating it. If the attack were to continue, the system would crash due to memory exhaustion. We used IP addresses to uniquely identify our test systems and each of its feature vectors.

Figure 7.16 illustrates a subset of the DAML+OIL specification of the assertions made from the the DoS attack listed as Reference 2:

Our queries took the following form:

```
(rdf:type IDS:sys_130.85.93.33 ?host)
```

JTP responded with all of the classes that `sys_130.85.93.33` had membership in:

```
Query succeeded.
```

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
xmlns:IDS ="http://security.umbc.edu/IDS#"
xmlns     ="http://security.umbc.edu/IDS#">

<IDS:System rdf:ID="sys_130.85.93.64">
</IDS:System>

<rdf:Description rdf:about="#sys_130.85.93.64">
   <IDS:hasProcess rdf:resource="#sys_130.85.93.64_ap2.0_359"/>
</rdf:Description>

<Process rdf:ID="sys_130.85.93.64_ap2.0_359">
<selfDist><Amount_WA_Normal /></selfDist>
<vmCodeSize><Amount_WA_Normal /></vmCodeSize>
<totVmSize><Amount_WA_Normal /></totVmSize>
<lockedVM><Amount_WA_Normal /></lockedVM>
<numMinFault><Amount_WA_Normal /></numMinFault>
</Process>

<rdf:Description rdf:about="#sys_130.85.93.64">
   <IDS:hasProcess rdf:resource="#sys_130.85.93.64_ap2.0_360"/>
</rdf:Description>

<Process rdf:ID="sys_130.85.93.64_ap2.0_360">
<selfDist><Amount_WA_Normal /></selfDist>
<vmCodeSize><Amount_WA_Normal /></vmCodeSize>
<totVmSize><Amount_WA_Normal /></totVmSize>
<lockedVM><Amount_WA_Normal /></lockedVM>
<numMinFault><Amount_WA_Normal /></numMinFault>
</Process>
```

FIG. 7.16. DAML+OIL Specification: DoS Assertions

```
Bindings 1:

   ?host = |http://security.umbc.edu/IDS#|::|System|

Bindings 2:

   ?host = |http://www.w3.org/2000/01/rdf-schema#|::|Resource|

Bindings 3:

   ?host = |http://security.umbc.edu/IDS#|::|procWMemCons|

Bindings 4:

   ?host = |http://security.umbc.edu/IDS#|::|SystemUnderDoSAttack|

Bindings 5:

   ?host = |http://www.daml.org/2001/03/daml+oil#|::|Thing|
```

```
Bindings 6:
   ?host = |http://security.umbc.edu/IDS#|::|SystemUnderMemoryAttack|
Bindings 7:
   ?host = |http://security.umbc.edu/IDS#|::|SystemUnderExploit|
```

The reasoning process informed us that `sys_130.85.93.33` belonged the *SystemUnderDoSAttack, SystemUnderMemoryAttack*, and *SystemUnderExploit* classes and that it was experiencing memory consumption (`MemConsum`). This procedure was carried out for each of the referenced attacks.

Similarly, we were able to query the KB regarding the class membership of each of the asserted instances. Those queries took the form:

```
(rdf:type IDS:sys_130.85.93.64_ap2.0_360 ?process)
```

JTP responded with all of the classes that `sys_130.85.93.33_ap2.0_360` had membership in:

```
Query succeeded.


