**CERIAS Tech Report 2004-26**

# A CATEGORIZATION OF COMPUTER SECURITY MONITORING SYSTEMS AND THE IMPACT ON THE DESIGN OF AUDIT SOURCES

by Benjamin A. Kuperman

Center for Education and Research in
Information Assurance and Security,
Purdue University, West Lafayette, IN 47907-2086

# A CATEGORIZATION OF COMPUTER SECURITY MONITORING SYSTEMS AND THE IMPACT ON THE DESIGN OF AUDIT SOURCES

A Thesis

Submitted to the Faculty

of

Purdue University

by

Benjamin A. Kuperman

In Partial Fulfillment of the

Requirements for the Degree

of

Doctor of Philosophy

August 2004

# ACKNOWLEDGMENTS

There are a number of individuals and organizations that I would like to recognize for their assistance and support throughout the many years of my graduate career.

First, I would like to note that portions of this work were supported by a gift from the Intel Corporations, Grant EIA-9903545 from the National Science Foundation, and the sponsors of the Purdue Center for Education and Research in Information Assurance and Security.

I would like to personally thank the various sponsors of COAST and CERIAS for both their financial and intellectual support. Many of the opportunities I have had to research computer security would not have been possible without you. I would like to especially thank Hewlett-Packard Corporation including Mark Crosbie, John Trudeau, and Martin Sadler for a wonderful and productive internship and continuing relationship over the years.

I would also like to thank my committee for all of their input, feedback, and support, especially my advisor Gene Spafford. Without his assistance and the opportunities he has made available, I doubt I would be where I am today. I owe a special debt of thanks to Professor Carla Brodley for her advice and encouragement. She kept me on the path whenever I seemed to be straying too far.

One of the unfortunate characteristics of academic institutions is that sometimes the staff is underrecognized for all of their hard work. You have all helped me in countless ways. However, I would like to thank Marlene Walls for keeping Spaf on schedule (well, as far as that is actually possible) and laughing with me at some of the more ridiculous situations that sometimes arose, and Mary Jo Maslin for our conversations and her willingness to share the excitement and optimism that fills her life.

I would be remiss were I not to recognize the technical staff for all of their work on the various systems I have used as well as putting up with my stream of problems, suggestions, and software requests. My thanks to Dan Trinkle, Vince Koser, Kent Wert, Adam Hammer, Ed Finkler, and Ed Cates.

There are a number of former Purdue students whose friendship and advice has been invaluable to me. I would like to thank Sofie Nystrøm for her willingness to drag me off to the gym or dinner for much needed breaks, Tom Daniels even though he sometimes was a "time vortex" that consumed many an afternoon, and Diego Zamboni for helping me keep alive my software and geek-toys habits.

I would also like to thank my friends for all of their support, assistance, and distractions throughout the years I've been working towards this goal: my roommates and cohorts in crime Ed Connell, James Pollard, Dennis Brylow, Chuck Pisula, Shann Rufer, and Craig Skoch; my [libation sharing] colleagues Krista Bennett, James Early, Florian Buchholz, Brian Carrier, Keith Frikken, Paul Williams, Mahesh Tripunitara, Adam Welc, and Natalia Nogiec; the beer and trivia gang Sarah Bonewits, Rebecca Meisenbach, and Jane Natt and her girls Maddie and Caty; and my faraway friends Elizabeth Reed and Nannette Belt.

I cannot express enough thanks for all of the love and support (and patience!) my family has offered me throughout this entire process. My heartfelt thanks to Gilbert, Joyce, Jereme, Andrea, Seth, Martha, Ava, and Zebulon. You have provided me with so much.

There are so many others that have helped me along the way, to all of you, my thanks.

TABLE OF CONTENTS

Page

Page

Page

Page

# LIST OF TABLES

Table Page

Table Page

# LIST OF FIGURES

Figure Page

# ABBREVIATIONS

| | |
|---|---|
| AIM | Attack, Intrusion, and Misuse |
| BSM | Basic Security Model |
| CCITSE | Common Criteria for Information Technology Security Evaluation |
| CSM | Computer Security Monitoring |
| DoD | Department of Defense |
| IDS | Intrusion Detection System |
| NRT | Near real-time |
| RT | Real-time |
| TCSEC | Trusted Computer Security Evaluation Criteria |
| TOCTTOU | Time of Check to Time of Use |

# ABSTRACT

Kuperman, Benjamin A. Ph.D., Purdue University, August, 2004. A Categorization of Computer Security Monitoring Systems and the Impact on the Design of Audit Sources . Major Professor: Eugene H. Spafford.

Traditionally, computer security monitoring systems are built around the audit systems supplied by operating systems. These OS audit sources were not necessarily designed to meet modern security needs. This dissertation addresses this situation by categorizing monitoring systems based on their goals of detection and the time constraints of operation. This categorization is used to clarify what information is needed to perform detection as well as how the audit system should be structured to supply it in an appropriate manner. A prototype audit source was designed and constructed based on the information from the categorization. This audit system supplies information based on the type of detection to be performed. The new audit source was compared against an existing OS audit source and shown to have less overhead in many instances, generate a smaller volume of data, and generate useful information not currently available.

# 1 INTRODUCTION

It is possible to classify computer security monitoring activities in terms of the goals of detection as well as the timeliness of detection being performed. From this classification, it is possible to significantly improve upon our understanding of what characteristics of audit data and audit data sources are necessary or highly desirable. This document examines this claim in detail and describes work performed to demonstrate its validity.

## 1.1 Background and Problem Statement

### 1.1.1 Background

Since their creation, computing systems have been scrutinized and monitored for a variety of reasons and purposes. The early systems were expensive to purchase and operate, motivating the collection of data to be used for performance and billing measurements. The collected data would be used to detect system problems, allow for the generation of bills for system usage, measure system utilization, and other similar behaviors. As these computer systems became more prevalent, they began to be used for sensitive military and governmental purposes.

In the 1970s, the United States Department of Defense (DoD) became increasingly concerned about security issues associated with the proliferation of computers handling sensitive and classified information, and therefore increased their examination of computer audit as a security mechanism [Bac00]. This effort led to the formation of the DoD Security Initiative of 1977, which produced a series of documents on trusted systems referred to as the "rainbow series" because of their brightly colored covers. As described in the Tan Book from the rainbow series, *A Guide to*

*Understanding Audit in Trusted Systems* [US 88], the data collected was to be used to not only detect undesired behavior, but also to serve as a discouragement in advance of its occurrence. For the purposes of this paper, we refer to such analysis as *Computer Security Monitoring* or CSM for short. By extension, the software that performs such monitoring is referred to as a CSM system. The data stream generated by the computing system intended for analysis is referred to as an *audit trail* and the individual components as *audit records*.

Organizations that handled sensitive or classified data were already subject to security audits in advance of the introduction of computer systems. With the movement of classified data to computer systems, these audits needed to be continued and extended into the electronic realm. Consequently, organizations were required to purchase computers designated as trusted systems based on the government specifications produced by the DoD initiative. As a result, many commercial operating systems were augmented with audit systems based on the specifications and testing methodologies described in the Orange Book, *Trusted Computer Systems Evaluation Criteria* [US 85], which described the necessary characteristics for auditing and audit data generation. These audit systems motivated the development of security monitoring systems in the government, academic, and commercial sectors. Much of this work can be described as seeing what could be determined based on the information provided by the operating system without necessarily augmenting the sources.

Orange Book based systems generate information based on kernel events. This limits the scope of observation for monitoring systems to also be at the kernel level. Additionally, the Orange Book specifications only describe *what* needs to be in the audit trail, not *how* the information needs to be represented. This means that a monitoring system needs to be ported to handle different operating systems' audit trails, including changes between versions of the same operating system. One technique that is used to overcome these limitations is to monitor network traffic. Unfortunately, this requires that the CSM infer what is taking place on the machines

based on the traffic seen, and there are techniques that can be used to evade such systems [PN98].

### 1.1.2 Problem Statement

There has been an ongoing pursuit of improved CSM systems. Past studies have indicated that the existing audit systems do not supply all of the data desired by CSM systems [Pri97]. Recently, there have been efforts to improve the available data either by embedding the detection system into the monitored system itself [Zam01, KSZ02] or by adding a new audit data source tailored to the detection system [CK01]. Recently, there have been attempts to apply data mining techniques to perform security monitoring [LS98], but these types of systems often discover non-sensical associations [McH00]. Others assume that the high false alarm rate will continue and they attempt to apply similar techniques to deal with the large volume of alerts generated by existing systems [JD02].

While it is possible to develop improvements that impact the general field of security monitoring, more specific enhancements and observations can be obtained by considering systems within a narrower operational view.

This paper proposes a categorization of computer security monitoring systems based on their detection capabilities to better identify and understand the characteristics and requirements of each type of system. This categorization is intended to augment the current practice of categorizing CSM systems based upon detection technique.

This categorization can then be used to design and build audit sources that are supplying information tailored to the specific detection goals of a computer security monitoring system. This would reverse the current process of building general purpose audit sources and then trying to perform security monitoring using that information. Instead, the audit source can be constructed based on the type of information that is going to be needed to perform the various types of detection.

## 1.2 Thesis Statement

This dissertation describes the work performed to show the validity of the following hypothesis:

> *It is possible to classify computer security monitoring activities in terms of the goals of detection as well as the timeliness of detection being performed. From this classification, it is possible to significantly improve upon our understanding of what characteristics of audit data and audit data sources are necessary or highly desirable.*

For our purposes, *significant improvement* is going to be shown by considering the amount of audit information generated, the overhead introduced by the audit system, and the inclusion of useful information not already part of the audit trail.

## 1.3 Document Organization

This dissertation is organized as follows: Chapter 1 presents the background and the thesis statement. Chapter 2 presents past work done to classify CSM systems. Chapter 3 describes our improvements to existing categorizations. Chapter 4 describes how the categorization can influence the design and construction of audit sources. Chapter 5 describes our implementation of a set of audit sources based on these ideas and compares their performance against an existing general purpose audit system. Finally, Chapter 6 presents the conclusions, summarizes the contributions of this work, and discusses directions for future research.

# 2 BACKGROUND AND RELATED WORK

Initially, CSM systems were described by the techniques used to decide whether or not to raise an alarm. As the number of systems grew, various classifications and taxonomies were produced to group these systems. Most of these focused on the structure of the system and their basic behavior. This chapter examines the various types of decision making techniques and the past classifications of CSM systems. additionally, it discusses the past work in specifying the audit systems that are present in operating systems.

## 2.1 Decision Making Technique

Traditionally, CSM systems have been partitioned into two groups based on the technique employed to determine if an alert should be generated from the audit data. These two categories are *anomaly detection* and *misuse detection*. There is a less frequently articulated third category called *target monitoring* that can be considered a special subcategory of both anomaly and misuse detection.

### 2.1.1 Anomaly Detection

#### Anderson

James P. Anderson was one of the individuals tapped by the US government to examine the issues of computer security monitoring. In 1980, he published the final report of his study to improve the computer security auditing and surveillance capability of computer systems for a classified client [And80]. His report noted that the existing security audit trails were in need of improvement with respect to both utility as well as data content. He considered as threats any deliberate, unauthorized

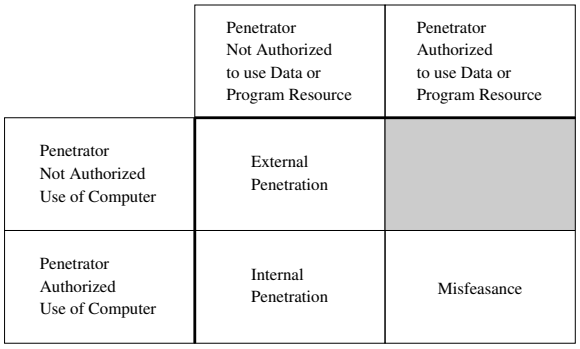


Figure 2.1. Anderson's general cases of threats against a computer system. He distinguished the threats based on the authorization of using the computer facility as well as the authorization to use whatever data or programs that are of concern. [And80]

attempt to access information, manipulate information, or render a system unreliable or unusable.

He proposed a grouping of threats based on the attackers' authorization to use the computer system as well as their authorization to use a set of data or program resources (see Figure 2.1). He defined an *external penetration* as either a outsider or unauthorized employee accessing a computer system. His *internal penetration* was broken down into three subcategories: *masqueraders* who have proper credentials of some other user who is authorized to access the data or program resources, *legitimate users* who access data or programs that they are not normally authorized to access as part of their job, and *clandestine users* who are able to avoid or disable the audit system. The final category of threat is that of *misfeasance* where users abuse their legitimately granted access.

Anderson concluded his report with a development plan including a functional description and task breakdown of both a surveillance subsystem and a system trace component. Included are existing and desired data characteristics for the audit trail and some proposed techniques of analysis. As part of this section of the report, Anderson discussed the possibility of using statistical anomaly techniques to perform

detection based on his hypothesis that masqueraders can be detected by abnormal time of use, frequency of use, volume of data referenced, or from patterns of reference.

Denning

In 1987, Dorothy Denning published a model of a detection system that could be used to perform the calculations suggested by Anderson's report [Den87]. The model was designed to be independent of any particular computer system, application environment, set of vulnerabilities, or types of intrusions. The implementation discussed and examples presented were based on a prototype of the Intrusion Detection Expert System (IDES) [DN85] developed at SRI International.

The basis for this model was the hypothesis that "exploitation of a system's vulnerabilities involves abnormal use of the system; therefore, security violations could be detected from abnormal patterns of system usage" [Den87]. To this end, she introduced the following five statistical models that she claimed can be used to determine if an anomaly has occurred:

1. *Operational Model* – The measurement of some metric $x$ in a new audit record can be compared against a fixed limit, presumably based upon past encounters of these types of records. For example, $x$ might refer to the number of failed login attempts within the past 10 seconds.

2. *Mean and Standard Deviation Model* – The mean and standard deviation for the past $n$ measurements of metric $x$ are stored $(x_1, x_2, \ldots, x_n)$, and a new observation $x_{n+1}$ is considered abnormal if it falls outside of $d$ standard deviations from the mean.

3. *Multivariate Model* – This is similar to the mean and standard deviation model except that it is based on the correlations among two or more metrics. For example, one could compare the number of CPU cycles used versus the length

of a login session. In theory, this could provide greater discrimination power over a single variable analysis.

4. *Markov Process Model* – This model applies only to event counters. In this scheme, a matrix of state transition frequencies is maintained (rather than simply the frequencies of state occurrences). A new observation is considered to be abnormal if the frequency of the transition observed is less than a particular threshold. This allows for the detection of unusual command and/or event sequences rather than single event occurrences.

5. *Time Series Model* – This model uses an interval timer together with an event counter or resource meter, and takes into account the order and inter-arrival times of $n$ observations as well as their values. A new observation is considered to be abnormal if its probability of occurring at that time is too low.

These models of computation have served as inspiration for many of the subsequent security monitoring systems including those which were not direct descendants of the IDES work. This paper still serves as a foundational reference and inspiration for many of the anomaly detection systems being developed today.

There are a number of potential threats to statistical anomaly techniques that have been identified over time (e.g., concept drift, the deliberate mis-training of the detector through slow changes in behavior [LB99]), however these risks can frequently be mitigated by other techniques.

### 2.1.2 Misuse Detection

Based on experience building and operating anomaly detection systems, some types of actions and activities were considered to always be worthy of notice. These are usually classified under the heading *misuse*. A *misuse detection* based system is generally based on a set of fixed rules used to determine if a particular audit

event should trigger an alarm. There are three sub-categories of misuse detection frequently encountered in the literature:

1. *Expert System* – A set of fixed rules that when violated will generate an alert. This system supposedly encodes the decision making process of a human expert in the subject domain into the decision making component of the computerized system. The MIDAS system was one of the earliest intrusion detection systems based on an expert system analysis core [SSHW88]. As these systems are based on modeling a human expert's behavior, they are limited by whatever information that a human expert is aware of, including any misconceptions.

2. *Signature Matching* – A process of looking for pre-specified, exact pattern matches to be contained within the data under analysis. This could be viewed as a subset of an expert system; however, signature matching systems are usually considered separately as they deal strictly with the pattern matching process and not any other decision making processes. Most virus scanning tools utilize a set of signatures as part of their detection [HB00]. Similarly limited by expert knowledge and awareness, it has been noted that not all misuse can be represented by a simple pattern [Kum95].

3. *Policy Violation* – This system involves the encoding of a set of administrative policies into a machine interpretable form. Actions that are being monitored are compared against this set of policy rules to detect violations or deviations. Many systems that are designed to handle classified information utilize this type of system to determine when an operator is exceeding his or her authority. Examples of such formalized systems of policy specification include the Bell-LaPadula model [BL73, BL74] and the Clark-Wilson model [CW87, CW88]. Some policies are difficult to model because of unstated assumptions that may be held by those creating the policy but are not represented in the formal model.

Frequently, discussions of misuse detection refer to one or more of the above as being considered to be the definition of misuse, and the terms are used interchangeably (e.g., a description of a system employing signature matching would use "signature" and "misuse" as if the terms were identical). While one could arguably consider all of the above to encompass the same set of analysis actions, the differing nuances of the analysis performed are often useful in understanding the intended actions of a particular system. For example, an expert system will likely be based on steps an expert uses to analyze a given piece of data, while a policy violation system will attempt to convert an audit record into an action that can be compared against the policy rules. This is an important conceptual distinction even though both systems may generate alerts on similar input data.

### 2.1.3 Target Based Monitoring

There are certain events that can always be labeled as an instance of misuse. In some instances, we can enumerate certain actions that should never be performed by any user of a specified system. In other cases, we can identify objects on the system that should never be accessed by any actions whatsoever. Taken together, these objects or actions are the *targets* of our monitoring.

Unlike anomaly or misuse based monitoring, *target based monitoring* is usually constructed from a set of binary (triggered/untriggered) sensors. These are akin to system assertions that particular actions should never take place under normal circumstances. Examples of target based monitoring include the following:

- A machine connected to the network that is not supposed to be accessed by either insiders or outsiders. This is also known as a "honeypot."

- System calls that are present in the operating system or system libraries, but are not invoked by any of the system programs or require prior registration to invoke.

- Files that are being monitored for access, modification, or execution. These are also known as "tripwires" [KS94].

- Executable programs that are present, and possibly with a tantalizing name, but are not supposed to be run (e.g., Cliff Stoll's tripwire programs [Sto91]).

- Replacing some of the standard binaries with alarm triggers, and giving the original programs new, non-standard names to detect when an outsider is attempting to use a system.

- Directories that are not supposed to be entered.

- Embedding randomized markers (canaries) at the boundary of data buffers to monitor for overruns [CPM+98, CBD+99].

- Monitoring data passed into buffers for executable code [KS99a, BTS99, TS01].

The described activities are certainly anomalous when compared to normal use. Most of them are objects that are not supposed to be accessed based on the expressed policy, and therefore are also misuses of the system when they occur. The important distinction is that there is never a policy instance where those accesses should go unnoticed. This means that target based monitoring can be used in instances where there is no formal policy, nor a known pattern of abuse. Depending on the implementation, there are sometimes issues with the overhead created by the frequency of the checks of the tripwires.

## 2.2 Classifications Based on Structure

As the number of CSM systems grew, efforts were made to survey them and to construct taxonomies to classify the various systems. What follows are various groupings of security monitoring systems that have been proposed and used.

### 2.2.1 AINT Misbehaving – 1995

A general division of "anti-intrusion techniques" is presented in [HB95] where these techniques are divided into "six high-level, mutually supportive techniques" specifically

- Prevention – techniques that prevent or severely reduce an attacker's chance of success.
- Preemption – an offensive strike against a possible attacker in advance of an attack.
- Deflection – redirect an attack from the item being protected.
- Deterrent – increase either the effort needed to succeed, the risk of making an attack, or reduce the reward for succeeding.
- Detection – determine when and how an attack is taking place and notify the authorities.
- Autonomous countermeasures – actively take action against an ongoing attack.

These techniques are intended to cover the entire range of activities including those that are "independent of the availability of a rich audit trail." Each of these six techniques will reduce the intrusion attempts being made on the computer system and are considered to be used in a layered fashion as shown in Figure 2.2. Each layer of anti-intrusion techniques will eliminate a set of intrusion attempts that the previous layer allowed through. Preemption takes place outside of the computer system; detection, deflection, and autonomous countermeasures take place within the computer system itself; and prevention and deterrence techniques have components that operate both within the computer system as well as outside of the system. Only attacks that pass through all layers of defense will be able to access the system resources.

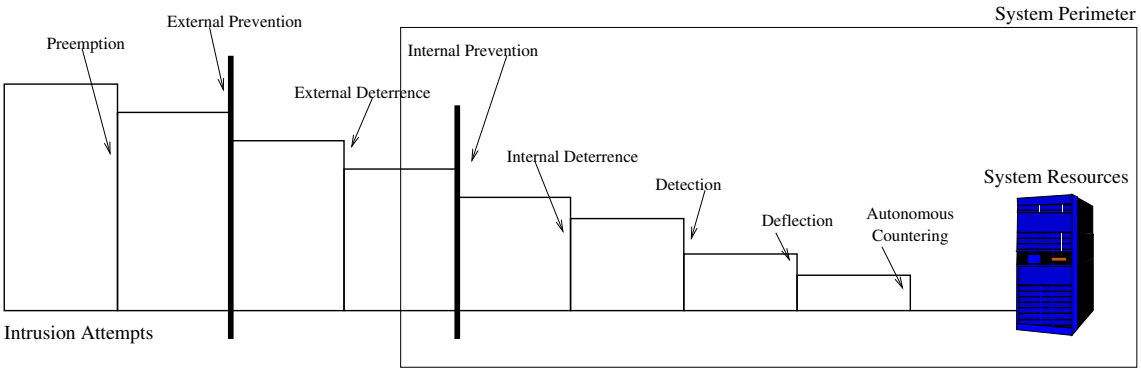


Figure 2.2. Halme and Bauer's layers of anti-intrusion techniques. [HB95]

This paper is one of the few categorizations that classifies items based on the intended goal of the countermeasure. However, many of these techniques take place outside of the computer system being monitored and are not intended to model only electronic behavior. For example, one type of external deterrence is the increase of penalty for being convicted of unlawfully intruding onto a computer system, another is the inculcation of people into a different moral framework that is less accepting of intrusion attempts. Our categorization is looking at computer security monitoring systems that are performing detection (or other collection of audit data), which corresponds to their description of detection techniques. However, some systems might also contain prevention, deflection, and countermeasure components as well.

The authors also discuss how researchers have divided into two "camps" based on the type of calculation being made in the anti-intrusion techniques, which is similar to the divisions described earlier in Section 2.1. One group uses attack signatures (corresponding to Misuse Detection) and the other is based on the analysis of anomalous behaviors (corresponding to Anomaly Detection). They also note that "intrusion detection products are as yet esoteric and not well designed to work together with complimentary approaches such as intrusion preventing firewalls." The

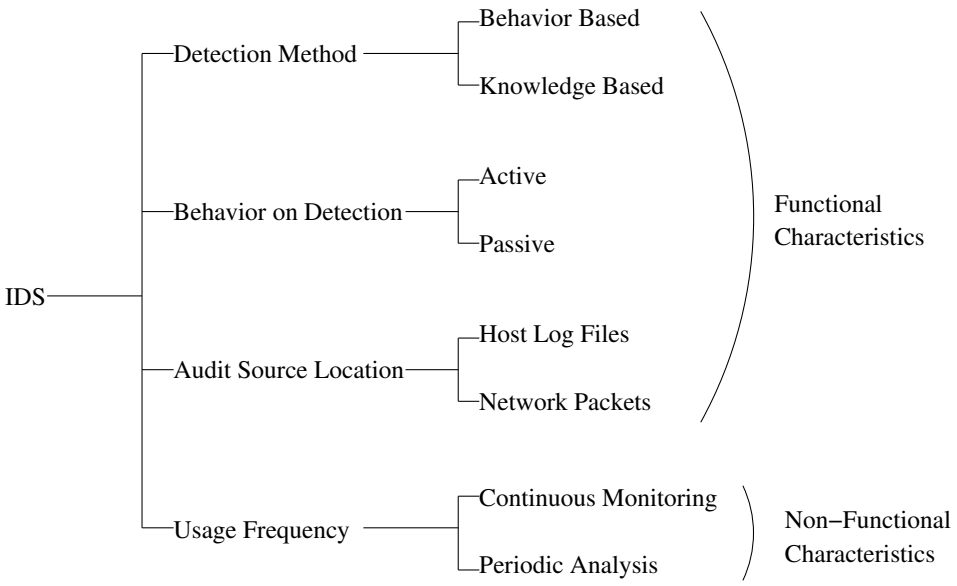


Figure 2.3. The taxonomy of intrusion-detection systems presented in [DDW99].

categorization we present in Chapter 3 can help identify which systems compliment another system.

### 2.2.2 Debar, Dacier, Wespi – 1999

In [DDW99], Debar, Dacier, & Wespi introduce a taxonomy of intrusion detection systems. By their definition "an intrusion-detection system dynamically monitors the actions taken in a given environment, and decides whether these actions are symptomatic of an attack or constitutes a legitimate use of the environment." They divide the characteristics into functional and non-functional categories. The functional characteristics are the method of detection, the behavior on detection, and the source of audit data. The only non-functional characteristic they include is the usage frequency which they note "is an orthogonal concept" to their functional characteristics. Figure 2.3 shows how they structured their taxonomy.

The first division they make is based on a characteristic of the detector – how does the detector make decisions. This is divided into two categories. The first is based on performing detection based on the appearance of the audit data. The system is programmed with knowledge of what attacks look like, and the system searches the audit data for instances of attacks. The authors note that this is what has traditionally been called misuse detection. The second category includes systems that make decisions based on analyzing the behavior taking place on the system. These systems know what normal behavior on the system should look like and generate an alarm if behavior is seen deviating from this norm. This they refer to as being "detection by behavior" or what has traditionally been labeled as anomaly detection.

The second criteria on which they make their classification is the behavior of the system after it makes some sort of detection (regardless of how that detection was made). They describe two types of responses, the first is to simply generate an alert or message but do nothing else. This they refer to as being a "passive" system. The second category is "active" systems that perform some sort of response other than the generation of an alert. This might included terminating a connection or reconfiguring the network topology.

The third criteria they use is based on what sources the detection system uses to collect audit data. They consider two possibilities. The first is that the system uses some type of host log files. The second possibility is that the system collects and analyzes network packets. Although this describes the data source, this characteristic does describe part of the structure of the detection system.

The final criteria used is based on how often the system is run. They consider two cases; either the system is always running and monitoring the audit data ("continuously monitoring"), or the system must be invoked periodically ("periodic analysis") and it analyzes a "static snapshot" of the system state. They consider this to be an orthogonal issue because it has more to do with how the system is [designed to be] run rather than how it operates.