Bindings 1:
   ?host = |http://www.w3.org/2000/01/rdf-schema#|::|Resource|
Bindings 2:
   ?host = |http://security.umbc.edu/IDS#|::|procWMemConsum|
Bindings 3:
   ?host = |http://security.umbc.edu/IDS#|::|Process|
Bindings 4:
   ?host = |http://www.daml.org/2001/03/daml+oil#|::|Thing|
```

7.6.2 Results and Discussion

The FCMdd clustering algorithm only misclassified 150 (0.468%) of the 32,000 instances of normal data and 1,248 instances of anomalous data. The reasoning process, therefore, is mostly an exercise in correctly characterizing the anomalous data.

Instances of normal data that were characterized as any type of an intrusion were counted as a false positive. Likewise, instances of anomalous data that were not characterizes as an intrusive were counted as false negatives. We use the *F-Measure* given in Table 5.7 as a performance measure. The confusion matrix that we used for our experiments is given in Table 7.6.2 below.

Actual Classification	**Predicted Classification**	
	Anomalous	¬Anomalous
Anomalous	True Positive	False Negative
Normal	False Positive	True Negative

Table 7.4. Confusion Matrix for Actual and Predicted Classifications

Our results, which are recorded in Table 7.5, show an average *F-Measure* of .977606 for the second phase (classification) of our process. The combined *F-Measure* of both Phase-1 and Phase-2 was .971878. The stealthy port scan was responsible for reducing the overall *F-Measure*.

Self-distance was the most significant factor when reasoning about processes. All processes that had a self-distance exceeding 2σ were classified as anomalous processes. When the instances of the buffer overflow attack in Reference 1 were reasoned over, the first 132 samples were classified as belonging to the more general class *ProcessUnderExploit*. This is because properties that were needed to specify the more specific category, *ProcessUnderBufferOverFlow*, did not immediately appear in the data. The reasoner excluded all of the instances of benign feature vectors that were misclassified during the initial clustering phase.

Similarly, the effects of the DoS (Reference 2 — induced memory leak) were not apparent until after 200 instances. We also misclassified 2 of the 84 false negatives. With our ontology, the reasoner was only able to characterize the process' behavior during the "Directory Traversal" attack in Reference 3 as anomalous. Both the buffer overflow in Reference 4 and the DoS in Reference 5 were almost immediately discernible because the required properties (classes that represented values greater than 3σ) manifested themselves from the outset.

Reference Number	**% False Positives**	**% False Negatives**	**# MisCharacterizations**	***F-Measure***
1	0	0	132	1.00
2	2	.06	200	0.9924
3	7	0	577	0.9847
4	50	0	11	0.9982
5	0	0	7	1.00
6	0	0	81	1.00
7	0	0	324	1.00
8	0	0	112	1.00
9	0	0	17	1.00
10	100	50	0	0.6646
11	0	0	0	1.00
12	0	0	0	1.00
13	0	0	0	1.00
14	0	0	211	1.00
15	0	0	0	1.00
16	0	0	0	1.00
			Average	**.977606**

Table 7.5. Experimental Results: Phase-2

The four trojaned binaries were easily characterized because of the variances between code size, resident set size, and self-distance properties of the test data and those of the model.

The properties and relationships of the ontology's characterization of the network protocol stack were sufficient to classify four of the five network attacks. Regarding the TCP port scan, we could not separate the false negatives and true negatives from the first phase nor could we detect all instances of the port scan. This poor performance is attributable to this being a stealthy probe, crafted to "fly under the statistical radar".

7.7 Chapter Conclusions

The primary function of the reasoning process was to classify data according to the type of attack or intrusion that the feature vector represented. The reasoner accurately classified the overwhelming majority of attacks and and intrusions, and it somewhat reduced the percentage of the already small number of false positives. Although it was successful in the

majority of the cases, the reasoner performed poorly on the stealthy port scan, consequently reducing the overall performance measure achieved during the first phase of the process.

Chapter 8

Intrusion Detection and Response Protocols for Mobile Ad Hoc Networks

Mobile ad hoc networks (MANETs) are fundamentally different from their wired-side counterparts. MANETs provide no fixed infrastructure, base stations or switching centers. Moreover, the nodes of a MANET are computationally constrained and have limited power. The routing protocols utilized in MANETS are dependent on each node serving as a router. Examples of these routing protocols include: *AODV [94], DSR [57], ZRP [47]* and *TORA* [91], as well as cluster based optimizations as described in [55], [69] and [74].

The nature of MANETs not only introduces new security concerns but also exacerbates the problem of detecting and preventing aberrant behavior. Whereas in a wired network an intruder could be a host that is either inside or outside of the network and could be subjected to varying degrees of access control and authentication, in a MANET, an intruder is part of the network infrastructure. Moreover, at the outset, an intruder in a MANET could be a trusted and integral component of the network infrastructure and only later exhibit aberrant behavior.

Existing intrusion detection and response mechanisms for MANETs capitalize on the collaborate nature of mobile ad-hoc routing. These mechanisms rely upon promiscuous packet snooping to detect the mishandling of data in mobile ad-hoc networks. Our work improves and enhances existing mechanisms. Our research also revealed that the routing

protocols typically employed by mobile ad-hoc networks lack sufficient functionality to enable robust intrusion detection, hence we have added modules that provide the necessary functionality. These modules are applicable to all of the routing protocols used in MANETs, not just *DSR*.

Snooping protocols leverage two properties inherent in most mobile ad hoc protocols. The first property is that each node in the network maintains a list containing the addresses of those nodes with which it is in immediate proximity or on the path from a source to a destination. The second property, as is the case in the 802.11 [52] and *MACAW* [9] link layer protocols, is that a node is able to "hear" the *RTS/CTS* negotiation of its neighbors. Accordingly, each node that participates in the intrusion detection process 'snoops" on its neighbor's transmissions in order to ensure that they have not been modified or mis-routed. The notion of "snooping" is also employed in *DSR*, which is used for "reflecting shorter routes" as an optimization of the route maintenance process.

In our extension, which is viable for DSR and other ad hoc routing protocols, the snooping nodes listen to all other nodes in their proximity. This is in sharp contrast to both *Watchdog* [81] and *Neighborhood Watch* [13], which only work with *DSR*, watching the forward node on the patch from source to destination. We have experimented with, and provide detailed results for, two response mechanisms. In the *passive* response mode, upon determining that another node is aberrant, a node will unilaterally cease interaction with that node. Although each node acts independently, eventually the intrusive node will be blocked from using all network resources. In the *active* response mode, each node relies upon a *Cluster Based* hierarchy. When a node detects an aberrant neighbor, it informs its *Cluster Head*, which in turn initiates a voting procedure. If the majority determine that the suspected node is in fact intrusive, an alert will be broadcast throughout the network and the intrusive node will be denied network resources.

8.1 Background

Watchdog, introduced by Marti et al. [81], was the first snooping intrusion detection pro-

tocol for MANETs . *Watchdog* relies upon *DSR* and each node participates by "watching" its downstream node, on the route from source to destination, to ensure that it has re-transmitted the packet without modification. Marti et al. hold that if source routing is not used then a misbehaving node could simply broadcast to a non-existent node to fool the watchdog. While this is true, packet modification is not covered up by simply broadcasting to a non-existent node. To mitigate the effects of a misbehaving node, Marti et al. also introduced *Pathrater*, which selects a path from source to destination based upon a "reliability" metric, instead of the shortest path. This approach, as observed in [13], relieves the malicious node from the requirement of participating in the routing process, which may be construed as a reward.

Buchegger and Le Boudec [13] build upon Marti et al.'s work by replacing *Watchdog* with *Neighborhood Watch*. Their work is also limited to *DSR*, and snoops its downstream neighbor. They introduce a *Trust Manager*, *Reputation System*, and a *Path Manager*. Essentially each node is required to run a finite state machine to calculate trust, which in turn is used to rank the other node's reputation and then determine routes with the highest security metric. Buchegger and Le Boudec did not seem to consider the resource constraints imposed upon most mobile ad hoc devices, nor did they provide analysis of their protocol with respect to network performance, true positives or false positives.

We believe that our work extends and improves both of these efforts by expanding the malicious detection to collaborate with routing protocols other than DSR, and offering a more robust identification procedure by the incorporation of a cluster voting scheme.

Zhang et al. [123] propose a distributed and collaborative approach to intrusion detection using an Anomaly Detection Model. They use information-theoretic measures to describe the characteristics of the normal flow of information across the mobile ad hoc network. They use the *RIPPER* [18] and *SVM Light* [56] classifiers, trained using normal data to predict what is to be the next normal event given n previous events. If the next event n is not what the classifier had predicted then it is deemed to be an anomaly.

They consider two classes of attacks: *Route Logic Compromise* and *Traffic Pattern Distortion*. Route Logic Compromise attacks include malicious packet mis-routing and dropping

while Traffic Pattern Distortion includes the malicious alteration of packet contents. In training the classifier they use (1) local routing information to include cache entries and and traffic statistics and (2) a position locater or some uncompromisable form of GPS. In response to a detected intrusion they suggest using a majority-based distributed consensus algorithm where nodes communicate intrusion specific information and then initiate multi-layered response procedures.

Zhang et al. advocate a static routing model, whereas the fundamental nature of a mobile ad hoc network is dynamic. Our approach differs from that proposed by Zhang et al. because we do not make prior assumptions about normal or anomalous behavior and by piggy-backing onto the network infrastructure, we allow for dynamic change within an ad hoc environment.

8.2 Attack Classes

We have identified message mis-routing and message modification as the primary concerns in MANETS. These classes of attack are due to the collaborative routing paradigm employed in MANETs. Our approach requires us to address one additional type of attack, a distributed denial of service attack, where two malicious nodes, acting in collusion, attempt to exploit the intrusion detection process to deny service to a third node.

8.2.1 Message Modification Attacks

It was pointed out in [124] that due to the absence of encryption and due to the vulnerable nature of the transmission medium (viz.: radio) MANETS are susceptible to attacks where a node can maliciously modify the contents of a packet if it is on the route between the source and the destination. Such alterations may lead to the subsequent failure of applications using that modified data, hence these attacks must be detected and responded to.

A message modification attack will need to recalculate the packet's *checksum* and modify that field in the packet's header appropriately. If the attacker fails to do so, the packet will be discarded when it reaches its destination because the checksum computation, which is performed to detect transmission errors, will fail.

Malicious checksum alteration is a type of denial of service attack. To mount it, the attacker modifies the checksum so that it will not match the packet, consequently, when the packet arrives at its final destination the checksum computation will fail and the packet will be discarded. Our protocol will detect this type of attack and the offending node will eventually be denied access to network resources.

8.2.2 Message Mis-Routing Attack

The message mis-routing attack occurs when a malicious node discards packets that need to be forwarded to the next hop node in the route from source to destination. The offending node exacerbates the situation by not sending "Route Error" messages to the sender of the message, consequently the source will continue to send messages which will be maliciously dropped. If a reliable transport level connection is not used, this will cause a huge loss of packets in the network until the source times out and initiates an unnecessary "re-Route Discovery" procedure.

Another type of message mis-routing attack happens when the malicious node changes an entry in the packet's route list and forwards the packet to the next, albeit wrong, hop. Ultimately this will cause network congestion and unnecessary "route truncation" and "re-Route Discovery" procedures to be initiated.

8.2.3 Denial of Service Attack

As will be detailed, the *active* approach to our intrusion detection method employs a voting scheme to raise an alarm condition resulting in the identity of an intruder being broadcast throughout the network. It is conceivable that two malicious nodes could act in collusion and falsely declare an otherwise innocent node as an intruder. The end result is that the innocent node would be denied service. As will be detailed in Section 8.4.3, our approach addresses and mitigates this type of attack.

8.3 Snooping Protocol Extensions

We assume the presence of symmetric omni-directional links within the ad hoc network. When a node that is not on the path from source to destination is able to hear transmissions of two intermediate nodes, A and B, that are on the *source route*, it becomes a snooping or *monitor* node for node B. Its intrusion detection function is to ensure that node B does not alter the contents of the packet or misroute the packet. It accomplishes this by comparing certain information contained in the packet p as it is inbound to intermediate node B with the same information as contained in packet p' as it is outbound from node B.

This section details the data structures and algorithms maintained and executed at each node to facilitate our Intrusion Detection and Response Protocol.

Figure 8.1 illustrates the scenario where *node B* snoops on *Node 3* by examining packet p as it is inbound to *node 3* from *node 2* and packet p' as it is outbound from *node 3* to *node 4*.

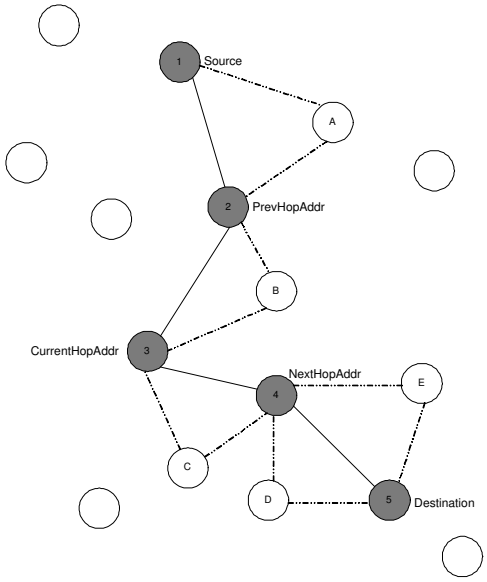


FIG. 8.1. Node A snoops on Node 2, Node B snoops on Node 3, etc.

8.3.1 Data Structures

Each node employing our Intrusion Detection and Response Protocol maintains four data structures: the *ID Snoop Table*, *IDStatus Table*, *BadNode Table* and the *Threshold Table*. For every packet snooped, the monitoring node makes an entry in its $ID\ Snoop\ Table$, recording information which will be used to detect intrusions. The format of the $ID\ Snoop\ Table$ follows:

Table $ID\ Snoop\ Table$

1: SrcAddr
2: DstAddr
3: PacketSeqNumber
4: PrevHopAddr
5: CurrentHopAddr
6: NextHopAddr
7: CHECKSum
8: TIMEStamp

An entry in the $ID\ Snoop\ Table$ is uniquely identified by the $SrcAddr$, $DstAdrr$ and the $PacketSeqNumber$.

Referring to Figure 8.1, when node B is snooping on node 3, node 2 is the $PrevHopAddr$, node 3 is the $CurrentHopAddr$ and node 4 is the $NextHopAddr$.

The monitoring node creates an entry in the *IDStatus Table* for every node that it is snooping upon. This table contains the total number of times that the monitoring node has detected an intrusion of a particular class for a particular node. The format of the *IDStatus Table* is as follows:

Table $ID\ Status\ Table$

1: NODE Address

2: ModificationCount

3: MisRouteCount

The *Threshold Table* holds threshold values for the attack classes. When a node exceeds the threshold value for a particular attack class, the protocol assumes that the anomalous behavior displayed by the node is in fact malicious and that link errors are not the cause of anomalies. The format of the *Threshold Table* is:

Table $Threshold\ Table$

1: ModThreshold

2: MisRouteThreshold

3: TimeOutPeriod

The *BadNode Table* holds the address of nodes that have been deemed to be intrusive. Whenever a node receives any packet or request from a node that is listed in the *BadNode Table* that request or packet is ignored. This effectively denies the intrusive node access to any resources in the MANET. The format of the *BadNode Table* is as follows:

Table $BadNodeTable$

1: NODEAddress

8.3.2 Algorithms

The $Snoop\ (Packet\ p)$ method forms the core of our intrusion detection protocol. All packets that the monitoring node receives (that are not explicitly addressed to the monitor) are passed to this method. The algorithm implemented by the $Snoop$ method follows:

method $Snoop(Packet\ p)$

1: **if** $p \neq$ *data* packet **then**

2: $\quad Return$

3: **end if**

4: **if** p = *Route Error Packet* $\wedge$ ($p \in$ *ID Snoop Table*) **then**

5: *ID Snoop Table* $\leftarrow$ *ID Snoop Table* - p

6: $Return$

7: **end if**

8: **if** ($p \in$ *ID Snoop Table*) $\wedge$ (CurrentHopAddr $\in$ *Neighbor Table*) **then**

9: $PerformID(p)$

10: $MakeEntry(p)$

11: $Return$

12: **end if**

13: **if** $p \in$ *ID Snoop Table* **then**

14: $PerformID(p)$

15: $Return$

16: **end if**

17: **if** CurrentHopAddr $\in$ *Neighbor Table* **then**

18: $MakeEntry(p)$

19: $Return$

20: **else**

21: Discard the packet

22: $Return$

23: **end if**

Accordingly, the *Snoop* algorithm does one of the following:

i. Ignores non-data carrying packets.

ii. If a Route Error Packet is detected, **AND** the *ID Snoop Table* contains an entry (or entries) for the node being reported as *unreachable*, the entry (or entries) are removed from the *ID Snoop Table*.

iii. If there is an entry for the packet in the *ID Snoop Table* (implying that the packet

was snooped during its previous hop), **AND** the *CurrentHopAddr* is also in the node's neighbor table, then run $PerformID\ (Packet\ p)$ and $MakeEntry\ (Packet\ p)$ on the packet.

iv. If there is an entry for the packet in the *ID Snoop Table*, implying that the packet was snooped during its previous hop, run $PerformID\ (Packet\ p)$ on the packet.

v. If there is no entry for the packet in the *ID Snoop Table* and if the monitor node has an entry for the next hop recipient in its neighbor table (implying that it will be able to hear the next hop relay the packet), make an entry in the *ID Snoop Table* by running $MakeEntry\ (Packet\ p)$ on the packet.

vi. If there is no entry for the packet in the *ID Snoop Table* and if the monitor node does not have an entry for the next hop recipient in its neighbor table, drop the packet.

As stated, the $Snoop$ method calls the $MakeEntry$ and the $PerformID$ methods. The $MakeEntry$ method creates an entry in the *ID Snoop Table*, storing the relevant header and routing information, as follows:

method $MakeEntry\ (Packet\ p)$

1: Store corresponding header and routing information from p into the $ID\ Snoop\ Table$
2: $TIMEstamp \leftarrow$ System time T.

The $TIMEstamp$ field in the *ID Snoop Table* is used to detect message mis-routing attacks where the node fails to forward the packet and to clear the table of "old " entries.

The $PerformID(p')$ method tests for message modification attacks and message mis-routing attacks. It does so by comparing the entry in the *ID Snoop Table* derived from the inbound packet, p, to information derived from the outbound packet, p'. To test for a suspected message modification attack, the $PerformID(p')$ method compares the checksum in the packet p' to that which was in packet p, as is recorded in the checksum field of the corresponding entry in the *ID Snoop Table*. To detect a mis-routing attack, the protocol performs