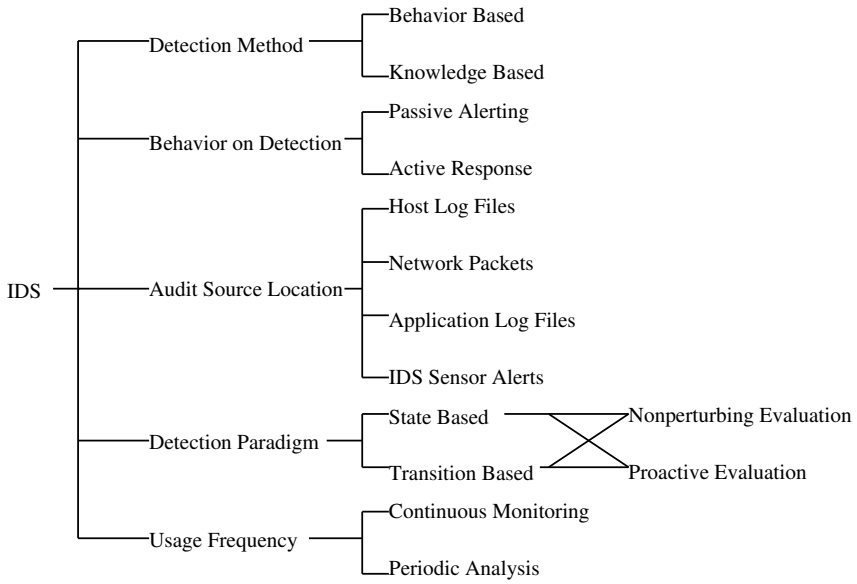


Figure 2.4. The revised taxonomy of intrusion-detection systems presented in [DDW00].

### 2.2.3 Debar, Dacier, Wespi (Revised) – 2000

In [DDW00], Debar, Dacier, & Wespi present a revised taxonomy which is shown in Figure 2.4. There are a number of modifications from the earlier version. First, they change their definition to read "an intrusion-detection system acquires information about its environment to perform a diagnosis on its security status ... the goal is to discover breaches of security, attempted breaches, or open vulnerabilities that could lead to potential breaches." The most notable differences include that they no longer require the system to be observing *actions* within an environment, but the *environment* itself instead; and that the systems are not making only a binary decision about normal usage, but actively looking for attacks, attempted attacks, and potential vulnerabilities that might be susceptible to an attack.

They elaborate that detection methods utilize three informational components:

1. Long term information related to the technique being used to detect intrusions;

2. Configuration information about the current state of the system; and

3. Audit information describing the events that are happening.

They have augmented the audit source location category to include systems that collect audit information from application log files and IDS sensor alerts. Although these sources would have likely been contained under their previous divisions, they chose to make them explicit.

The only new category in the taxonomy is that of "detection paradigm" which describes, in part, how the detection system interacts with it's environment. They describe a "state based" detection system as one that is performing some sort of security assessment to determine which state (secure or insecure) the system is currently in. The other category, "transition based," looks for known events that are associated with transitions from a secure state to a non-secure state. These assessments are further subdivided based on the type of evaluation that is being performed. Either it performs a "non-perturbing evaluation" (e.g., reading configuration files) or it performs a "proactive evaluation" (e.g., actively looks for vulnerabilities such as the scanner Nessus [Der03]).

### 2.2.4 Axelsson – 1999,2000

Stefan Axelsson performed a survey of intrusion detection systems [Axe98] that was later used to develop a taxonomy of the same [Axe00]. He described the taxonomy as being based first on "detection principle" and then by "certain operational aspects." He noted that often he was forced to attempt to reverse engineer the detection system to ascertain what the underlying principle was. He stated "often it is the mechanism that is used to implement the detector rather than the detection principle itself" that he used to classify the system. Figure 2.5 shows his classification of the detection principles used.

There are two major categories of detection principle used. The first is anomaly detection. He breaks this into two subcategories based on how the rules are built.

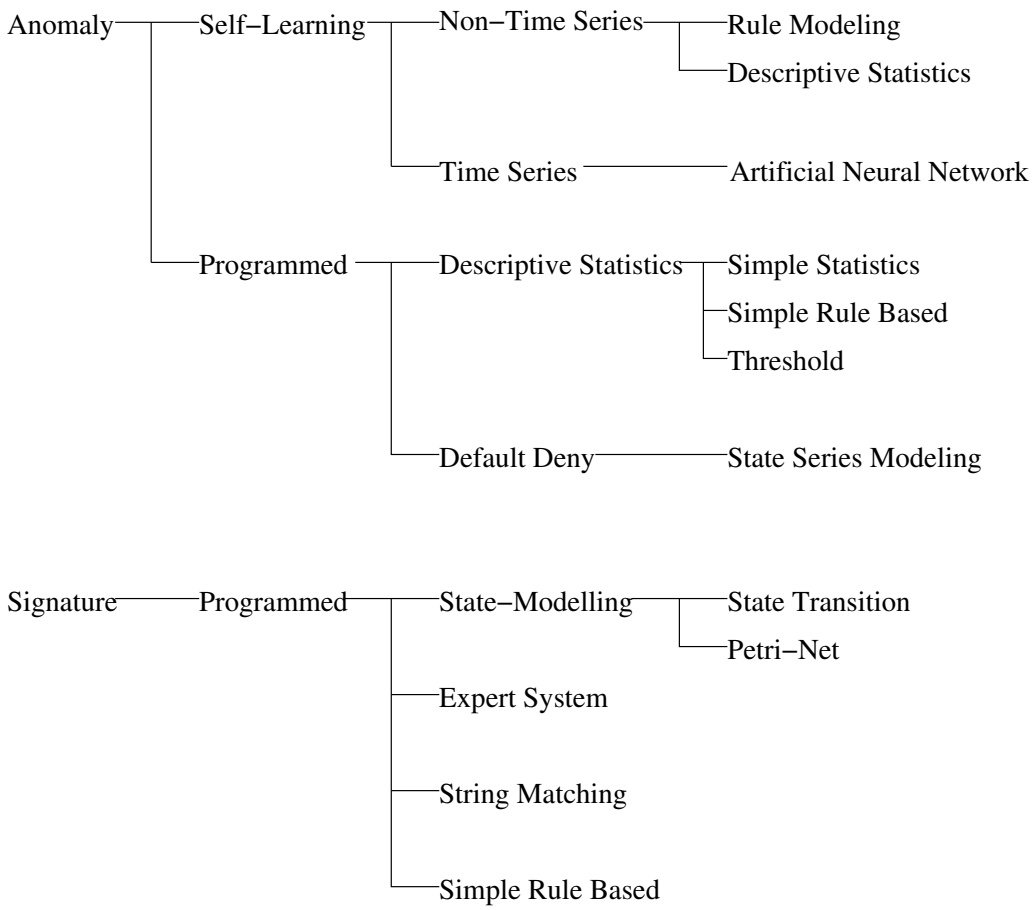


Figure 2.5. The categories of detection principles described by Axelsson in [Axe00].

The first case is termed "self-learning" where the detection system generates it own set of rules. This can be done using either time series information to construct an artificial neural network, or through non-time series data that can be used for a system that is based on either rule modeling or descriptive statistics. In the programmed anomaly detection case, there are two categories. The first is detectors based on descriptive statistics (using either simple statistics, rule based, or threshold techniques). The second is based on using state series modeling to implement a "default deny" policy (anything that is not in the set of known state transitions is considered to be an attack).

The second detection principle is based on using programmed signatures. There are four subcategories identified, state-modeling (based on either state-transitions or petri-nets for detection), expert systems, string matching, or simple rule based systems. This is extremely similar to the subcategories of misuse detection systems that was described in Section 2.1.2.

The final category of detection principle identified is called "signature inspired" and is described as being a hybrid system. This system is self-learning and performs automatic feature selection when presented with labeled instances of both normal and intrusive behavior.

This taxonomy is based on a pair of orthogonal concepts: anomaly or signature based detection, which can be either self-learning or programmed. However, in his survey, he ran across no detectors in the class of signature based, self-learning systems. (Immunological techniques such as [FHS97] appear to be in this category, but were not included in the survey.)

The paper includes a second taxonomy based on system characteristics. Figure 2.6 shows the set of system characteristics used in [Axe00]. Of interest to this dissertation is the indication that the time of detection should be divided into only two categories. The labels used are "real-time" and "non-real-time" but these labels are used to distinguish between systems that can run at the same time as the data is generated or not. This taxonomy does mention the notion that the data might be

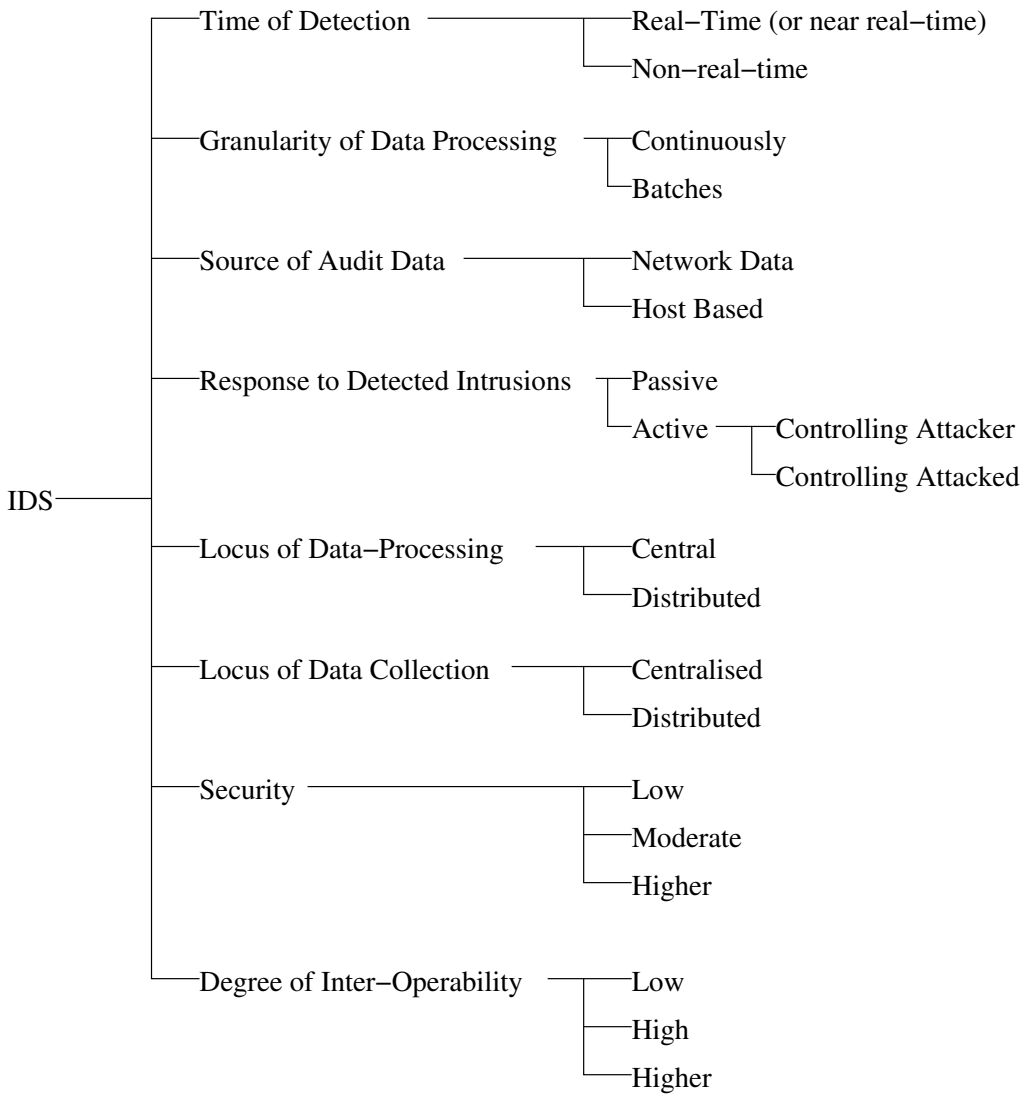


Figure 2.6. The categories of system characteristics described by Axelsson in [Axe00].

sent either continuously or in batches, but indicates that this is linked, but not overlapping the divisions based on the time of detection. The taxonomy uses structural characteristics such as the source of audit data, the locus of data-processing, or the locus of data collection to classify systems.

### 2.2.5 Kvarnström – 1999

In 1999, Håkan Kvarnström performed a survey of 17 commercial tools for intrusion detection and provided a comparison of these systems to illustrate the state of current systems [Kva99]. Figure 2.7 shows the various criteria he used when comparing systems.

These various criteria are collected into groups depending on the type of characteristic described including the following: functional, security, architectural, operational, and management as shown in Figure 2.8.

Again we see that systems are viewed as being either rule based (misuse detection) or anomaly based (anomaly detection), get their data from either host or network sources, and either are passive or active in response to an alarm. This taxonomy does include a more detailed description of the various components in the field "degree of interoperability" (although in practice the paper only rates things as "Low," "Medium," or "High").

### 2.2.6 CMU Survey – 2000

Allen, Christie, et al. [ACF+00] is a survey of a the field of intrusion[1] detection (CSM) including research findings, commercial and research tools, and market entry. They present an overview of the current state of intrusion detection in terms of both the technology that is known and the software that is available on the market. They note where there are significant gaps and make suggestions as to where the

[1]They note that there is substantial disagreement over the meanings of the terms "intrusion" and "attack" especially in regards to detection.

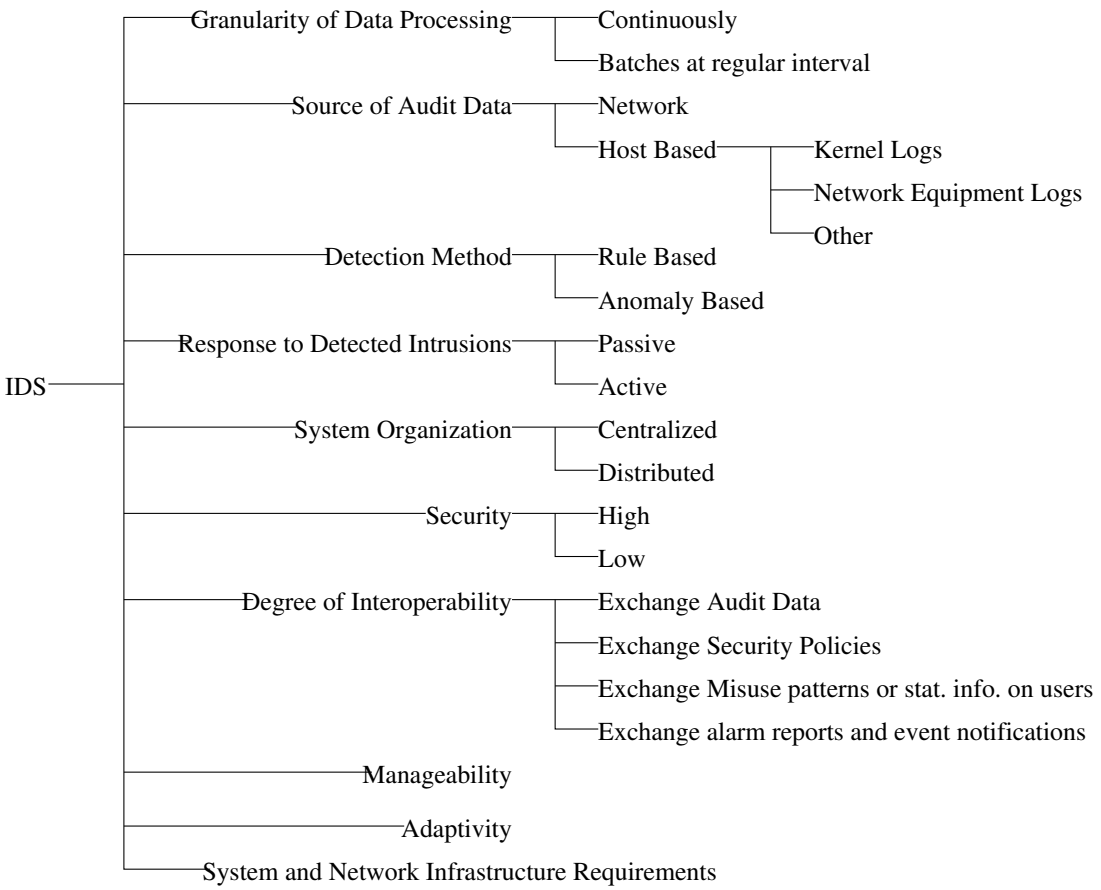


Figure 2.7. The comparison criteria for detection tools used by Kvarnström [Kva99].

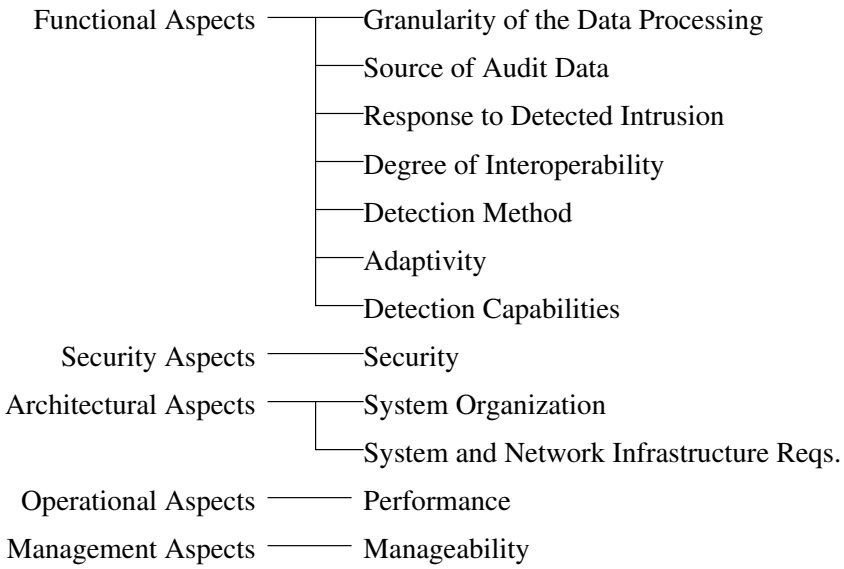


Figure 2.8. The grouping of comparison criteria used in Kvarnström [Kva99].

field might next proceed. They include a discussion of the problems that modern organizations experience as related to the deployment of intrusion detection software. They conclude with a set of recommended next steps for research sponsors, users, vendors, and researchers.

Figure 2.9 contains the comparison criteria used. Again, we see that the organization is based on the method for analyzing data, the type of response to an alert, and the structural characteristics. This taxonomy does not classify the systems based on a timeliness characteristic.

One of the observations presented is that existing security monitoring systems are unable to detect many types of intrusions (or attacks) in their early stages – something that would be vital in building a system designed to minimize the damage caused. Another set of observations presented is that current intrusion detection (CSM) systems provide little or no support for damage recovery, or for forensics, but in the authors opinions, they could provide such support. A third point raised is

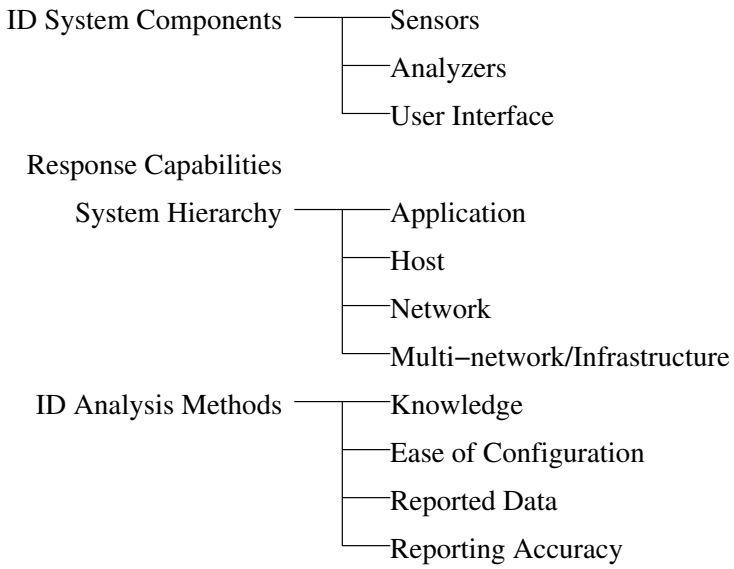


Figure 2.9. The comparison criteria for detection tools used in the CMU report [ACF+00].

the "bandwagon effect" wherein corporations look at the offerings from their peers and competitors and look to provide something similar or slightly better (i.e., no motivation to try to create something different from the existing offerings).

They also discuss that tools to manage audit data are lacking. They note that "storing raw real-time audit data for retrospective ID analysis may require impractically large storage capacities." This seems to support the notion that the data needed for retrospective and real-time detection posses different characteristics.

### 2.2.7 MAFTIA – 2001

Another taxonomy of detection systems was published by IBM in 2001 [ACD+01] as part of the MAFTIA project. This taxonomy is intended to describe detection systems by characteristics (but not necessarily the underlying algorithms used) yet also "aims at describing the capabilities that result from the algorithms used" be-

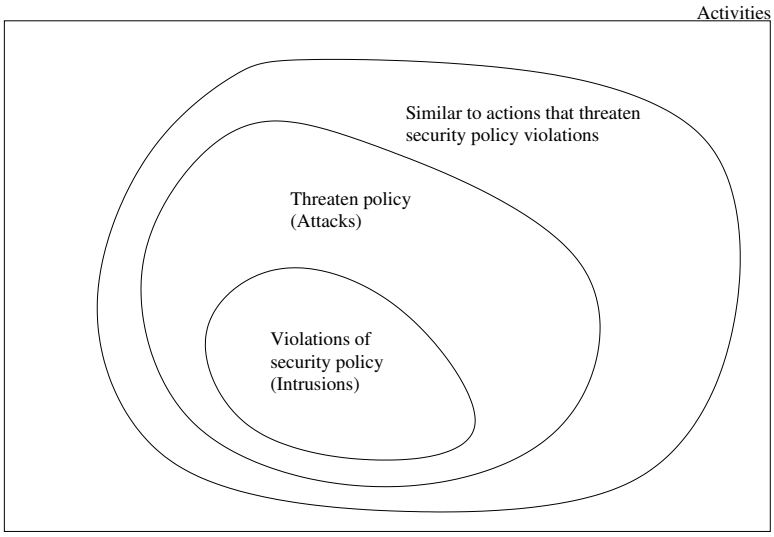


Figure 2.10. The view of system activities taken by MAFTIA [ACD+01].

cause they "describe the criteria an IDS has to meet in order to detect a given attack, or more generally, to process activities observed." Figure 2.10 shows the way in which MAFTIA views activities on a system. Part of MAFTIA is a taxonomy of attacks classified by the activity initiator, the interface objects, and the affected objects which are connected by some sort of dynamic activity that is classified by the invocation method, the type of communication, and other attributes. They use *activity scopes* for creating general descriptions which can become more specific. For instance, the affected object might be generally described as "host," more narrowly "environment," and specifically as the "system registry." Or the communication might be described generally as "networking," which can be more specified as "application" then the protocol "HTTP" and finally the version "HTTP 1.1". The taxonomy is focused on these categories of objects and protocols because it uses such information to determine what attacks an IDS is capable of detecting based on the type of information it is collecting. Therefore the first set of classification rules they apply are based on the structure of the detection engine and the sensors

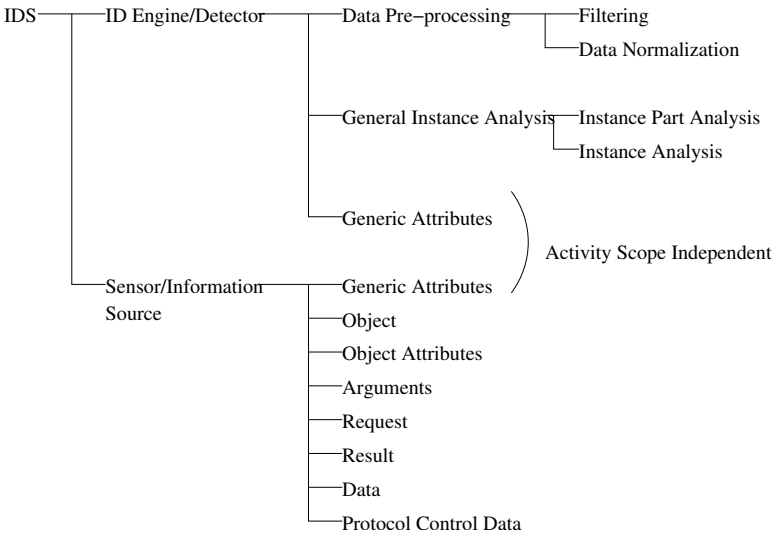


Figure 2.11. The classification of systems based on structure as performed in MAFTIA [ACD+01].

as shown in figure 2.11. The "activity scope independent attributes" for detection engines and sensors are shown in figures 2.12 and 2.13, respectively.

MAFTIA is one of the few taxonomies to try to describe time delays in more concrete terms than either online or offline. However, they use arbitrary divisions for delay that are rooted in current processing speeds and expectations, an assumption that will likely need to be revised as computer systems continue to increase in speed and processing power.

### 2.2.8 Zamboni – 2001

As part of his doctoral dissertation, Zamboni [Zam01] presents a classification of existing computer security monitoring systems based on the structure of data generation and analysis (distributed or centralized) as well as a classification based on the data analysis structure and structure of the data collection mechanisms (direct or indirect subdivided by host or network based). Although the classification sys-

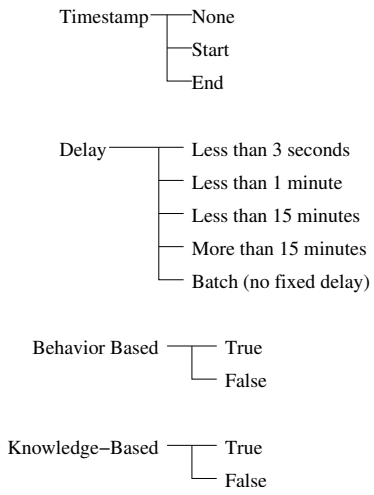


Figure 2.12. Activity scope independent attributes of detection engines in MAFTIA [ACD+01].

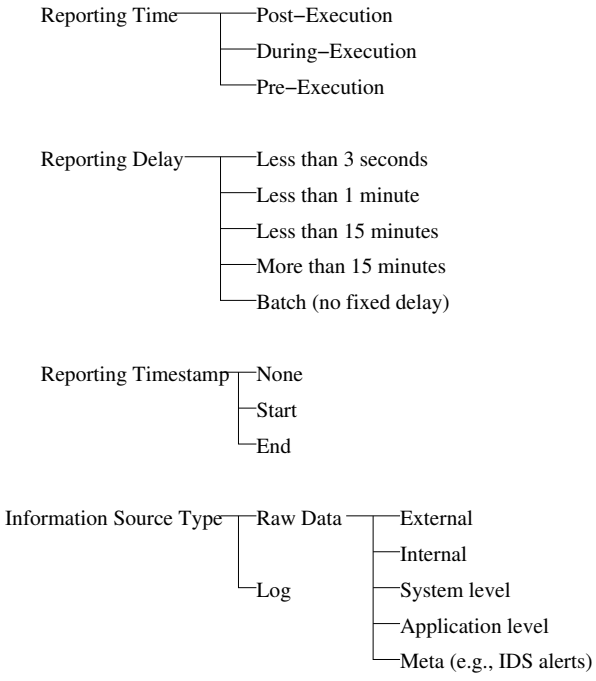


Figure 2.13. Activity scope independent attributes of data sensors in MAFTIA [ACD+01].

tem used makes no reference to time, the dissertation presents an architecture that appears to support *real-time detection* in terms of our proposed terminology.