a test to ensure that the specified path was followed.

The $PerformID$ method makes entries into the $IDStatusTable$ when it detects a node exhibiting anomalous behavior. The algorithm implemented by the $PerformID$ method follows:

method $PerformID\ (Packet\ p')$

1: Find the entry in *ID Snoop Table* for p'
2: Compare the checksum contained in p' with that in the entry found above.
3: **if** checksum(p') $\neq$ checksum(p) **then**
4: For the PrevHopAddr's entry in the *ID Status Table*
ModCount $\leftarrow$ ModCount + 1
5: **end if**
6: **if** $ModCount > ModThreshold$ **then**
7: RAISEAlarm(PrevHopAddr)
8: **end if**
9: **if** $p(NextHopAddr) \neq p'(CurrentHopAddr)$ **then**
10: For the PrevHopAddr's entry in the *ID Status Table*
MisRouteCount $\leftarrow$ MisRouteCount + 1
11: **end if**
12: **if** $MisRouteCount > MisRouteThreshold$ **then**
13: RAISEAlarm(PrevHopAddr)
14: **end if**
15: $ID\ Snoop\ Table \leftarrow ID\ Snoop\ Table$ - p

To test for message mis-routing attacks, for each time period *T*, the monitor node calls the $PerformMisRoute$ method to test for entries in the *ID Snoop Table* that have exceeded the *TimeOutPeriod* specified in the *Threshold Table*. Whenever a node displays anomalous behavior by dropping or mis-routing a packet, the *MisRouteCount* entry in that node's entry in the $ID\ Status\ Table$ is incremented.

method $PerformMisRoute$

1: **for all** $p \in$ *IDS Snoop Table* **do**
2: **if** [$(Systemtime - TIMEStamp) > MisRouteThreshold$] $\wedge$ [$NextHopAddr \in$ *Neighbor List*] **then**
3: For the NextHopAddr's entry in the *ID Status Table*
MisRouteCount $\leftarrow$ MisRouteCount + 1
4: *IDS Snoop Table* $\leftarrow$ *IDS Snoop Table* - p
5: **end if**
6: **if** [$(Systemtime - TIMEStamp) > MisRouteThreshold$] $\wedge$ [$NextHopAddr \notin$ *Neighbor List*] **then**
7: *IDS Snoop Table* $\leftarrow$ *IDS Snoop Table* - p
8: **end if**
9: **if** $MisRouteCount > MisRouteThreshold$ **then**
10: RAISE Alarm(CurrentHopAddr)
11: **end if**
12: **end for**

In addition to testing for message mis-routing attacks, the *PerformMisRoute* method also clears old entries in the *ID Snoop Table*. If, in the event that a node moves out of range of its "monitoring" node after it has received a packet but before it forwards the packet, it could appear to the monitoring node that the packet was maliciously dropped. The *PerformMisRoute* method tests and corrects for this condition.

Depending on which approach is being used (*passive* or *active*), the $RAISEAlarm$ method in the above algorithm results in one of two different responses. We detail the responses in the following section.

8.4 Response to Intrusions

Our intrusion detection protocol allows for either an active or passive response to intrusions. With either response mode, the outcome is the isolation of the offending node from the network. In the passive mode, a node makes a unilateral decision based on its own observations of anomalous behavior. The more frequent and aberrant the behavior on the part of an intrusive node, the sooner the intrusive node will be isolated and denied access to the underlying network infrastructure.

The active response mode offers a higher level of assurance than does the passive mode. The increased assurance level is due to a majority voting scheme and consequently, the flooding of the intrusive node's identity throughout the network. The active mode, however, is more complex to implement.

8.4.1 Passive Response

Once the *threshold value*, which mitigates the effects of link error, for message misrouting or message modification has been exceeded, an alarm is raised. In the passive mode, the node that raised the alarm removes the intrusive node from its neighbor table and will no longer participate in *route discoveries*, *Hello Messages* or collaborative routing with the intrusive node. Additionally, the intrusive node's address is recorded in the *BadNode Table*. As we will show in the section detailing our experiments, the more dense the network, the more nodes that simultaneously declare a node intrusive and prevent the malicious node from utilizing network resources. If the node in question continues to act intrusively each node in the network will eventually make a unilateral decision to disassociate itself with the intruder.

8.4.2 Active Response

Tay et al. [55] propose the *Cluster Based Routing Protocol* (CBRP) where nodes form clusters, each with an elected cluster head. The role of the cluster head is to optimize the route discovery process. We utilize the cluster heads to enable a voting protocol and active

responses to intrusions. In CBRP, a neighbor cluster head is a minimum of 2 hops away.

When a node raises an alarm it forwards that alarm to all of its cluster heads. In turn, the cluster head initiates the voting scheme described below. It is important that no node be able to spoof identities of other nodes, as this will enable it to foil the voting scheme by generating spurious votes. Accordingly, we assume that some kind of mechanism to authenticate each node is available. Secondly the voting scheme may fail if the majority of the cluster heads are in fact malicious nodes. If this were to be the situation, the malicious cluster heads could vote in an incorrect manner and foil the protocol. However, we feel that the likelihood of malicious nodes being elected as cluster heads to the majority of the clusters is relatively small.

8.4.2.1 Data Structures Each cluster head participating in the voting scheme is required to maintain the following four data structures. The first two data structures are available from the underlying *Cluster-Based Protocol* while the remaining two are exclusively used for the voting process:

i. Neighboring Cluster Head Information. This table maintains information about all the cluster heads within its vicinity. This is used by the node that initiates the voting process to calculate which other cluster heads to include in the voting process for the current suspect.

ii. Two-hop Neighborhood Information. This table contains two hop topology information for every node. This information will be used by the cluster heads that participate in the voting process to decide if the suspect node, for which the voting process has been initiated, is in their neighborhood. If the suspect node is within two hops of a cluster head it will either vote positive or negative, otherwise it will vote neutral.

iii. Suspect Table. An entry to this table is made whenever a member node raises an alarm about a suspected node. This table holds the identity of the suspected node, the identity of the monitoring node and the type of intrusion detected.

iv. Voting History Table. This table records all the voting processes currently in progress (both initiated and voted by the cluster head), and all those voting processes whose outcome was either positive or negative.

The last two data structures are required to avoid multiple instances of the same voting process from being initiated for a single suspected node. These tables also prevent a single monitoring node from raising an alarm at different cluster heads and all of them voting positively based on the information obtained exclusively from a single monitoring node.

8.4.3 The Voting Protocol

The voting protocol employs two key strategies: Distributed Voting and Majority Voting. They are detailed as follows:

i. Distributing Votes:

   Whenever the voting process is initiated, all of the participating nodes send their votes to all other participating nodes. Each node, upon receipt of the votes, locally decides the outcome of the vote. This avoids the need for a voting coordinator.

ii. Majority Voting. Any vote is successful if a majority of the participating nodes vote positively.

The Protocol:

i. When the threshold is reached at a node, the node sends this information to all of its cluster heads. This information contains the identity of the monitoring node and the identity of the suspected node.

   If a node suspects its cluster head of being an intruder it will only send the alarm information to its alternative cluster head, if it exists. If the node does not have an alternative cluster head it will forward the alarm information to a cluster head that is two hops away. This two-hop information is contained in its *Cluster Adjacency Table* as described in [55].

ii. When a cluster head receives information about a potentially intrusive node, it adds this information to its suspect table. This information is used to respond to voting requests from other cluster heads.

iii. The cluster head checks the voting history to ascertain if a vote is currently in process for this suspect node:

   (a) If the cluster head finds that a vote is in progress, it does not initiate a new round of voting. This prevents issuing concurrent voting requests for the same suspect node.

   (b) If no vote is currently in process for the suspect node, the cluster head initiates the voting process. It sends a `VOTE-REQ` packet to all its neighboring cluster heads. The `VOTE-REQ` includes a list of cluster heads that are to participate in this vote. The `VOTE-REQ` also contains the identity of the suspect node and the identity of the monitoring node which raised the alarm.

iv. When a cluster head receives a `VOTE-REQ` for the same suspect node that it has just initiated the voting process for, it resolves the conflict by giving preference to the initiator with the higher address. The non-initiating cluster heads vote in the following manner:

   (a) Vote positive if it finds an entry in the suspect table for the same suspected node but reported by a different monitoring node from that included in the `VOTE-REQ`.

   (b) Vote neutral if the suspected node is not in its Two hop neighborhood because this means that the suspected node is not a neighbor of any of adjacent cluster. Hence this cluster head cannot judge the behavior of the suspected node.

   (c) Vote negative if the suspected node is in its Two hop neighborhood but does not find an entry in the suspect table for that suspected node. This indicates that the members of this cluster head have not noticed anything malicious about the

suspected node even though the suspected node is a neighbor to some of the members.

v. Every participating cluster head decides the outcome of the voting independently. The vote is positive if it has received a majority of votes in the affirmative, where a majority is calculated from the number of participating cluster heads listed in the original `VOTE-REQ`. Otherwise, the vote is deemed to be negative.

vi. If the vote is deemed positive at a cluster head, it sends out a `FINAL-RESPONSE` packet which is flooded throughout the network. This `FINAL-RESPONSE` is to instruct all the nodes in the network to stop communicating with the malicious node. It includes the identity of the malicious node and a list of cluster heads that voted positive in the voting process.

vii. A node in the network that is unaware of these process cannot arbitrarily trust a single `FINAL-RESPONSE` message because the message could have been sent by a malicious node in order to make other nodes stop communicating with an innocent node. Hence a node, upon receiving the `FINAL-RESPONSE`, waits to receive the `FINAL-RESPONSE` from enough participating cluster heads to conclusively verify positive results.

viii. Upon receiving `FINAL-RESPONSE` from all of the required nodes, a node enters the malicious node in its *BadNode Table* as described in Section 8.4.1

8.5 Protocol Modules

As previously stated, prior work has based efforts on the ad hoc routing protocol DSR. An obvious advantage is gained by each packet carrying the source route. This means that any snooping node can quickly determine where the packet should have come from, and where the packet should go to, on the next packet hop. Although using DSR is advantageous to our host based IDS, it has also drawn criticism for being insecure. Be that as it may, it

is easier to detect malicious activity when correct packet handling can be determined based solely on the contents of the packet header.

We have extended our base snooping algorithm to work with other routing schemes such as AODV. While each packet does not carry the route from source to destination, a snooping node can determine if the current hop is the final destination. This allows the snooping node to listen for the packet to be forwarded without modification. Obviously, a mis-route can not be determined, but any modification to the packet, or packet dropping, can easily be determined and logged.

In the case of dropped packets, we chose to ignore the broadcast packets in this particular effort. Protocols within ad hoc networks that broadcast can have mechanisms to limit/prevent widespread flooding. For example, a simple rule can prevent a node from re-broadcasting a packet that the node has already handled. This type of behavior could be misinterpreted as malicious. An idea for incorporation would be to log these as broadcast misbehaviors, and use them only in conjunction with other misbehaviors for determining bad nodes. In other words, misbehaviors with dropping broadcast packets would not be considered, by themselves, enough evidence on which to alarm. We do log broadcast packets for determining packet modifications.

In order to implement the algorithms, two additional pieces of support code need to be in place. It is important to recognize that performance of the intrusion detection algorithm is only as good as the underlying protocol that keeps track of the node's current one hop neighbors. The only routing protocol in the GlomoSim network simulator [5] having a neighbor table is AODV. Unfortunately, the table is only updated when nodes are expected to route traffic. The fundamental basis for our algorithms is knowing current one hop neighbors in order to determine correct packet handling. The implementation of AODV's neighbor table was determined to be woefully inadequate for our purposes.

We chose to implement a neighbor function that periodically sends `Hello` messages to announce its presence. The messages are received and tracked in a one hop neighbor table. If a node does not receive a `Hello` packet from one of its neighbors for three consecutive

`Hello` periods, then the neighbor is assumed to have moved out of range and is removed from the neighbor table.

The second piece of code added was a dynamic clustering scheme based on the Distributed and Mobility-Adaptive Clustering (DMAC) algorithm as described in [6]. The algorithm was slightly modified to use the Neighbor function to determine changes in Clusters and to initiate the appropriate actions (i.e. new Cluster Head elections). It should be noted that we are using DMAC to maintain a cluster hierarchy for voting, and not as a routing protocol.

8.6 Experiments

The algorithms were simulated using GlomoSim version 2.03. We used the simulation environment detailed in [81] as a starting point. The following subsection details our simulation environment, metrics, and experimental results.

8.6.1 Simulation Environment

i. Grid Size. We set the grid size to 2,000 by 2,000 meters.

ii. Nodes. We had 50 nodes in total. There were always 16 nodes involved in constant bit rate (CBR) connections, and we varied the number of bad nodes.

iii. Packet Traffic: 10 CBR connections were generated simultaneously, where 4 nodes were the source for two streams each, and 2 nodes were the source for a single stream each; destination nodes only receive one CBR stream.

iv. Mobility. Mobility used the random waypoint model with maximum speed set to 20 meters per second, and the pause time set to 15 seconds.

v. Routing Protocol. The routing protocol was set to AODV and DSR.

vi. MAC Layer. We used 802.11, peer-to-peer mode for the MAC layer.

vii. Radio. We used the "no fading" radio model with the radio range set to 376 meters.

viii. Simulation Time. We set the simulation time to 200 seconds.

ix. Dropped Packet Time Out. We set the timeout period for dropped packets to 10 seconds.

x. Dropped Packet Threshold. We set the dropped packet threshold to 10 packets.

xi. Clear Delay. An event expiration timer. We set it to 100 seconds. (e.g.: the amount of time that a node considered an event without coming to a final determination).

xii. Misroute Threshold. We set this parameter to 5 events. This is applicable to routing protocols using Source Routes such as DSR, but is not applicable in AODV.

xiii. Modification Threshold. We set this to 5 events.

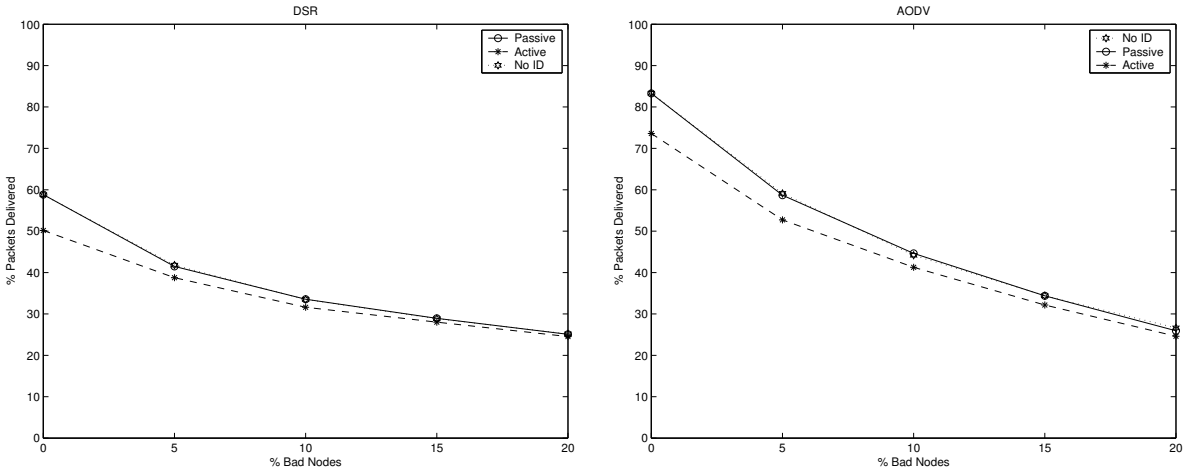
xiv. Neighbor `Hello` period. We set this to 30 seconds.

8.6.2 Metrics

We measure false positives, true positives, and packet throughput each as a function of the percentage of bad nodes in the network. False positives and True positives are counted as a single tally for each node making the identification. By using this method there may be greater than 50 total False or True positives counted. All results are averaged over a number of simulation runs.

8.6.3 Results and Discussion

Results were obtained by averaging 100 simulation runs for 200 seconds each. The graphs in Figure 8.2 illustrate the successfully delivered packets. The graphs in Figure 8.4 show the true positives, while those in Figure 8.3 show false positives. Each graph plots its respective metric as a percentage of bad nodes in the network. This measurement was chosen so that we could compare our performance to the work of Marti et al.



(a) (b)

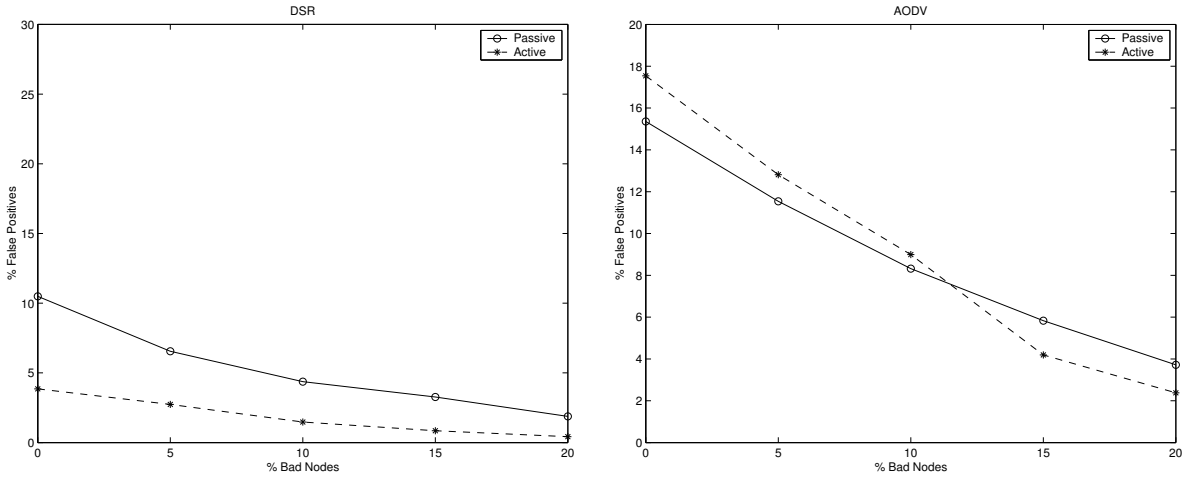
FIG. 8.2. Percentage of Packets Delivered: (a) DSR, (b) AODV

Given our simulation environment, only 60% of the packets using DSR were successfully delivered and 83% of the packets using AODV were delivered. As expected, the packet delivery rate is the same for when there are no intrusion detection mechanisms active on the network and when the passive response protocol is employed. This indicates that the passive response incurs no additional overhead. As the density of malicious nodes increases, the percentage of packets successfully delivered for both the Active and Passive response protocols converges at 25%. Because Marti et al. selected paths based upon a reliability rating (e.g.: *Pathrater*) their malicious node avoidance mechanism incurred more overhead than ours.

False positives are those nodes that were incorrectly labeled as malicious. As expected, the performance of both the Passive and Active response protocols improved, with respect to false positives as the density of the malicious nodes increased.

According to [81] there are two contributing factors that influence the rate of false positives — speed and collisions. The node's speed can cause monitoring nodes to believe packets have been dropped, when the mobiles move out of range prior to packet relaying. Collisions at the monitoring node may also lead to a node's failure to detect a packet relay.

True positives are malicious nodes that were correctly identified. In our protocol, the



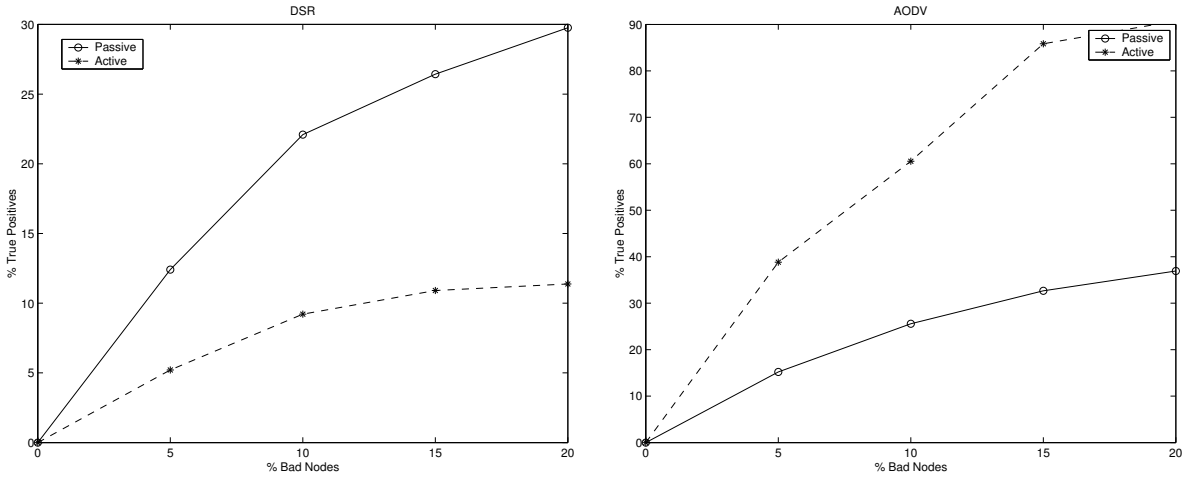
(a) (b)

FIG. 8.3. Percentage of False Positives: (a) DSR, (b) AODV

successful identification of a malicious node is dependent upon the likelihood that a third node will be in proximity to the relaying node and the relaying node's -1 hop neighbor. Consequently, our results show that as the density of malicious nodes increases so does the detection rate. Our experiments indicated that AODV outperforms DSR and that the detection rates of the passive response protocol reach 90% under AODV. As is the case in any intrusion detection system, there is a Min-Max trade off between false negatives and false positives. By requiring a majority vote, the Active Response attempts to minimize false positives, which is not the case with the Passive Response.

8.7 Chapter Conclusions

We have extended our snooping algorithm to work not only with DSR, but also AODV. While each packet does not carry the route from source to destination in AODV, a snooping node can determine whether the current hop is the final destination. This allows the snooping node to listen for the packet to be forwarded without modification. Obviously, a mis-route cannot be determined, but any modification to the packet, or dropping of the packet can easily be recognized and logged.



(a) (b)

FIG. 8.4. Percentage of True Positives: (a) DSR, (b) AODV

Our intrusion detection and response protocol for MANETs performed better than the one proposed by Marti et al. in terms of false negatives and percentage of packets delivered. It should be noted that Marti et al. do not report on the rate of true positives. We do not know how it compared in regard to true positives because they did not report their detection rate. Because Buchegger and Le Boudec did not report any results, the work of Marti et al. is the only work available for comparative purposes.

Chapter 9

Intrusion Detection in Wireless Sensor Networks

Improvements in wireless networking and micro-electro-mechanical systems (MEMS) have contributed to the formation of a new computing domain – distributed sensor networks. These distributed sensor networks are characterized by limited power supplies, low bandwidth, small memory sizes and a different traffic model. The traffic model of the mobile ad-hoc networks addressed in Chapter 8 is typically many-to-many whereas the traffic model of a sensor network is more of a hierarchical model or many to one. MEMS are significantly more resource constrained than the typical "mobile" or "handheld" device. The threat to a sensor network is different from the threat to a mobile ad-hoc network. As such, existing network security mechanisms, including those developed for Mobile Ad-Hoc Networks, are a poor fit for this domain. Research into authentication and confidentiality mechanisms designed specifically for sensor data and network control protocols is needed. Given the fact that little prior work ([95] being the exception) exists in this space, there is a need to identify the problems and challenges and to propose solution techniques.

This chapter addresses the security problems that sensor networks face. We identify the threats and vulnerabilities to sensor networks, starting from the radio layer and progressing to the application layer. We state why the security mechanisms that are presently used in mobile ad-hoc environments are inadequate or not appropriate for sensor networks. Our contribution to this area is the creation of lightweight techniques for securing existing sensor network routing and data movement approaches, such as directed diffusion [53], spin [71]

and data dissemination [11, 79]. Our protocol includes a mechanism to detect aberrant and intrusive sensor nodes and remove them from the sensor network.

9.1 Background

There is relatively little work in the area of securing sensor networks. Like their mobile ad-hoc counterparts, sensor networks lack a fixed infrastructure and the topology is dynamically deployed. Addressing the security of mobile ad-hoc networks, Kostov et al. [68] point out that if the routing protocol can be subverted and messages altered in transit, then no amount of security on the data packets can mitigate a security threat at the application layer. Consequently, they introduce "*Security Aware Ad-hoc Routing*" (SAR). SAR characterizes and explicitly represents the trust values and relationships associated with ad-hoc nodes and uses these values to make secure routing decisions. They address two problems: Ensuring that data is routed through a secure route composed of trusted nodes and the security of the information in the routing protocol. To motivate their scenario, they use the example of two military generals wishing to communicate via an ad hoc network using a generic form of the AODV protocol. They employ a route discovery protocol where only nodes with a security metric equivalent to the sender and receiver participate in the routing process. Their work appears to be based on the *Bell-La Padula Confidentiality Model* [7]. Their model, however, is dependent on self-enforcement where nodes with a lower than required security level voluntarily opt out of participating in the hop-by-hop routing process.

Perrig et al. [95] introduce "*SPINS: Security Protocols for Sensor Networks*" comprised of *Sensor Network Encryption Protocol* (SNEP) and μTESLA. The function of SNEP is to provide confidentiality (privacy), two-party data authentication, integrity and freshness. μTESLA is used to provide authentication to data broadcasts. SPINS presents an architecture where the base station accesses nodes using source routing.

In SNEP, each $node_j$ shares a unique master key K_j with the base station. This master key is used to derive all other keys. For data encryption, SNEP employs a one time encryption key produced by using a key derived from K_j and an incremental counter (message indicator)

as inputs to the RC5 cryptographic algorithm. The RC5 algorithm outputs a binary string that is used as the one time key. The message is XORed with the one time key, transmitted, and the counter is incremented in preparation for the next message. The base station, aware of the node's counter value and the derived key, produces the identical one time key, and XORs the encrypted message with the one time key to produce the clear text.

Our security protocol differs from SPINS in two fundamental and essential ways.

i. SPINS uses source routing, making the network vulnerable to traffic analysis. Our protocol relies upon broadcasts where the entire communication is end-to-end encrypted in order to mitigate the threat posed by traffic analysis.

ii. We provide a mechanism for detecting certain types of aberrant behavior, behavior that may be due to either a compromise or malfunction of an individual node. In either case, we are able to remove the node from the network.

9.1.1 Threats to Sensor Networks

There are many vulnerabilities and threats to a WSN. They include outages due to equipment breakdown and power failures, non-deliberate damage from environmental factors, physical tampering, and information gathering. We have identified the following threats to a WSN:

i. Passive Information Gathering: If communications between sensors, or between sensors and intermediate nodes or collection points are in the clear, then an intruder with an appropriately powerful receiver and well designed antenna can passively pick off the data stream.

ii. Subversion of a Node: If a sensor node is captured, it may be tampered with, electronically interrogated and perhaps compromised. Once compromised, the sensor node may disclose its cryptographic keying material, and access to higher levels of communication and sensor functionality may be available to the attacker. Secure sensor

nodes, therefore, must be designed to be tamper proof and should react to tampering in a *fail complete manner* where cryptographic keys and program memory are erased. Moreover, the secure sensor needs to be designed so that its emanations do not cause sensitive information to leak from the sensor.

iii. False Node: An intruder might "add" a node to a system and feed false data or block the passage of true data. Typically, a false node is a computationally robust device that impersonates a sensor node. While such problems with malicious hosts have been studied in distributed systems, as well as ad-hoc networking, the solutions proposed there (group key agreements, quorums and per hop authentication) are in general too computationally demanding to work for sensors.

iv. Node Malfunction: A node in a WSN may malfunction and generate inaccurate or false data. Moreover, if the node serves as an intermediary, forwarding data on behalf of other nodes, it may drop or garble packets in transit. Detecting and culling these nodes from the WSN becomes an issue.

v. Node Outage: If a node serves as an intermediary or collection and aggregation point, what happens if the node stops functioning? The protocols employed by the WSN need to be robust enough to mitigate the effects of outages by providing alternate routes.

vi. Message Corruption: Attacks against the integrity of a message occur when an intruder inserts itself between the source and destination and modifies the contents of a message.

vii. Denial of Service: A denial of service attack on a WSN may take several forms. Such an attack may consist of jamming the radio link or exhausting resources or mis-routing data. Karlof and Wagner [60] identify several DoS attacks including: "Black Hole", "Resource Exhaustion", "Sinkholes", "Induced Routing Loops", "Wormholes", and "Flooding" that are directed against the routing protocol employed by the WSN.

viii. Traffic Analysis: Although communications might be encrypted, an analysis of cause and effect, communications patterns and sensor activity might reveal enough informa-

tion to enable an adversary to defeat or subvert the mission of WSN. Addressing and routing information transmitted in the clear often contributes to traffic analysis.

9.2 Security Protocol

Security is a broad term that encompasses the characteristics of authentication, integrity, privacy, non-repudiation, and anti-playback. In the case of our sensor network the security requirements are comprised of authentication, integrity, privacy (or confidentiality), anti-playback and an intrusion detection and correction mechanism. The recipient of a message needs to be able to be unequivocally assured that the message came from its stated source. Similarly, the recipient needs to be assured that the message was not altered in transit and that it is not an earlier message being re-played in order to veil the current environment. Finally, all communications need to be kept private so that eavesdroppers cannot intercept, study and analyze, and devise counter measures in order to circumvent the purposes of the sensor network.

9.2.1 Single Collection and Authentication Point (Base Station) Model

Our model assumes the family of sensor routing protocols where each sensor communicates either directly or indirectly with a base station. In turn, the base station correlates and aggregates information from each sensor. Accordingly, the base station will need to verify the authenticity of the sensor, the integrity of the communication, and to ascertain that it is not a replay of an earlier communication. Recall the assumption that the base station is computationally robust and secure. In our protocol each sensor $\boldsymbol{j}$ shares a unique 64-bit Key $\boldsymbol{Key}_j$ with the base station. Our protocol provides for a multi-hop scenario where the range of a base station is extended employing nodes that are adjacent to the base station to serve as intermediaries for non-adjacent nodes. Figure 9.1 depicts an example of such a sensor network topology.

Given our security requirements, the entire communication between the base station and a node is encrypted. The format of all communications (sensor nodes and the base station)

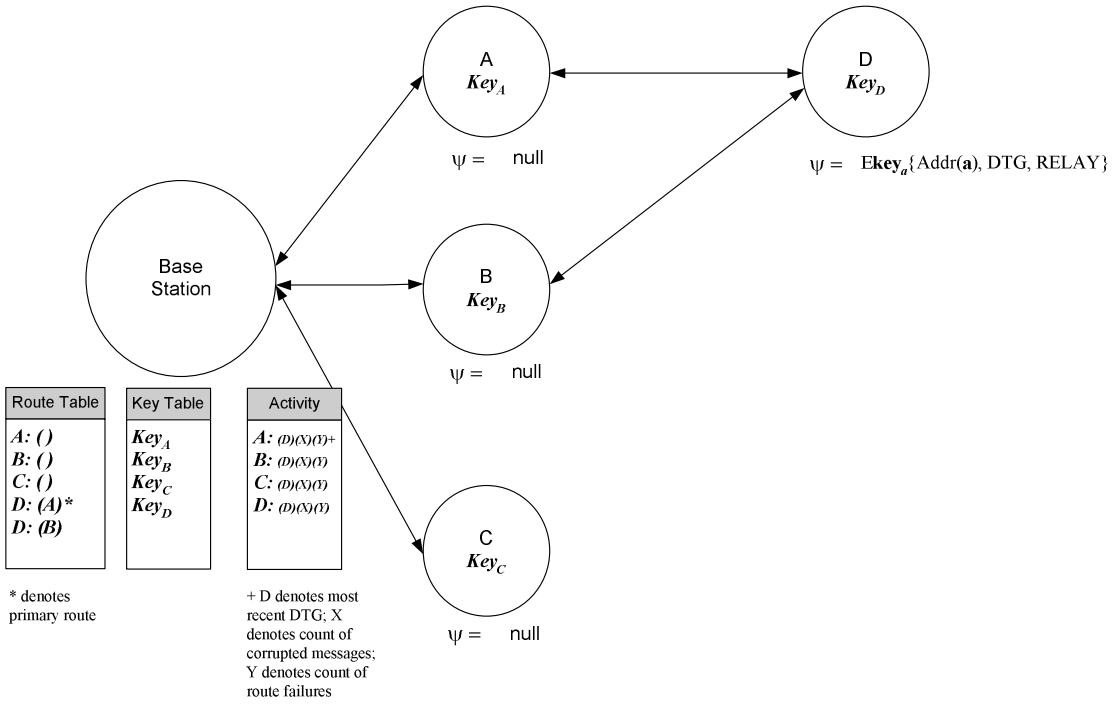


FIG. 9.1. Example Network Topology

consist of a preamble, header and payload. The preamble is empty if the communication originates from the base station and is directed to a sensor, otherwise it contains the address of the sending node. The header contains the recipient's address, nonce and a command and is encrypted under key $\boldsymbol{K}_j$, which is shared between the base station and node j. The payload contains data exchanged between the node and the base station. As will be explained, the payload is encrypted under the shared key of the destination node, which may be different from the key used to encrypt the header. This difference comes into play when the communication needs to be relayed by an intermediate node. Figure 9.2 depicts the communication format.

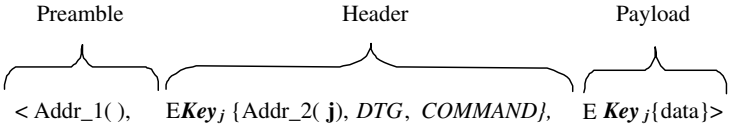


FIG. 9.2. Message Format

Where:

- Addr_1 is empty if the communication is from the base station to a sensor.
- Addr_1 contains the address of the transmitting node if the communication is directed to the base station. The inclusion of Addr_1 enables the base station to immediately select the correct key; instead of trying keys until it locates the correct one.
- Addr_2 contains the address of the destination node if the communication is from the base station to a node. If the communication is from a node to the base station, Addr_2 will contain the address of the sending node.
- DTG is the date-time-group and is a nonce used to prevent replay attacks.
- COMMAND is a command to the sensor.

9.2.2 Topology Discovery and Network Setup

The base station is deployed with the unique ID and symmetric encryption key of each node in the micro sensor network. Similarly, each node is deployed with the unique key that it shares with the base station and, as in SPINS, its clock is synchronized with the base station's clock. We note that sensors do not obtain their keys "over the air" from the base station; rather every sensor is programmed with a unique key. Upon initialization of the sensor network the base station learns the network topology, creates and optimizes a routing table and provides a mechanism to non-adjacent (out of radio range) nodes that enables them to securely reach the base station.

At start up, the base station sends a *HELLO* message to each node. If the node replies with a *HELLO-REPLY*, then the node is adjacent to the base station and the base station adds that node to its route table. Those nodes that did not reply are assumed to be more than one hop away and non-adjacent to the base station. For these non-adjacent nodes, the base station sends a message containing the *RELAY* command and a payload, to be forwarded to the non-adjacent node, to each of the adjacent nodes. In a *RELAY* message, the header is encrypted under the adjacent node's key and the payload, which encapsulates a header and payload intended for the non-adjacent node, is encrypted under the non-adjacent node's key. The relaying (adjacent) node prepends the original preamble to the payload and transmits the new message. The header of the message received by the non-adjacent node contains a *HELLO* command and the payload contains a mechanism that will be used by the non-adjacent node to reach the base station through the intermediate (adjacent) node.

This mechanism, referred to as Ψ, is a header containing the *RELAY* command encrypted under the adjacent node's key. The adjacent node simply places the header of the incoming message containing the *RELAY* command in the payload of the outgoing message containing the *HELLO* command. To respond to the *HELLO* message, the non-adjacent node constructs a *HELLO-REPLY* message encrypting it under the key it shares with the base station and places it in the payload. The preamble containing the base station's address and Ψ are pre-pended to the payload and the message is transmitted. In turn, the adjacent node receives the transmission, decrypts the header and upon seeing the *RELAY* command, prepends the preamble to the payload and transmits it to the base station. Once the base station discovers which nodes are adjacent to it and all of the paths by which the non-adjacent nodes are reachable, it optimizes its route table so as to not overburden an adjacent node with the task of relaying messages. If the optimization process results in a different route, the base station sends the affected non-adjacent node an updated Ψ. The secure topology discovery and network setup algorithm is presented in Figure 9.3:

C $\leftarrow$ all sensors in Sensor Network
Route Table $\leftarrow \phi$
Temp Route Table $\leftarrow \phi$
$\forall$ **j** ϵ C do
  Base Station $\rightarrow$ j : <Addr_1(), E**Key**$_j$ {Addr_2(**j**), DTG, *HELLO*}, null>
  if (**j** $\rightarrow$ Base Station : <Addr_1(j), E**Key**$_j$ {Addr_2(**j**), DTG, *HELLO-REPLY*}, null>) then
    Route Table $\leftarrow$ Route Table + **j**()
    C $\leftarrow$ C - **j**
$\forall$ **k** ϵ C do
  $\forall$ **j** ϵ Route Table do
   Base Station $\rightarrow$ j : <Addr_1(), E**Key**$_j$ { Addr_2(**j**), Null, *RELAY*}, E**Key**$_k$[Addr_2(**k**), DTG, *HELLO*}>
   j $\rightarrow$ k : <Addr_1( ),E**k**$_k$ { Addr_2(**k**), DTG, *HELLO*}, ψ}>
   if (k $\rightarrow$ j : <Addr_1( ), header, payload> where:
     header = ψ = E**Key**$_j$ {Addr_2(j), null, *RELAY*}
     payload = Addr_1(**k**), E**Key**$_j$[DTG, Addr_2(**k**),*HELLO-REPLY* }, null >
   and
   j $\rightarrow$ Base Station: < Addr_1(**k**), E**Key**$_k$ { DTG, Addr_2(**k**), *HELLO-REPLY*}, null>) then
    Temp Route Table $\leftarrow$ Temp Route Table + **k**(j)
Optimize(Temp Route Table)
Route Table $\leftarrow$ Route Table + Temp Route Table
$\star$ Note: The DTG is only verified by the final destination; consequently it is null for intermediate nodes.

FIG. 9.3. Secure Topology Discovery and Network Setup Protocol

As depicted in Figure 9.1, the base station maintains three tables. Their purpose and function follows:

The **Route Table** contains the primary route, indicated by an *, and alternate routes to a node. An entry of the form A:() indicates that the node is directly connected to the base station whereas an entry such as D:(A) indicates that A is an intermediate node between the base station and node D.

The **Key Table** contains the unique key shared by node j and the base station.

The **Activity Table** contains the most recent Date Time Group (DTG) received by the base station from a particular node, a count (X) of corrupted messages sent by the node, and a count (Y) of other nodes dependent upon this node to relay messages. The values of X and Y are used to detect aberrant behavior on the part of an individual node.

We use a cipher text auto-key system employing a 64-bit key (detailed in [26]) for data encryption. Accordingly, the strength of a cryptosystem is dependent upon both its key length and the soundness of the encryption algorithm. Current cryptographic doctrine recommends using keys of 128 bits, however, this requirement is predicated upon the notion that the encrypted communication remains secure against cryptanalysis and brute force attacks for a 30 year period. In contrast, we require that the sensor network's communications withstand a brute force attack for the life of the network and a short period thereafter.

To prevent traffic analysis, the entire communication is encrypted (with the exception of the preamble which is null except for traffic intended for the base station). Accordingly, a node will need to decrypt all communications that it "hears". This adds very little overhead because when the node decrypts the first 64 bits of the the message, the recipient's address (Addr_2) is revealed. If a valid address is present then the node will continue to decrypt the message, otherwise it will discard it.

As previously stated, authentication is achieved through the use of a shared secret, which is the 64-bit key $\mathbf{K}_j$, shared between the base station and node $\boldsymbol{j}$. Message integrity is achieved through the selection of an encryption algorithm that exhibits strong properties of diffusion and confusion. Accordingly, an attack aimed at altering the message will only be against

the form of the message and not its substance. Anti-Play back is achieved by the use the *Date-Time-Group*. Finally, privacy is achieved as a result of encrypting all communications.

9.2.3 Inserting Additional Nodes into the Network

The insertion of an additional node into the existing sensor network is easily accomplished. In our model the unique identity and the key $\boldsymbol{K}$ of the node to be added are loaded into the base station, the new node's clock is synchronized with that of the existing network and the base station repeats the topology discovery algorithm.

9.2.4 Isolating Aberrant Nodes

An aberrant node is one that is not functioning as specified. Identifying and isolating aberrant nodes that are serving as intermediate nodes is important to the continued operation of the sensor network.

A node may cease to function as expected for several reasons:

- It has exhausted its source of power.
- It is damaged.
- It is dependent upon an intermediate node and is being blocked because the intermediate node has fallen victim to the previous two bullets.
- It is dependent upon an intermediate node and is being deliberately blocked because the intermediate node has been compromised.
- An intermediate node has been compromised and it is corrupting the communication by modifying data before forwarding it.
- It has been compromised and it communicates fictitious information to the base station.

Our protocol, detailed in Figure 9.4, effectively mitigates against the class of attack (or failure) where an intermediate node is involved.