### 2.2.9 SecurityFocus.Com – 2003

For a modern understanding of how general practitioners partition the space of detection systems, we look at a two part article on terminology used in Intrusion Detection published by the security portal SecurityFocus. In [Cuf03a, Cuf03b], the author lists the following types of detection systems:

- **Application IDS** which use signatures for specific applications rather than being concerned with operating system level messages;
- **Consoles** which are used for reporting and management from detection systems as well as other data sources;
- **File Integrity Checkers** that compare file permissions and hash values against an existing database of information;
- **Honeypots** which present an attractive target and can be used to collect information on what attackers are doing;
- **Host-based IDS** that monitor system event logs for suspicious activities;
- **Hybrid IDS** that use both network node IDS and host IDS in a single software package;
- **Network IDS** that monitor all traffic on the network segment where deployed;
- **Network Node IDS** that collect data from the end hosts of a network connection (rather than the network backbone itself);
- **Personal Firewall** which they consider to be another term for a HIPS system;
- **Target-Based IDS** which they describe as being ambiguously defined, but utilize examples that are similar to what is described in section 2.1.3;

- **Network Intrusion Prevention System (NIPS)/Inline IDS** which are firewalls that use IDS signatures to allow or deny network traffic;
- **Host Intrusion Prevention System (HIPS)** which is software that decides if a particular host should accept or reject each inbound connection; and
- **Attack/DDoS Mitigation Tool** that is designed to block DDoS attacks as far upstream as possible to maintain the availability of bandwidth.

For a majority of systems, they are still being classified by the structure of the system, specifically where they collect data from. There are instances of niche tools that are identified by their function rather than form (file integrity, DDoS mitigation tools, etc.) but they are the exception rather than the rule.

## 2.3 Summary

Past classifications of computer security monitoring systems have been focused on the following 5 things:

1. What type of data classification is taking place, anomaly or misuse?
2. Does the system operate online or offline?
3. Where does the data come from, host or network?
4. Is the structure for the sensor(s) and detector(s) centralized or distributed?
5. Does the system do anything else for response other than generating an alert message?

Other than a few entries in [Cuf03a, Cuf03b], no taxonomy attempts to describe the systems based on the type of activity that is to be detected. Timeliness of operation is usually only discussed as being online or offline with occasional descriptions of the delays involved.

## 2.4 Audit

Part of this dissertation covers the generation of a new audit source. This section reviews the current audit trail specifications and their predecessors.

### 2.4.1 TCSEC

In the 1980s, The United States Department of Defense published a set of computer security standards known as the Trusted Computer Security Evaluation Criteria (TCSEC) as a set of manuals each detailing a particular aspect of the evaluation criteria. Each manual's cover was a distinct color, leading to the practice of referring to a particular volume by its color instead of proper name. (Collectively, the TCSEC manuals are referred to as the "Rainbow Series" for the many colors present.)

The Orange Book [US 85] is the best known volume from the TCSEC documentation. This manual covers the various requirements that a system must meet to be able to be certified at a particular level of trust. These levels are referenced by a letter-number combination ranging from the lowest category, D – Minimal Protection (for systems that fail evaluation), to A1 (Verified Design) with the upper limit left open for future expansion. The Orange Book details the criteria a system must have to be certified at each level as well as the manner in which evaluation must take place to determine if a system meets the criteria.

Any system that is intended to pass this criteria must contain an audit system. The Tan Book (*A Guide to Understanding Audit in Trusted Systems*) describes in detail the requirements and rationale for such auditing facilities. It offers the definition that "The audit process of a secure system is the process of recording, examining, and reviewing any or all security-relevant activities on the system" and indicates that the text is intended to help guide manufacturers on the design and incorporation of audit mechanisms into their system as well as suggestion how to effectively use such audit capabilities. Additionally, it describes what should be

recorded, how the audit should be conducted, and what protective measures should be applied to the audit resources.

The general goal of audit within the context of these military documents is to "assure individual accountability whenever mandatory or discretionary security policy is invoked" specifically in "systems that are used to handle or process classified or other sensitive information" [US 88]. The Tan Book also contains a list of explicit goals for security audit mechanisms:

1. To allow the review of patterns of access (on a per-object and per-user basis) and the use of protection mechanisms of the system.

2. To allow the discovery of both insider and outsider attempts to bypass protection mechanisms.

3. To allow the discovery of a transition of a user from a lesser to a greater privilege level; for example, when a user moves from clerical to system administrator roles.

4. To serve as a deterrent to users' attempts to bypass system-protection mechanisms.

5. To serve as yet another form of user assurance that attempts to bypass the protection will be recorded and discovered, with sufficient information recorded to allow damage control.

There are explicit descriptions of the requirements for performing the required auditing functions for each of the levels of evaluation enumerated within the TCSEC.

In December 2000, at the 16th Annual Computer Security Applications Conference, the Orange Book (and by implication, the rest of the Rainbow Series) was retired and given a formal wake to mark its passing from the United States national policy. Although no longer considered to be a current document, the TCSEC remains a strong influence in the design and implementation of computer operating systems.

### 2.4.2 Common Criteria

In June 1993, the US, Canadian, and European computer security criteria organizations (TCSEC, CTCPEC, and ITSEC) began work on creating a single unified specification which became known as the Common Criteria for Information Technology Security Evaluation (CCITSE). Version 1.0 was released in January 1996. After extensive external review and trials, a revised Version 2.0 was released in May 1998. This version was adopted as an international standard in 1999 (ISO 15408). A number of minor changes were made as a result of the ISO process leading to a revised Version 2.1 specification which was adopted in August 1999. [Com, CCI99, Nat03]

The CCITSE is not a set of specifications for systems, rather it is a framework in which specifications for trusted computer systems can be drafted and a set of evaluation criteria for the evaluation thereof. In the development and construction of a trusted system, first a Protection Profile (PP) must be written. A Protection Profile provides a set of security requirements that a consumer (possibly a customer or software developer) to use to create a set of standardized security requirements to meet their needs. They may need to select more than one PP to meet all of their desired criteria.

Here is what the CCITSE has to say about auditing:

> "Security auditing involves recognizing, recording, storing, and analyzing information related to security relevant activities ... The resulting audit records can be examined to determine which security relevant activities took place and whom (which user) is responsible for them."

It further defines a section for the generation of audit data called *FAU_GEN*. This family is described as the following:

> "This family identifies the level of auditing, enumerates the types of events that shall be auditable by the TSF, and identifies the minimum set of audit-related information that should be provided within various audit record types."

The only specification made in the CCITSE itself is that there are two levels of the FAU_GEN family, FAU_GEN.1 and FAU_GEN.2. FAU_GEN.1 which requires the specification of the "level of auditable events, and specifies the list of data that shall be recorded in each record." At level FAU_GEN.2 those events must be associated with individual user entries. Protection Profiles must detail the actual contents based on the above requirements.

In response to the release of the Common Criteria, the National Security Agency (NSA) has produced two protection profiles that are conversions of the TCSEC requirements into the CC language: *Controlled Access Protection Profile* (CAPP) [Age99a] corresponding to C2 ratings and *Labeled Security Protection Profile* (LSPP) [Age99b] corresponding to B1. These two profiles are currently the most frequently used profiles for evaluation of operating systems.

### 2.4.3 BS 7799, ISO 17799

Outside of the United States' military and security community, the most frequently recognized document is British Standard 7799 [bs799a, bs799b] (introduced in 1995, revised in 1999), which describes various aspects of information security management. (This was fast-tracked into an ISO standard 17799 within the same year as the release of the revision.) It begins with a justification of information security and the document covers a wide range of topics including physical security, writing policy, personnel, access control, system development, and audit. The requirements specified are of a general nature as they are intended to cover many areas within the document. In [bs799a], section 9.7 describes the suggested monitoring of system use. It suggests logging of user IDs, timestamps of log-on and log-off, terminal identity and location (if possible), records of successful and rejected system access attempts, and records of successful and rejected data and other resource access attempts. They also suggest monitoring (but not necessarily logging) authorized accesses, privileged operations, unauthorized access attempts, and sys-

tem alerts or failures. This document is similar to that of the Common Criteria, in that it attempts to provide advisory checklists and frameworks without supplying great amounts of detailed requirements.

### 2.4.4 Current Implementations

Most modern commercial operating systems including Solaris, HP-UX, AIX, IRIX, and Windows 2000 have been evaluated against the CAPP or LSPP (or the C2 and B1 TCSEC equivalent) and therefore provide an audit system that meets those requirements. There are a number of attempts to provide a similar audit functionality to the open source Linux operating system [sal03, sna03, scs03, lin03] in varying states of development, but none surveyed are currently incorporated into a mainstream distribution.

# 3 MODEL OF CLASSIFICATION

As described above, CSM systems are currently described by the technique employed in the detection phase; anomaly, misuse, or target, along with other characteristics such as location of data collection and location of data analysis.

These systems (especially research proposals or prototypes) are also self-identified based on the type of decision model employed leading to paper titles that end in descriptions such as "... using Autonomous Agents" or "... based on the X Algorithm." The most probable reason for this is that such characteristics were what set them apart from the rest of the CSM systems, therefore making them worthy of note. Although there is still a great deal of room for further research based on detection technique, it would also be useful to have terminology that allows us to discuss the various systems based on the purposes of the detection being performed, and allowing meaningful comparisons to be made among CSM systems designed to perform similar tasks.

It is likely that the categorizations discussed in Chapter 2 were *structural* in nature because the original projects had a structural focus – the way in which the CSM system was built was the novel portion of the work. This makes sense if you consider that such developments were an exploration of a new area. (Also, a structural approach is a commonly used technique when constructing a taxonomy.) Consider what might happen if we discovered a new ecosystem on another planet. The initial descriptions of the discovered life forms would focus on the structural characteristics of each new creature. How tall are they? How many limbs? What types of colorations? What sort of internal structures do they have? We would probably build a classification system based on these structural characteristics to help us determine if a given specimen was new or not.

After a period of time, we would have a fairly good idea of the overall structural types and variances that exist. Then our focus would be more on how these creatures behaved and how the ecosystem fits together. Is this creature a predator? Are they prey to something else? What sort of behaviors does it have? We would begin classifying things based on the behavioral characteristics such as predator/prey, colony/solo, hunter/gather and so forth. We would not throw out the existing structural categorizations, but we would use the behavioral classification systems to help us augment our overall understanding of the ecosystem. The remainder of this chapter presents our new categorization designed to supplement the existing structural taxonomies of CSM systems.

## 3.1 Categorization vs. Taxonomy

Our model is intentionally a categorization instead of a taxonomy. As pointed out by Krsul [Iva98] and Lough [Lou01] many of the existing "taxonomies" of computer security systems do not meet all the defining criteria. Borrowing Krsul's definitions, a *taxonomy* is the theoretical study of classification, including its bases, principles, procedures, and rules. A *classification* is the separation or ordering of objects (or specimens) into classes. Krsul presents the following four characteristics that must be present in the characteristics of a taxonomy:

1. **Objectivity:** The features must be identified from the object known and not from the subject knowing. The attribute being measured should be clearly observable.

2. **Determinism:** There must be a clear procedure that can be followed to extract the feature.

3. **Repeatability:** Several people independently extracting the same feature for the object must agree on the value observed.

4. **Specificity:** The value for the feature must be unique and unambiguous.

What we present is not based on the *theory of classification* because we have not attempted to build a system that would allow us to satisfy all four of the aforementioned requirements. Specifically, we do not present a "clear procedure" from which a given CSM system could be classified into a category which means that we do not meet the standard of "determinism." Secondly, our categories are not designed to be mutually exclusive, nor are they designed to be all encompassing. This means that it fails the "specificity" requirement. Rather, we present a set of categories (or classes) into which a CSM system can fall. However, a given instance of a system may belong to more than one category simultaneously. While this could be eliminated by factorially expanding the categories to include all possible combinations of elements, and we could obtain total coverage by including the categories "other" and "unknown," such modifications would decrease the utility of the system by adding in unneeded complexity. Rather than contort the system to meet a higher (and unneeded) specification, we have opted for a simpler version for the sake of clarity and utility.

## 3.2 Goal of Detection

The first way in which we wish to examine existing CSM systems is based on their operational characteristics. We want to be able to answer the question *For what security purpose is this system monitoring behavior?* We propose that the following four categories describe the major areas of focus within the computer security monitoring field:

- Detection of Attacks
- Detection of Intrusions
- Detection of Misuse
- Computer Forensics

We will discuss each of these in the following sections. The above list does not completely cover the range of activities under the umbrella of computer monitoring. There are certainly other topics such as system "health" monitoring, fault recovery and tolerance, performance monitoring, system accounting, and so forth; however, the focus of many of these additional areas falls outside of the traditional scope of security and thus of this category.[1] As mentioned previously, it is possible for a system to be a member of more than one of these areas.

By subdividing the field of computer security monitoring into different categories based on the type of detection that is being performed, we gain a more refined set of expectations regarding the behavior of such systems. With a more carefully described grouping of systems, we will be able to improve our comparisons and differentiations between systems by being able to tell if the two are designed to perform similar functions. Presently, it is not uncommon to see comparisons between systems that are designed to detect attacks and those designed to detect misuse. Such comparisons may not make sense as the underlying models of detection may differ widely between the two. Additionally, the audit information desired may differ significantly. This section describes the various goals of detection as well as explaining some of the differences between the categories.

### 3.2.1 Detection of Attacks

One type of analysis focuses on the detection of attempts to exploit a vulnerability in a computer system. This exploitation may be through hostile code or a Trojan horse, programs that deliberately exploit a flaw, intentionally malformed or specially crafted network packets, etc. We call such CSM systems *attack detection systems*. We use the definition of an *attack* in the sense presented by Anderson [And80] – "a specific formulation or execution of a plan to carry out a threat." Thus an attack is defined by the plan from its inception, to its creation and existence, and then on

[1] All of the security related components of such monitoring (e.g., performance monitoring for DoS detection) are contained within our enumerated list.

through its enactment. Therefore, the goal of these systems are the detection of that attack at any stage.

The word *attack* refers to both actions or objects whose goals are to exploit a vulnerability in a computer system. In the action form, an attack refers to the actions that attempt to exploit a vulnerability in a computer system. The vulnerability need not be currently present in the system under attack: It is possible for an attack to be against a vulnerability that was removed from an older version of the system, or against a different platform (e.g., an attack against Solaris being sent to a MacOS X machine). As an object, an attack is a piece of code, data, or some sort of tool (either software or hardware) whose purpose is to attempt to exploit a vulnerability. An attack tool (object) performs an attack (action) against a computer system. We note these two forms of attack detection to allow our definition to include systems that monitor for exploit attempts as well as systems that look for the existence of exploit tools.

If the threats of concern to a system are carefully enumerated and sufficiently expanded, all of the attacks to be detected could be described *a priori* (at least in theory). However, this task is generally considered to be infeasible for a computer system of any size. Therefore, there are some attacks that will only be enumerated *a posteriori*. *A priori* enumeration offers the potential ability to prevent an attack's success either by being able to detect it while it is occurring or by being able to remove a vulnerability from the target system. *A posteriori* detection capabilities allow us to go back through existing records once we have identified a new attack and detect it after it has already occurred.

There is no requirement in our definition for an attack to be successful or unsuccessful. Nor is there a requirement for the existence of the vulnerability for which the attack is intended to exploit to exist either presently or historically on either the system under observation or any other system. The definition of an attack exists outside of any specific computer system (although specifying some of the characteristics of a computer system may be useful when attempting to identify or enumerate

attacks). While there is an implication of intent ("... attempt to exploit ...") this is an unmeasurable (not directly observable) characteristic, and therefore outside of the realm of what can be expected to be collected by any sort of electronic monitoring. However, in many instances such intent can reasonably be inferred.

An *attack* can be a single instance or a sequence of network datagrams, shell commands, library or kernel service requests, file system modifications, machine instructions, hardware modifications, etc. Any sequence or sub-sequence may or may not be chronologically ordered. There have been some recent attempts to coalesce individual attack alerts into an single *attack scenario* [DW01, VS01]. Many of the existing systems described as "Intrusion Detection Systems" are actually designed to detect attacks. This is especially true for network based systems.

Frequently, attack detection can be performed without needing to consider who is causing the attack, or by assigning an attack to a generic external user. For example, a "ping-of-death" attack [CER96] or a "land" attack [CER97] are network oriented attacks. Network traffic does not posses the concept of users, and it is not necessary for these attacks to be bound to a user to be detected. While we could create a generic outside user to whom we ascribe all unclaimed attacks, this introduces an extra, unneeded assumption, which might cause later problems. Any attack that *must* be bound to a particular user is going to be an instance of either misuse or an intrusion as well.

We use the term `Attack Detection System` and the abbreviation `ADS` to refer to a CSM system that has a goal of detecting attacks.

### 3.2.2 Detection of Intrusions

As described in Section 2.1.1, Anderson proposed a categorization of penetrations based on the intruder's authorization to use the computer system as well as the data or program resources being consumed [And80]. We use the term *intrusion detection* to refer to the detection of the first three classes of penetrations presented by

Anderson: external penetrations, internal penetrations (with the possible exception of *legitimate users*), and clandestine users. These correspond to the left column of Anderson's permission matrix (Figure 2.1) where the subject is not authorized to use the data or resource in question. These intruders may be accessing the computer system by one of the following means:

- Gaining the use of services – such as a printer or making calculations
- Accessing data or programs – such as accessing a database or executing a control program
- Modification of data – including deletion and truncation

The critical condition is that regardless of the type of access occurring, such access is not allowed by the administrative policy of the computer system. Therefore, an intrusion takes place only within a particular system because it is dependent upon that system's notion of users, resources (computers and networks), and administrative domains. This is unlike attacks which can exist outside of any particular computer system reference.

Systems that detect anomalous command usage or other behavior-based profiles are performing intrusion detection. The various models enumerated by Denning [DN85] are all based on the detection of this type of threat.

We use the term `Intrusion Detection System` and the abbreviation `IDS` to refer to a CSM system that has a goal of detecting intrusions. It is important to note that many of the existing systems and categorizations described in Chapter 2 used "IDS" as a generic term. Outside of the descriptions in that chapter, the acronym IDS is used as described here.

### 3.2.3 Detection of Misuse

The detection of penetrations described in [And80] but not covered by intrusion detection (namely *misfeasance* but also some instances of internal penetrations by

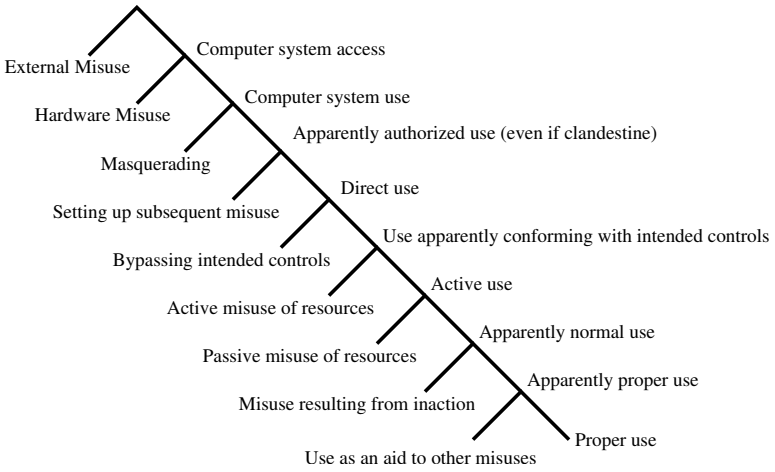


Figure 3.1. Neumann and Parker's classes of computer misuse. The diagram represents what they believed to be the typical steps and modes of computer systems. The leftward branches all refer to misuse, while the rightward branches represent potentially acceptable use.

legitimate users) falls under the category of *misuse detection*. This is different from the usage of misuse detection in Section 2.1.2 that described a processing technique used by CSM systems. Therefore, we will use the phrase "detection of misuse" when describing the goal of such a system.

Neumann and Parker introduced a hierarchy of behaviors that they considered to be computer misuse (see Figure 3.1), which was based on their combined personal experience of handling 3000 cases of system misuse over 29 years [NP89]. They use the following classes to categorize misuse:

- Improper disclosure of data
- Modification of data
- Denial of service
- Trojan horse
- Playback attacks

- Misrepresentation
- Impersonation
- Inferences that permit the derivation of unrevealed data
- Failure to act appropriately

Their stated goal was to improve upon the notion that misuse is simply a violation of the CIA model (Confidentiality, Integrity, and Availability), and were using the term "misuse" in a generic manner. Generally speaking, the top five classes describe some type of unauthorized use (and therefore can be considered an intrusion), while the remaining four describe misuse of legitimately conferred authority.

For our purposes, systems whose goal is the detection of misuse are concerned with examining the behaviors of users who are authorized to use the computer system under observation. However, they might might not have been authorized to use the particular program or data, or perhaps not authorized to do so in a particular manner. For instance, an IRS auditor needs to be able to access any citizen's tax record to do her job. However, she is not supposed to use this access to look up information for personal use (such as searching for celebrities or neighbors). This issue is sometimes referred to as the "insider threat," which was for many years the most prevalent type of reported security incident. Recent news articles such as [Mes03] indicate that there is still much concern about the detection of such misuse, and reports such as [Ric03] indicate that at least 56% of responding organizations were victims of computer misuse in the past year.

We use the term `Misuse Detection System` and the abbreviation `MDS` to refer to a CSM system that has a goal of detecting misuse.

### 3.2.4 Computer Forensics

There are instances where the operators/administrators of a computer system want to reconstruct actions that have already taken place on a computer system.

Among these would be instances of computer intrusion, misuse, or attack. Of course, these are not the only reasons. One might also want to verify the authenticity of a message, keep track of changes made to a system by the administrator, or keep track of all transactions processed by a given terminal. In some cases such logging is required by law (e.g., to comply with SEC regulations), and in most it is viewed as either a deterrent or failsafe against wrongdoing. More specifically, consider the following two examples:

1. A trading system where information is collected, not to detect attacks, intrusions, or misuse, but is kept to be used for higher levels of fraud analysis such as money laundering; or

2. A system designed to trigger an alarm (without attempting to collect any other information) whenever a "ping-of-death" packet is observed on the network, or something that looks for known email worms and viruses in all incoming email.

The first is purely forensic in nature – the information is collected solely for record keeping purposes, while the second is concerned only with detecting a specific attack without concern to who was affected, or where the attack originated from.

This attempt to reconstruct the past activity of a computer system is referred to as *computer forensics*. Computer forensics seeks to answer a number of questions including the following:

- What happened?
- When did the events occur?
- In what order did the events occur?
- What was the cause of these events?
- Who caused these events to occur?
- What enabled these events to take place?

- What was affected? How much was it affected?

It is impractical, if not impossible, to store all of the states of a computer system, so at best we have a set of snapshots of the state of the system at particular points in time [FV99]. One way to view computer forensics is that it is the process of taking these various snapshots and by examining them in sequence, using them to construct a plausible scenario of events. In general, we cannot divine with complete certainty which events took place; however, we can frequently collect an amount of corroborating evidence to either support or refute our hypothesis of what occurred.

In a similar manner, a CSM system using computer forensics might examine the filesystem on a disk, the audit logs generated by the operating system, alerts generated by an attack detection system, and so forth to reconstruct a probable timeline of events along with reinforcement and support for various hypotheses.

Most forensic activities have limitations in either the type of data and events that can be determined or the relative ordering of those events. For instance, we might be able to determine that event $B$ occurred, but we know that $B$ depends on an event $A$ as a precondition. From this we might infer that $A$ probably took place, but this would be without any sort of direct observation. In other instances, we might be able to establish that both $A$ and $B$ occurred, but cannot discern in what order they took place. What is able to be determined is frequently dependent upon what information sources are made available and how trustworthy their information is.

We use the term `Computer Forensics System` and the abbreviation `CFS` to refer to a CSM system that has a goal of supporting computer forensics.

### 3.2.5 Differentiation Based on Scope

As noted previously, a system might have multiple security monitoring goals, that is, it would be properly categorized with more than one of the above labels. Similarly, a given event on a computer system might be of concern to more than one type of CSM system. However, the perspective taken would differ greatly depending

on the goal of the monitoring. The following might describe how each type of system would view an event where user Alice was logged in from node `Wintermute` to run program `DataMine` to access Bob's customer record:

- **Detection of Attack:** Is program `DataMine` an attack tool? Are the actions being taken by user Alice part of an attack?

- **Detection of Intrusion:** Is this really user Alice logging in? Does she normally login from node `Wintermute` at this time? Does she normally run program `DataMine`? To access Bob's record? Is she allowed to access that program or data?

- **Detection of Misuse:** Is Alice supposed to running program `DataMine`? Is she supposed to be accessing Bob's customer record? Has she run other programs that make running `DataMine` a policy violation? Has she accessed other records that make accessing Bob's records a policy violation?

- **Computer Forensics:** What is happening on the system? Where is node `Wintermute`? Who is using the account called "Alice" and where is she located? What program is being called by the name `DataMine`? What part of the database is being accessed by the label "Bob's customer record" and where is it being stored? What changes happen as a result of this action?

Imagine a set of four very focused researchers pouring over an audit trail. Each researcher has a hypothesis that she is intent on either proving or disproving. They each will have a different set of information that they consider necessary, useful, and extraneous. Any event that they view will be colored by their single-minded goal allowing them to apply a specialized perspective on the data that might be different from how others see it. A given CSM system might have more than one of these goals without needing them to be separate entities. A real-world example of this type of perspective is a teacher evaluating a student's book report. You could imagine that there are two separate graders, The first is reading the paper to see if proper

grammar and style were used. The second is examining the content regardless of the mechanics looking to see if the facts are correct and that a valid argument is being presented. The fact that there is only one physical grader does not mean that they cannot have more than one goal.

The next sections examine the differences in perspective that exist between the various goals of detection and the difference in audit data requirements.

### Detection of Intrusions vs. Detection of Misuse

Intrusion detection attempts to detect masqueraders or others that gain unauthorized access to the system. The detection of misuse is concerned with determining if an authenticated user is abusing their authority without regard to whether this is the "actual" user controlling the account or not. Intrusion monitoring might not be concerned with which files are accessed during a given session, but the detection of misuse would be. Intrusion detection is asking "Is this user allowed to access this system?" and "Is this actually the user they claim to be?" while a MDS believes this user to be who they claim, but wants to know "Based on the existing policy, should this user be doing these actions?"

### Detection of Intrusions vs. Detection of Attacks

An ADS might be content to only examine the payload of network packets, but an IDS would be concerned with where the packets come from or how the system is responding to them. Additionally, attack detection need not include the notion of "users" which is a fundamental component of intrusion detection. Although an ADS can operate without the notion of users, one could imagine that there is a generic, unspecified user performing all of the unattributed events. This is functionally equivalent to having such events not attributed to a user. However, an IDS cannot operate without having a notion of users, and in fact, would require a category for an unknown or unauthenticated user.

### Detection of Intrusions vs. Computer Forensics

Although there are many similarities between the types of data being examined, there are some differences between the interpretive focus of these systems. Intrusion detection looks more at actions without concern for their consequences, while computer forensics is also concerned with the results of such actions. An IDS is determining if these actions are being performed by an authorized user – in some cases, even an unordered list of actions would be acceptable. However, a CFS is going to be collecting information that can be used to determine which events preceded others, especially if it will be used as a support to a causality inference.

### Detection of Misuse vs. Detection of Attacks

In most instances, an attack is a type of misuse. However, an attack can come from outside of a computer system and therefore may not be tied to any particular user. Also, the detection of misuse is concerned with non-destructive data access (e.g., file reading), which generally would be of no concern to attack detection. For example, the detection of a `phf` exploit [AUS96] would be handled differently by each type of system. The ADS would be looking for possible attack strings without consideration of their eventual effects. However, a MDS would be concerned with the results of a successful attack (i.e., the misuse of the system by the web server account) and may not consider why the user is misbehaving (i.e., because of an attack).

### Detection of Misuse vs. Computer Forensics

These two groups are the most similar in comparison. Both are attempting to understand what sort of behavior is taking place on the system. Computer forensics is less closely tied to the notion of users, although both are concerned with what action has taken place and are interested in knowing who was performing the actions.

However, a MDS needs the system policy to make decisions, while a CFS will possibly take the system policy as one more piece of data to analyze. In terms of audit data, computer forensics has stricter time and ordering requirements than what can be used for the detection of misuse, and will collect information that is part of legitimate system use to allow it to be analyzed.

### Detection of Attacks vs. Computer Forensics

Computer forensics involves examining a much broader range of issues than attack detection. Computer forensics is concerned with not only what attack took place (although it can function without this information) but also with the after-effects of such attacks as well as normal, non-malicious behavior. Attack detection has a much more narrowly focused scope of activity observation, specifically, deciding if each action or object is an attack. It does not need to determine who was attacking or what effects the attack had (although in many cases that information would be considered useful).

## 3.3 Timeliness of Detection

The second feature that we use to characterize CSM systems is the timeliness of the data processing, measured by the time between data generation to its correlation and the raising of alarms. The rate at which the monitoring occurs is dependent upon several factors including the following:

- Efficiency of audit data generation
- Efficiency of the correlation technique

but should be independent of the following:

- Rate of event occurrence
- Underlying hardware architecture

The goal is to create a timeliness specification that can exist independent of a specific hardware platform or operating system and describes systems able to operate at a particular level of performance given some minimal set of computing power and above. Additionally, it should not have any caveats requiring proper behavior of users or differing behaviors based on the homogeneity of data sources. In other words, the timeliness specification should be applicable to both active and quiescent systems and apply regardless of whether security related events are occurring or not.

In the early publications on CSM, one of the unobtainable (though desired) goals was the ability to perform "real-time" detection. At the time, the overhead of data generation, network latency, and processing speed of the analysis engine limited the performance of the implemented systems. This led to many of them being run on a dedicated host with audit data collected from the systems of concern and delivered at the end of a day or a week. As the speed and power of computing systems improved, the time period between data generation and analysis was reduced to a point where detection can take place as the system is operating. Historically, the literature describes this simultaneous data generation and processing as taking place "in real time." Unlike past discussions, we will be introducing a stricter set of definitions for the timeliness of detection.

It is common for a CSM system to be able to be configured to operate under more than one timeliness category. In this case, such a system has multiple *modes of operation*, and our categorization can be used to describe either the range of modes a system is capable of or the current behavior of such a system.

### 3.3.1 Notation

When discussing the time constraints of CSM systems, we will be using the following notation and assumptions. The operation of a computer system can be considered to be an ordered sequence of *events* that take place on the various machines and networks that make up the system. An event may be considered to be

any activity or state transition such as the signaling of an interrupt, invocation of a command, termination of a process, transmission/reception of network data, and so forth depending on the level of abstraction being used. A subset of these events are of concern to our security monitoring schemas and a subset of that describes those events for which we wish to generate alerts. As this detection is generally thought of as the detection of malicious or undesired behavior, we refer to these as "bad events." We recognize that some of these events do not carry any sort of negative connotation (e.g., forensic data) but will use this term as it allows us to employ a simpler mnemonic in the discussion that follows.

The set of events that can take place on a computer system is represented by the set $E$. For a set of events $E$ that can take place on a computer system, there exists a subset of bad events $B$ such that

$$B \subseteq E$$

and there exists events $a$, $b$, and $c$ such that

$$\begin{aligned} a, b, c &\in E \\ b &\in B \end{aligned}$$

We use the notation $t_x$ to represent the absolute time of the occurrence of event $x$. This measure is based on some external observation reference point and is independent of the local measurements of time. This allows us to ignore the issue of clock synchronization for the purposes of our discussion. While this permits us to maintain a certain degree of simplicity for our model, this should not be taken as in anyway minimizing the importance of clock synchronization within a computer system (either single- or multi-machine). See [Mil91] for a discussion of clock synchronization and a possible solution.

We use the notation $x \rightarrow y$ to indicate that $y$ is causally dependent upon the occurrence of $x$, meaning that event $y$ cannot occur until after event $x$. Unless

otherwise noted, we will assume that the dependence of the events in question will take place in alphabetic order, namely

$$a \rightarrow b \rightarrow c$$

which implies that the absolute times are similarly ordered

$$t_a < t_b < t_c$$

This indicates that event $b$ cannot occur before $a$ takes place, and both $a$ and $b$ are preconditions for $c$. However, this relationship does not indicate that $a$ causes either $b$ or $c$ to occur, merely that $b$ and $c$ cannot occur without $a$ having previously occurred.

We also define a detection function $D(x)$, which represents the determination of the truth of the statement $x \in B$ by the CSM system. Ideally, we would like to be able to evaluate $D(x)$ from the knowledge of the membership of $B$. This would indicate an ability to pre-determine what all of the "bad" events are (the *a priori* enumeration discussed in Section 3.2.1). Unfortunately, in most implementations, the membership of $B$ is determined based on the output of $D(x)$ as $D(x)$ represents the detection capabilities of the CSM system, which is the converse of our ideal case. In part, this is caused by the difficulty of enumerating the membership of set $B$ in advance and the complexity of the possible states and events of a modern computing system. Furthermore, *false positives* are instances where an event $x$ is not actually a bad event, but does trigger an alert from the CSM system.

$$\text{False positive: } x \notin B, D(x) = true$$

Similarly, *false negatives* are instances where $x$ is a bad event, but the CSM system fails to classify it as such.

$$\text{False negative: } x \in B, D(x) = false$$

Using this notation, we now present four categories of timeliness that a CSM system could be operating in: real-time, near real-time, periodic, and retrospective.

### 3.3.2 Real-Time

*Real-time detection* takes place as the system is operating. Originally, this meant that detection occurred concurrently with the data generation. However, we further restrict the limits such that the detection needs to occur before any of the events logically dependent on the bad event take place. This can be represented by the ordering

$$t_a < t_{D(b)} < t_c$$

or alternatively in interval notation

$$t_{D(b)} \in (t_a, t_c) \tag{3.1}$$

indicating that the detection of the bad event takes place after any precondition event, but before any dependent event. Our usage of "real-time" is intended to be much stricter than the historical usage that implied that both data generation and processing were happening simultaneously. Our real-time restrictions allows for a CSM system to make a guarantee of detecting a bad event before the subsequent events instead of simply making a "best-effort" attempt to process the information.

### 3.3.3 Near Real-Time

A *near real-time detection* system detects an event with membership in $B$ close to when the event occurs, but does not guarantee as tight a bound on when the determination takes place as compared to real-time detection. Specifically, the detection of the bad event $b$ occurs within some finite time $\delta$ of the actual occurrence of the event. Using the notation described above we stipulate that

$$\left| t_b - t_{D(b)} \right| \quad \leq \quad \delta \tag{3.2}$$

which can be represented in interval notation as

$$t_{D(b)} \quad \in \quad [t_b - \delta, t_b + \delta]$$

This indicates that detection need only take place within $\delta$ of the actual event, either before or after, and not necessarily before any causally related events take place. In current CSM systems, $\delta$ would be on the order of seconds or possibly minutes. While it might not make intuitive sense that detection of $b$ can occur before $b$ itself, consider the case of network attack detection using a passive sniffer where the sniffer is located upstream from the victim machine. In this case, it is possible to detect a malicious packet on the wire before it is delivered to the end host.

It has been suggested that we are not concerned with how far in advance the detection of $b$ takes place, leading to the expression

$$t_{D(b)} \quad \in \quad [-\infty, t_b + \delta]$$

While we are not opposed to having the detection take place as early as possible, having an unbounded upper limit might be used as a justification for the generation of false positives, because one could argue that the alerts that were generated *might* occur at some time in the future.

### 3.3.4 Periodic (or Batch)

In *periodic detection* systems (or *batch analysis*), blocks of records are submitted to a CSM system once every time interval rather than the on-demand, per-record basis that the previous two techniques likely employ. The time length of the interval is commonly on the order of minutes or hours (although a low number of days would not be unheard of). If we assume that records are delivered every interval of length $p$ then we would like

$$t_{D(b)} \quad \leq \quad t_b + 2 * p \tag{3.3}$$

The $2*p$ limit is based on the fact that there may be a delay of duration $p$ in sending the indicators of bad event $b$, and that we want our CSM system to perform the detection before the next batch of records are delivered, which will happen in another interval of duration $p$. In a worst case scenario, the event $b$ will occur immediately

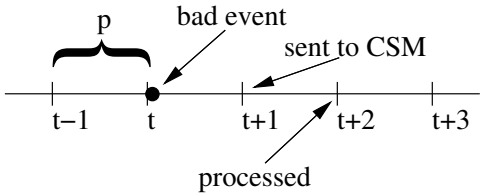


Figure 3.2. A graphical representation of the reason for the upper bound limit on a periodic CSM system.

after the last batch of records was sent to the CSM system and will not be sent until time $t_b + p$. The next batch of records will be sent after another interval of $p$, which would be the time $(t_b + p) + p$, and we want the decision $D(b)$ to be made no later than that time. See Figure 3.2 for a graphical illustration of this requirement. If the CSM cannot process the audit data at this rate, then as long as audit records are being continuously generated, the CSM will have an ever-increasing backlog of records to examine. This can exhaust the CSM space resources potentially causing the CSM to fail.

Generally, anything that delivers audit records via a file or over a non-local network connection will be in this category.

The fact that the upper limit is represented by $2 * p$ means that the audit data must be transmitted or polled twice as often as the guaranteed time. In other words, to ensure that detection takes place within 4 minutes of an event, the CSM must operate with a period of 2 minutes.

### 3.3.5 Retrospective (or Archival)

*Retrospective* (or *archival*) *detection* generally takes place outside of any particular time bounds. These systems are frequently designed to operate on non-"live" data or data from the operation of a computer system at a previous point in time (i.e., not data reflecting the current state of the system). As a result of this con-

Table 3.1
A chart summarizing the various time constraints needed for each of the various temporal categories a CSM system might be a member of.

| | |
|---|---|
| **Real-time** | $t_{D(b)} \in (t_a, t_c)$ |
| **Near real-time** | $\left\|t_b - t_{D(b)}\right\| \leq \delta$ |
| **Batch** | $t_{D(b)} \leq t_b + 2 * p$ |
| **Retrospective** | Unconstrained |

dition, retrospective systems cannot depend on being able to make queries of any systems, especially as the systems that generated the data might no longer exist.

Many of the CSM systems designed to operate under the previously discussed time bounds are able to be used in a retrospective mode. If it is possible to simulate the arrival of audit data, or specify a file or database containing such data to be processed, then it logically follows that the system can be fed recorded data. However, there may be additional considerations that need to be applied to the data generation and recording steps to enable such processing to take place later (e.g., relative timestamps, self-contained data references, usernames as well as user ID numbers, and so forth). Much of the research on data mining for CSM utilizes features that can only be generated after the entire session is complete (e.g., average value per session), not simply a running measurement. Archival systems have the advantage of being able to look ahead in the data stream to see if there is support for a particular hypothesis, giving such systems additional analytical power.

### 3.3.6 Summary of Timeliness

Table 3.1 summarizes the various constraints based on timeliness introduced in this section. In Section 3.4 we will examine examples of security monitoring systems from each of these categories.

Table 3.2
This table provides a cross-reference to the relevant portion of the text where the systems are explained.

| | **Type of Detection** | | | |
|---|---|---|---|---|
| **Timeliness** | Attack | Intrusion | Misuse | Forensics |
| Real-Time | 3.4.1 | 3.4.5 | 3.4.9 | 3.4.13 |
| Near Real-Time | 3.4.2 | 3.4.6 | 3.4.10 | 3.4.14 |
| Periodic | 3.4.3 | 3.4.7 | 3.4.11 | 3.4.15 |
| Archival | 3.4.4 | 3.4.8 | 3.4.12 | 3.4.16 |

These timeliness bounds can be used to determine the degree of interaction that needs to take place between an audit source and its CSM consumer. A CSM system that is designed to perform real-time detection will likely need to be able to directly interact with the data source to possibly suspend the execution of the system or process until an audit record can be processed, while an archival system will view the system as a one-way diode-like universe (either producing or consuming data without any interaction in the opposite direction).

## 3.4 Examples (Matrix)

By combining the two classification categories described in Sections 3.2 and 3.3 as axes, one forms a matrix of sixteen different types of computer security monitoring systems. In this section we will present illustrative examples of each and then describe the differences that exist among the varying groups. Table 3.2 indicates in which section each cell of the matrix is described.

### 3.4.1 Real-Time – Detection of Attacks

Real-time attack detection systems possess the characteristic ability to detect an attack or malicious code either right before delivery or immediately after impact. A widely available example would be that of a TSR anti-virus scanner. This software resides in memory examining every binary program that is about to be executed and searches it for known viruses (attacks) [HB00].

### 3.4.2 Near Real-Time – Detection of Attacks

Near real-time attack detection systems must detect an attack within a short interval of its occurrence. Many current systems that claim to be an "Intrusion Detection System" are based on network monitoring using a promiscuous network interface looking for attacks and are therefore in this category. Snort [Roe99] and NFR [NFR01] are instances of this type of system.

### 3.4.3 Periodic – Detection of Attacks

Periodic attack detection systems will either periodically review activity records (e.g., tcpdump logs) or scan storage areas in search of malicious code or other indicators of attacks. The scanner portion of most anti-virus software [HB00] will periodically examine mounted filesystems for known malicious software and infected programs. Similarly, there were a number of shell scripts designed to periodically detect Code Red attacks on webservers (e.g., [Car01b, Asa01]).

### 3.4.4 Archival – Detection of Attacks

Archival attack detection systems sieve through collections of audit records looking for traces of an attack. In 1998 and 1999, DARPA hosted an "Offline Intrusion Detection Evaluation" in which entrants sifted through collections of network data looking for a variety of attacks [LFG+00, HLF+01].

### 3.4.5 Real-Time – Detection of Intrusions

A real-time intrusion detection system needs to be able to detect intruders before they are able to actually utilize their access. Most software systems that perform authentication are instances of such a system, as an intruder would not possess the proper characteristics or credentials and therefore be detected as an intruder. For example, SSH stores known public keys of remote machines and performs a verification during a connection to detect man-in-the-middle attacks [The02,SSH02]. Similarly, biometric systems can use a thumbprint or other biometric measure to identify the legitimate user and detect impostors [JHP00].

### 3.4.6 Near Real-Time – Detection of Intrusions

Near real-time intrusion detection systems will determine if a particular user is unauthorized (masquerader, etc.) by examining the audit trail created shortly after events take place. Systems that attempt to perform user identification through anomaly analysis of command sequences (e.g., [LB97]) fall into this category.

### 3.4.7 Periodic – Detection of Intrusions

A periodic intrusion detection system collects its input data at regular intervals and determines if any of the observed actions are coming from an unauthorized user. Distributed collection/centralized analysis software that performs user profiling (e.g., UNICORN [CJN+95]) will be of this category if the audit data is sent in batches instead of being part of a continuous stream.

### 3.4.8 Archival – Detection of Intrusions

Archival intrusion detection systems analyze collected audit data and attempt to determine which actions (if any) indicate that an unauthorized access of the system

has taken place. Many of the user profiling programs can operate on archival data as well (e.g., [Sma88]).

### 3.4.9 Real-Time – Detection of Misuse

A real-time misuse detection system monitors the various user behaviors and actions that take place on a computer system and compares them against a given policy. It does so for each action as it occurs to enable the detection to take place in real-time. Janus [GWTB96] is one example of such a system. Janus uses the Solaris debugging facility to intercept all system calls performed by an executable. These are then compared against a policy file to determine their acceptability. If found to be in violation, either the call is prevented, an alert is raised, or both.

### 3.4.10 Near Real-Time – Detection of Misuse

Near real-time misuse detection systems perform similar comparisons of activities against specified policy; however, they frequently do so by reading system generated audit data instead of by direct observation. Polycenter [Dig94] and Hewlett-Packard's IDS/9000 [Hew01] are both instances of this type of detection system. IDS/9000 collects a variety of system audit data and passes them into a correlator, which performs the needed comparisons and generates alerts for violations.

### 3.4.11 Periodic – Detection of Misuse

Periodic misuse detection systems operate by either periodically reading a collection of records from log files, or by periodically receiving data from remote hosts. As with all misuse detection systems, these records are examined to determine if any sort of policy violation has occurred. Early versions of ISS RealSecure [Int01] on certain hardware platforms collected system audit records and, when polled, sent those records to an analysis machine. Another example is the syslog monitoring

programs that are run once a day via a cron job in most Linux distributions such as LogWatch [Bau02] or LogCheck [Psi02].

### 3.4.12 Archival – Detection of Misuse

An archival misuse detection system is designed to analyze stored records of computer usage and examine them for evidence of unauthorized activities. In some instances, these systems need meta-information on the audit records such as user ID to username mappings and system version information. According to Bace, the early detection systems constructed in the 1980's were designed to operate on both batches of audit records as well as archived stores [Bac02]. Another common system of archival misuse detection is the fraud detection software employed by credit card companies to detect card theft and abuse [CFPS99].

### 3.4.13 Real-Time – Computer Forensics

A real-time computer forensic system needs to be able to collect forensic information as an activity is occurring. This information would be related to the who, what, when, and why of the activity in question. An example of such a system is Carrier's **S**ession **To**ken **P**rotocol (STOP) [Car01a]. In STOP, when an incoming TCP connection is made, the operating system attempts to recursively track down the source of the connection to the originating machine and user across all intermediate machines. All activity for the connection request is suspended until the information has been collected.

### 3.4.14 Near Real-Time – Computer Forensics

Near real-time computer forensic systems collect information as it is being generated by the system. The FBI's DCS1000 software (a.k.a. Carnivore) [Fed01] collects digital information as part of a passive electronic wiretap. It is collecting the data

shortly after it has been generated and transmitted, but without an intervening storage stage.

### 3.4.15 Periodic – Computer Forensics

Periodic forensic systems collect computer system information after regular intervals. For instance, Tripwire [KS94] can be used to periodically analyze a file system and detect any unauthorized changes to files or directories.

### 3.4.16 Archival – Computer Forensics

Archival computer forensic tools make up the most common category of forensic software. These programs frequently take a computer system as a whole or major subcomponent thereof as input for analysis (e.g., filesystem, raw disk, memory dump). These tools construct timelines of activity, detect file modifications, attempt to recover deleted information, and so forth. The Coroners Toolkit [FV99] and Encase [Gui01] are two of the better known examples of such software.

## 3.5 Summary

In this chapter we presented two categories that can be used to describe CSM systems based on their design goals. The scope of monitoring for a system can be described as being the detection of attacks, the detection of intrusions, the detection of misuse, or computer forensics. The timeliness of detection describes how long it takes for a "bad" event to be detected by a CSM monitoring system. Our divisions of real-time, near real-time, periodic, and archival are much more descriptive than the traditional labels "online" and "offline."

We identified instances of all sixteen categories that are formed by using these two categories as axes for classification. This indicates that our categories describe existing systems.

Table 3.3
A summary of the sixteen different categories of CSM systems that can be described based on the characteristics of "Goal of Detection" and "Timeliness of Detection."

| | Real-time | Near real-time | Periodic | Archival |
|---|---|---|---|---|
| Attack | Detect an attack before any of the effects from the attack occur. | Detect an attack within a short interval of its occurrence. | After the audit data has been sent, detect an attack before the next batch of data is sent. | Detect an attack based on stored audit data. |
| Intrusion | Detect an intruder before they are able to affect system resources. | Detect an intruder within a short time of their accessing a computer system. | Detect an intruder within one interval period after audit data has been sent to the CSM system. | Detect an intruder based on stored audit data. |
| Misuse | Detect someone violating system policy before they are able to perform any other actions. | Detect someone violating system policy shortly after the violation takes place. | Detect someone violating system policy within one interval period of audit data being sent to the CSM system. | Detect instances of policy violation after-the-fact. |
| Forensics | Collect information on system activities before any following activities take place. | Collect information on system activities within a short interval of their occurrence. | Collect information on system activities periodically. | Reconstruct system activities (possibly from a non-live system). |

One way that this categorization improves upon current practice is that it allows for a better comparison of systems. By grouping CSM systems according to scope of monitoring, systems that are intended to have similar behaviors can be more easily identified and compared amongst one another. Additionally, this classification helps establish metrics with which to measure such systems because their purpose and timeliness of operations are more clearly articulated.

Another useful outcome is for the ability of an organization to determine if they have selected CSM systems that will give them the desired security coverage. When deciding which systems to use, one can look to the type of detection each CSM system is providing and select from among them to give the desired coverage. Consider an instance where a corporation has a mandate to use a CSM system that provides both attack and intrusion detection, which will be deployed throughout the company. The accounting division might have additional requirements involving misuse detection and can augment their detection coverage by selecting a misuse detection system to use as well. Another instance might be where the organization's security policy requires redundant systems for detection. By using our categorization, such an organization might see that to obtain adequate two level coverage, they would need to deploy more than two systems as the CSM systems each might only cover part of the desired space.

Requirements for security are generally "non-functional." An example would be the requirement for the "detection of misuses of the system." Our categorization improves this situation by describing more functional requirements for the timeliness behavior and how delivery of the data can be made and measured. It is important to note that our current categorization does not specify content, only parts of the delivery structure. The location of the data generation components and other requirements would drive content specifications. However, this categorization can also assist in the writing of such content specifications.

Our categorization also allows us to address the deficiencies in audit data specifications by re-examining them based on the intended end usage of the data. By

knowing for what the data is going to be used, new audit sources can be built that supply data tailored to one or more of our detection goals, with a delivery architecture that may support one or more timeliness categories. This allows us to reverse the traditional practice of building CSM systems to consume existing audit data and instead build better sources of audit information.

# 4 SPECIFICATIONS OF AUDIT SYSTEMS

In this chapter, we explain how the categorization of CSM systems helps the derivation of specifications for audit systems.

Again, we look at this issue from two complementary perspectives: the goals of detection and the timeliness of detection. The goals of detection influences *what* data should be collected in terms of content. The timeliness of detection describes *how* that data should be collected and stored along with influencing what additional information might need to be collected.

## 4.1 Types of Audit Records

Consider a generic computer system and the possible auditable events that can occur within that system. Table 4.1 lists some of those events and the data associated with them.

Table 4.1: Events that may possibly be audited in a generic computer system, and some of the data that could be associated with those auditable events.

**Identification and Authentication**

- password changed
- failed login events
- successful login attempts
- terminal type
- login location
- user identity queried
- login attempts to non-existent accounts
- terminal used
- login type (interactive/automatic)
- authentication method
- logout time
- total connection time
- reason for logout

*(continued)*

**OS operations**

- auditing enabled
- attempt to disable auditing
- attempt to change audit config.
- change in privilege
- change in group label
- "sensitive" command usage
- putting an object into another users memory space
- deletion of objects from other users memory space

**Successful program access**

- command names & arguments
- time of use
- day of use
- CPU time used
- wall time elapsed
- files accessed
- number of files accessed
- maximum memory used

**Failed program accesses**

Systemwide CPU activity (load)

Systemwide disk activity

Systemwide memory usage

**File accesses**

- file creation
- file read
- file write
- file deletion
- failed file accesses
- permission change
- label change
- directory modification
- attempt to access another users files
- attempt to access "sensitive" files

**Info on files**

- name
- timestamps
- type
- content
- group
- permissions
- label
- physical device

*(continued)*

| | |
|---|---|
| • owner | • disk blocks |
| **User interaction** | |
| • typing speed | • mouse movements |
| • typing errors | • mouse clicks |
| • typing intervals | • idle times |
| • typing rhythm | • connection time |
| • analog of pressure | • data sent from terminal |
| • window events | • data sent to terminal |
| • multiple events per location | |
| • multiple locations with events | |
| Hardcopy printed | |
| **Network activity** | |
| • packet received | • port opened |
| - protocol | • port closed |
| - source address | • connection requested |
| - destination address | • connection closed |
| - source port | • connection reset |
| - destination port | • machine going down |
| - length | |
| - payload size | |
| - payload | |
| - checksum | |
| - flags | |

Not all of these events are going to be necessary for every type of security monitoring. If we examine each group of events, we can see that audit systems that are designed to support a single type of monitoring will be supplying different audit records and different data with those audit records.

Table 4.2
Audit events related to identification and authentication

| **Identification and Authentication** | |
|---|---|
| Password changed | MF |
| Failed login events | AI F |
| Successful login attempts | IMF |
| Terminal type | IMF |
| User identity queried | AIMF |
| Terminal used | IMF |
| Login type (interactive/automatic) | IMF |
| Login location | IMF |
| Authentication method | I F |
| Logout time | IMF |
| Total connection time | I F |
| Reason for logout | AIMF |
| Login attempts to non-existent accounts | AI F |

We can label these events based on the type of CSM consumer that would be interested in them. We have done so with a subset of these events in Tables 4.2 through 4.5.

### 4.1.1 Identification and Authentication

These events are related to the actions and records generated when a user logs into the system establishing a session. The events that are useful to an ADS are those related to failures to properly authenticate. As most of these events are directly related to both user identification and authentication, an audit system supplying information to an IDS would be generating these various records. This information

is essential in establishing the current user identity on the system and the details regarding the type and location. An IDS needs most of this information as these events have to do with user identification and authentication, both of which are essential to the establishment of user identities and authorization. Data that is useful to an MDS comes from actions that take place after a user has authenticated herself to the system as the system cannot determine if a user is violating policy until after she has been identified. Information related to the location from which a user is connecting from would be useful to both IDS and MDS systems as such may be indicative of either an external penetrator or someone accessing the system remotely from an unauthorized location. All events are useful for a CFS and therefore would be generated and transferred.

### 4.1.2 OS Operations

The events in Table 4.3 occur within the kernel or operating system of the computer system. In contrast to the above, most of these events would be meaningful to an ADS because they are security related and have the potential to escalate privilege, violate the confidentiality or integrity of information, or impact other processes. The events that would be useful in detecting an intruder are those that involve changing either the identity or group label, and those events relating to attempting to invoke a restricted or sensitive operation. The former are important as they will result in a change in system identity, and the latter are important in that use of sensitive commands can help identify a masquerader. It is probable that the system policy will have rules addressing the use of these system events, so they should also be recorded for use in an MDS. Again, all of these events would be useful in some way to a CFS.

Table 4.3
Audit events related to OS operations

| **OS operations** | |
|---|---|
| Auditing enabled | F |
| Attempt to disable auditing | A MF |
| Attempt to change audit configuration | A MF |
| Change in privilege | AIMF |
| Change in group label | AIMF |
| "Sensitive" command usage | AIMF |
| Putting an object into another users memory space | A MF |
| Deletion of objects from other users memory space | A MF |
| Systemwide CPU activity (load) | A MF |
| Systemwide disk activity | MF |
| Systemwide memory usage | A MF |

Table 4.4
Audit events related to program access

| **Program access** | |
|---|---|
| Successful program access | AIMF |
| • command names | AIMF |
| • time of use | IMF |
| • day of use | IMF |
| • CPU time used | AIMF |
| • wall time elapsed | AIMF |
| • files accessed | AIMF |
| • number of files accessed | IMF |
| • maximum memory used | AIMF |
| Failed program accesses | IMF |

Table 4.5
Audit events related to file accesses

| **File accesses** | |
|---|---|
| File creation | MF |
| File read | MF |
| File write | MF |
| File deletion | A MF |
| Failed file accesses | IMF |
| Permission change | IMF |
| Label change | IMF |
| Directory modification | MF |
| Attempt to access another users files | AIMF |
| Attempt to access “sensitive” files | AIMF |

### 4.1.3 Program Access

The events in Table 4.4 pertain to both successful and unsuccessful execution of programs. An ADS is going to be interested in successful program execution events and information regarding the behavior of that program such as files accessed, resources used, and so forth. The use, and attempted use, of programs has been used as a technique to establish the identity of the person controlling a particular session. An IDS can use information on what programs a user attempted to access (including when access was denied) and how that program behaved. Similarly, this information can also be used to verify a user is using authorized resources properly.