```
∀ j ∈ { (Current Time T - DTG) > Δ } do
    if (j is adjacent) then
        Base Station → j : POLL
        if j ↛ Base Station : POLL-REPLY then
          j Activity_Y ++
    else
    if (j is non-adjacent) then
        Base Station → k_primary : POLL
        if k_primary ↛ Base Station : POLL-REPLY then
          k_primary Activity_Y ++
          Base Station → k_alternate → j : POLL
          if j → Base Station : POLL-REPLY then
            Base Station → k_alternate → j : UPDATE-PSI
          else
            Route Table = Route Table - j
        else
          Base Station → k_primary → j : POLL
          if j ↛ Base Station : POLL-REPLY then
          Route Table = Route Table - j

∀ j ∈ { Activity_X > Threshold } do
    if (j is adjacent) then
        ∀ n ∈ Primary-Adjacent-List(j)
          Base Station → k_alternate → n : UPDATE-PSI
        Route Table = Route Table - j
    else
        Base Station → k_alternate → j : POLL
        if j ↛ Base Station : POLL-REPLY then
          Route Table = Route Table - j

∀ j ∈ Route Table do
    if Activity_Y > Threshold then
        Route Table = Route Table - j
```

FIG. 9.4. Network Repair Algorithm

In order to mitigate against an intermediate node that corrupts the data being relayed to the base station, the base station keeps a counter of corrupted packets. Such a tactic constitutes a denial of service against the sending node because the base station will in ll probability, request a retransmission, consequently depleting the node of power.

Periodically, the base station checks the activity table associated with a node, testing for a prolonged period of inactivity and for a high incidence of corrupted messages originating from the node. If the node is directly connected (i.e., it does not rely upon an intermediate node), this could be evidence of aberrant behavior on the part of the node. If the node relies upon an intermediate node, this could be evidence of aberrant behavior on either the part of the node or the intermediate node.

In either case, the base station polls the node. If the base station does not receive a poll-reply within the time out period, it will re-poll the node via an alternative path, if it exists. If it receives a reply, it will transmit a new ψ to the node which reflects the alternate route and increment the intermediate nodes counter (Y) of route failures.

9.3 Experiments

Our experimental goals were to measure the efficiency of the network setup protocol and the intrusion detection and repair algorithm. We used power consumption and time to converge as our metrics. We used the SensorSim [90] extension of the NS network simulator [89] for our experiments. All of our protocol, to include cryptographic functionality, is implemented at the routing layer.

9.3.1 Assumptions

We make the following assumptions regarding our simulated sensor network:

- We make no attempt to counter the threat from a widespread denial of service attack against the RF layer. As pointed out in [95], such attacks are fairly straightforward to mount against fixed frequency RF communication links that are found in the sensors.

A denial of service is in itself an alarm.

- The base station is computationally robust, having the requisite processor speed, memory and power to support the cryptographic and routing requirements of the sensor network. The base station is part of a trusted computing environment.
- The communication paradigm is either base station to sensor or sensor to base station.
- The radio range of a sensor is 15 meters.
- Given the radio range of a sensor, the single hop area of coverage with the base station at the center is: 706 square meters, πr^2.
- The sensing range of a sensor is 1 meter, providing an area of coverage of 3.141 square meters.
- Given the single hop area of coverage provided by the base station and a sensor's area of coverage, it will require 224 sensors to saturate the one hop area.

9.3.2 Simulation

We used the simple radio and battery models of the simulator. This model assumes a current draw of 12 mA to transmit a message, 1.8 mA to receive a message and 2.9 mA for the CPU to process a message. We also assume a data rate of 19,200 kbps and message length of either 24 or 48 bytes.

We conducted experiments to measure energy expenditure of each sensor function (*Tx, Rx* and *CPU*) during network setup time for four different network topologies. We simulated a geographic environment measuring 5654 square meters. We divided this environment into two concentric circles: the inner circle, with a radius of 15 meters, and the outer circle, with a radius of 30 meters. The base station was located at the center. The sensor placement within our experimental topologies is as follows:

1. 30 nodes randomly placed in the inner circle and 70 nodes randomly placed in the outer circle.

2. 50 nodes randomly placed in the inner circle and 50 nodes randomly placed in the outer circle.

3. 70 nodes randomly placed in the inner circle and 30 nodes randomly placed in the outer circle.

4. 100 nodes randomly placed across the entire area.

9.3.3 Results

9.3.3.1 Network Setup We measured the energy consumed (*Y axis*) by each component of the sensor (the transmitter, receiver and CPU) for the entire network of 1 base station and 99 sensors, plotting it against the time taken (*X axis*) for the particular network topology to converge. We used a log plot so that small values would be discernible.

Figure 9.5 shows the results for Topology #1. Topology discovery and network setup occurred in 54 seconds of simulation time with a total energy expenditure of 37.8 $Asec$ for all nodes in the sensor network.

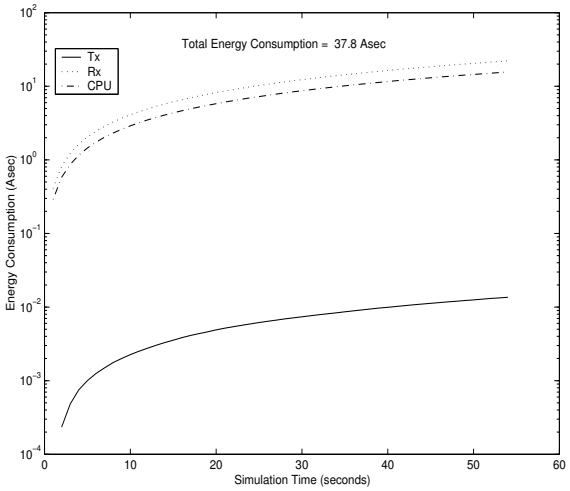


FIG. 9.5. Network Setup: 30 Adjacent and 70 Non-adjacent Nodes

Figure 9.6 shows the results for Topology #2. It took 55 seconds of simulation time for topology discovery and network setup. The total network energy consumption was 38.5

$Asec$.

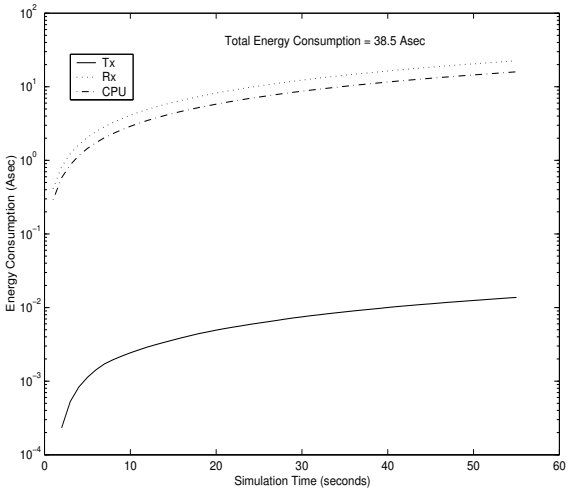


FIG. 9.6. Network Setup: 50 Adjacent and 50 Non-adjacent Nodes

The energy expenditure and time taken for Topology #3 is illustrated in Figure 9.7. Energy consumption was 22.4 $Asec$ and it took 32 seconds of simulation time for the network to converge.

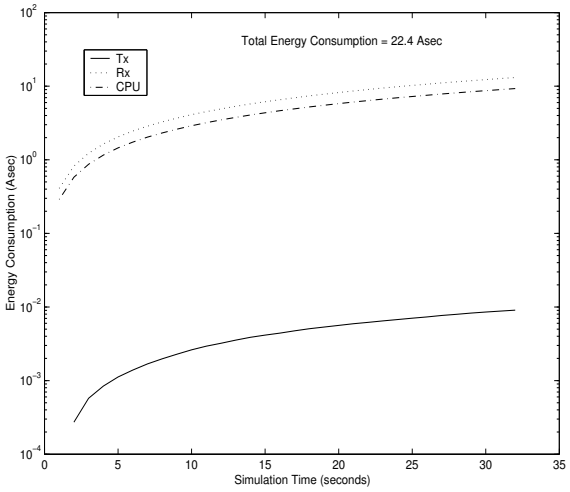


FIG. 9.7. Network Setup: 70 Adjacent and 30 Non-adjacent Nodes

Figure 9.8 illustrates the random distribution of sensors in Topology #4. It took 75 seconds of simulation time and energy consumption was 52.5 $Asec$.

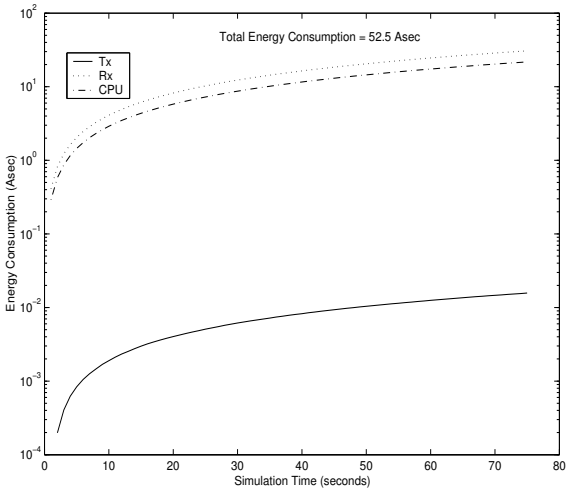


FIG. 9.8. Network Setup: Random Distribution

In all cases, and as anticipated, the receiver (*Rx*) component consumed the highest amount of energy, closely followed by the CPU. The transmitter (*Tx*) component consumed the least amount of energy. This is intuitive, as the number of messages received greatly outweighs those transmitted.

The topology with the densest inner circle and the sparsest outer circle consumed the least amount of energy and converged the fastest. The topology scenario that was most representative of the methods used for physical protection (30 inner nodes and 70 outer nodes) was near the median for time and energy consumption. Our results indicate, and it is intuitive, that as the ratio of adjacent to non-adjacent nodes increases in favor of adjacent nodes, energy consumption for topology discovery and network setup decreases.

Network setup is the most expensive in terms of energy consumption, which is due to the volume of messages. However, energy consumption decreases from this point forward for the life of the sensor network. To put the energy requirements into perspective, suppose that a sensor network using our security protocol were to maintain its peak rate for a protracted period. If this were the case, then each sensor equipped with a battery similar to the Eveready *X91* with a capacity of 3,135 mAh would be sustained for approximately 435 hours (Note: the Berkeley Renee Mote uses two of these batteries [8]). Since our network would have a

required lifespan of a few days, this time period is well within the bounds of the requirements for our application class.

9.3.3.2 Intrusion Detection and Network Repair We simulated the causal effects of aberrant nodes on the network. We used a node's inactivity during the last Δ time units as a measure of its state. Accordingly, we chose values of 15, 5 and 5 for Δ, X and Y, respectively. The simulation consisted of causing 5 adjacent and 5 non-adjacent nodes to become inactive during the simulation, for each of the four topologies described above. During the first 3 seconds of the simulation, and periodically afterward, the sensors transmit data to the base station, which populates the **activity table**. For the remaining simulation time period, the base station periodically launches the network repair protocol described in Figure 9.4 to update the routing table and transmit *UPDATE-PSI* messages as required.

Figures 9.9 — 9.12 illustrates the energy consumption over the 20 second simulation time period for each of the four topologies.

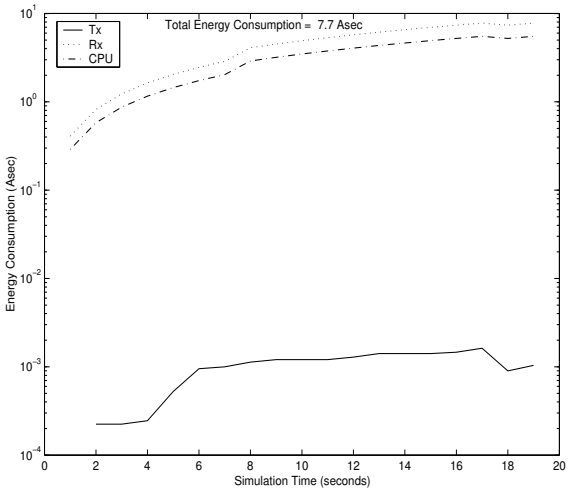


FIG. 9.9. Network Repair: 30 Adjacent and 70 Non-adjacent Nodes

We found that in each case the network repair algorithm took constant time and energy. The main reason for the short simulation time is the constant (simulation) time of 0.03 seconds required by the base station for polling each node and repairing the network based on

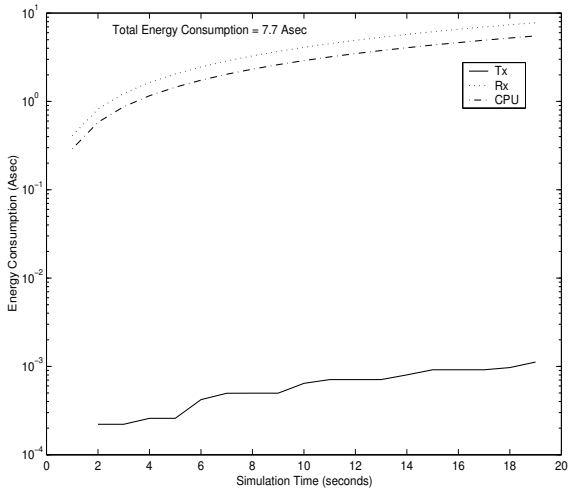


FIG. 9.10. Network Repair: 50 Adjacent and 50 Non-adjacent Nodes

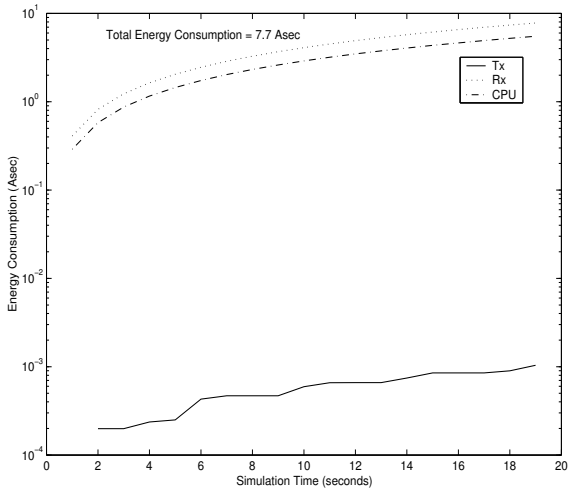


FIG. 9.11. Network Repair: 70 Adjacent and 30 Non-adjacent Nodes

the responses (either direct or relayed). For each topology, this time corresponds to energy consumption of approximately 7.7 $Asec$ to repair the entire network.

9.4 Chapter Conclusions

We have identified the threats and vulnerabilities that a sensor network might face. In response, we have created a lightweight communications protocol and intrusion detection

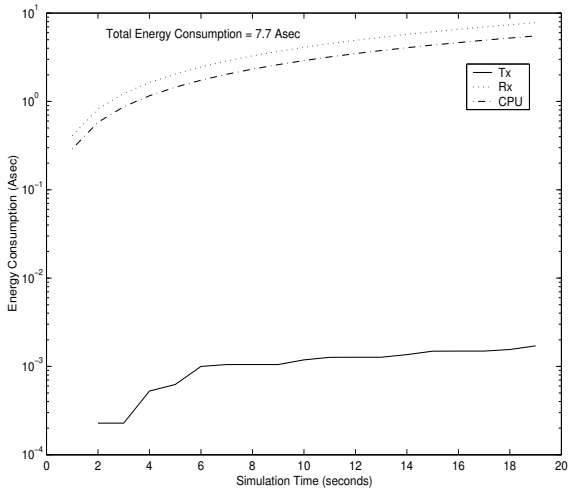


FIG. 9.12. Network Repair: Random Placement

and correction mechanism that mitigates those threats. Our protocol considers the diminished computational capacity that is characteristic of sensor networks. Our experiments have demonstrated that our protocol is feasible in terms of functionality and considering to a sensor node's limited resources.

Chapter 10

Conclusions

According to the empirical analysis that we conducted, the most serious attacks are effected by "insiders" who carry out their attacks via an attached terminal — not via the network. Consequently, a network-based IDS will fail to detect the most consequential and damaging attacks. Moreover, the most pervasive network-based IDSs are of the signature type and they are only able to detect known attacks.

We are aware of the tendency among security practitioners to rely on network-based signature detectors almost exclusively. To justify their choice, they state that host-based systems cannot be trusted to provide accurate information. The reason for their mistrust rests upon the notion that the underlying operating system is not provably secure.

To deflect these criticisms, we have anticipated a worse case scenario — *hidden processes*, which by their nature have the potential to cause a considerable amount of harm. We have implemented a technique to detect this type of attack and have expanded it to instrument the Linux kernel. We do not claim to have made the operating system impervious to attack, but we offer reasonable assurances regarding the integrity of the data captured by our solution which provides identifiable artifacts of an attack. Hopefully this will alleviate the general concerns voiced about host-based intrusion detectors.

The instrumented kernel gave us access to the kernel's data structures, which in turn provided us with potentially hundreds of metrics and measures to evaluate system state. One of the measures that became available to us was the stream of system calls that are generated

by a running process. Stephanie Forrest et al. were the first researchers to use system calls as a measure of a process' state. In their model, they viewed sequences of system calls as *n-grams* and built a lexicon of all of the contiguous words of length n in the stream of system calls. Tan et al., after extensive analyses, have concluded that this type of a measure will always result in blind spots. To counter their affects, we developed a unique system call measurement that we refer to as *self-distance*. We then used the Kullback-Leibler dissimilarity metric to measure the dissimilarity of the *self-distance* distributions between a baseline exemplar of a process and an unclassified (aberrant or benign) sample taken from the same type of process. Our experiments strongly indicate that the Kullback-Leibler dissimilarity measure, weighted by information gain of the *self-distance* distributions between an unknown sample and a baseline exemplar is a suitable measure for distinguishing between processes that are under attack and those that are not.

We proposed a dual-phase host-based intrusion detection methodology. Our methodology is intended to be employed throughout a domain or enterprise, where each system is responsible for itself and communicates with other systems in order to give each other a 360° view of the environment.

During the first phase of our method, we compared samples of low-level kernel data to a model of normal behavior. Accordingly, we experimented with different clustering techniques, vector measurements, and normalization methods to determine an optimal modeling strategy. Our experiments showed that the *Fuzzy c-Medoid* clustering algorithm using the *Mahalanobis* distance performed best. We experimented with *z-normalization* but learned that the Mahalanobis distance precluded its use. We presented a proof stating the reason for this.

Once a data sample failed to conform to the model, what to call it or how to classify it became an issue. The second phase of our intrusion detection process, therefore, is concerned with classifying the instances of anomalous data so that it can be communicated to other systems in meaningful form.

Based upon the results of our empirical analysis, we created a *target-centric* taxonomy

of attacks categorized by *Input*, *Means* and *Consequences*. These categories serve as the core for our *target-centric* ontology.

We argued the case for moving away from taxonomies and their syntactical representation languages, to ontologies and semantically rich ontology specification languages. We compared the ontology specification language DAML+OIL to *XML*, which is currently advocated by the IETF in its emerging IDMEF standard. Our comparison highlighted the deficiencies of XML and demonstrated how a language like DAML+OIL mitigates those deficiencies. We also demonstrated how an ontology specification language, in concert with a shared ontology, can simultaneously serve as an attack recognition, attack reporting and attack correlation language.

Using DAML+OIL, we created a data model of the relationships that hold between the low-level data that we captured while during our experiments. We initiated real attacks against our experimental systems and collected the data. We devised a means to map the instances of the attack data (data that failed to conform to our model of normal behavior), which were represented as numeric values, to instances of our ontology, which need to be represented as logical symbols. We used the *Java Theorem Prover*, a sound and complete *First Order Logic* theorem prover, to reason over and classify instances of anomalous data. Our experiments proved that we were able to classify the attacks and intrusions with a high degree of accuracy.

The *F-Measures* of our dual-phase process were .982249 and .977606 respectively, and the overall *F-Measure* was .971878. Because of the base-rate fallacy, the limiting factor an IDS's performance is not its ability to correctly identify intrusions, but rather its ability to suppress false alarms. The high *F-Measures* in conjunction with the posterior probability of .998 have demonstrated our methodology's overall effectiveness.

We have presented new and novel techniques that advance the field of intrusion detection in several areas. We have designed novel mechanisms to detect and mitigate aberrant behaviors encountered in Mobile Ad Hoc (MANET) and Wireless Sensor (WSN) networks. Since both of these networks are comprised of resource constrained devices, we designed

our intrusion detection mechanisms as protocols that monitor network state rather than system state. We have experimented with a "snooping" protocol for MANETS, extending prior research to work with all mobile ad hoc routing protocols, not just *DSR*.

We have identified the threats and vulnerabilities that a sensor network might face. To mitigate them, we have created a lightweight communications protocol that includes an intrusion detection and correction component. Our experiments and simulations have demonstrated that our protocol is functionally feasible given a sensor node's limited resources.

10.1 Future Work

Although our research shows promise, there is considerable work yet to be done. Our work has been "stand alone", focusing on a single detector. Research regarding the dynamic building of distributed ID coalitions needs to begin. These coalitions are to be modeled on loosely coupled social networks such as those in human environments. Better analytical methods need to be developed in order to detect attacks that are crafted to "fly under the statistical radar". The data set that we extract from a running kernel is highly dimensional. Although some attributes appear to never fluctuate beyond the statistical norm, could their exclusion result in a missed attack? Research into reducing the dimensionality of the data set is needed. In our model of a distributed IDS, the ID units interact as both producers and consumers of information. Communications between the ID units needs to be supported by a secure infrastructure that incorporates distributed access control. Policy based access control, which is in stark contrast to static access control lists, was developed at UMBC under NSF contract 0242403. Whether or not it is a viable approach for a distributed IDS needs to be investigated.

In order to detect attacks that are crafted to be evasive by hiding in the statistical "noise", requires sensitive and discerning analytical tools. The following describes some of the issues pertaining to the data analysis and model generation that are in need of future research.

i. Give a data set, what is the optimal number of partitions (states) represented in the set?

ii. If the actual number of states is known how do you partition the data set to best model the different states?

One possible approach is motivated by the *information-theoretic* engineering literature (see e.g. [105] where the approach is explored for the Mahalanobis distance).

Let $\pi_1, \ldots, \pi_k$ be a partition of the dataset into k clusters. If $\mathbf{c}_i$ is the centroid of cluster π_i define the quality (or distortion) of π_i, as $q(\pi_i) = \sum_{\mathbf{x} \in \pi_i} d(\mathbf{c}, \mathbf{x})$. Likewise, define the quality of the partition of k as $Q_k = q(\pi_1) + \ldots + q(\pi_k)$. To calculate the optimal number of partitions we denote the quality of the optimal partition k of the dataset as Q_k^o and plot Q_k^o versus k. This plot is referred to as the *distortion curve* of the cluster and according to Sugar et al. [104] the optimal number of clusters is at the "kink" of the distortion curve (a more sophisticated approach based on the analysis of the distortion curve is suggested by Sugar in [105]).

A different approach to identifying the "right" number of clusters is based on optimization theory. The problem of finding k optimal clusters partition of a dataset by a variety of k–means algorithms can be stated as a constrained optimization problem (see [64]) with the quality of an optimal partition given by Q_k^o. The constrained optimization problem is associated with the convex *dual* optimization problem which, in many cases, is much easier to solve (see e.g. [113]). We denote the value of the corresponding dual problem by D_k^o, where it is known that $Q_k^o \geq D_k^o$ (see [97]). To identify the "right" number of clusters we will analyze the curve $Q_k^o - D_k^o$.

Likewise, partition optimality is also an issue. Families of k–means clustering algorithms attempt to generate an optimal partition by applying a gradient type method that leads to a local minimum of the function Q_k. Enhancements of the batch k–means introduced recently by Kogan et al. in [29, 65] lead to significant improvement over the classical batch k–means algorithms.

Optimizing normalization methods and distance functions remains as a research issue. One such distance function worth investigating was recently published in [64]. This function, given in Equation 10.1, is motivated by optimization theory and is a generalization of a

distance function based upon Kullback-Leibler divergence. It measures the distance between a data vector $\mathbf{x}$ and a cluster centroid $\mathbf{c}$.

$$d(\mathbf{c}, \mathbf{x}) = \sum_{j=1}^{n} \mathbf{x}[j] \log \frac{\mathbf{x}[j]}{\mathbf{c}[j]} + \sum_{j=1}^{n} \mathbf{c}[j] - \sum_{j=1}^{n} \mathbf{x}[j] \tag{10.1}$$

References

[1] Julia Allen, Alan Christie, William Fithen, John McHugh, Jed Pickel, and Ed Stoner. State of the Practice of Intrusion Detection Technologies. Technical Report 99tr028, Carnegie Mellon - Software Engineering Institute, 2000.

[2] James P. Anderson. Computer Security, Threat Monitoring, and Surveillance. Technical report, James P. Anderson, Co., April 1980.

[3] William A. Arbaugh, D. J. Farber, and J. M. Smith. A secure and reliable bootstrap architecture. In *IEEE Symposium on Security and Privacy*, pages 65 – 71, 1997.