### 4.1.4 File Accesses

The events in Table 4.5 are related to various sorts of operations on files on a computer system. Most of these actions would not be useful in the detection

of attacks except for file deletions and accessing files belonging to other users or otherwise described as "sensitive." File accesses that are noteworthy to an IDS include those that change the permissions or labels on files as well as attempts to access files belong to others. All of these events would be useful to a MDS because they relate to the behavior of the user. Most of these events are successful attempts, and misuse is based on authorized users abusing that authorization.

### 4.1.5 Audit Content Summary

In this brief example, we have shown that the set of audit records that should be generated for the various forms of detection are not identical. The type of content that is generated by an audit system should be selected based on the goals of detection that it intends to support. In Section 4.3 we discuss some of the broader concerns for audit sources tailored to meet the specific goals of detection.

## 4.2 Timeliness

As discussed in Chapter 3, the type and amount of data that an audit system generates may be highly dependent upon the time frame in which the CSM system is designed to operate. In some implementations, too much data may be counterproductive as this may supply more data that can be adequately handled within the desired timeliness bounds. However, a conflicting requirement exists in that much of the information is ephemeral to varying degrees and if it is not reported, it might not be available for analysis later.

### Volatility

Farmer and Venema give an example of items with differing volatility in [FV99]. While they are specifically discussing computer forensics, the issues of volatility they

raise are still relevant in other security monitoring contexts. Specifically they rank the following items of a system from most volatile downward:

- Registers, peripheral memory, caches, etc.
- Memory (kernel, physical)
- Network state
- Running processes
- Disk
- Floppies, backup media, etc.
- CD-ROMs, printouts, etc.

One of the issues in data collection is the fact that the act of collecting information often causes incidental modification of the system itself, possibly compromising the data. For example, generating an audit record and sending it out to an analysis or storage system will cause changes in registers, CPU cache, memory of the current process, kernel state changes (both in process tables and device drivers), network buffers, and the generation of a number of packets of network data that might affect ARP tables and other similar structures This is not a problem that can be solved through a recategorization of systems, and therefore outside of the scope of this dissertation. However, it does affect the specification of an audit system in that if a CSM system requires some of the more volatile information, then the audit system must be designed in such a way to capture that information first, and in a manner that does so without disturbing other information that needs to be recorded.

### System State Changes

Another consideration that must be made involves the data representation and the inclusion of meta-information along with the required audit data. Consider an

audit record that includes the process ID of the parent process, and assume that the CSM system requests additional information on that process. As time progresses from the initial record generation, the CSM system risks that the information might no longer be available, for instance the parent process might terminate. Further, it is also possible that a later query for information might return valid information that refers to something other than what was desired. For instance, if the parent process ID was $X$, a query for information about that process might return information about a different process that happens to have the same PID at the time that the query is made. The information is valid, and the response to the query returns what is requested, but does not return the information that was intended. needed This has impacts both the type and amount of information that is included in an audit record. All information about the state of a system is subject to change over time, and the longer the interval between audit record generation and CSM query, the greater risk of not getting the desired information. Consequently, the amount and detail of information that is included will depend on the mechanism and timeliness of the audit source and the CSM consumer.

More ephemeral information needs to be recorded versus simply being available. For example, given a data buffer in a real-time system, the information regarding *where* the buffer is located might not need to be recorded. However, the information would need to be recorded if it is to be used in systems operating in other than RT mode as it is highly unlikely that such information will still be available for a query. Additionally, the longer the interval (or for archival purposes) more details should be included in the audit records. In an archival system, the audit trail should include not only the UID value, but also the mapping from UID to username if that information is going to be used. UID to username mappings are generally stable, but after a period of time, there can be many changes and information can either be lost or reused, for instance as might happen on a university server between semesters.

### 4.2.1 Real-Time Detection and Levels of Abstraction

Recall that in section 3.3.2 the definition of a real-time CSM system is one that operates under the condition that

$$t_{D(b)} \in (t_a, t_c)$$

where the detection of the "bad" events takes place after the predecessor events, but before consequent events. It is also necessary for each of these three events to occur in distinct time intervals that are separate from each other. The implication of this is that

**Insight 1** *Precursor and consequent events must exist in groups that are mutually exclusive with respect to the events under analysis at the level of abstraction currently being audited.*

For instance, if the execution of a program both causes the bad event to occur as well as subsequent actions, then real-time detection could not take place by monitoring events at the program execution request level of abstraction.

The events on a computer system can be viewed at different levels of abstraction representing different levels of granularity in the audit trail. The finer the granularity, the greater the number of events and audit data are generated by the system. Sometimes by changing the level of abstraction at which audit data is generated, an audit system can support real-time detection or not. So, in the instance of a program that both causes the bad event and the subsequent effects, we can design an audit source that can supply information at the proper level of abstraction such that real-time detection can take place.

Similarly, the granularity can be made finer to the level of program lines, machine instructions, or CPU cycles. One of the side effects of using finer granularity is that there are more audit records being generated. In Table 4.6, you can see that by moving from a coarse level (such as program execution) to a finer grained level (such as library calls), the number of events begins increasing. Were the resolution to

Table 4.6
A table showing how an event can be broken down into sub-events depending on the level of abstraction.

| **Program** | **Shell** | **System Call** | **Function Call** |
|---|---|---|---|
| 1. prog. execution | 1. prog. start | 1. exec(program) | 1. .init() |
| | 2. prog. end | 2. syscall1() | 2. main() |
| | | 3. syscall2() | 3. libcall1() |
| | | ... | 4. libcall2() |
| | | ... | ... |
| | | n. exit() | m. fini() |

be taken down to the level of CPU cycles, then a current computer system would be generating in excess of 1,000,000,000 audit records per second. While finely grained audit records would guarantee that each event occurs in groupings that are mutually exclusive, the volume of data generated would overwhelm most CSM implementations unless they were designed for this type of data or specifically needed such information. Consequent events are not allowed to occur until after analysis has taken place. Therefore, using an unneeded level of detail in audit data generation is wasteful of resources (time, CPU, storage, etc.) and should be avoided. This leads to our next recommendation for the design of an audit system:

**Insight 2** *Granularity of audit should be made finer [only] until such point that the events to be observed occur in mutually exclusive groups.*

While this statement is not a requirement, it should factor in as a design principle when constructing audit systems. It is especially relevant for audit systems designed to support real-time CSM systems.

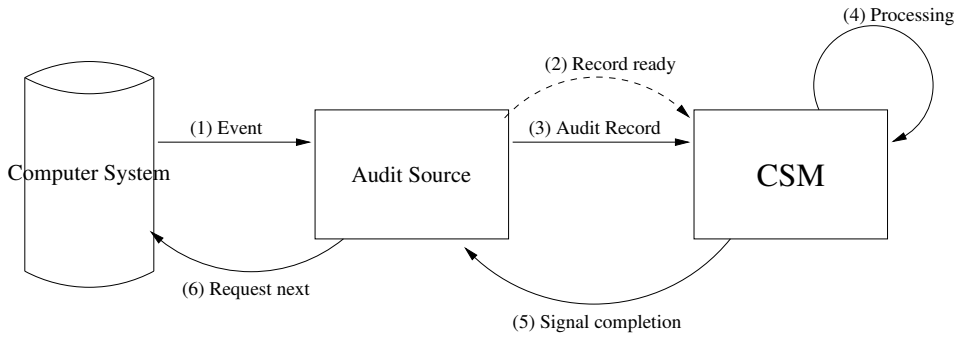


Figure 4.1. Steps needed for the communication between an audit source and a CSM to perform real-time detection.

Handshaking

If we assume a generic audit event at the appropriate level of abstraction, then figure 4.2.1 represents a model of the communication that needs to take place between the various components in an audit system. The operating system generates an event and either sends information of that event to the audit system or the audit system collects the information from the OS itself (path 1). From there, the audit system can optionally signal the CSM to indicate that an audit record is ready to send (path 2). When ready, either the audit source sends the audit record to the CSM or the CSM reads the audit record from the source (path 3). As the CSM processes the record, it may generate one or more additional events that feed back into the CSM system either for further processing or storage until a final decision has been made (path 4). Examples of generated events include correlated/coalesced events, alerts, timer information, and so forth. When ready for the next record, the CSM optionally signals the audit source (path 5). If a CSM uses polling, then paths 2, 3, and 5 are collapsed into the polling read that the CSM makes of the audit source. In many cases, the only paths that are present are 1, 3, and 4.

To ensure that real-time detection can take place, there needs to be a guarantee that no subsequent event takes place before the end of processing a given event,

including meta-events, completes (i.e., path 4). There are two ways in which this can take place.

1. The operating system blocks all further execution until it receives an acknowledgement signal (path 6); or,

2. The audit source and CSM guarantee that steps 2–5 can be traversed before the next possible event can take place (step 1).

In the first instance, by blocking further execution we guarantee that no further events occur, therefore allowing detection to take place in whatever timespan is needed by the particular CSM system. We are arbitrarily delaying $t_c$ until after $t_{D(b)}$ is known to have occurred to successfully meet our definition.

The second solution incorporates what might be described as "hardware real-time" in that it more closely reflects the behaviors that take place in "real-time systems." Frequently, such systems utilize a set of timers and known operating times for routines to guarantee that an event or set of events will take place within a known amount of wall-clock time. If we assume that we are operating with audit records corresponding to system calls, that our CSM system operates independently of the system being monitored, and that the time it takes to process a system call request and context switch is 50 msec, then steps 1–4 must be guaranteed to take place in less than 50 msec. If we can make such a guarantee, then we can operate without needing to have the back-channel path 6.

This second approach ties the CSM system much closer to the underlying hardware in that the same code will have differing execution times depending on the underlying speed of the system. It is possible that a system that is operating in real-time might not be able to function in that manner if the hardware is modified requiring careful attention to hardware detail when constructing the system. For instance, adding a new network card might increase the length of time spent handling interrupts, or an increase in memory size might increase the latency of data accesses.

There are some modern audit sources that communicate state with the operating system. One example would be Hewlett-Packard's Intrusion Detection Data Source (IDDS) [Hew00,CK01] which has a "blocking" mode of operation in which the system will stop processing events that generate audit data whenever the IDDS output buffers are full. This suspends operations until the data consumer has time to read the next block of records. However, this system only uses the signaling between the audit source and the operating system – it does not have any indication whether the CSM system has processed any particular record, only that the audit source's output buffer is no longer full.

The relative speed and computing power of both the system generating audit records and the system on which the CSM is located are important. To ensure real-time operation, the CSM must be able to consume audit records at a faster rate than the audit system can generate them. The difficulty of guaranteeing sufficient computing power becomes much harder when there is distributed data generation but centralized analysis. Also, CSM systems that are passively monitoring events, such as a network sniffer, may not be able to observe all of the events that are occurring.

One technique that has been used to mitigate the problem of differing CPU requirements is to have both the audit source and the CSM share the same computing resources, but with the CSM running at a higher priority. As more audit records are generated, the CSM has more work to do. Because of the higher priority, the CSM consumes more of the available computing time resulting in fewer events occurring and fewer audit records to be generated. However, this technique is ineffective in instances of passive monitoring.

Other issues in real-time audit systems include:

1. **Buffering**: If path 1 uses buffered I/O, then it is possible that event $c$ might have occurred before the buffer was written to the audit record. Similarly, if path 3 is buffered, then the audit source might not have sent the record on to the CSM until after event $c$ occurs. This buffering might skew any guarantee

of detection time enough that a real-time system will be transformed into a near real-time system or even a periodic system with a low period interval.

This is of concern when a system is under high load as such a situation cause a significant degradation on system throughput and performance.

2. **Granularity:** Deciding the correct granularity is difficult. If a CSM is only using part of the event reported as an audit record, then it might not be able to make the detection before the consequent events take place.

3. **Where data is collected**: As discussed in [Zam01] the location of audit data collection elements is important in the effectiveness and type of data collected. For instance, if network data is collected in user space, but an attack is launched that affects the network driver, it might not be possible to ever see event $b$. Or consider the instance of a land attack [CER97] where the audit source can see the packet containing identical source and destination IP addresses (event $b$), but may never see the subsequent event $c$ because the network stack is sending the RST packets to itself through internal channels.

### 4.2.2 Near Real-Time Audit Systems

Recall that near real-time (NRT) computer security monitoring takes place under the restriction that the detection of an "bad" event occur within some $\delta$ of the occurrence.

$$\left| t_b - t_{D(b)} \right| \leq \delta$$

One of the goals of NRT system design is to keep the value of $\delta$ to a reasonably small value. This translates into an audit system designed in such a manner that it makes a best effort attempt to get the data out as quickly as possible.

The idea is to collect the *necessary* information and then any other related information that can be expediently included. There may not be sufficient time to perform translations between binary and printable formats, nor to perform transformation

from numeric values to an expanded representation. For instance, in the case of an audit record with a UID, there may not be time to perform a lookup of username from the ID number, nor might there be justification to delay the generation of the record. Similarly, a process ID might be reported rather than looking up the arguments and executable name from the process table. Many of these transformations require one or more filesystem accesses that could impose significant delays. If the information is not *required* then there must be consideration of the delay introduced by collecting and including that information.

Many existing audit systems operate in a manner that makes them compatible for NRT usage. One common design is an audit source that generates data that is written out to a file descriptor. Often, this is a file on disk, but through the use of named pipes or `/proc/fd` files, it is possible to have the data be sent directly to a CSM consumer or network transmitter.

Usually, audit data is generated as events occur and then fed into the audit output stream without any consideration of the state of the data consumer. The underlying assumption of the audit system is that the data stream is constantly being consumed, and the detection system assumes that it is able to process the data as fast as it becomes available. Often, there is little or no effort made to synchronize the behavior of these two components. This leads to NRT audit and monitoring systems operating in a stream-like manner – they rely on the blocking and buffering features of modern I/O systems to handle many of the issues in their communication.

Another issue that needs to be considered during the design of a NRT audit source is how to handle issues of concurrency. For instance, a system might have multiple CPUs, each of which might be generating audit records. There are a number of possible solutions to handling these issues including mutex locks, multiple streams, and CPU binding.

The first technique is to employ some sort of mutex lock on the audit system output to prevent more than one process from trying to access it at the same time. The standard issues of deadlock prevention or detection apply. Consideration must

be made in terms of the commingled audit streams that are generated. It is possible that the information about a child will appear before the record of the parent `fork`ing a child has been entered or similar problems. It is possible that the other CPUs might block while waiting for the mutex lock to become available, reducing the performance of the overall system to that of a uni-processor machine or worse.

The second technique is to avoid the issue of locking by simply having each CPU generate its own audit trail. This avoids the potential issue of performance degradation as a consequence of blocking on a mutex lock. However, this solution introduces the additional problem of trying to determine how to merge the various audit trails. This may require the CSM to become more complex as it will be consuming multiple audit streams, or for the introduction of a log merging function in either the operating system or the audit system.

The final technique is to bind the audit record generation to a single processor. This serves both as a mutex type lock (it can only generate one record at a time) and to merge multiple auditable events that may take place on other processors. This solution pushes the responsibility for handling the concurrency issues onto the audit source and away from the need to be handled by CSM themselves.

### 4.2.3 Periodic Audit Systems

Audit systems that operate in a periodic mode supply data such that it can be processed within an interval of the data being delivered from the audit system.

$$t_{D(b)} \leq t_b + 2 * p$$

Recall that the upper limit for NRT systems is $t_b + \delta$. Generally, $p$ is larger than $\delta$ ($p > \delta$), if not substantially larger ($p >> \delta$). This leads to a much greater interval between transmissions of audit data. Also, the audit data builds up for an interval of size $p$ before being transmitted leading to "bursty" delivery of audit data. Contrasted against the stream-like behavior of NRT systems, periodic audit systems operate in

a block-oriented mode; relatively large amounts of data are sent after a significant time interval. This leads to a number of design considerations.

The block data transmission requires that the audit system transmit complete audit records. (Streaming systems might have incomplete records, but it should be handled by the buffering mechanisms in the I/O functions.) This requires that the audit system be concerned with buffer sizes to determine if a record will completely fit inside an output buffer and locking audit files or buffers so that incomplete records are not transmitted.

As audit records are being sent as part of a larger data block, the impact of transmission latency is lessened and therefore increasing the amount of data transmitted will have a smaller impact. This means that additional contextual information can be included along with the audit records. Not only can such information be included with little additional cost, it should be included because the cost of requesting additional information is much higher in periodic systems. A request for additional information can usually be serviced within three intervals of data transmission (the initial send, the transmission of the query, and the reply), which in periodic systems can be quite substantial. Because the interval might be on the order of hours or days, the possibility of the information having changed is significant.

A possible audit system can be designed to take advantage of the delay between generation and transmission of audit data to incorporate additional information into the audit record. An audit record could be generated containing space for additional information. A second process could be performing the various lookups and fill in as many of the blanks as possible before the next transmission cycle. In this manner, the audit system can both supply information in a timely manner as well as include as much context as possible without introducing delays.

### 4.2.4 Archival Audit Data Generation

Audit systems that are designed to supply information for use in retrospective or archival mods of operation need to generate records with the understanding that the system on which the information was collected might no longer be available for analysis. The audit trail should be as self-contained as possible and be able to be used by CSM systems without them needing to make additional queries from the audit source. This has a different nuance from the above instances of attempting to make audit records self-contained. The importance of such in this case is not for an efficiency reason, rather, the concern is that the information will not be available.

One technique that can be used to incorporate more of the system state into the audit trail is *journaling*. In a journaling-type audit trail, the initial system state can be recorded during initialization and as changes occur, they are included in the audit trail. It is the initial state-dump that most strikingly distinguishes an archival audit trail from the others. Table 4.7 offers an idea of what events might be recorded as part of the initial system state dump as well as the subsequent events. The information collected in the initial dump as well as the subsequent events recorded need to be tailored to reflect the various goals of detection that this audit source is going to support.

In other timeliness modes, the amount of information is lessened because of the need to consider response time as well as the ability to make queries. However, archival systems can make no such assumptions. The data that is part of the audit trail may be the only record of the events that took place on the system. Consequently, audit systems that are being designed to operate with retrospective systems need to take such concerns into consideration.

## 4.3 Goals of Detection

Historically, there has been a marked division between the specification, design, and implementation of audit data sources and the specification, design, and imple-

Table 4.7
Information that might be collected as part of the state-dump that could be incorporated into an archival audit trail for a Unix system. [FV99, Dit01]

| | |
|---|---|
| Initial dump: | • `dmesg` output (current hardware configuration) |
| | • `/etc/rc` scripts being used (boot-time system configuration) |
| | • current process table |
| | • dump of `/etc/passwd` (current users and UID mappings) |
| | • kernel configuration files and state |
| | • `/etc/fstab` (mounted filesystems) |
| Later events: | • process executions (dump `argv` and `environ`) |
| | • mounting and unmounting of filesystems |
| | • filesystem changes (`mv`, `symlink`, `link`, etc.) |

mentation of Computer Security Monitoring systems. Frequently, the audit sources present in an operating system are based upon a set of standards intended to allow the system to pass some sort of security evaluation (e.g., TCSEC [US 85] or Common Criteria [Com]). The designers of CSM systems identify what information is available on the target system and then decide how they can use that information and what additional data collection the CSM needs to perform on its own.

As the audit systems are usually designed and built before the various CSM systems, it is reasonable to assume that they were designed to be general in nature and not tailored towards any particular type of security monitoring other than that which is inherent in the requirements met as part of the security evaluation. This has led to some CSM systems being designed based on the information available, rather than the more logical design of having the audit data produced based on the data needs of the consumers. Additionally, the general nature of the audit data implies that the audit source is providing information that is able to be used by multiple systems. The side effect of trying to support multiple detection goals is

that you supply information that is unneeded by some monitoring applications as well as (possibly) not providing all of the information that would be useful within a specific area.

This section discusses how an audit system might be designed to supply information based on the goal of detection that a CSM system could be performing. This can help in the design and construction of tailored audit sources that can either be separate structures or possibly a single system that can be configured to support the multiple goals of data consumers.

### 4.3.1 Detection of Attacks

Based on the definition presented in Section 3.2.1, an attack can either be described by a set of actions or by a particular sequence of data, such as payload, program, malicious data file, etc. Therefore, the detection of attacks can be performed by looking at either the actions or the objects on the system, or both. Where the information should be collected from depends on the type of attacks that are being looked for and the level of abstraction being used.

On a computer, user actions are best examined at either the kernel or application level. Many existing audit systems collect information about user actions on the kernel level. Information from the application level is often limited to the information that specific programs generate. Sometimes that has information about user behavior, but usually it is reporting about its own actions.

One of the most straightforward attack detection techniques involves the collection and comparison of all data (or at least a subset of a particular type of data) against a set of signature patterns. For instance, some virus detection tools monitor email traffic looking for a predefined set of strings that are indicative of email viruses. This can be expanded to more descriptive instances (e.g., sequence of `NOP` codes followed by machine instructions), but they necessarily need to be carefully described and finitely bounded or they risk being non-computable.

The type of data to be collected for attack detection will depend upon the type of attacks and nature of the detection being performed. However, we can suggest general categories of data corresponding to existing classes of attacks.

Systems that are looking at network based attacks need to have access to the packet header information and/or payload contents. Systems that are looking for known instances of malware (worms, viruses, etc.) will need access to the content of executable files either on disk or when they are loaded to be executed. Systems that are looking for buffer overflow attacks can either find instances of attack tools in a manner similar to malware or it can be looking for instances of attack by monitoring application level data copying functions. To address one type of race conditions, the audit source needs to include information on file accesses both by symbolic and resolved names.

### 4.3.2 Detection of Intrusions

The detection of intrusions takes place by identifying users with permission to access a computer system accessing things that they are not allowed to, or someone accessing a system without authorization. To perform either of these functions, the audit system needs to be supplying information about the user that is performing the actions. Modern operating systems perform user authentication and use access control lists as techniques to prevent such intrusions from occurring.

There are two types of detection that can be performed as part of the detection of intrusions. The first is to identify when the user authentication system has failed in some way. The second is to identify when there is a failure of the access control system.

When the authentication system fails, that means that someone is using an account that does not belong to him. For a CSM to detect this, it has to have sufficient information to decide if the user performing actions is the one who should be using this account. There have been a number of techniques proposed for authenticating a

user including the following: biometrics such as fingerprints, retinal images, or hand shapes; behavior such as typing patterns or mouse movements; and shared secrets such as simple passwords, image based passwords, or cryptographic keys. Other types of user behavior can be used for a profile-based system. This behavior can be collected at any level of abstraction, but to be useful, a user needs to have somewhat consistent behavior.

In the second instance (that of detecting someone with authorization to use the machine, but not having authorization to access the programs or files), a CSM needs to either determine that someone is using an account other than his own or that the access control mechanism either failed or was not properly specified. The first case is usually detected in the same manner as detecting authentication failures. The information that would be useful for determining ACL failures would involve the access policy which usually cannot be supplied by the audit system and the list of user attempts to access objects – both success and failure.

### 4.3.3 Detection of Misuse

The detection of misuse involves determining when a user is abusing his authorization. To that end the audit system needs to relay audit records about what information users are accessing. The level of information that should be reported depends upon the level at which the policy is written. In the case of system operators abusing authority, there might be sufficient information supplied at the kernel level reporting of file accesses. For "regular" users of the system, the information needed will depend upon the level at which the policy is written and how the information is stored.

If the information is divided into separate files for each record, then information collected at the kernel level regarding file accesses might be sufficient. If the information is commingled within a single file, then kernel level information may be insufficient. It might be possible to use kernel level information regarding file read

and write to determine what information has been accessed, but it generally would be better to collect audit information at the application level. If the access control policy operates based on procedure oriented access control, or if the mechanism for accessing information is standardized (e.g., a DBMS), then collecting information based on these procedures and methods would be recommended. However, without a standardized method for accessing the data, it would be difficult to generate appropriate information at the application level.

If detection of misuse is to take place in the retrospective time frame, it is important that the information regarding the computer system's policy is stored along with the rest of the audit records. Without the policy, it may not be possible to perform the detection of misuse because without knowing what is considered acceptable use, it is impossible to determine misuse. Consider an audit trail that is based on recording the queries of a health care database. Certain actions might have been acceptable at the time, but a change in policy (e.g., HIPAA implementation) might make those same actions misuse. If the policy is stable and long lasting, then it might be possible to reconstruct it based on the current policy. If the policy is dynamic, then periodic based systems would need to have copies of the policy, or at least information included in the audit trail indicating when a change in policy occurred.

### 4.3.4 Computer Forensics

Describing the audit information that should be collected to support computer forensics is difficult because almost any piece of information might be useful as part of forensic analysis. As the goal of computer forensics is to reconstruct the state of the system as well as events on the system, then any piece of information might be useful to that end.

In general criminology, forensics is a scientific process. There are generally accepted practices and procedures that are used as part of evidence collection as well

as describing what evidence should be collected. Computer forensics currently lacks much of this rigor as it is operating on transient electronic media being controlled by multiple types of operating systems, and this creates systems that are subject to multiple sets of "laws" or operating rules.

Dissimilarly, physical forensics only has one set of physical laws that are in play. Even radical location changes such as being on another planet will only affect some of the parameters of the system, not the basic model. However, for something like a removable flash drive, then the exact changes that take place to it will depend upon what operating system is affecting such changes, and possibly what version or patches have been applied to that specific machine. This lack of a common frame of reference suggests that much information about the environment in which the audit information is being generated should be included in the audit record, especially if this is to be used with an archival system.

The type of information that is collected to be used for computer forensics needs to include not only what the users are doing, but it must also include information as to what the computer system is doing on its own. This can include generating records about the physical hardware state (register values, memory contents, and machine instructions), the state and operation of physical devices (printers, network cards, etc.), storage media, operating system state (kernel configuration and state variables), process tables, daemons, and user actions. Table 4.8 gives an idea of both the scope and types of information that are available. The data should be collected in a manner consistent with its volatility.

If the forensic information is intended to be used in court, then there are additional concerns about the method used to collect information and a need to record the chain of custody for the evidence itself. There are publications describing how practices in physical evidence collection can be applied to digital evidence collection [CS03, CS04].

Table 4.8
A partial list of the types of information that can be collected as part of an audit trail designed to support computer forensics.

| | |
|---|---|
| Registers | Stack contents |
| Data Buffers<br>Size<br>Content<br>Location | Network Traffic<br>State of network<br>Addresses (IP, MAC, etc.)<br>Interface type |
| UID to name mappings | Running network services |
| Password fields (home directory, shell, UID, GID, etc.) | Open and `TIME_WAIT` network ports |
| Instructions (based on granularity)<br>System calls<br>Library calls<br>Machine instructions | Pathname resolutions<br>symlinks<br>disk and inode mappings |
| Processes<br>signal handlers<br>Parent PID<br>Children<br>Exit status<br>Resources used (self and children, both current and total)<br>CWD, PWD, controlling terminal | Filesystem state<br>Mounted filesystems<br>Mount points<br>Exported filesystems<br>Filesystem types |
| Other running processes | Executable code (binary) |
| Process transitions (`fork` and `exec`) | Users currently logged in |
| Interrupt occurrences (mouse, keyboard, etc.) | Network events |

# 5 AUDIT SOURCE

This chapter describes the implementation of a new audit source designed to provide data based on the type of detection the end user will be applying.

It is our hypothesis that our categorization of CSM systems will allow us to improve upon the design of audit systems. The idea is that by using the goal of detection as part of the design criteria, we can build audit systems that are providing more relevant information than an audit system that is designed to supply only general purpose data. To test this, we designed and constructed a set of audit sources. We elected to use Unix for our proof of concept platform because of our experience with the operating system and the availability of several machines for development and testing.