[4] Taimur Aslam, Ivan Krusl, and Eugene Spafford. Use of a Taxonomy of Security Faults. In *Proceedings of the 19th National Information Systems Security Conference*, October 1996.

[5] Lokesh Bajaj, Mineo Takai, Rajat Ahuja, Rajive Bagrodia, and Mario Gerla. Glomosim: A scalable network simulation environment. Technical Report 990027, 13, 1999.

[6] Stefano Basagni. Distributed clustering for ad hoc networks. In *Fourth International Symposium on Parallel Architectures, Algorithms, and Networks*, pages 310–315, June 1999.

[7] D. Bell and L. LaPadula. Secure computer system unified exposition and multics interpretation. Technical Report MTR-2997, MITRE, Bedford, MA, 1975.

[8] Mote. http://kingkong.me.berkeley.edu/ nota/RunningMan/Mote.htm. Page from the Smart Dust program giving an overview of the Berkeley Renee Mote.

[9] Vaduvur Bharghavan, Alan Demers, Scott Shenker, and Lixia Zhang. MACAW: a media access protocol for wireless LAN's. In *Proceedings of the Conference on Com-*

munications Architectures, Protocols and Applications, pages 212–225. ACM Press, 1994.

[10] Daniel Boley. Principal direction divisive partitioning. In *Data Mining and Knowledge Discovery*, number 2 in 4, pages 325 – 344, 1998.

[11] Philippe Bonnet, J. E. Gehrke, and Praveen Seshadri. Toward sensor database systems. In *Second International Conference on Mobile Data Management*, Hong Kong, January 2001.

[12] Dan Brickley and R.V. Guha. RDF Vocabulary Description Language 1.0: RDF Schema. http://www.w3c.org/TR/rdf-schema/, 2003.

[13] Sonja Buchegger and Jean-Yves Le Boudex. Nodes bearing grudges: Toward routing security, fairness, and robustness in mobile ad hoc networks. In *Proceedings of the 10th Euromicro Workshop on Parallel, Distributed and Network-based Processing*. IEEE Computer Society, 2002.

[14] Security Focus Bugtraq. Vulnerabilities::help. http://www.securityfocus.com/bid/4877/help, 2003.

[15] CERT Coordinating Center. 2002 annual report. http://www.cert.org/annual200s.

[16] David D. Clark. IP datagram reassembly algorithms. IETF, July 1982.

[17] P. C.Mahalanobis. *On Tests and Measures of Groups Divergence*. International Journal of the Asiatic Society of Bengal, 1930.

[18] W. Cohen. Fast effective rule induction. In *Proceedings of the 12th International Conference on Machine Learning*. Morgan Kaufman, 1995.

[19] Mitre Corporation. Common vulnerabilities and exposures. http://cve.mitre.org/, 2003.

[20] Paul J. Criscuolo. Distributed denial of service. Technical Report CIAC-2319, Lawrence Livermore National Laboratory, February 2000.

[21] D. Curry and H. Debar. ”intrusion detection message exchange format data model and extensible markup language (xml) document type definition. http://www.ietf.org/internet-drafts/draft-ietf-idwg-idmef-xml-10.txt, January 2003.

[22] Dark-Angel. Phantasmagoria. http://online.securityfocus.com/archive/1/290724, September 2002.

[23] Randall Davis, Howard Shrobe, and Peter Szolovits. What is Knowledge Representation? *AI Magazine*, 14(1):17 – 33, 1993.

[24] Herve Debar and Benjamin Morin. Evaluation of the diagnostic capabilities of commercial intrusion detection systems. In *Recent Advances in Intrusion Detection*, LNCS 2516. Springer, September 2002.

[25] Transmission control protocol. IETF, September 1981.

[26] Dorothy Denning. *Cryptography and Data Security*. Addison-Wesley, Boston, MA, 1982.

[27] Dorothy E. Denning. An Intrusion Detection Model. *IEEE Transactions on Software Engineering*, SE 13(2):222–232, February 1987.

[28] Dorothy E. Denning. *Information Warfare and Security*. Addison Wesley, 2001.

[29] I.S. Dhillon, J. Kogan., and C. Nicholas. Feature selection and document clustering. In M.W. Berry, editor, *A Comprehensive Survey of Text Mining*, pages 73–100. Springer-Verlag, 2003.

[30] Jon Doyle. Some Representational Limitations of the Common Intrusion Specification Language. Technical report, MIT, November 1999.

[31] Jon Doyle, Isaac Kohane, William Long, Howard Shrobe, and Peter Szolovits. Event Recognition Beyond Signature and Anomaly. In *2nd IEEE-SMC Information Assurance Workshop*, June 2001.

[32] S. Eckmann, G. Vigna, and R. Kemmerer. STATL: An Attack Language for State-based Intrusion Detection. *Journal of Computer Security*, 10(1/2):71 – 104, 2002.

[33] Eleazar Eskin, Wenke Lee, and Salvatore Stolfo. Modeling system calls for intrusion detection with dynamic window sizes. In *Proceedings of DARPA Information Survivability Conference and Exposition II (DISCEX I)*, volume 1, pages 165 – 175, June 2001.

[34] Rich Feiertag, Cliff Kahn, Phil Porras, Dan Schackenberg, Stuart Staniford-Chen, and Brian Tung. A Common Intrusion Specification Language. http://www.isi.edu/ brian/cidf/drafts/language.txt, June 1999.

[35] Richard Fikes, Jessica Jenkins, and Gleb Frank. Jtp: A system architecture and component library for hybrid reasoning. In *Proceedings of the Seventh World Multi conference on Systemics, Cybernetics, and Informatics*. Stanford, July 2003.

[36] E. Forgy. Cluster analysis of multivariate data: Efficiency vs. interpretability of classifications. *Biometrics*, 21(3):768, 1965.

[37] Stephanie Forrest, Steven A. Hofmeyr, Anil Somayaji, and Thomas Longstaff. A sense of self for unix processes. In *1996 IEEE Symposium on Security and Privacy*, pages 120 – 128. IEEE Computer Society Press, 1996.

[38] Ernest J. Friedman-Hill. Jess, The JAVA Expert System Shell. http://herzberg.ca.sandia.gov/jess/docs/52/, November 1997.

[39] Gartner information security hype cycle declares intrusion detection systems a market failure. http://yada-yada.com, June 2003.

[40] Joseph Giarratano and Gary Riley. *Expert Systems Principles and Programming*. PWS Publishing Company, third edition, 1998.

[41] Robert L. Glass and Iris Vessey. Contemporary Application-Domain Taxonomies. *IEEE Software*, pages 63 – 76, July 1995.

[42] Dina Q. Goldin and P.C. Kanellakis. On similarity queries for time series data: Constraint specification and implementation. In *First International Conference on the Principles and Practice of Constraint Programming*, pages 137 – 153. Springer - LNCS 976, September 1995.

[43] Jean Goubault-Larrecq. An Introduction to LogWeaver (v2.8). http://www.lsv.ens-cachan.fr/ goubault/DICO/tutorial.pdf, September 2001.

[44] NSS Group. Intrusion Detection and Vulnerability Assessment Group Test, Edition 2. Technical report, NSS Group, The NSS Group Ltd., 483 Green Lanes, London N13 4BS, England, 2002.

[45] T. F. Gruber. A Translation Approach to Portable Ontologies. *Knowledge Acquisition*, 5(2):199–220, 1993.

[46] Volker Haarslev and Ralf Moller. RACER: Renamed ABox and Concept Expression Reasoner. http://www.cs.concordia.ca/ faculty/haarslev/racer/index.html, June 2001.

[47] Z. J. Haas and M. Perlman. The performance of query control schemes for zone routing protocol. In *SIGCOMM'98*, 1998.

[48] Joshua W. Haines, Lee M. Rossey, Richard P. Lippman, and Robert K. Cunningham. Extending the DARPA Off-Line Intrusion Detection Evaluations. In *DARPA Information Survivability Conference and Exposition II*, volume 1, pages 77 – 88. IEEE, 2001.

[49] I. Horrocks, U. Sattler, and S. Tobies. Reasoning with Individuals for the Description Logic SHIQ. In *Proceedings of the 17th International Conference on Automated Deduction*, number 1831. Springer-Verlag, 2000.

[50] John Howard. *An Analysis of Security Incidents on the Internet*. PhD thesis, Carnegie Mellon University, 1997.

[51] Galen Hunt and Doug Brubacher. Detours: Binary interception of win32 functions. In *Proceedings of the Third USENIX Windows NT Symposium*, 1999.

[52] IEEE. *Std 802.11, 1999 Edition*, R2003 edition, 1999.

[53] Chalermek Intanagonwiwat, Ramesh Govindan, and Deborah Estrin. Directed diffusion: a scalable and robust communication paradigm for sensor networks. In *Mobile Computing and Networking*, 2000.

[54] J. Hendler. DARPA Agent Markup Language+Ontology Interface Layer. http://www.daml.org/2001/03/daml+oil-index, 2001.

[55] Mingliang Jiang, Jinyang Li, and Y.C. Tay. *INTERNET-DRAFT : Cluster Based Routing Protocol (CBRP)*. IETF.

[56] T. Joachims. *Making Large-Scale SVM Learning Practical*, chapter 11. MIT Press, 1999.

[57] David B Johnson and David A Maltz. Dynamic source routing in ad hoc wireless networks. In Imielinski and Korth, editors, *Mobile Computing*, volume 353. Kluwer Academic Publishers, 1996.

[58] I.T. Jolliffee. *Principal Component Analysis*. Springer Series in Statistics. Springer, second edition, 2002.

[59] Cliff Kahn, Dan Bolinger, and Don Schackenberg. Communication in the Common Intrusion Detection Framework v 0.7. http://www.isi.edu/ brian/cidf/drafts/communication.txt, June 1998.

[60] C. Karlof and D. Wagner. Secure Routing in Sensor Networks: Attacks and Countermeasures. In *Proc. of First IEEE International Workshop on Sensor Network Protocols and Applications*, May 2003.

[61] Richard A. Kemmerer and Giovanni Vigna. Intrusion Detection: A Brief History and Overview. *Security and Privacy a Supplement to IEEE Computer Magazine*, pages 27 – 30, April 2002.

[62] Kristopher Kendall. A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems. Master's thesis, MIT, 1999.

[63] Florian Kerschbaum, Eugene Spafford, and Diego Zamboni. Using Embedded Sensors for Detecting Network Attacks. In *Proceedings of the First ACM Workshop on Intrusion Detection Systems*, November 2000.

[64] J. Kogan, M. Teboulle, and C. Nicholas. Optimization approach to generating families of k–means like algorithms. In I. Dhillon and J. Kogan, editors, *Proceedings of the Workshop on Clustering High Dimensional Data and its Applications (held in conjunction with the Third SIAM International Conference on Data Mining)*, 2003.

[65] J Kogan, M. Teboulle, and C. Nicholas. Data driven similarity measures for k–means like clustering algorithms. *Information Retrieval*, to appear.

[66] Anita Komladi, Penny Rheingans, Andrew Sears, John Goodall, and Jeff Undercoffer. Visualization of snort data. Tech Report, September 2003.

[67] Joe Kopena. DAMLJessKB. http://edge.mcs.drexel.edu/ assemblies/software/damljesskb/ articles/DAMLJessKB-2002.pdf, October 2002.

[68] Yordan Kostov and Govind Rao. Low cost optical instrumentation for biomedical measurement. *J. Review of Scientific Instruments*, pages 4361–4373, December 2000.

[69] P. Krishna and D.K Pradhan N.H. Vaidya, M. Chatterji. A cluster-based approach for routing in dynamic networks. Technical report, Department of Computer Science, Texas A and M University, 2000.

[70] Raghu Krishnapuram, Anupam Joshi, Olfa Nasraoui, and Liyu Yi. Low-Complexity Fuzzy Relational Clustering Algorithms for Web Mining. In *IEEE Transactions on Fuzzy Systems*, volume 9, August 2001.

[71] Joanna Kulik, Wendi Rabiner, and Hari Balakrishnan. Adaptive protocols for information dissemination in wireless sensor networks. In *5th ACM/IEEE Mobicom Conference*, Seattle, WA, August 1999.

[72] Carl E. Landwehr, Alan R. Bull, John P. McDermott, and William S. Choi. A Taxonomy of Computer Program Security Flaws. *ACM Computing Surveys*, 26(3):211 – 254, September 1994.

[73] David D. Lewis and William A. Gale. A sequential algorithm for training text classifiers. In W. Bruce Croft and Cornelis J. van Rijsbergen, editors, *Proceedings of SIGIR-94, 17th ACM International Conference on Research and Development in Information Retrieval*, pages 3–12, Dublin, IE, 1994. Springer Verlag, Heidelberg, DE.

[74] Ben Liang and Zygmunt J. Haas. Virtual backbone generation and maintenance in ad-hoc network mobility management. Technical report, School of Electrical Engineering, Cornell University, Ithaca, NY 14850, 2000.

[75] Ulf Lindqvist and Erland Jonsson. How to Systematically Classify Computer Security Intrusions. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy*, pages 154 – 163, May 1997.

[76] Ulf Lindqvist and Phillip A. Porras. Detecting computer and network misuse through the production-based system toolset (p-best). In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 146 – 161. IEEE, May 1999.

[77] Richard Lippmann, David Fried, Isaac Graf, Joshua Haines, Kristopher Kendall, David McClung, Dan Weber, Seth Webster, Dan Wyschogrod, Robert Cunningham, and Marc Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-line Intrusion Detection Evaluation. In *Proceedings of the DARPA Information Survivability Conference and Exposition,2000*, pages 12 – 26, January 2000.

[78] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, and C. Jalali. Ides: A progress report. In *Proceedings of the Sixth Annual Security Applications Conference*, pages 273 – 285, 1990.

[79] Sam Madden and Michael J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *ICDE Conference*, San Jose, CA, February 2002.

[80] Mathew Mahoney and Philip K. Chan. An Analysis of the 1999 DARPA/Lincoln Laboratory Evaluation Data for Network Anomaly Detection. In *Recent Advances in Intrusion Detection*, LNCS 2516. Springer, September 2003.

[81] Sergio Marti, Thomas J. Giuli, Kevin Lai, and Mary Baker. Mitigating routing misbehavior in mobile ad hoc networks. In *Proceedings of MOBICOM*, pages 255 – 265, 2000.

[82] Deborak L. McGuinnes and Frak van Harmelen. Owl web ontology language. http://www.w3c.org/TR/owl-features, August 2003.

[83] John McHugh. Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory. *ACM Transactions on Information and System Security*, November 2000.

[84] John McHugh, Alan Christie, and Julia Allen. Intrusion Detection Implementation and Operational Issues. www.stsc.hill.af.mil/crosstalk/2001/jan/mchugh.asp, January 2001.

[85] Internet catalog of assessable technologies. http://icat.nist.gov, 2003.

[86] Stephen Northcutt, Mark Cooper, Matt Fearnow, and Karen Frederick. *Intrusion Signature and Analysis*. SANS GIAC. New Riders, 201 West 103rd Street, Indianapolis, IN 46290, first edition, 2001.

[87] Stephen Northcutt, Judy Novak, and Donald McLachlan. *Network Intrusion Detection - An Analyst's Handbook*. SANS GIAC. New Riders, 201 West 103rd Street, Indianapolis, IN 46290, second edition, 2000.

[88] Natalya F. Noy and Deborah L. McGuinnes. Ontology development 101: A guide to creating your first ontology. Stanford University.

[89] Network simulator. http://www-mash.berkeley.edu/ns, 1996.

[90] S. Park, A. Savvides, and M.B. Srivastava. Sensorsim: A simulation framework for sensor networks. In *The Third ACM International Workshop on Modeling, Analysis and Simulation of Wireless and Mobile Systems*, 2000.

[91] Vincent D. Park and M. Scott Corson. A highly adaptive distributed routing algorithm for mobile wireless networks. In *Proceedings of IEEE INFOCOM '97*, 1997.

[92] Samuel Patton, William Yurcik, and David Doss. An Achilles' Heel in Signature Based IDS: Squealing False Positives in SNORT . In H. Debar, L. Me, and F. Wu, editors, *Recent Advances in Intrusion Detection (RAID 2000)*, number 1907 in Lecture Notes in Computer Science, pages 80–92, Toulouse, France, October 2000. Springer-Verlag.

[93] Vern Paxson. Bro: A system for Detecting Network Intruders in Real Time. In *Proceedings of the 7th Symposium on USENIX Security*, 1998.

[94] Charles E. Perkins and E. M. Royer. Ad hoc on demand distance vector routing. In *IEEE WMCSA'99*, 1999.

[95] Adrian Perrig, Robert Szewczyk, Victor Wen, David Culler, and J.D.Tygar. Spins: Security protocols for sensor networks. *Wireless Networks*, 8:521 – 534, 2002.

[96] Phillip A. Porras and Peter G. Neumann. EMERALD: Event Monitoring Enabling Responses to Anomalous Live Disturbances. In *1997 National Information Systems Security Conference*, Oct 1997.

[97] R. T. Rockefeller and R.J.B Wets. *Variational Analysis*. Springer–Verlag, New York, 1998.

[98] Marty Roesch. Snort, version 1.8.3. available via www.snort.org, August 2001. an open source NIDS.

[99] Muriel Roger and Jean Goubault-Larrecq. Log Auditing through Model Checking. In *Proceedings of 14th the IEEE Computer Security Foundations Workshop (CSFW'01)*, pages 220 – 236, 2001.

[100] M. Rose. Management Information Base for Network Management of TCP/IP-based Internets. IETF, May 1990.

[101] Anil Somayaji and Stephanie Forrest. Automated response using system-call delays. In *Proceedings of the Ninth USENIX Security Symposium*, pages 120 – 128, August 2000.

[102] Steffan Staab and Alexander Maedche. Ontology Engineering Beyond the Modeling of Concepts and Relations. In *Proceedings of the 14th European Congress on Artificial Intelligence*, 2000.

[103] Stealth. Adore. http://teso.scene.at/releases/adore-0.42.tgz. The Adore Root Kit.

[104] C. Sugar, L. Lenert, and R. Olshen. *An Application of Cluster Analysis to Health Services Research: Empirically Defined Health States for Depression from the SF-12*. http://www-stat.stanford.edu/ olshen/manuscripts, 2000.

[105] C.A. Sugar. and J.M. Gareth. Finding the number of clusters in a dataset: an information-theoretic approach. *Journal of the American Statistical Association*, 98(463):750–763, 2003.

[106] George Gaylord Sumpson. *Principles of Animal Taxonomy*. Columbia University Press, 1961.

[107] Kymie Tan, John McHugh, and Kevin Killourhy. Hiding intrusions: From the abnormal to the normal and beyond. In *Proceedings of the Information Hiding: 5th International Workshop, IH 2002*, volume 2578, pages 1 – 17. Springer, January 2003.

[108] Kymie M.C. Tan, Kevin S. Killourhy, and Roy A. Maxion. Undermining an anomaly-based intrusion detection system using common exploit. In *Fifth International Symposium on Recent Advances in Intrusion Detection (RAID-2002)*, LNCS-2516, pages 54–73. Springer-Verlag, Berlin, October 2002.

[109] Kymie M.C. Tan and Roy A. Maxion. Why 6? Defining the Operational Limits of stide, an Anomaly-Based Intrusion Detector. In *IEEE Symposium on Security and Privacy*, pages 188–201. IEEE Computer Society Press, May 2002.

[110] Kymie M.C. Tan and Roy A. Maxion. Determining the operational limits of an anomaly-based intrusion detector. *IEEE Journal on Selected Areas in Communications, Special Issue on Design and Analysis Techniques for Security Assurance*, 21(1):96–110, January 2003.

[111] Tim Bass. Intrusion Detection Systems and Multisensor Data Fusion, April 2000.

[112] Bob Toxen. *Real World Linux Security*. Prentice Hall, 1999.

[113] V. Vapnik. *Statistical Learning Theory*. Wiley, New York, 1998.

[114] W3C. Extensible Markup Language. http://www.w3c.org/XML/, 2003.

[115] David Wagner and Paolo Soto. Mimicry attacks on host-based intrusion detection systems. In *Ninth ACM Conference on Computer and Communications Security*, pages 255 – 264, 2002.

[116] Christina Warrender, Stephanie Forrest, and Barak Pearlmutter. Detecting intrusion using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy*, pages 133 – 145. IEEE Computer Society Press, 1999.

[117] David S. Watkins. *Fundamentals of Matrix Computations*. Wiley Inter Science, New York, New York, second edition, 2002.

[118] D. Weber. A taxonomy of computer intrusions. Master's thesis, MIT, 1998.

[119] Chris Welty. Toward a Semantics for the Web. www.cs.vassar.edu/faculty/welty/papers/dagstuhl-2000.pdf, 2000.

[120] Xue Wu and Michael Cukier. A practice of intrusion detection system evaluation. Technical report, University of Maryland, College Park, August 2003. www.cs.umd.edu/ wu/paper/paper3.pdf.

[121] William Wulf. Statement before the House Science Committee, U.S. House of Representatives, October 10 2001.

[122] William Wulf. A grand challenge in information security. Keynote Address: NSF Cyber Trust Point Meeting. Johns Hopkins Information Security Institute, Baltimore, MD, August 2003.

[123] Yongguang Zhang, Wenke Lee, and Yi-An Huang. Intrusion detection for wireless ad-hoc networks. In *Mobile Networks and Applications*. ACM, 2002.

[124] Lidong Zhou and Zygmunt J. Hass. Securing ad hoc networks. *IEEE Network*, 13(6):24 – 30, Nov/Dec 1999.

Appendix A

Format of the ICAT Meta-base

Table A.1 presents the fields of the ICAT meta-base and their possible values. Values enclosed in “quotes” are explicit values, otherwise they are a short description of the possible content.

Field	**Values**
Vulnerability Name	A numeric identifier linked to the Common Vulnerabilities and Exposures database maintained by Mitre
Published Before	When the vulnerability was discovered
Summary	An overview of the vulnerability
Severity	"high" "medium" "low"
Exploitable Range	"local" "remote"
Vulnerability Type	"Input validation Error" "Access validation error" "Exceptional condition handling error" "Environmental error" "Configuration error" "Race condition" "Design error" "Other"
Exposed System Component	"Operating system" "Protocol stack" "Server Application" "Non-server application" "Hardware" "Communication Protocol" "Encryption module" "Other"
Loss Type	"Availability" "Confidentiality" "Integrity" "Security protection" "root level access" "user level access" "other access"
Reference n	the source of the advisory(e.g. Cert or Bugtraq)
Vulnerable Software and Version	(e.g. Solaris 2.3)

Table A.1. Fields and Possible Values of the ICAT Meta-base

Appendix B

Format of the CERT Advisories

CERT Advisories, although following a prescribed format, are fairly free form. Table B.1 provides the format of a CERT/CC Advisory.

Field	**Values**
Name	Advisory number and vulnerability name
Original Release Date	Date first posted
Last Revised	date of latest revision
Source	Source of the information
Systems Affected	Systems and software affected
Overview	A synopsis of the vulnerability
Description	A detailed description of the vulnerability
Impact	The potential impact of the vulnerability
Solution	A solution to the vulnerability
Appendices	Information provided by vendors and References

Table B.1. Format of a CERT Advisory

Appendix C

Format of the Linux Process Descriptor

The Linux process descriptor is defined in the "C" structure task_struct and includes structures of type mm_struct, list_head, Linux_binfmt, completion, timer_list, tms, user_struct, tty_struct, thread_struct, fs_struct, files_struct, signal_struct, sigpending. The purpose of these structures is included in the listing of the task_struct in the form of comments. The listing of the memory structure, mm_struct, follows the process descriptor.

```
struct task_sruct {

        volatile long state;
        unsigned long flags;
        int sigpending;
        mm_segment_t addr_limit;
        struct exec_domain *exec_domain;
        volatile long need_resched;
        unsigned long ptrace;

/* Lock depth */
        int lock_depth;

        long counter;
        long nice;
        unsigned long policy;
        struct mm_struct *mm;
        int processor;

        unsigned long cpus_runnable, cpus_allowed;

        struct list_head run_list;
        unsigned long sleep_time;

        struct task_struct *next_task, *prev_task;
        struct mm_struct *active_mm;
        struct list_head local_pages;
        unsigned int allocation_order, nr_local_pages;

/* task state */
        struct linux_binfmt *binfmt;
        int exit_code, exit_signal;
```

```
        int pdeath_signal;  /*  The signal sent when the parent dies  */

/* related PIDs */
        unsigned long personality;
        int did_exec:1;
        pid_t pid;
        pid_t pgrp;
        pid_t tty_old_pgrp;
        pid_t session;
        pid_t tgid;

/* boolean value for session group leader */
        int leader;

/*
 * pointers to (original) parent process, youngest child, younger sibling,
 * older sibling, respectively.  (p->father can be replaced with
 * p->p_pptr->pid)
 */
        struct task_struct *p_opptr, *p_pptr, *p_cptr, *p_ysptr, *p_osptr;
        struct list_head thread_group;

/* PID hash table linkage. */
        struct task_struct *pidhash_next;
        struct task_struct **pidhash_pprev;

        wait_queue_head_t wait_chldexit;
        struct completion *vfork_done;
        unsigned long rt_priority;
        unsigned long it_real_value, it_prof_value, it_virt_value;
        unsigned long it_real_incr, it_prof_incr, it_virt_incr;
        struct timer_list real_timer;
        struct tms times;
        unsigned long start_time;
        long per_cpu_utime[NR_CPUS], per_cpu_stime[NR_CPUS];

/* faults */
        unsigned long min_flt, maj_flt, nswap, cmin_flt, cmaj_flt, cnswap;
        int swappable:1;

/* process credentials */
        uid_t uid,euid,suid,fsuid;
        gid_t gid,egid,sgid,fsgid;
        int ngroups;
        gid_t   groups[NGROUPS];
        kernel_cap_t   cap_effective, cap_inheritable, cap_permitted;
        int keep_capabilities:1;
        struct user_struct *user;

/* limits */
        struct rlimit rlim[RLIM_NLIMITS];
        unsigned short used_math;
        char comm[16];

/* file system info */
        int link_count, total_link_count;
        struct tty_struct *tty; /* NULL if no tty */
        unsigned int locks; /* How many file locks are being held */

/* ipc */
        struct sem_undo *semundo;
        struct sem_queue *semsleeping;

/* CPU-specific state of this task */
        struct thread_struct thread;

/* filesystem information */
        struct fs_struct *fs;
```

```
/* open file information */
        struct files_struct *files;

/* signal handlers */
        spinlock_t sigmask_lock;
        struct signal_struct *sig;

        sigset_t blocked;
        struct sigpending pending;
        unsigned long sas_ss_sp;
        size_t sas_ss_size;
        int (*notifier)(void *priv);
        void *notifier_data;
        sigset_t *notifier_mask;

/* Thread group tracking */
        u32 parent_exec_id;
        u32 self_exec_id;

/* Protection of (de-)allocation: mm, files, fs, tty */
        spinlock_t alloc_lock;

/* journalling filesystem info */
        void *journal_info;
};
```

Appendix D

Format of the Linux Memory Structure

```
struct mm_struct {

         struct vm_area_struct * mmap;
         rb_root_t mm_rb;
         struct vm_area_struct * mmap_cache;
         pgd_t * pgd;
         atomic_t mm_users;
         atomic_t mm_count;
         int map_count;
         struct rw_semaphore mmap_sem;
         spinlock_t page_table_lock;

         struct list_head mmlist;

 /* List of all active mm's.  These are globally strung
  * together off init_mm.mmlist, and are protected
  * by mmlist_lock
  */
         unsigned long start_code, end_code, start_data, end_data;
         unsigned long start_brk, brk, start_stack;
         unsigned long arg_start, arg_end, env_start, env_end;
         unsigned long rss, total_vm, locked_vm;
         unsigned long def_flags;
         unsigned long cpu_vm_mask;
         unsigned long swap_address;

         unsigned dumpable:1;

/* Architecture-specific MM context */
         mm_context_t context;
 };
```

Appendix E

DAML+OIL Target Centric Ontology

```
<?xml version='1.0' encoding='ISO-8859-1'?>
<rdf:RDF
  xmlns:rdf ="http://www.w3.org/1999/02/22-rdf-syntax-ns#"
  xmlns:rdfs="http://www.w3.org/2000/01/rdf-schema#"
  xmlns:daml="http://www.daml.org/2001/03/daml+oil#"
  xmlns:xsd ="http://www.w3.org/2000/10/XMLSchema#"
  xmlns     ="http://security.umbc.edu/IDS#">


<daml:Ontology rdf:about="">
  <daml:versionInfo>$Id: att-cent-ont.tex,v 1.7 2004/01/05 23:23:19 junder2 Exp $</daml:versionInfo>
     <rdfs:comment>
       UMBC IDS ontology
       Jeffrey L. Undercoffer   12 December 2003
     </rdfs:comment>
      <daml:imports rdf:resource="http://www.daml.org/2001/03/daml+oil"/>
</daml:Ontology>


<!--  ############ ########### ############ ############ ########### -->
<!--
                           Base Classes
-->
<!--  ############ ########### ############ ############ ########### -->



<daml:Class rdf:ID="System">
  <rdfs:label>System</rdfs:label>
</daml:Class>

<daml:Class rdf:ID="Process">
  <rdfs:label>Process</rdfs:label>
</daml:Class>

<daml:Class rdf:ID="Network">
  <rdfs:label>Network</rdfs:label>
</daml:Class>


<daml:Class rdf:ID="Input">
</daml:Class>

<daml:Class rdf:ID="Means">
```

```
</daml:Class>

<daml:Class rdf:ID="Consequence">
</daml:Class>

<daml:Class rdf:ID="UserLocation">
</daml:Class>

<!--  ############ ########### ############ ############ ########### -->
<!--
  Consequence: Denial of Service
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="DoS">
  <rdfs:subClassOf rdf:resource="#Consequence"/>
</daml:Class>

<daml:Class rdf:ID="SynFlood">
  <rdfs:subClassOf rdf:resource="#DoS"/>
</daml:Class>

<daml:Class rdf:ID="PoD">
  <rdfs:subClassOf rdf:resource="#DoS"/>
</daml:Class>

<daml:Class rdf:ID="IpFrag">
  <rdfs:subClassOf rdf:resource="#DoS"/>
</daml:Class>

<daml:Class rdf:ID="NetFlood">
  <rdfs:subClassOf rdf:resource="#DoS"/>
</daml:Class>

<daml:Class rdf:ID="SysCrash">
  <rdfs:subClassOf rdf:resource="#DoS"/>
</daml:Class>

<daml:Class rdf:ID="ExcessForks">
  <rdfs:subClassOf rdf:resource="#DoS"/>
</daml:Class>


<!--  ############ ########### ############ ############ ########### -->
<!--
         Consequence:  Probes
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="Probe">
  <rdfs:subClassOf rdf:resource="#Consequence"/>
</daml:Class>

<daml:Class rdf:ID="TcpConnect">
  <rdfs:subClassOf rdf:resource="#Probe"/>
</daml:Class>

<daml:Class rdf:ID="PingScan">
  <rdfs:subClassOf rdf:resource="#Probe"/>
</daml:Class>

<daml:Class rdf:ID="SynScan">
  <rdfs:subClassOf rdf:resource="#Probe"/>
</daml:Class>

<daml:Class rdf:ID="RSTProbe">
```

```
  <rdfs:subClassOf rdf:resource="#Probe"/>
</daml:Class>


<!--  ############ ########### ############ ############ ########### -->
<!--
          Consequence: Privledge Escelation & Access
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="PrivledgeEsc">
  <rdfs:subClassOf rdf:resource="#Consequence"/>
</daml:Class>

<daml:Class rdf:ID="unAuthRoot">
  <rdfs:subClassOf rdf:resource="#Consequence"/>
</daml:Class>

<daml:Class rdf:ID="unAuthUser">
  <rdfs:subClassOf rdf:resource="#Consequence"/>
</daml:Class>


<!--  ############ ########### ############ ############ ########### -->
<!--
       Consequence: Loss of Confideniality
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="LossOfConf">
  <rdfs:subClassOf rdf:resource="#Consequence"/>
</daml:Class>


<daml:Class rdf:ID="DirectoryExposure">
  <rdfs:subClassOf rdf:resource="#LossOfConf"/>
</daml:Class>



<!--  ############ ########### ############ ############ ########### -->
<!--
       Means: Input Validation Errors
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="InputValidErr">
  <rdfs:subClassOf rdf:resource="#Means"/>
</daml:Class>

<daml:Class rdf:ID="BoundCond">
  <rdfs:subClassOf rdf:resource="#InputValidErr"/>
</daml:Class>

<daml:Class rdf:ID="MalformedIn">
  <rdfs:subClassOf rdf:resource="#InputValidErr"/>
</daml:Class>


<!--  ############ ########### ############ ############ ########### -->
<!--
       Means: Logic Exploits
  -->
<!--  ############ ########### ############ ############ ########### -->
```