## 5.1 Audit System Design

Our design goals included building a set of audit sources that were:

- **Standalone**: Each audit source (attack, intrusion, misuse) must be able to operate independently of the other modes of operation.

- **Composable**: The audit sources should be such that we could use one or more of them simultaneously, preferably without requiring any major changes in architecture.

- **Not require recompilation:** While it is possible to build audit sources into an operating system at many different levels, many of those additions would require having access to the source code and recompiling it. While this is possible, a simpler solution would be one that did not require this step.

- **Portable:** Ideally, these audit sources would be able to be used on various Unix-like operating systems without needing substantial modifications other than a possible recompilation.

Based on past experience [KS98, KS99b], we knew that it was possible to attain these goals through the the use of a technique known as *library interposition*. Interposition is the "process of placing a new or different library function between the application and its reference to a library function" [Cur94]. This technique allows a programmer to intercept function calls to code located in shared libraries by directing the dynamic linker to first attempt to reference a function definition in a specified set of libraries before consulting the normal library search path.

In many operating systems, most applications make use of a standard set of routines and functions that are provided on the system and stored in libraries. This code is used by many applications and most systems use a technique known as *linking* to allow each program to utilize function within the library without requiring the code for the function to be duplicated in every copy of the source code. Additionally, a technique known as *dynamic linking/loading* is used to eliminate the need for these library functions to be copied into every compiled application and provide a way for libraries to be changed or updated without requiring the recompilation of all programs making use of the library.

What happens is that at compile time, a program that is dynamically linked contains its own functions but only references to shared library functions are stored within the program. The compiler will check along a *library search path* for libraries that contain references to unresolved symbols. If all functions are resolved, the compiled program is created (as an executable) with a list of what libraries are needed to execute the program with all external references.

At execution time, the dynamic loader copies the executable into memory and locates and maps all library dependencies. Many times, these libraries are already in memory, and instead of loading another copy of them, the programs can share a copy thus saving both space on disk for stored programs but also memory space

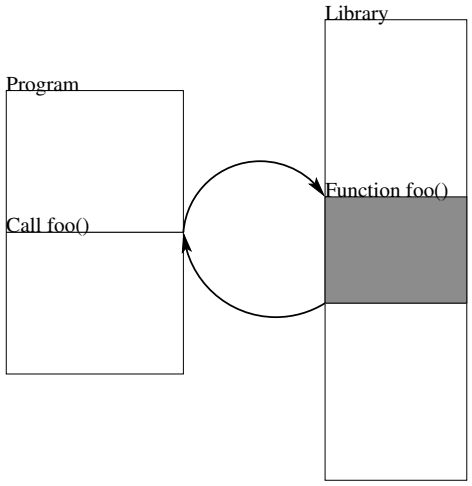


Figure 5.1. The normal way in which a dynamic library makes a library call

for executing programs. During the execution of the program if a reference to a library function is encountered then the linker traverses the shared library search path until it encounters a library that contains the function (loading it into memory if necessary). Figure 5.1 shows how this affects program flow.

Because the code is not stored in the executable program itself (and loaded dynamically at runtime), an administrator can upgrade/update/install new versions by replacing the existing libraries with another that has the same name and function entry points.

*Library interposition* is a technique that takes advantage of dynamic linking by causing a specially constructed dynamic library to be accessed instead of the standard library. An interposed library can be used to change the behavior of an application, collect additional information, modify arguments, or any other programmable task. Many times, the library is designed in such a way that the interposing function will call the actual library function in a manner that allows the program to behave as it would otherwise. Additionally, libraries that posses this sort of behavior are able to be composed in a manner such that more than one library call is interposing on the same function (see Figure 5.1).

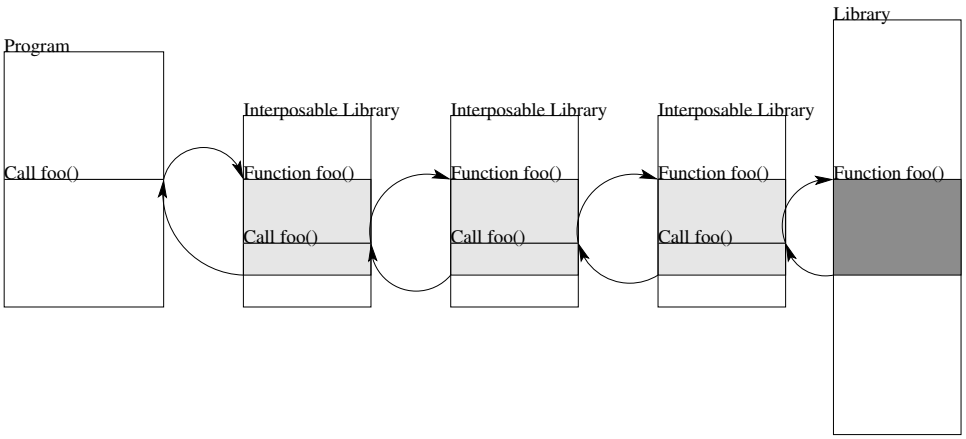


Figure 5.2. The way in which a program's library call can be resolved through a composition of interposable libraries.

Using interposition, we can collect information both before and after the next layer of library reference. Additionally, the values can be modified if desired (either the arguments or the return values) as part of the interposing code. More information about library interposition and the construction of interposable libraries can be found in [Nak01, Jon93, Sun96, Sun94].

### 5.1.1 Timeliness

This system only supports near real-time detection as anything that would support real-time detection would need to have collaboration with the data consumer as well as the data source. Additionally, any locking mechanism for the audit source would potentially interfere with performance benchmarks.

As this system is designed to support near real-time detection, the output mechanism will simply write the output to a file stream. By default, output will be sent to `stderr`, which usually is an unbuffered output stream. However, by setting an environment variable, the output of the library can be sent to a specified file for storage.

## 5.2 Implementation

The next sections describe the design of each of the audit systems created. The audit libraries were written in a deliberately naive manner. Specifically, there is no filtering that is applied to the event selection or audit record generation. If the design of the library is such that a particular event is to be recorded, then all instances of that event are recorded, even those that are unlikely to be of use within the particular scope of detection. This will cause the results generated to reflect the upper bound in terms of amount of data generated and overhead.

The key thing we wish to prove via our implementation is that more narrowly focused audit sources are capable of generating better audit data and a reduced sized audit trail. Detractors might criticize our implementation as being too expensive in terms of overhead. We acknowledge that performance improvements can be made by using more optimized, kernel-based methods of logging and data generation. These techniques are lacking in portability, but could potentially be added in at a later time. However, our implementation should adequately demonstrate our primary goals.

## 5.3 Attack Detection

There are a number of data items that are of concern to various attack detection CSM systems. For our prototype, we focused on building a library that was capable of supplying information that would be useful for the detection of buffer overflow attacks, format string attacks, and race conditions.

### 5.3.1 Buffer Overflows

**Buffer overflow attacks** are usually the undesirable side-effects of unbounded string copy functions. The most common examples from the C programming language are `strcpy()` and `gets()`, which copy each character from a source buffer to

a destination buffer until a null or newline character is reached, respectively. The vulnerability arises because neither checks whether the destination buffer is large enough to hold the source buffer's contents, nor can they take a parameter specifying the buffer size. In C programs, there is no bounds checking and what happens is that data is written into memory starting with the base address of the destination buffer and continuing sequentially until there is no more data to be copied. If more data is copied than space was allocated, adjacent memory locations will have been overwritten. If the buffer was a local variable (and therefore allocated on the program stack) then an overflowing buffer could overwrite other variables as well as stored frame pointers and return addresses.

Typically an attacker exploits this software fault in the following manner. A local buffer is filled in excess of its bounds to directly overwrite a return address on the stack. The data used to overwrite the saved return address is selected to redirect execution to code of the attackers choice (often contained within the same overflowed buffer and copied along with the rest of the data). Whenever program execution of the function responsible for the frame containing the modified return completes, the modified value will be loaded into the instruction pointer register, thereby directing execution to the malicious code.

If a CSM system is designed to detect buffer overflow attacks, then it is concerned with the following information on string copying functions:

- How long is the data to be copied?
- Where does the data come from?
- Where is the data going? (e.g., stack, heap, global storage)
- What is the length of the initial string? The final string?
- What is the content of the data string?

The following library functions were interposed for data collection:

- `strcpy()`, `strcat()`

- `gets()`
- `scanf()`, `vscanf()`, `fscanf()`, `vfscanf()`, `sscanf()`, `vsscanf()`
- `sprintf()`, `vsprintf()`
- `realpath()`, `getwd()`

The interposing audit library generates records for each consisting of

- record size
- function name
- source information (length, Global/Heap/Stack information, and value)
- destination information (length, Global/Heap/Stack information, and value)
- format string (for `printf` and `scanf`-like functions)

### 5.3.2 Format String Attacks

Format string attacks are a type of attack first recorded in the middle of 2000 [Scu01, Arc00, Bou00, New00]. These are similar to buffer overflow attacks in that data from the user of a program is incompletely mediated and that malicious input can be used to modify the program stack to cause the execution of an attacker's code.

Certain C library functions use format strings that allow a programmer to format data input and output using a predetermined set of conversion specifications. (A commonly encountered example is the `printf()` function.) When a function using these format specifications is invoked, a stack frame is created and the specification arguments are pushed onto the stack along with a pointer to the format string. When the function executes, the conversion specifiers will be replaced by the arguments on the stack. The vulnerability arises because programmers write statements such

as `printf(string)` instead of the proper form: `printf("%s",string)`. The statements operate identically unless `string` contains conversion specifiers. In this case, for each conversion specifier, `printf()` will pop an argument from the stack. There is no programmatic way to determine the number of parameters passed to `printf()` other than guessing based on the number of conversion specifications included in the format string.

If a user can specify the format string to be used, then an attacker can input an argument string using the conversions specifiers. Consider this short C program with a vulnerable `sprintf()` function.

```
#include <stdio.h>

int main(int argc, char *argv[]) {
    char buf[100];

    sprintf(buf,argv[1]);
    printf("-> %s <-\n",buf);
    return 0;
}
```

The first argument passed on the command line is being copied into the character array `buf`. If a user passes in the string "fluffy bunnies" then the output will be the following:

```
% ./a.out "fluffy bunnies"
-> fluffy bunnies <-
%
```

However, if a user includes any of the formatting specifiers, then `sprintf()` will interpret them. Because only two arguments are passed to the call to `sprintf()`, the values used by the specifiers will come from data already on the stack. An attacker

might decide to examine the values on the stack using the %x format specifier to print the value in hex.

```
% ./a.out "%x %x %x %x %x %x"
-> ffbee714 300 2234c ff29bc20 0 66666265 <-
%
```

Each time a %x is encountered, a word is popped from the stack and displayed in hex format. In this case, the value of the next six words on the stack are printed. Using this technique, the attacker can dump the contents of the entire stack.

The key to using this attack to modify stored data is the "%n" conversion specifier, which pops four bytes off the stack and writes the number of characters in the format string before "%n" to the address pointed to by the popped four bytes. Basically, if we move up the stack popping values using "%08x", and then when we reach a known address value we insert a "%n" in the format string, we would write the number of characters we had in the format string so far onto the return address on the stack. Note that length specifiers allow the creation of arbitrarily long format string "lengths" without needing the string itself to be an equal length. For example, the string "%.500x" is only six characters long, but would result in five hundred characters being printed.

One of the arguments that is passed to sprintf() is the format string itself. An attacker can use this to select an arbitrary address and store it in a known location on the stack as the following example shows.

```
% ./a.out "AAAA %x %x %x %x %x %x"
-> AAAA ffbee714 300 2234c ff29bc20 0 41414141 <-
%
```

The value of the character A is 41, so the sixth %x is displaying the first 4 characters of the format string, AAAA. If that %x were changed to a %n then the program would try to write the value 35 to memory address 0x41414141. By changing the first four

bytes and adding length specifiers, an attacker can write almost any value to almost any location in memory.

Ideally, the audit data necessary to detect these functions would be the format string and the number of arguments passed to that function. As previously mentioned, we cannot determine how many arguments were passed to that function, so instead we simply report the name of the function and the format string.

The following functions are interposed in this manner as part of the attack audit data generation library: `printf`, `fprintf`, `sprintf`, `snprintf`, `vprintf`, `vfprintf`, `vsprintf`, and `vsnprintf`.

### 5.3.3 Race Condition Attacks

**Race condition attacks** are more formally known as TOCTTOU attacks. *TOCTTOU* is an abbreviation for Time Of Check To Time Of Use and are formally defined in [BD96]. This class of attacks is possible because most operating systems have a weak binding between file names and actual files. This type of attack can take place during sequences of file access based on the name of a file rather than a file descriptor. In part, vulnerabilities of this type exist because the POSIX standard does not include some of the system calls that would be necessary to eliminate them. However, in many cases these faults are introduced by programmers in an non-deliberate manner.

Specifically, what happens is that a program accesses a file based on its pathname (either relative or absolute) and later attempts to access the same file via the same pathname expecting both references to resolve to the same file on disk. However, it is possible for a user to modify the file system in such a manner that the second request resolves to a different entity than the first request. To generate data that can be used to detect such a situation, the audit library collects information from the library functions that reference files by name rather than through a file descriptor. Those functions include the following:

- `access`
- `acl`
- `chmod`
- `chown`, `lchown`
- `creat`
- `execv`, `execve`, `execvp`
- `link`, `symlink`
- `mknod`
- `open`
- `rename`
- `stat`, `statvfs`, `lstat`
- `utime`, `utimes`

## 5.4 Intrusion Detection

For the detection of intrusions, we utilized the model proposed by Lane and Brodley [LB97, LB99], which uses command sequences to determine whenever an imposter is accessing someone else's account. This is based on generating a user profile of normal behavior that can then be used to detect anomalous changes through a computation of sequence similarity. The command sequence information is recorded by the library, but it has been augmented to include additional information about the program that is being executed and the state of the system at the time of that execution.

Additionally, process accounting information is also logged such that basic anomaly detection as proposed by Denning [Den87] can be performed.

Information recorded includes the following:

- The initial real, effective, and saved user identification values
- The final real, effective, and saved user identification values
- The initial real, effective, and saved group identification values
- The final real, effective, and saved group identification values
- The program name (as invoked by the user)
- The arguments from the command line
- The `stat()` information of the binary itself (to prevent user command aliasing problems)

- The amount of user CPU time spent executing
- The amount of system CPU time spent executing
- The total user CPU time used by all children
- The total system CPU time used by all children

## 5.5 Misuse Detection

The types of misuse we are looking for include file accesses such that a CSM system similar to Janus [GWTB96] can detect policy violations, the reading or accessing of another user's files, and information on read or write attempts on the same. Consequently, file accesses including accesses that are based on file handles are interposed to generate audit information including the following:

- `access`
- `open`, `fopen`
- `creat`, `mknod`
- `chown`, `lchown`, `fchown`
- `stat`, `lstat`, `fstat`, `statvfs`
- `truncate`, `ftruncate`
- `rename`
- `dup2`, `dup`
- `read`, `pread`
- `acl`, `facl`
- `close`, `fclose`
- `chmod`, `fchmod`
- `execv`, `execve`, `execvp`
- `unlink`
- `symlink`, `link`
- `tmpfile`, `mktemp`, `mkstemp`
- `write`, `pwrite`
- `utime`, `utimes`

A CSM system that is to use this information would also need to have a policy file that describes what are the acceptable file access policies. Because policies differ between organizations, the audit source cannot supply policy information on its own. Because we have not filtered out any of these events, the information from this audit source should be able to be used with any specified policy.

### 5.5.1 Audit Data Reporting

Audit information is reported by the interposing library via labeled variable size audit records with the following format:

```
TYPE : FNAME : PID SIZE PAYLOAD
```

where `TYPE` is a one character marker indicating which audit library generated the record ('A' for attack, 'I' for intrusion, and 'M' for misuse). `FNAME` is the name of the function or type of record represented by ASCII characters surrounded with delimiting ':' characters (there are no spaces between any of the fields in the actual audit record). The `PID` is the process number at the time when the program is initially invoked. The `SIZE` field is of type `size_t` and contains the length of the entire audit record in characters including the previously described header information. `PAYLOAD` contains the rest of the audit record in a variable format depending on the value of `FNAME`. For a full description of all audit records see CERIAS TR 2004-25 [Kup04].

Each library uses an environment variable to indicate where the audit records should be written. Those variables are `AUDLIB_ATTACK`, `AUDLIB_INTRUSION`, and `AUDLIB_MISUSE` for the respective library. If these variables are not set, output defaults to `stderr`.

## 5.6 Benchmark Testing

Testing was performed on a Sun Microsystems SunFire v100 with 256MB of memory. Comparisons were made against the Solaris BSM (Basic Security Module) audit system that is an integral part of Solaris. Information about the BSM module can be found in chapters 20–23 of [Sun03]. Two testing methodologies were applied. The first set of tests were done with a system benchmark and they are described in this section. The second set of tests were done using the performance of an

application and are described in Section 5.7. System benchmarks can magnify smaller effects caused through implementation choices.

Benchmarking was performed by using the BYTE Unix benchmark [SY91]. The documentation describes it as follows: "The BYTE UNIX benchmarks consist of eight groups of programs: arithmetic, system calls, memory operations[1], disk operations, Dhrystone, database operations[2], system loading, and miscellaneous. These can be roughly divided into the low-level tests (arithmetic, system calls, memory, disk, and Dhrystone) and high-level tests (database operations, system loading, and the C-compiler test that is part of the miscellaneous set)." The benchmark is designed to execute a particular task as many times as possible within a set time interval. It then repeats the test a number of times and averages the results to determine the value for a particular test. A run of the benchmark including all program groups takes approximately one hour.

### 5.6.1 Experimental Setup

The three interposable audit libraries were compiled on the target machine and installed into the system library directory (`/usr/lib`). To support both 32- and 64-bit applications, a 64-bit compilation was made and stored in `/usr/lib/64`. The system dynamic loader configuration was updated to include both of these directories in the system search path.

To measure the performance impact of the audit libraries, the benchmark was run as a normal user with no other operations taking place on the computer during the course of the execution. The benchmark was run once to establish a baseline, then run with each of the audit libraries in turn, and finally the baseline was run again making a total of 6 runs.[3] To enable BSM, the system needed to be rebooted.

[1] Memory access tests were deleted from the benchmark in January 1990.

[2] Database operation tests were removed from the distribution of the BYTEBench Benchmark in May 1991.

[3] 3 different library instantiations were tested yielding a set of 6 baseline runs. All of these were used to generate the average benchmark value.

At this point, BSM auditing was enabled for the user running the benchmark and a run with BSM auditing was performed. Then, auditing was configured to record no events and disabled, at which point in time another baseline run of the benchmark was made.

The general design for each of these benchmark sections is that a particular task is run in a loop for a fixed period of time, with the measurement being made of the number of loops that are successfully performed within that interval. This allows for a fixed runtime (approximately one hour for all tests to complete) as well as producing results in which larger values indicate "better" performance.

The tables in Section 5.6.2 to 5.6.8 use the following labeling conventions. The column labeled **Baseline Avg.** reports the mean value of the baseline runs. The column labels **A-Lib**, **I-Lib**, **M-Lib** refer to the values based on the run of the BYTEBench benchmark while using the audit source designed to generate data for the detection of attacks, intrusions, and misuse, respectively. The label **AIM** refers to the values based on the run of the benchmark with all three audit libraries generating data simultaneously. The column marked **BSM** has values from the run of the benchmark with BSM auditing enabled.

### 5.6.2 Results – Arithmetic Functions

The arithmetic tests are based on a source program that performs assignment, addition, subtraction, multiplication, and division with each test simply changing the data type that is being used in the performance of this task. These tests are all CPU intensive and the auditing systems probably only affected the performance during process initialization and conclusion so we expected minor effects, if any at all. The results of these tests are shown in Table 5.1 and the percentage difference from the average between the baseline runs is shown in Table 5.2.

As expected, all runs are close to the average baseline with a number of them actually performing *better* than the baseline. The largest variation is a decrease

Table 5.1
The results of the arithmetic tests of the BYTEBench 3.1 Benchmark. The results are expressed in terms of loops-per-second, and a higher value is considered to be better.

| **Data type** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** | **BSM/disabled** |
|---|---|---|---|---|---|---|---|
| Overhead | 4354281.9 | 4357155.7 | 4356670.4 | 4354866.8 | 4353858.4 | 4353987.0 | 4352582.5 |
| Register | 64684.0 | 64751.0 | 64701.6 | 64653.6 | 64814.5 | 64682.8 | 64619.1 |
| Short | 61500.2 | 61513.3 | 61509.1 | 61534.2 | 61565.8 | 61535.3 | 61481.9 |
| Integer | 64674.5 | 64733.1 | 64698.1 | 64653.4 | 64650.5 | 64603.9 | 64573.8 |
| Long | 64664.5 | 64702.5 | 64737.4 | 64638.6 | 64714.9 | 64725.4 | 64654.9 |
| Float | 168701.4 | 168804.4 | 168770.4 | 168718.3 | 168728.2 | 168699.2 | 168727.3 |
| Double | 131897.4 | 131950.0 | 131911.1 | 131869.4 | 131910.4 | 131876.0 | 131912.7 |

Table 5.2
The percentage difference from the average of the baseline runs of the arithmetic tests of the BYTEBench 3.1 Benchmark. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average.

| **Type** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|
| Overhead | -0.05% | -0.04% | 0.00% | 0.02% | 0.02% |
| Register | -0.12% | -0.04% | 0.03% | -0.22% | -0.01% |
| Short | -0.02% | -0.01% | -0.06% | -0.11% | -0.06% |
| Integer | -0.06% | -0.00% | 0.07% | 0.07% | 0.14% |
| Long | -0.06% | -0.12% | 0.04% | -0.08% | -0.10% |
| Float | -0.06% | -0.04% | -0.01% | -0.02% | -0.00% |
| Double | -0.04% | -0.01% | 0.02% | -0.01% | 0.01% |

of 0.22%, which occurs when computing with register variables and all three audit libraries enabled. The largest overhead is 0.14%, which occurs when BSM is being used with integer variables. The lack of significant change is notable because this test is CPU-bound and variations would have indicated that the audit libraries were interfering in an unexpected manner.

### 5.6.3 Results – Dhrystone

The Dhrystone benchmark was developed originally in 1984 [Wei84] and revised in 1988 [Wei88] by Dr. Reinhold P. Weicker while he was a part of Siemens AG. This benchmark is designed to model the mix of mathematical and other operations present in the typical programs of that era. This includes integer mathematics, no floating point calculations, array manipulation, character strings, indirect addressing, conditional operators, and program flow control operations.

The results of this benchmark are shown in Table 5.3 and the percentage change is shown in Table 5.4.

Table 5.3
The results of the Dhrystone 2 benchmark tests of the BYTEBench 3.1 Benchmark. The results are expressed in terms of loops-per-second, and a higher value is considered to be better.

| **Data type** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Without register variables | 1050044.8 | 1050594.3 | 1050162.6 | 1050011.0 | 1049827.0 | 1050292.4 |
| Using register variables | 1049918.0 | 1050414.0 | 1049701.3 | 1050150.6 | 1050347.5 | 1049637.5 |

Table 5.4
The percentage difference from the average of the baseline runs of the Dhrystone 2 tests of the BYTEBench 3.1 Benchmark. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average.

| **Type** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|
| w/o Reg. Vars. | -0.05% | -0.01% | 0.00% | 0.02% | -0.02% |
| Reg. Vars. | -0.05% | 0.02% | -0.02% | -0.04% | 0.03% |

In this case, we again see that the various audit systems have a negligible impact on performance, and in many cases increasing it slightly above average. The largest variation took place during the times when the library designed to generate data related to attacks were occurring. In both cases, the benchmark was 0.05% faster. It is possible that this performance improvement was caused by the library's technique of caching the function pointer to the interposed function. (A similar improvement was noticed in [KS99b].) The only library calls are `malloc`, `strcpy`, `strcmp`, and `printf` of which only `strcpy` and `printf` are interposed, and only in the Attack library.

A 2001 white paper [Wei02] by EEMBC (the Embedded Microprocessor Benchmark Consortium), it is pointed out that the Dhrystone benchmark can spend between 10% and 20% of its time in the `strcpy` and `strcmp` functions, so it was expected that the overhead of the interposing library would be noticeable, and only affecting the Attack library (the others do not collect information on these functions). Although noticeable, the performance difference is likely not significant.

### 5.6.4 Results – System Calls

The next set of tests are designed to measure the overhead of making system calls. The System Call Overhead Test performs a tight loop of the following calls:

```
close(dup(0));
getpid();
getuid();
umask(022);
```

of which `close()` and `dup()` are interposed by the misuse library. The Pipe Throughput Test creates a pipe and then reads and writes from it. Again, only the misuse library interposes on the `read()` and `write()` calls. The Pipe-based Context Switching Test behaves similarly; it uses two pipes and a set of processes to do the reading and writing. The Process Creation Test has the parent process `fork()` and then `wait()` for the child to `exit()`. The Execl Throughput Test recursively invokes itself via the `execl()` system call, keeping track of the number of iterations by modifying the command line arguments.

The results of these tests are shown in Table 5.5 and the percentage difference from the average baseline is shown in 5.6.

For the system call overhead test, the results for our audit libraries are as expected. Both the attack and intrusion libraries perform slightly better than the average baseline, however the misuse library incurred a substantial cost of approximately 40%. Surprisingly, this is only half of the overhead of using Solaris BSM (80%).

For the two pipe-based tests, the attack and intrusion libraries have limited impact (though the intrusion library manages to improve performance 7%) while the misuse library incurs significant cost. In this case, it is nearly 90%, probably because pipe-based communication takes place entirely in memory, but the audit library is making several I/O requests for logging purposes. The BSM audit system only has a 19% overhead, probably because the logging is taking place within the kernel avoiding the need for additional context switches when writing data to disk.

In the process creation test, the three audit libraries all had similar, substantial overhead costs (on the order of 40%–50%) which was worse than the 9% overhead that BSM had. The construction of this test is a tight loop of a forking parent

Table 5.5
The results of the system call benchmark tests of the BYTEBench 3.1 Benchmark. The results are expressed in terms of loops-per-second, and a higher value is considered to be better.

| **Test** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| System Call Overhead | 118385.1 | 119435.1 | 119621.0 | 72885.8 | 69151.5 | 22899.3 |
| Pipe Throughput | 74890.4 | 76344.9 | 80304.4 | 8232.3 | 8229.0 | 60324.2 |
| Pipe-based Context Switching | 33062.7 | 32336.8 | 32106.6 | 3951.0 | 3946.2 | 26697.5 |
| Process Creation | 661.9 | 404.9 | 352.0 | 346.6 | 265.0 | 604.0 |
| Execl Throughput | 274.6 | 160.3 | 161.6 | 157.9 | 111.9 | 218.2 |

Table 5.6
The percentage difference from the average of the baseline runs of the system call benchmark tests of the BYTEBench 3.1 Baseline. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Sys. Call Overhead | 118385.1 | 0% | -1% | 38% | 42% | 81% |
| Pipe Throughput | 74890.4 | -2% | -7% | 89% | 89% | 19% |
| Pipe-based Ctx. Sw. | 33062.7 | 2% | 3% | 88% | 88% | 19% |
| Process Creation | 661.9 | 39% | 47% | 48% | 60% | 9% |
| Execl Throughput | 274.6 | 42% | 41% | 42% | 60% | 21% |

process that then waits for the child to exit. The audit libraries are designed to record information on process start and termination involving a number of file `open()` requests that are probably dominating the overhead increase.