```
<daml:Class rdf:ID="LogicExploit">
  <rdfs:subClassOf rdf:resource="#Means"/>
</daml:Class>

<daml:Class rdf:ID="ExceptCond">
  <rdfs:subClassOf rdf:resource="#LogicExploit"/>
</daml:Class>

<daml:Class rdf:ID="RaceCond">
  <rdfs:subClassOf rdf:resource="#LogicExploit"/>
</daml:Class>

<daml:Class rdf:ID="SerialError">
  <rdfs:subClassOf rdf:resource="#LogicExploit"/>
</daml:Class>

<daml:Class rdf:ID="AtError">
  <rdfs:subClassOf rdf:resource="#LogicExploit"/>
</daml:Class>



<!--  ############ ########### ############ ############ ########### -->
<!--
       Means: Configuration Error
  -->
<!--  ############ ########### ############ ############ ########### -->

<daml:Class rdf:ID="ConfigError">
  <rdfs:subClassOf rdf:resource="#Means"/>
</daml:Class>


<!--  ############ ########### ############ ############ ########### -->
<!--
      Locations
  -->
<!--  ############ ########### ############ ############ ########### -->

<daml:Class rdf:ID="viaTTY">
  <rdfs:subClassOf rdf:resource="#UserLocation"/>
</daml:Class>

<daml:Class rdf:ID="viaNetwork">
  <rdfs:subClassOf rdf:resource="#UserLocation"/>
</daml:Class>




<!--  ############ ########### ############ ############ ########### -->
<!--
      Quantifying Classes
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="Rate">
  <daml:oneOf rdf:parseType="daml:collection">
    <Rate rdf:ID="Rate_WB_Normal"/>
    <Rate rdf:ID="Rate_B_Normal"/>
    <Rate rdf:ID="Rate_Normal"/>
    <Rate rdf:ID="Rate_A_Normal"/>
    <Rate rdf:ID="Rate_WA_Normal"/>
  </daml:oneOf>
</daml:Class>
```

```
<daml:Class rdf:ID="Amount">
  <daml:oneOf rdf:parseType="daml:collection">
    <Amount rdf:ID="Amount_WB_Normal"/>
    <Amount rdf:ID="Amount_B_Normal"/>
    <Amount rdf:ID="Amount_Normal"/>
    <Amount rdf:ID="Amount_A_Normal"/>
    <Amount rdf:ID="Amount_WA_Normal"/>
    <Amount rdf:ID="Amount_Inf"/>
    </daml:oneOf>
</daml:Class>

<daml:Class rdf:ID="BoolValue">
  <daml:oneOf rdf:parseType="daml:collection">
    <BoolValue rdf:ID="True"/>
    <BoolValue rdf:ID="False"/>
  </daml:oneOf>
</daml:Class>



<!--  ############ ########### ############ ############ ########### -->
<!--
      Lower Ontology
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:Class rdf:ID="anomSelfDist">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
  </daml:unionOf>
</daml:Class>


<daml:Class rdf:ID="BufferOverFlow">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#unkRetAdd"/>
      <daml:hasClass rdf:resource="#True"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#vmCodeSize"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#totVmSize"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
      <daml:Restriction>
      <daml:onProperty rdf:resource="#numMinFault"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
```

```
</daml:Class>


<daml:Class rdf:ID="anomProcess">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#unkRetAdd"/>
      <daml:hasClass rdf:resource="#True"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#resSetSize"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
     <daml:onProperty rdf:resource="#codeSize"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="netAnomPackets">
  <daml:unionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipInRecvs"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipInRecvs"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:unionOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipInDisc"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipReasmOk"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="sysWMemConsum">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#memUsed"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#swapUsed"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#conSwitch"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#swapIn"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
```

```
    <daml:Restriction>
      <daml:onProperty rdf:resource="#swapOut"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#pagesIn"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#pagesOut"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="procWMemCons">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
     <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
   <daml:Restriction>
     <daml:onProperty rdf:resource="#stackSize"/>
     <daml:hasClass rdf:resource="#Amount_A_Normal"/>
       </daml:Restriction>
     <daml:Restriction>
       <daml:onProperty rdf:resource="#resSetSize"/>
       <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
     </daml:Restriction>
     <daml:Restriction>
       <daml:onProperty rdf:resource="#numChildProcs"/>
       <daml:hasClass rdf:resource="#Amount_A_Normal"/>
     </daml:Restriction>
     <daml:Restriction>
       <daml:onProperty rdf:resource="#numMinFault"/>
       <daml:hasClass rdf:resource="#Amount_A_Normal"/>
     </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="rstProbe">
  <rdfs:subClassOf rdf:resource="#Probe"/>
  <rdfs:subClassOf rdf:resource="#Means"/>
  <rdfs:subClassOf rdf:resource="#Network"/>
    <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpOutMsg"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpEchoResp"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstabRst"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
   <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpOutRst"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>
```

```
<daml:Class rdf:ID="synFlood">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <rdfs:subClassOf rdf:resource="#Network"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstb"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpSynRec"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="exIcmpEchoReq">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <rdfs:subClassOf rdf:resource="#Network"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstb"/>
      <daml:hasClass rdf:resource="#Amount_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpInRecvs"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpInEcho"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="tcpPortscan1">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpOutMsg"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpOutEchoResp"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstabRst"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpOutRst"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="tcpPortscan2">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpOutMsg"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#icmpOutEchoResp"/>
```

```
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstabRst"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpOutRst"/>
      <daml:hasClass rdf:resource="#Rate_A_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="exIpPacketSize">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipInOutReq"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipReAsmReq"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="ipFrag">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <rdfs:subClassOf rdf:resource="#Network"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#tcpEstb"/>
      <daml:hasClass rdf:resource="#Amount_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#ipInOutReq"/>
      <daml:hasClass rdf:resource="#Rate_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="trojan1">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Restriction>
      <daml:onProperty rdf:resource="#selfDist"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#codeSize"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#resSetSize"/>
      <daml:hasClass rdf:resource="#Amount_A_Normal"/>
    </daml:Restriction>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#numChildProcs"/>
      <daml:hasClass rdf:resource="#Amount_WA_Normal"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>
```

```
<daml:Class rdf:ID="trojan2">
  <rdfs:subClassOf rdf:resource="#Means"/>
  <daml:intersectionOf rdf:parseType="daml:collection">
     <daml:Restriction>
        <daml:onProperty rdf:resource="#unkRetAdd"/>
         <daml:hasClass rdf:resource="#True"/>
     </daml:Restriction>
     <daml:Restriction>
         <daml:onProperty rdf:resource="#selfDist"/>
         <daml:hasClass rdf:resource="#Amount_Inf"/>
     </daml:Restriction>
   </daml:intersectionOf>
</daml:Class>







<!--  ############ ########### ############ ############ ########### -->
<!--
      Middle Ontology
  -->
<!--  ############ ########### ############ ############ ########### -->



<daml:Class rdf:ID="ProcessUnderMemoryExploit">
  <rdfs:subClassOf rdf:resource="#ProcessUnderExploit" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#procWithMemProb"/>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="ProcessUnderInputValidErr">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasinducedstate"/>
      <daml:hasClass rdf:resource="#InValidError"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="ProcessUnderExploit">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasinducedstate"/>
      <daml:hasClass rdf:resource="#LogicExploit"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="ProcessUnderBufferOverFlow">
  <rdfs:subClassOf rdf:resource="#ProcessUnderInputValidErr" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#BufferOverFlow"/>
  </daml:intersectionOf>
</daml:Class>
```

```
<daml:Class rdf:ID="ProcessWithAnomSelfDist">
  <rdfs:subClassOf rdf:resource="#Process" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#anomSelfDist"/>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="ProcessUnderTrojan">
  <rdfs:subClassOf rdf:resource="#ProcessUnderInputValidErr" />
  <daml:unionOf rdf:parseType="daml:collection">
   <daml:Class>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#trojan1"/>
  </daml:intersectionOf>
 </daml:Class>
   <daml:Class>
   <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Process"/>
    <daml:Class rdf:about="#trojan2"/>
  </daml:intersectionOf>
</daml:Class>
</daml:unionOf>
</daml:Class>


<daml:Class rdf:ID="NetworkUnderSynFlood">
  <rdfs:subClassOf rdf:resource="#NetworkUnderDoS" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Network"/>
    <daml:Class rdf:about="#synFlood"/>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="NetworkUnderTcpPortscan">
  <rdfs:subClassOf rdf:resource="#ProcessUnderInputValidErr" />
  <daml:unionOf rdf:parseType="daml:collection">
   <daml:Class>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Network"/>
    <daml:Class rdf:about="#tcpPortScan1"/>
  </daml:intersectionOf>
 </daml:Class>
   <daml:Class>
   <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Network"/>
    <daml:Class rdf:about="#tcpPortScan2"/>
  </daml:intersectionOf>
</daml:Class>
</daml:unionOf>
</daml:Class>


<daml:Class rdf:ID="NetworkUnderSynProbe">
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue><SYNProbe /></daml:hasValue>
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
```

```
      <daml:hasClass rdf:resource="#SYNProbe"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="NetworkUnderRstProbe">
  <rdfs:subClassOf rdf:resource="#NetworkUnderProbe" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass><Probe /></daml:hasClass>
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
      <daml:hasClass rdf:resource="#Probe"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="NetworkUnderExIpPacketSize">
  <rdfs:subClassOf rdf:resource="#Network" />
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#Network"/>
    <daml:Class rdf:about="#exIpPacketSize"/>
  </daml:intersectionOf>
</daml:Class>




<!--  ############ ########### ############ ############ ########### -->
<!--
      Upper Ontology
  -->
<!--  ############ ########### ############ ############ ########### -->



<daml:Class rdf:ID="SystemUnderUnAuthRoot">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#unAuthRoot"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderUnAuthUser">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#unAuthUser"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:ID="SystemUnderDoS">
  <daml:intersectionOf rdf:parseType="daml:collection">
```

```
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#DoS"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderProbe">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#Probe"
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderLossofConf">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#LossofConf"
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderExploit">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#SystemUnderMemoryAttack"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasProcess"/>
      <daml:hasClass rdf:resource="#ProcessUnderMemoryExploit"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderMitnickAttack">
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
     <daml:hasValue rdf:resource="#unAuthRoot" />
     </daml:Restriction>
    </rdfs:subClassOf>
    <daml:unionOf rdf:parseType="daml:collection">
    <daml:Class>
      <daml:intersectionOf rdf:parseType="daml:collection">
        <daml:Class rdf:about="#SystemUnderDoSAttack"/>
        <daml:Restriction>
          <daml:onProperty rdf:resource="#connectedTo"/>
          <daml:hasClass rdf:resource="#SystemUnderProbe"/>
        </daml:Restriction>
      </daml:intersectionOf>
    </daml:Class>
    <daml:Class>
      <daml:intersectionOf rdf:parseType="daml:collection">
        <daml:Class rdf:about="#SystemUnderProbe"/>
        <daml:Restriction>
          <daml:onProperty rdf:resource="#connectedTo"/>
          <daml:hasClass rdf:resource="#SystemUnderDoSAttack"/>
        </daml:Restriction>
      </daml:intersectionOf>
    </daml:Class>
```

```
  </daml:unionOf>
</daml:Class>


<daml:Class rdf:ID="SystemCompromisedByTrojan">
  <rdfs:subClassOf rdf:resource="#SystemUnderInputValidErr" />
  <rdfs:subClassOf rdf:resource="#System" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#unAuthRoot" />
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasProcess"/>
      <daml:hasClass rdf:resource="#ProcessUnderTrojan"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderMemConsAttack">
  <rdfs:subClassOf rdf:resource="#SystemUnderDoSAttack" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#DoS" />
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Class rdf:about="#MemConsum"/>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderDoSAttack">
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#DoS" />
      <daml:hasClass><DoS /></daml:hasClass>
    </daml:Restriction>
    <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#DoS"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderProbe">
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#Probe" />
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasClass rdf:resource="#Probe"/>
    </daml:Restriction>
```

```
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderRstProbe">
  <rdfs:subClassOf rdf:resource="#SystemUnderProbe"/>
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#RSTProbe" />
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
      <daml:hasClass rdf:resource="#RSTProbe"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderSynFloodAttack">
  <rdfs:subClassOf rdf:resource="#System" />
  <rdfs:subClassOf rdf:resource="#SystemUnderDoSAttack" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue   rdf:resource="#DoS"/ >
      </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
      <daml:hasClass rdf:resource="#NetworkUnderSynFlood"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemCompromisedByBufferOverFlow">
  <rdfs:subClassOf rdf:resource="#SystemUnderInputValidErr" />
  <rdfs:subClassOf rdf:resource="#System" />
    <rdfs:subClassOf>
    <daml:Restriction>
     <daml:onProperty rdf:resource="#experiencing"/>
     <daml:hasValue rdf:resource="#unAuthRoot" />
    </daml:Restriction>
   </rdfs:subClassOf>
    <daml:intersectionOf rdf:parseType="daml:collection">
      <daml:Class rdf:about="#System"/>
      <daml:Restriction>
<daml:onProperty rdf:resource="#hasProcess"/>
<daml:hasClass rdf:resource="#ProcessUnderBufferOverFlow"/>
      </daml:Restriction>
    </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemWithAnomProcess">
  <rdfs:subClassOf rdf:resource="#SystemUnderDoSAttack" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#unAuthUser" />
    </daml:Restriction>
  </rdfs:subClassOf>
```

```
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasProcess"/>
      <daml:hasClass rdf:resource="#ProcessWithAnomSelfDist"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemWithAnomNet">
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
      <daml:hasClass rdf:resource="#netAnomPackets"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>

<daml:Class rdf:ID="SystemUnderTcpPortScan">
  <rdfs:subClassOf rdf:resource="#System" />
  <rdfs:subClassOf>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#experiencing"/>
      <daml:hasValue rdf:resource="#Probe" />
    </daml:Restriction>
  </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
      <daml:hasClass rdf:resource="#NetworkUnderTcpPortscan"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<daml:Class rdf:ID="SystemUnderExIpPacketSizeAttack">
   <rdfs:subClassOf rdf:resource="#System" />
  <rdfs:subClassOf>
  <daml:Restriction>
   <daml:onProperty rdf:resource="#experiencing"/>
   <daml:hasValue rdf:resource="#DoS" />
  </daml:Restriction>
 </rdfs:subClassOf>
  <daml:intersectionOf rdf:parseType="daml:collection">
    <daml:Class rdf:about="#System"/>
    <daml:Restriction>
      <daml:onProperty rdf:resource="#hasNetwork"/>
      <daml:hasClass rdf:resource="#NetworkUnderExIpPacketSize"/>
    </daml:Restriction>
  </daml:intersectionOf>
</daml:Class>


<!--  ############ ########### ############ ############ ########### -->
<!--
      Properties
  -->
<!--  ############ ########### ############ ############ ########### -->


<daml:ObjectProperty rdf:ID="experiencing">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Consequence"/>
</daml:ObjectProperty>
```

```
<daml:ObjectProperty rdf:ID="connectedTo">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#System"/>
  <daml:inverseOf rdf:resource="#connectedTo" />
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="hasNetwork">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Process"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="hasProcess">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Process"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="hasinducedstate">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Means"/>
</daml:ObjectProperty>


<!--  System Properties -->


<daml:ObjectProperty rdf:ID="memUsed">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="swapUsed">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="cpuOne">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="cpufive">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="cputen">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numProcs">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numUsers">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
```

```
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="timeUserMode">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="timeUserModeLow">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="timeSysMode">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="timeIdle">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="swapIn">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="swapOut">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="pagesIn">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="pagesOut">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="conSwitch">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="rateProcs">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#System"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>



<!--  Process Properties -->
```

```
<daml:ObjectProperty rdf:ID="selfDist">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="unkRetAdd">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#BoolValue"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="niceValue">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="codeSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="vmCodeSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="libSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="dataSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="stackSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="vmEnvSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="totVmSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="resSetSize">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="lockedVm">
```

```
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numOpenFiles">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numChildProcs">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numMajFault">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numMinFault">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numChMajFault">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numChMinFault">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numCnSwap">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="numLinkCount">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="chgPPid">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#BoolValue"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="rateUTime">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="rateSTime">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
```

```
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="rateChUTime">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="rateChSTime">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="chgUid">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#BoolValue"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="chgGid">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#BoolValue"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="chgSUid">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Process"/>
  <rdfs:range rdf:resource="#BoolValue"/>
</daml:ObjectProperty>



<!--  Network Properties -->

<daml:ObjectProperty rdf:ID="tcpEstb">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="tcpSynRec">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="tcpListen">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Amount"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="ipInRecvs">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="ipRasmOk">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="ipInDisc">
```

```
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="ipInDeliv">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="ipInOutReq">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="icmpInMsg">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="icmpInEcho">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="icmpInEchoRep">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="udpIn">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="udpNoPort">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="udpInErr">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="udpOut">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="icmpOutMsg">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="icmpOutEchoResp">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
```

```
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="tcpToAlg">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="tcpEstabRst">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

<daml:ObjectProperty rdf:ID="tcpOutRst">
  <rdf:type rdf:resource="http://www.daml.org/2001/03/daml+oil#UniqueProperty"/>
  <rdfs:domain rdf:resource="#Network"/>
  <rdfs:range rdf:resource="#Rate"/>
</daml:ObjectProperty>

</rdf:RDF>
```