The `execl()` throughput test is interesting in that the program continuously executes itself, but passes information on the number of iterations by changing the command line arguments for each invocation. Again, we see that the audit libraries have over 40% overhead, which is double what is incurred by the BSM audit system (21%). It seems likely that the start-up routines for the audit libraries are responsible for this overhead.

### 5.6.5 Results – Filesystem

The next set of benchmark tests are designed to measure the capabilities of the filesystem. They consist of reading, writing, or copying a file for an interval of time. The resulting scores are based on data transfer rates (KBps) instead of the number of loops-per-second which the other tests use. The misuse audit library is watching for all file accesses (including reads and writes) and so will probably have a noticeable performance impact. Of the other two libraries, the only interposition will be from

the attack library tracking the `open()` and `fprintf()` requests, both of which only occur during the setup of the test.

The scores from these tests can be seen in Table 5.7 and the percentage of overhead incurred is in Table 5.8.

The file read and copy tests behaved as we expected with respect to the attack and intrusion audit libraries in that they had no performance overhead (rather, it seems they have improve performance). The misuse library did incur some performance cost, but it is much higher than anticipated. It is only performing a tenth of the throughput in the file read tests and one quarter of the throughput in the other two. In contrast, the BSM audit system has approximately 22% overhead for file read, 15% overhead on file write, and 8% overhead on file copy. The misuse library has a similar high/medium/low performance, but with differing ratios.

The misuse library is reporting every file `read()` request including information on where the buffer is located. In our experimental setup, there was only one disk meaning that the audit log is being written to the same filesystem that is used for the throughput tests. Every benchmark `read()` is going to cause a number of audit system `write()` requests. To test to see if this was possibly responsible, another run of the benchmark was performed using a set of audit libraries where the routine that was to write the log information to disk simply returned success without making the actual `write()` call. Table 5.9 shows the performance overhead of this configuration.

It is important to note the behavior of the BSM audit system when it is enabled, but no records are selected for logging, is not known. Our audit system is still generating the audit records, it simply does not write them to disk. The BSM system might not be generating any audit records in this situation. However, we are including this information for completeness.

The performance overhead for the misuse library during the file read test dropped from being over 90% to about 19%. The file write also decreased substantially from 75% to 26%. Surprisingly, the file copy test dropped even further, which is interesting considering that it is composed of multiple read and write requests. The attack and

Table 5.7
The results of the filesystem benchmark tests of the BYTEBench 3.1 Benchmark. The results are expressed in terms of kilobytes-per-second, and a higher value is considered to be better.

| **Test** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| File Read (10 seconds) | 201680.8 | 211138.0 | 213264.0 | 16973.0 | 17097.0 | 158800.0 |
| File Write (10 seconds) | 60910.8 | 49732.0 | 49300.0 | 12566.0 | 12566.0 | 51899.0 |
| File Copy (10 seconds) | 24130.2 | 24882.0 | 24489.0 | 6377.0 | 6353.0 | 21932.0 |
| File Read (30 seconds) | 202560.0 | 212605.0 | 213758.0 | 17077.0 | 17142.0 | 155930.0 |
| File Write (30 seconds) | 60603.2 | 49110.0 | 49244.0 | 12455.0 | 12322.0 | 51310.0 |
| File Copy (30 seconds) | 23715.7 | 24147.0 | 24065.0 | 6377.0 | 6364.0 | 21901.0 |

Table 5.8
The percentage difference from the average of the baseline runs of the filesystem tests of the BYTEBench 3.1 Benchmark. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| File Read (10s) | 201680.8 | -5% | -6% | 92% | 92% | 21% |
| File Write (10s) | 60910.8 | 18% | 19% | 79% | 79% | 15% |
| File Copy (10s) | 24130.2 | -3% | -2% | 74% | 74% | 9% |
| File Read (30s) | 202560.0 | -5% | -6% | 92% | 92% | 23% |
| File Write (30s) | 60603.2 | 19% | 19% | 79% | 80% | 15% |
| File Copy (30s) | 23715.7 | -2% | -2% | 73% | 73% | 8% |

Table 5.9
The percentage difference from the average of the baseline runs of the filesystem tests of the BYTEBench 3.1 Benchmark. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average. These tests were run using audit systems that were not actually writing the data out to disk.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| File Read (10 seconds) | 201680.8 | -5% | -5% | 19% | 19% | -5% |
| File Write (10 seconds) | 60910.8 | 20% | 20% | 27% | 26% | -2% |
| File Copy (10 seconds) | 24130.2 | -1% | -2% | 7% | 8% | -3% |
| File Read (30 seconds) | 202560.0 | -5% | -6% | 19% | 19% | -5% |
| File Write (30 seconds) | 60603.2 | 19% | 19% | 26% | 26% | -3% |
| File Copy (30 seconds) | 23715.7 | -1% | -2% | 7% | 8% | -2% |

intrusion libraries did not experience a decline in overhead during the file write test. This would seem to indicate that the overhead in these systems during this test is not being caused by the logging mechanism.

### 5.6.6 Results – System Loading

The next set of benchmark tests are based on seeing how the system behaves under variations in system load with a different number of concurrent processes. The system executes a number of shell scripts simultaneously and record the number of loops of the scripts that are able to be performed. The script that is executed sorts a short file, searches through that, and counts the results. The results of this portion of the benchmark can be seen in Table 5.10 and the performance overhead for each of the audit systems can be found in Table 5.11.

One of the most notable characteristics is that the performance overhead is consistent without regard to the number of concurrent processes. The attack and misuse audit libraries are over 29% overhead. The intrusion library has approximately 25% overhead. All three of these are larger than the overhead of 17% caused by the BSM system.

### 5.6.7 Results – Miscellaneous

The last few portions of the benchmark are based on miscellaneous tests. The first is a C Compiler test that performs iterations of the compilation of a sample program. The second is the calculation of the square root of 2 to 99 decimal places using the UNIX dc (desk calculator) utility. The third test is based on solving the Tower of Hanoi puzzle repeatedly. The raw scores for these tests can be seen in Table 5.12. The compiler and square root values are expressed in terms of loops-per-minute while the value for the Tower of Hanoi test is in loops-per-second. The percentage of overhead for each of the audit systems are shown in Table 5.13.

Table 5.10
The results of the system loading benchmark tests of the BYTEBench 3.1 Benchmark. The results are expressed in terms of loops-per-minute, and a higher value is considered to be better.

| **Test** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Shell scripts (1 concurrent) | 534.7 | 379.0 | 401.0 | 377.0 | 297.7 | 448.7 |
| Shell scripts (2 concurrent) | 276.8 | 196.0 | 208.0 | 195.0 | 154.0 | 229.0 |
| Shell scripts (4 concurrent) | 140.8 | 99.0 | 106.0 | 99.0 | 78.0 | 116.0 |
| Shell scripts (8 concurrent) | 70.7 | 50.0 | 53.0 | 50.0 | 39.0 | 58.0 |

Table 5.11
The percentage difference from the average of the baseline runs of the system loading tests of the BYTEBench 3.1 Benchmark. The number in parenthesis indicates the number of concurrent shell scripts being executed. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Shell scripts (1) | 534.7 | 29% | 25% | 30% | 44% | 16% |
| Shell scripts (2) | 276.8 | 29% | 25% | 30% | 44% | 17% |
| Shell scripts (4) | 140.8 | 30% | 25% | 30% | 45% | 18% |
| Shell scripts (8) | 70.7 | 29% | 25% | 29% | 45% | 18% |

For the C compiler test, the attack library had an 8% overhead, which was better than the 10% overhead incurred by the BSM system. The misuse audit system had slightly under 14% overhead. Surprisingly, the attack library had significantly higher overhead – 66% – which is the only time that the attack library had significantly worse performance than the others. This is probably because compilation involves many string operations that are only interposed in the the attack library.

For the calculation of the square root of two, all three of our libraries performed similarly. The overhead was similar to that of process execution. This is probably because the calculation involved is able to be performed quickly on current hardware, causing this portion of the benchmark to be reduced to that of process execution. However, the calculation does take long enough to reduce the impact that the library initialization and finalization routines have on system performance.

### 5.6.8 Results – Benchmark

The BYTEBench benchmark calculates an index value based on the scores of six of the individual benchmarks. They were selected in such a manner to span the various categories. The values are normalized against the performance of a Sun

Table 5.12
The results of the miscellaneous benchmark tests of the BYTEBench 3.1 Benchmark. The results are expressed in terms of loops-per-minute for the first two and loops-per-second for the third. A higher value is considered to be better.

| **Test** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| C Compiler | 319.0 | 108.3 | 293.2 | 275.3 | 103.3 | 287.2 |
| Dc: sqrt(2) to 99 decimal places | 10072.3 | 6341.7 | 6565.7 | 6220.0 | 4629.7 | 7977.6 |
| Recursion – Tower of Hanoi | 20336.9 | 20323.5 | 20329.1 | 20322.0 | 20337.5 | 20339.3 |

Table 5.13
The percentage difference from the average of the baseline runs of the miscellaneous tests of the BYTEBench 3.1 Benchmark. A smaller value is considered better, with negative values indicating a faster runtime than the baseline average.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| C Compiler | 319.0 | 66% | 8% | 14% | 68% | 10% |
| Sqrt(2) to 99 digits | 10072.3 | 37% | 35% | 38% | 54% | 21% |
| Tower of Hanoi | 20336.9 | 0% | 0% | 0% | 0% | 0% |

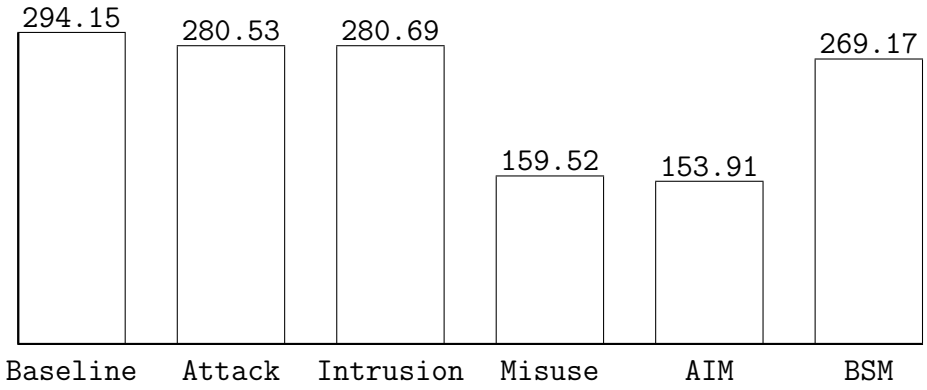


Figure 5.3. The Index values for the BYTEBench 3.1 benchmark.

Microsystems Sparc computer system (circa 1990) such that a computer performing identically will have an index of six (one point per test), twice as many operations would yield a twelve, and so forth. Table 5.14 shows the normalized scores for the various tests with Figure 5.3 representing the index values graphically. Table 5.15 shows the percentage difference from the average baseline score.

One of the interesting things to note is that the index value is dominated by the file system test. Unfortunately, this is a test on which the misuse library has a high amount of overhead cost, resulting in an overall index score that is 45.77% less than the average baseline index. The other two audit libraries only were slightly under 5%, which was better than the 8.49% overhead incurred by the BSM audit system.

Table 5.14
The index scores and normalized results that are used to calculate them for the BYTEBench 3.1 benchmark.

| **Test** | **Baseline Avg.** | **Attack** | **Intrusion** | **Misuse** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Arithmetic – double | 51.89 | 51.91 | 51.90 | 51.88 | 51.90 | 51.88 |
| Dhrystone (w/o register vars) | 46.94 | 46.97 | 46.95 | 46.95 | 46.94 | 46.96 |
| Execl Throughput | 16.82 | 9.72 | 9.79 | 9.57 | 6.78 | 13.22 |
| File Copy (30 seconds) | 134.72 | 134.90 | 134.44 | 35.63 | 35.55 | 122.35 |
| Pipe Throughput | 26.03 | 24.53 | 24.35 | 3.00 | 2.99 | 20.25 |
| Shell scripts (8 concurrent) | 17.75 | 12.50 | 13.25 | 12.50 | 9.75 | 14.50 |
| **Total** | 294.15 | 280.53 | 280.69 | 159.52 | 153.91 | 269.17 |

Table 5.15
The percentage overhead incurred by each auditing system as compared to the average index score of the baseline runs of the BYTEBench 3.1 system benchmark.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Index | 294.15 | 5% | 5% | 46% | 48% | 9% |

Table 5.16
The percentage of overhead that the various logging mechanisms generated on the BYTEBench 3.1 benchmark when auditing was enabled, but logging itself was not taking place.

| **Type** | **Avg** | **A-Lib** | **I-Lib** | **M-Lib** | **AIM** | **BSM** |
|---|---|---|---|---|---|---|
| Index | 294.15 | 4% | 4% | 10% | 12% | 7% |

Utilizing the data collected from the run of the various audit libraries without actually writing data to disk, we can attempt to eliminate the issue caused by the misuse library writing to disk whenever a read operation is taking place. Table 5.16 shows the overhead that takes place from the audit library without it actually writing data to disk.

The performance of the attack and intrusion libraries improved, but by less than 1%. However, the overhead of the misuse library improved by 36% to 10%. While this is a substantial improvement, it is still worse than the performance of BSM both with and without logging. Figure 5.4 shows the index values graphically.

### 5.6.9 Results – Log Sizes

One of the other goals of the specialized audit systems was to produce less audit data than a general purpose audit system. Table 5.17 and Figure 5.5 present the various log sizes created by the various audit systems during the run of the BYTEBench 3.1 benchmark.

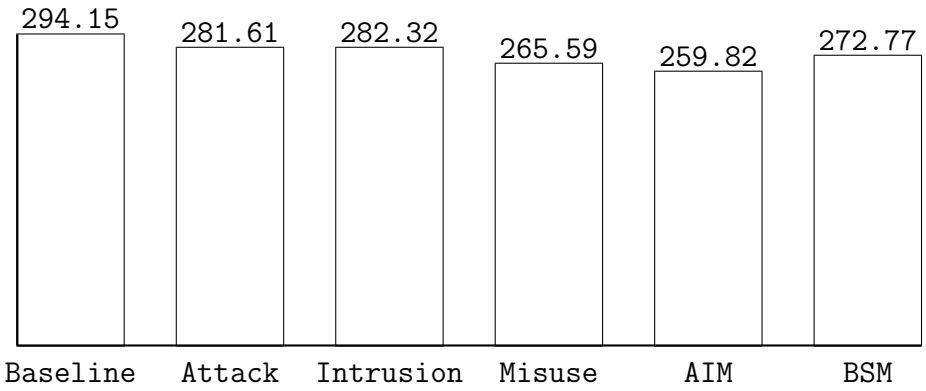


Figure 5.4. The Index values for the BYTEBench 3.1 benchmark with logging disabled.

Table 5.17
The file sizes of the various log files generated reported in terms of the number of bytes stored on disk.

| **Audit sources used** | **Size of log file** |
|---|---|
| Detection of Attack | 67785671 |
| Detection of Intrusion | 33653421 |
| Detection of Misuse | 554961827 |
| Detection of AIM (Total)<br>(Attack: 61176998)<br>(Intrusion: 26304418)<br>(Misuse: 536891216) | 624372632 |
| Solaris BSM | 961045085 |

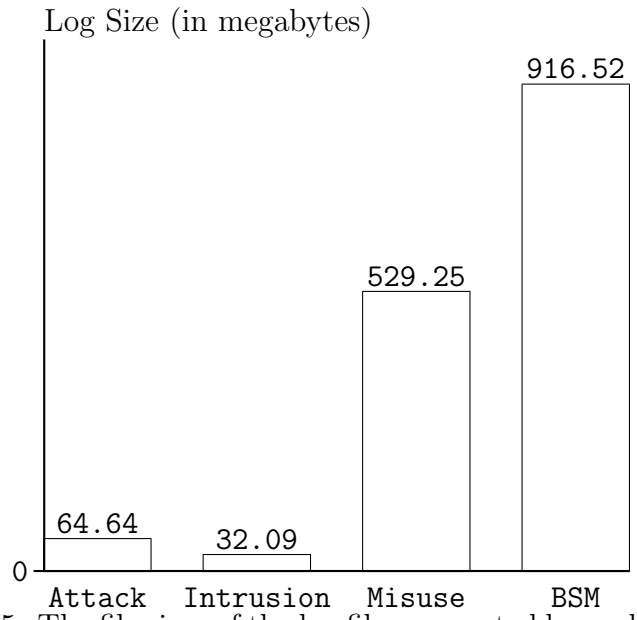


Figure 5.5. The file sizes of the log files generated by each of the audit sources during a run of the complete BYTEBench 3.1 benchmark.

The attack library generated 64.64MB of data, the intrusion library generated 32.09MB of data, and the misuse library generated 529.25MB of audit data. Whether taken singly, or all together, the amount of data generated is less than the 916.52MB produced by the BSM audit record. In all cases the audit log is still in its non-printable binary state.

### 5.6.10 Results – Number of Records

Another advantage of specialized audit systems is that they potentially generate fewer audit records that need to be handled by a CSM. Figure 5.6 presents the number of audit records generated during the run of the BYTEBench 3.1 benchmark.

### 5.6.11 Audit Log Content

Another one of the stated goals for improving audit systems was the inclusion of more useful information in the audit log to allow for security monitoring systems to

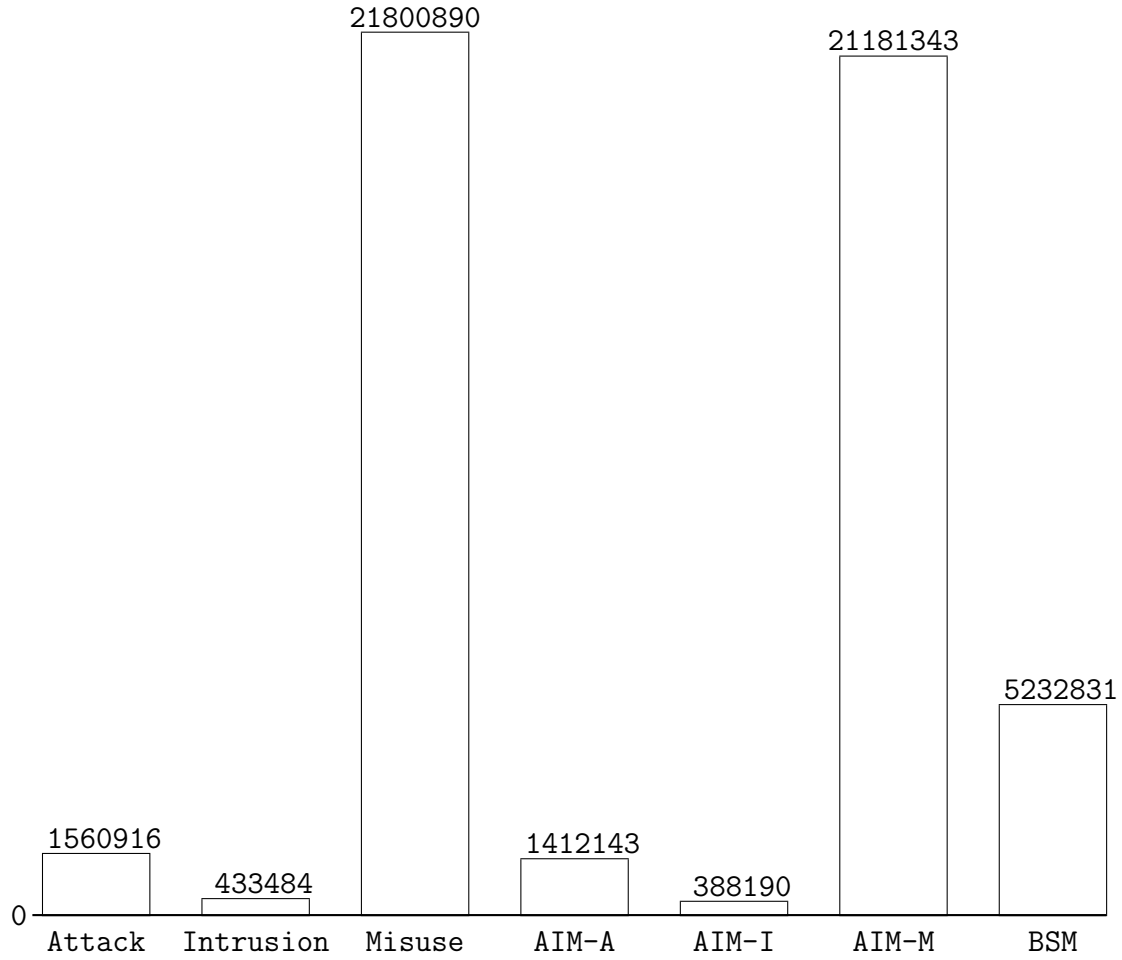


Figure 5.6. The number of audit records generated.

improve their accuracy and detection abilities. Although the audit trails generated by the three prototype libraries are much smaller than the BSM audit trail, there is information in each that is able to be used for the various goals of detection that is not present in the BSM audit trail.

Most of the data generated by the attack audit library is information that is not present within the BSM audit trail. Most of the records are based on information collected at the application level while BSM only has information collected by the kernel. Application level information is essential for the detection of a number of attacks, and the information regarding buffer size, content, and location is not available at the kernel level. The information regarding the symbolic names used in file accesses is included in our library, but is not present in the BSM record. Finally, there is no information in BSM regarding the format strings used during input or output routines. Almost all of the information in the attack library audit trail is not present in the BSM audit trail.

The data collected by the intrusion library has some overlap with the information contained in the BSM audit log, but much of it is novel. Information not present in the BSM audit trail includes the saved user identification values, information on the resources consumed by this process and its children, and some of the `stat` information regarding the executable.

The misuse library was looking at file accesses, all of which are mediated by the kernel and therefore should be present within the BSM audit trail. However, BSM does not record when a file is actually being read from or written to. This information is unique to our misuse library, and makes up approximately 56% of the total number of audit records generated.

## 5.7 Application Testing

The goal of these audit sources are that they will generate fewer records, consume less disk space, and provide new data not currently available for each of the

goals of detection. To test this, experimentation was performed with the Apache webserver and accompanying benchmarking tool `ab`. Version 1.3.27 of the webserver was downloaded from `http://www.sunfreeware.com/` and installed on the test machine. Two iterations were run, the first was to only have auditing enabled for the webserver. The second involved auditing the benchmark tool.

### 5.7.1 Apache Webserver

The Apache webserver package was installed and left in the default configuration. In this state, a default webpage is displayed when a web client connects to port 8080 on the server. The server pre-forks 8 children to handle incoming web requests and logs all URL requests in an application log file. The benchmarking tools were run without auditing and set to fetch the homepage 1000 times and present averaged results.

In terms of the number of records generated, we were interested in seeing that the server is generating more information regarding the internal operations so that detection of attacks can be performed. It is less likely that it will be generating information that is related to the detection of either intrusions or misuse. Table 5.18 shows the number of records generated, and Table 5.19 shows the size of the various audit logs.

The attack library generated significantly more audit records than the other libraries. These were almost all (97.4%) for the `strcpy()` library call. As this audit library records the contents of the data buffers, the size of the audit file is dependent on each instance of the call. However, it does generate an audit log that is ten times the size of the one generated by BSM, but it does so while generating thirty-two times more records. The intrusion library generates only eight records taking only 2KB of space. The misuse library generates approximately 70000 records, of which 90% are `stat()` calls.

Table 5.18
The number of audit records generated by the Apache webserver.

| **Audit sources used** | **Records generated** |
|---|---|
| Detection of Attack | 2483709 |
| Detection of Intrusion | 8 |
| Detection of Misuse | 69348 |
| Detection of AIM (Total)<br>(Attack: 2484219)<br>(Intrusion: 32)<br>(Misuse: 69501) | 2553752 |
| Solaris BSM | 77413 |

Table 5.19
The file sizes of the audit logs generated in terms of the number of bytes stored on disk generated by running the Apache webserver.

| **Audit sources used** | **Size of log file** |
|---|---|
| Detection of Attack | 128314520 |
| Detection of Intrusion | 2496 |
| Detection of Misuse | 4062970 |
| Detection of AIM (Total)<br>(Attack: 128314548)<br>(Intrusion: 2720)<br>(Misuse: 4063850) | 132381118 |
| Solaris BSM | 12973075 |

In terms of performance, the baseline setup is able to serve pages at a rate of 90.5 pages per second. With the attack library enabled, the server had a rate of 3.6 pages per second, with the intrusion library the server had a rate of 90.8 pages per second, with the misuse library the server handled 62.3 pages per second, and with BSM enabled the server throughput was 66.5 pages per second. A second iteration was performed using our non-logging audit sources and the performance was substantially better. In this instance when the attack library was enabled, the server had a rate of 67.6 pages per second, using the misuse library yielded 90.2 pages per second, and with the misuse library it served 86.9 pages per second. Again, much of the performance degradation appears to be coming from the recording of audit records, not the collection and generation of the information.

### 5.7.2 Apache Benchmark Application

The Apache webserver package includes a tool named `ab` which will rapidly request a given URL from a server. This tool was used to generate the traffic to the test webserver. For this set of tests, audit records were generated based on the operation of the `ab` tool. Table 5.20 shows the number of records generated, and Table 5.21 shows the size of the logfiles generated.

For these tests the attack and intrusion libraries generated few audit records, which were collectively under 5KB total size. The misuse library generated approximately 30000 records and took up 664KB. While the BSM audit library generated 40000 audit records (one third again what our libraries generated), they were substantially larger taking 5MB of disk space.

In terms of performance, the baseline was able to service 108.9 pages per second. Our attack library resulted in a rate of 108.6 pages per second, with our intrusion library the rate was 108.5 pages per second, and with our misuse library it served 94.9 pages per second. In contrast, with BSM enabled the benchmark requested only 77.3 pages per second. A second run with our non-logging libraries yielded a

Table 5.20
The number of audit records generated by the Apache webserver benchmarking tool `ab`.

| **Audit sources used** | **Records generated** |
|---|---|
| Detection of Attack | 112 |
| Detection of Intrusion | 3 |
| Detection of Misuse | 30007 |
| Detection of AIM (Total)<br>(Attack: 112)<br>(Intrusion: 4)<br>(Misuse: 30009) | 30125 |
| Solaris BSM | 40211 |

Table 5.21
The file sizes of the various log files generated reported in terms of the number of bytes stored on disk generated by running the Apache benchmark tool.

| **Audit sources used** | **Size of log file** |
|---|---|
| Detection of Attack | 4763 |
| Detection of Intrusion | 336 |
| Detection of Misuse | 680447 |
| Detection of AIM (Total)<br>(Attack: 4763)<br>(Intrusion: 364)<br>(Misuse: 680557) | 685684 |
| Solaris BSM | 5385900 |

baseline of 92 pages per second, and rates of 92.1, 92.1, and 92.3 pages per second for the attack, intrusion, and misuse libraries, respectively.

### 5.7.3 Apache Tests Summary

In the Apache tests, other than the attack library used on the webserver, our libraries generated fewer audit records, smaller audit logs, and had comparable or better performance. In the one instance where more audit records were generated, this was desirable because the audit records generated gave information that would be able to be used to detect attacks that were previously not able to be detected.

# 6 CONCLUSIONS, CONTRIBUTIONS, AND FUTURE WORK

In this chapter we will review the conclusions reached and summarize the contributions that have been made by this dissertation. It then discusses directions for future research on this subject.

## 6.1 Conclusions

Historically, computer security monitoring (CSM) systems have been classified based on their structural characteristics. One category used is the type of analysis being performed to make a decision whether to generate an alert or not. Techniques used include the detection of statistical anomalies; rule based detection through the encoding of an expert's decision making process, a set of signatures, or electronic representation of policy; and binary triggers. Additional categorizations are made based on where data is gathered and where it is processed. While useful in describing how the systems are constructed, these categorizations fail to classify systems such that they could be reasonably compared in terms of detection coverage and performance.

The sources of audit information in current operating systems were designed to meet TCSEC requirements put forth by the US Department of Defense. The TCSEC requirements were designed to support multi-level, secure computing with a particular worry about classified information being leaked to uncleared individuals. While leakage of information is still a concern, there are other threats that also need to be addressed.

The authors of CSM systems are building systems that perform a variety of types of detection based on these modern concerns. However, the audit information that modern systems make available was designed to supply information based on

the older concerns of TCSEC and designers of modern systems have been forced to make do with what is available. It is time for the information needs of the detection systems to decide what information should be in an audit trail instead of the current practice of deciding what can be detected based on the audit information supplied.

We have observed an increase in the amount of information despite seeing a reduction in the size of audit records. Improved quality of information is vital to assist in the reduction of false alarms in CSM systems. This improvement works towards correcting the problems expressed by "Garbage-In, Garbage-Out" – CSM systems can only be as accurate as the information with which they are provided.

## 6.2 Summary

In this dissertation we have demonstrated that it is possible to categorize computer security monitoring systems based on the goal of detection. We identified four categories: detection of attacks, detection of intrusions, detection of misuse, and computer forensics. The categorization is reasonable because there are instances of CSM systems that exemplify each goal, and there are identified differences between the categories. However, these categories are useful and complementary in terms of detection coverage (e.g., selecting a set of CSM systems such that you have one of each category) and layering (e.g., selecting more than one CSM system within the same category). Categorizing systems based on the goals of detection makes for more useful comparisons because items within the same group should be looking for similar activities.

We also demonstrated that it is possible to more rigorously define the timeliness of detection instead of using the current online/offline categories. We identified four modes of operation: real-time, near real-time, periodic, and archival. We formally defined the timeliness bounds for each category and described the differences that exist between each.

The utility of this categorization was further demonstrated by examining how audit sources can be designed to support both the goals of detection and timeliness of operation. We described what information each would need and discussed several of the architectural considerations that need to be made to support the various timeliness modes.

To verify that a special-purpose audit system was an improvement over existing general-purpose, TCSEC inspired audit systems, a set of three prototype audit sources were created – one each supplying audit information useful for the detection of attacks, intrusions, and misuse. These were then compared against Sun Microsystems' BSM audit system while running a system benchmark. All three audit systems generated less data, but with content that is not present in BSM. Additionally, in most cases the new audit libraries had less performance overhead on the operation of the computer system.

Another set of tests were performed using the Apache webserver and accompanying benchmark tool. When generating audit data during the operation of the webserver, the attack library generated substantially more data than the BSM audit system, but the intrusion and misuse libraries generated less in terms of number of records and log size. However, the information that was generated was related to the internal behavior the the webserver, and is useful for the detection of buffer overflow attacks. When generating audit information for the benchmarking tool, all three of our prototype libraries generated fewer records, smaller log files, and better performance than BSM.

## 6.3 Summary of Major Contributions

There were several specific contributions made by this dissertation including the following:

- **Goals of detection:** The various goals of CSM detection were described in detail with examples of each identified and described.

- **Timeliness of classification:** Formal bounds for the various modes of operation for CSM systems were identified and described.

- **Categorization of** CSM **systems:** The two above descriptions were used to classify CSM systems. This classification augments the existing classifications by grouping systems based on their behaviors instead of the current practice of classifying based on structure. This leads to a more specific language for describing a CSM system.

  While a CSM system might have behaviors that fit into multiple compartments (i.e., multiple goals of detection, multiple modes of operation), this categorization allows similar items to be grouped together. This can be used to establish better metrics by which to measure them by as their purpose and operation is more clearly articulated.

  This classification also allows an organization to select CSM systems such that they have appropriate *coverage* and *layering* in terms of types of detection to be performed. If an organization is unsure of their needs, then the classification can offer them a template from which to select systems.

  Requirements for security are often "non-functional," but the timeliness aspects allows for a more functional specifications.

- **Design of audit sources:** The effects of the above categorization on the design of audit sources was examined. Differences in the type of information and audit source behavior were described.

- **Requirements for real-time detection:** The behaviors necessary for real-time audit sources and CSM interaction are described.

- **Prototype audit sources:** A set of three prototype audit sources were constructed to demonstrate the validity of this approach and utility of the categorization. These were shown to impose less overhead (in general), to generate

significantly smaller amounts of audit data, and generate audit information that is more useful for the stated goal of detection.

While there are other contributions contained within the dissertation, these are the most salient points.

## 6.4 Future Work

There are number of directions in which this work can be expanded. While we have classified a number of existing CSM systems, there are certainly many more that can be classified using our categorization. One difficulty that is likely to arise is that many of these CSM systems have a closed or proprietary design, making it difficult to accurately ascertain goals of detection and timeliness modes of operation. However, a large scale application of this categorization might result in a useful tool for organizations to help them select appropriate CSM systems for their desired coverage.

The existing audit system prototype can be improved in a number of ways. First, each audit source can be expanded to included more events. For instance, the attack library currently supports three families of attacks, but this can be expanded to include more. Second, the audit libraries can be made to be configurable and support the enabling and disabling of specific audit events. At present there is no filtering capabilities; however, adding such should continue to improve the performance of the system and reduce the amount of information that is being generated. Third, the logging mechanism can be improved to reduce the amount of overhead incurred, especially during filesystem intensive tasks. This might involve using non-portable operating system routines. Finally, the audit generation capabilities can be incorporated into the standard set of libraries. This will allow information to be generated for all applications without the use of interposition, and can lead to having audit data generation capabilities being incorporated into statically linked binaries as well.

As mentioned earlier, audit information can be used for more than computer security monitoring. Additional research needs to be performed to examine if similarly focused special-purpose audit sources can support goals such as process accounting, health and performance monitoring, and system recovery.

Finally, prototypes based on some of the proposed architectures for audit systems can be constructed and examined. Among the possible designs are the following: a real-time audit system that supports the suspension of processes until being signaled; an audit source designed to operate with periodic CSM systems that utilizes a second process to fill in optional information while waiting for the audit data to be transmitted; and an audit system designed to supply information for use in an archival computer forensics based system.

# LIST OF REFERENCES

# LIST OF REFERENCES

[ACD+01] D. Alessandri, C. Cachin, M. Dacier, O. Deak, K. Julisch, B. Randell, J. Riordan, A. Tscharner, A. Wespi, and C. Wüest. Malicious- and Accidental-Fault Tolerance for Internet Applications: Towards a Taxonomy of Intrusion Detection Systems and Attacks. Technical Report RZ 3366, IBM Research, September 2001.

[ACF+00] Julia Allen, Alan Christie, William Fithen, John McHugh, Jed Pickel, and Ed Stoner. State of the Practice of Intrusion Detection Technologies. Technical Report CMU/SEI-99-TR-028, ESC-99-028, Networked Systems Survivability Program, Carnegie Mellon Software Engineering Institute, Pittsburgh, PA, January 2000.

[Age99a] National Security Agency. Controlled Access Protection Profile, 8 October 1999. Version 1.d.

[Age99b] National Security Agency. Labeled Security Protection Profile, 8 October 1999. Version 1.b.

[And80] James P. Anderson. Computer Security Threat Monitoring and Surveillance. Technical report, James P. Anderson Co., Fort Washington, PA, April 1980.

[Arc00] Iván Arce. UNIX Locale Format String Vulnerability. Posting to BugTraq mailing list `http://www.securityfocus.com/archive/1/80154`, September 2000.

[Asa01] Paul Asadoorian. Shell Script Searching for Code Red and Nimda. Posting to incidents.org mailing list, September 2001. `http://www.incidents.org/archives/intrusions/msg01035.html`.

[AUS96] AUSCERT. Vulnerability in NCSA/Apache CGI example code. AUSCERT Advisory AA-96.01, March 1996. CVE-1999-0067.

[Axe98] Stefan Axelsson. Research in Intrusion-Detection Systems: A Survey. Technical Report 98-17, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, December 1998. Revised 19 August 1999.

[Axe00] Stefan Axelsson. Intrusion Detection Systems: A Survey and Taxonomy. Technical Report 99-15, Department of Computer Engineering, Chalmers University of Technology, Göteborg, Sweden, March 2000.

[Bac00] Rebecca Gurley Bace. *Intrusion Detection*. Technology Series. Macmillan Technical Publishing, 2000.

[Bac02] Rebecca Bace. Personal communication via email, April 2002.

[Bau02] Kirk Bauer. LogWatch. Web page at `http://www.logwatch.org/`, 2002.

[BD96] Matt Bishop and Michelle Dilger. Checking for Race Conditions in File Accesses. *Computing Systems*, 9(2):131–152, Spring 1996.

[BL73] D. E. Bell and L. J. LaPadula. Secure Computer Systems: Mathematical Foundations and Model. Technical Report M74-244, The MITRE Corp., Bedford, MA, May 1973.

[BL74] D. Bell and L. LaPadula. Secure Computer Systems: A Refinement of the Mathematical Model. Technical Report MTR-2547, Vol 3, MITRE Corp., Bedford, MA, April 1974.

[Bou00] Pascal Bouchareine. More Info on Format Bugs. Webpage at `http://julianor.tripod.com/kalou-formats.txt`, October 2000.

[bs799a] Information Security Management – Part 1: Code of Practice for Information Security Management. British Standard BS 7799-1:1999, British Standards Institute, BSI, 289 Chiswick High Road, London, W4 4AL, May 1999.

[bs799b] Information Security Management – Part 2: Specification for Information Security Management Systems. British Standard BS 7799-2:1999, British Standards Institute, BSI, 289 Chiswick High Road, London, W4 4AL, May 1999.

[BTS99] Arash Baratloo, Timothy K. Tsai, and Navjot Singh. Libsafe: Protecting Critical Elements of Stacks. Technical report, Bell Labs, Lucent Technologies, Murray Hill, NJ 07974 USA, December 1999.

[Car01a] Brian Carrier. A Recursive TCP Session Token Protocol for Use in Computer Forensics and Traceback. Master's thesis, Purdue University, West Lafayette, IN, 47907-1315, USA, May 2001. CERIAS TR 2001-19; Masters Thesis.

[Car01b] Tom Carroll. code_red_check.sh. Posting to DShield mailing list, August 2001. `http://www.dshield.org/pipermail/list/2001-August/001194.php`.

[CBD+99] Crispin Cowan, Steve Beattie, Ryan Finnin Day, Calton Pu, Perry Wagle, and Erik Walthinsen. Protecting Systems from Stack Smashing Attacks with StackGuard. In *Proceedings of the 5th Linux Expo*, Raleigh, NC, May 1999.

[CCI99] Common Criteria: An Introduction. Booklet distributed at NISSC 1999, 1999. Booklet produced by Syntegra on behalf of the Common Criteria Project Sponsoring Organisations.

[CER96] CERT/CC. Denial-of-Service Attack via ping. CERT advisory at `http://www.cert.org/advisories/CA-1996-26.html`, December 1996. CVE-1999-0128.

[CER97] CERT/CC. Land IP Denial of Service. CERT advisory at `http://www.cert.org/advisories/CA-1997-28.html`, December 1997. CVE-1999-0016.

[CFPS99] Philip K. Chan, Wei Fan, Andreas Prodromidis, and Salvatore J. Stolfo. Distributed Data Mining in Credit Card Fraud Detection. *IEEE Intelligent Systems' Special Issue on Data Mining*, 14(6):67–74, November/December 1999.

[CJN+95] Gary G. Christoph, Kathleen A. Jackson, Michael C. Neuman, Christine L. B. Siciliano, Dennis D. Simmonds, Cathy A. Stallings, and Joseph L. Thompson. UNICORN: Misuse Detection for UNICOS$^{TM}$. In Sidney Karin, editor, *Proceedings of the 1995 ACM/IEEE Supercomputing Conference, December 3–8, 1995, San Diego Convention Center, San Diego, CA, USA*, New York, NY 10036, USA and 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 1995. ACM Press and IEEE Computer Society Press.

[CK01] Mark J. Crosbie and Benjamin A. Kuperman. A Building Block Approach to Intrusion Detection. In *4th International Symposium on Recent Advances in Intrusion Detection*, Davis, CA, October 2001.

[Com] Common Criteria for Information Technology Security Evaluation. Website at `http://www.commoncriteria.org/`.

[CPM+98] Crispin Cowan, Calton Pu, Dave Maier, Heather Hinton, Peat Bakke, Steve Beattie, Aaron Grier, Perry Wagle, and Qian Zhang. StackGuard: Automatic Adaptive Detection and Prevention of Buffer-Overflow Attacks. In *Proceedings of the 7th USENIX Security Conference*, San Antonio, TX, January 1998. USENIX.

[CS03] Brian Carrier and Eugene H. Spafford. Getting Physical With The Digital Investigation Process. *International Journal of Digital Evidence*, 2(2), Fall 2003.

[CS04] Brian D. Carrier and Eugene H. Spafford. Defining Event Reconstruction of Digital Crime Scenes. *Journal of Forensic Sciences*, 2004.

[Cuf03a] Andy Cuff. Intrusion Detection Terminology (Part One). Available online at `http://www.securityfocus.com/infocus/1728`, September 2003.

[Cuf03b] Andy Cuff. Intrusion Detection Terminology (Part Two). Available online at `http://www.securityfocus.com/infocus/1733`, September 2003.

[Cur94] Timothy W. Curry. Profiling and Tracing Dynamic Library Usage Via Interposition. In *USENIX Summer 1994 Technical Conference*, pages 267–278, Boston, MA, Summer 1994.

[CW87] D.D. Clark and D.R. Wilson. A Comparison of Commercial and Military Computer Security Policies. In *1987 IEEE Symposium on Security and Privacy*, pages 184–194. IEEE Computer Society Press, 1987.

[CW88] D.D. Clark and D.R. Wilson. Evolution of a Model for Computer Integrity. In *Proceedings of the 11th National Computer Security Conference*, Baltimore, MD, October 1988.

[DDW99] Hervé Debar, Marc Dacier, and Andreas Wespi. Towards a Taxonomy of Intrusion-Detection Systems. *Computer Networks*, 31(8):805–822, 1999.

[DDW00] Hervé Debar, Marc Dacier, and Andreas Wespi. A Revised Taxonomy of Intrusion-Detection Systems. *Annales des Télécommunications*, 55(7-8):83–100, 2000.

[Den87] Dorothy E. Denning. An Intrusion-Detection Model. *IEEE Transactions on Software Engineering*, SE-13(2):222–232, February 1987.

[Der03] Renaud Deraison. Nessus. Available online from `http://www.nessus.org/`, 2003.

[Dig94] Digital Equipment Corporation. POLYCENTER Security Intrusion Detector for SunOS, Version 1.0. Online description at `http://www.geek-girl.com/ids/0015.html`, August 1994.

[Dit01] Dave Dittrich. Basic Steps in Forensic Analysis of Unix Systems. Webpage at `http://staff.washington.edu/dittrich/misc/forensics/`, December 2001.

[DN85] Dorothy E. Denning and Peter G. Neumann. Requirements and Model for IDES – A Real-Time Intrusion Detection Expert System. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, 1985.

[DW01] Hervé Debar and Andreas Wespi. Aggregation and Correlation of Intrusion-Detection Alerts. In Lee et al. [LMW01], pages 85–103. Online program at `http://www.raid-symposium.org/raid2001/`.

[Fed01] Federal Bureau of Investigation. Carnivore Diagnostic Tool. Web page at `http://www.fbi.gov/hq/lab/carnivore/carnivore.htm`, March 2001.

[FHS97] Stephanie Forrest, Steven A. Hofmeyr, and Anil Somayaji. Computer Immunology. *Communications of the ACM*, 40(10):88–96, October 1997.

[FV99] Dan Farmer and Wietse Venema. Murder on the Internet Express. Free class on computer forensic analysis, IBM T.J. Watson Research Center, Yorktown Heights, NY, August 1999. Slides available at `http://www.porcupine.org/forensics/handouts.html`.

[Gui01] Guidance Software. EnCase. Online description at `http://www.guidancesoftware.com/html/forensic_software.html`, December 2001.

[GWTB96] Ian Goldberg, David Wagner, Randi Thomas, and Eric Brewer. A Secure Environment for Untrusted Helper Applications (Confining the Wily Hacker). In *Sixth USENIX Security Symposium, Focusing on Applications of Cryptography*, San Jose, California, July 1996.

[HB95] Lawrence R. Halme and R. Kenneth Bauer. AINT Misbehaving: A Taxonomy of Anti-Intrusion Techniques. In *Proceeds of the 18th National Information Systems Security Conference*, pages 163–172, October 1995.

[HB00] David Harley and Bruce Burrell. alt.comp.virus (Frequently Asked Questions). Posted periodically to alt.comp.virus, April 2000.

[Hew00] Hewlett-Packard, 3000 Hanover Street, Palo Alto, CA. *HP Praesidium Intrusion Detection System/9000*, 1 edition, July 2000.

[Hew01] Hewlett-Packard. Intrusion Detection System 9000 (IDS/9000). Web page at `http://www.hp.com/security/products/ids/`, June 2001.

[HLF+01] Joshua W. Haines, Richard P. Lippman, David J. Fried, Eushiuan Tran, Steve Boswell, and Marc A. Zissman. 1999 DARPA Intrusion Detection System Evaluation: Design and Procedures. Technical Report TR-1062, MIT Lincoln Laboratory, Lexington, MA 02420, USA, February 2001.

[Int01] Internet Security Systems. RealSecure. Web page at `http://www.iss.net/securing_e-business/security_products/intrusion_detection/`, June 2001.

[Iva98] Ivan Krsul. *Software Vulnerability Analysis*. PhD thesis, Department of Computer Sciences, Purdue University, 1998.

[JD02] Klaus Julisch and Marc Dacier. Mining Intrusion Detection Alarms for Actionable Knowledge. In *Proceedings of the eighth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 366–375. ACM Press, 2002.

[JHP00] Anil Jain, Lin Hong, and Sharath Pankanti. Biometric Identification. *Communications of the ACM*, 43(2):90–98, 2000.

[Jon93] Michael B. Jones. Interposition Agents: Transparently Interposing User Code at the System Interface. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles*, Asheville, NC, December 1993.

[KS94] Gene H. Kim and Eugene H. Spafford. The Design and Implementation of Tripwire: A File System Integrity Checker. In Jacques Stern, editor, *2nd ACM Conference on Computer and Communications Security*, pages 18–29, COAST, Purdue, November 1994. ACM Press.

[KS98] Benjamin A. Kuperman and Eugene H. Spafford. Generation of Application Level Audit Data via Library Interposition. Technical Report COAST TR 98-17, 1998.

[KS99a] Benjamin A. Kuperman and Eugene Spafford. Generation of Application Level Data via Library Interposition. Technical Report CERIAS TR 1999-11, COAST Laboratory, West Lafayette, IN, October 1999.

[KS99b] Benjamin A. Kuperman and Eugene H. Spafford. Generation of Application Level Data via Library Interposition. Technical report, COAST Laboratory, 1999. CERIAS TR 1999-11.

[KSZ02] Florian Kerschbaum, Eugene H. Spafford, and Diego Zamboni. Embedded Sensors and Detectors for Intrusion Detection. To appear in the Journal of Computer Security, 2002. Project web page at `http://www.cerias.purdue.edu/homes/esp/`.

[Kum95] Sandeep Kumar. *A Pattern Matching Approach to Misuse Intrusion Detection*. PhD thesis, Purdue University, Department of Computer Sciences, 1995.

[Kup04] Benjamin A. Kuperman. Audlib: Special Purpose Audit Sources for Computer Security Monitoring. Technical Report CERIAS TR 2004-25, CERIAS, 2004.

[Kva99] Håkan Kvarnström. A Survey of Commercial Tools for Intrusion Detection. Technical Report 99-8, Department of Computer Engineering, Chalmers University of Technology, Götenborg, Sweden, October 1999.

[LB97] Terran Lane and Carla Brodley. Sequence Matching and Learning in Anomaly Detection for Computer Security. In *Proceedings of the AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997.

[LB99] Terran Lane and Carla E. Brodley. Temporal Sequence Learning and Data Reduction for Anomaly Detection. *ACM Transactions on Information and System Security (TISSEC)*, 2(3):295–331, 1999.

[LFG+00] Richard P. Lippmann, David J. Fried, Isaac Graf, Joshua W. Haines, Kristopher R. Kendall, David McClung, Dan Weber, Seth E. Webster, Dan Wyschogrod, Robert K. Cunningham, and Marc A. Zissman. Evaluating Intrusion Detection Systems: The 1998 DARPA Off-Line Intrusion Detection Evaluation. In *Proceedings of the 2000 DARPA Information Survivability Conference and Exposition*, volume 2, Lexington, MA 02420, USA, 2000. MIT Lincoln Laboratory.

[lin03] The Linux Basic Security Module Project. Website at `http://linuxbsm.sourceforge.net/`, 2003.

[LMW01] Wenke Lee, Ludovic Mé, and Andreas Wespi, editors. *Recent Advances in Intrusion Detection : 4th Internation Symposium*, volume LNCS 2212, Davis, CA, USA, October 2001. Springer. Online program at `http://www.raid-symposium.org/raid2001/`.

[Lou01] Daniel Lowry Lough. *A Taxonomy of Computer Attacks with Applications to Wireless Networks*. PhD thesis, Virginia Polytechnic Institute and State University, May 2001.

[LS98] Wenke Lee and Salvatore J. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, Texas, January 1998.

[McH00] 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 (TISSEC)*, 3(4), 2000.

[Mes03] Ellen Messmer. Security Experts: Insider Threats Loom Largest. *Network World*, December 2003.

[Mil91] David L. Mills. Internet Time Synchronization: the Network Time Protocol. *IEEE Transactions on Communications*, 39(10):1482–1493, October 1991.

[Nak01] Greg Nakhimovsky. Building Library Interposers for Fun and Profit. *Unix Insider*, July 2001. Available online at `http://developers.sun.com/solaris/articles/lib_interposers.html`.

[Nat03] National Institute of Standards and Technology. Common Criteria for IT Security Evaluation: Common Language to Express Common Needs . Website at `http://csrc.nist.gov/cc/`, 2003.

[New00] Tim Newsham. Format String Attacks. Whitepaper, Guardent, Inc., September 2000.

[NFR01] NFR Security. *Overview of NFR Network Intrusion Detection*, June 2001. Manual available online `http://www.nfr.com/products/NID/docs/NID_Technical_Overview.pdf`.

[NP89] Peter G. Neumann and Donn B. Parker. A Summary of Computer Misuse Techniques. In *Proceedings of the 12th National Computer Security Conference*, pages 396–407, Baltimore, Maryland, October 1989.

[PN98] Thomas H. Ptacek and Timothy N. Newsham. Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection. Technical report, Secure Networks, Inc., Suite 330, 1201 5th Street S.W, Calgary, Alberta, Canada, T2R-0Y6, January 1998.

[Pri97] Katherine M. Price. Host-Based Misuse Detection and Conventional Operating Systems' Audit Data Collection. Master's thesis, Purdue University, December 1997.

[Psi02] Psionic Technologies. LogCheck. Web page at `http://www.psionic.com/`, 2002.

[Ric03] Robert Richardson. CSI/FBI Computer Crime and Security Survey. Technical report, Computer Security Institute, 2003.

[Roe99] Martin Roesch. Snort: Lightweight Intrusion Detection for Networks. In *Proceedings of the LISA'99 conference*. USENIX, November 1999.

[sal03] Secure Auditing for Linux. Website at `http://secureaudit.sourceforge.net/`, 2003.

[scs03] SCSLog. Website at `http://www.suse.de/~thomas/tools/scslog/`, 2003.

[Scu01] Scut, Team Teso. `http://teso.scene.at/articles/formatstring`, September 2001.

[Sma88] Stephen E. Smaha. Haystack: An Intrusion Detection System. In *Fourth Aerospace Computer Security Applications Conference*, pages 37–44, Tracor Applied Science Inc., Austin, TX, December 1988.

[sna03] SNARE for Linux. Website at `http://www.intersectalliance.com/projects/Snare/`, 2003.

[SSH02] SSH Communications Security. SSH Secure Shell, 2002. `http://www.ssh.com/products/security/`.

[SSHW88] M. Sebring, E. Shellhouse, M. Hanna, and R. Whitehurst. Expert Systems in Intrusion Detection: A Case Study. In *Proceedings of the 11th National Computer Security Conference*, October 1988.

[Sto91] Clifford Stoll. *The Cuckoo's Egg*. Pan Books Ltd., 1991.

[Sun94] Sun Microsystems. *Linker and Libraries (Solaris 2.4 Software Developer AnswerBook)*, 1994. SunOS 5.5.1.

[Sun96] Sun Microsystems, Inc., Mountain View, CA. *Shared Library Interposer (SLI) User's Guide*, SLI version 1.2 edition, January 1996.

[Sun03] Sun Microsystems, Inc. *System Administration Guide: Security Services*. Sun Microsystems, Inc., 2003.

[SY91] Ben Smith and Tom Yager. BYTE Unix Benchmarks, Version 3.6. Available online at `http://www.silkroad.com/public/bench.tar.gz`, May 1991.

[The02] The OpenBSD Project. OpenSSH. Project webpage online at `http://www.openssh.org/`, 2002.

[TS01] Timothy Tsai and Navjot Singh. Libsafe 2.0: Detection of Format String Vulnerability Exploits. Technical Report ALR-2001-019, Avaya Labs, Avaya Inc., Basking Ridge, NJ 07920, August 2001.

[US 85] US Department of Defense. Trusted Computer Systems Evaluation Criteria. Technical Report DoD 5200.28-STD, DoD Computer Security Center, Fort Meade, MD, December 1985. Also known as the 'Orange Book'.

[US 88] US Department of Defense. A Guide to Understanding Audit In Trusted Systems. Technical Report NCSC-TG-001, Version 2, National Computer Security Center, Fort George G. Meade, Maryland 20755-6000, June 1988. Also known as the 'Tan Book'.

[VS01] Alfonso Valdes and Keith Skinner. Probabilistic Alert Correlation. In Lee et al. [LMW01], pages 54–68. Online program at `http://www.raid-symposium.org/raid2001/`.

[Wei84] Reinhold P. Weicker. Dhrystone: A Synthetic Systems Programming Benchmark. *Communications of the ACM*, 27(10):1013–1030, 1984.

[Wei88] Reinhold P. Weicker. Dhrystone Benchmark: Rationale for Version 2 and Measurement Rules. *ACM SIGPLAN Notices*, 23(8):49–62, 1988.

[Wei02] Alan R. Weiss. Dhrystone Benchmark: History, Analysis, S̈coresänd Recommendations. White paper, ECL, LLC, Austin, TX, October 2002.

[Zam01] Diego Zamboni. *Using Internal Sensors for Computer Intrusion Detection*. PhD thesis, Department of Computer Sciences, Purdue University, August 2001.

VITA

# VITA

Benjamin Asher Kuperman was born December 2, 1974 in Beavercreek, Ohio. He graduated from Beavercreek High school in 1993 and went on to matriculate at The University of Toledo in Toledo, Ohio. In 1997 he graduated magna cum laude with a BSE in Computer Science Engineering and a BS in Mathematics, both with departmental and college honors. In 1999, he completed his MS in Computer Sciences at Purdue University in West Lafayette, Indiana. He earned his PhD in Computer Sciences from Purdue University in 2004. He will be joining the Computer Science Department at Swarthmore College as a Visiting Assistant Professor.