Paradigms for the Reduction of Audit Trails

by

Bradford Rice Wetmore
B.A. (Humboldt State University) 1989

THESIS

Submitted in partial satisfaction of the
requirements for the degree of

MASTER OF SCIENCE

in

Computer Science

in the

GRADUATE DIVISION

of the

UNIVERSITY OF CALIFORNIA
DAVIS

Approved:

_____

_____

_____

Committee in Charge

1993

Abstract

Most automated packages for intrusion detection focus on determining if a collection of audit data is suspicious. Package developers assume that the System Security Officer (SSO) will combine the results of their tools with a careful inspection of the logs to determine if indeed there is evidence of intrusive activity. In practice, most administrators rely exclusively on the conclusions generated by such packages. As a result, very few methods have been developed to browse the raw audit trails. This thesis presents a new approach to this problem.

By treating conceptual entities in an audit trail as objects, a framework for observing how entities interact can be developed. All of the records of interest are first scanned to determine the objects and actions of interest. During this initial scanning phase, the objects are interconnected based on how each affects the other, much like a directed graph. The vertices and edges represent the objects and actions respectively. Then, by focusing initially on one object of interest, a SSO can quickly determine how that object affected or was affected by any other object by noting the direction and type of edge connecting the nodes. Say, for example, a process with limited privilege was able to create a new process with unlimited privileges by executing one action. The two

processes are be represented by the vertices, and the action of gaining privilege could be represented by a directed edge from the first process to the second. Thus by focusing on these new objects, the SSO can then determine how other nodes were directly or indirectly affected by the first object simply by following the next set of edges.

An initial prototype program was produced and focused on the UNIX operating system model, and was fairly successful in following entities in the audit trail. Later efforts tried to extrapolate the model to more general computational systems.

Of course, the SSO must still possess technical knowledge of any system to fully analyze the data and realize the implications of the actions therein: there is no substitute for such expertise. This thesis presents a new methodology for browsing such data.

Table of Contents

1 Introduction 1

2 Background 5

2.1 Steps in Audit Trail Analysis 6

2.2 Purposes of Auditing 8

2.3 Auditing for Intrusions 10

2.4 Audit Trail Formats 13

2.5 Methods of Data Collection 15

2.6 Determining What to Audit 17

2.7 Examples of Sun BSM Audit Data 19

2.8 Problems with Audit Analysis 23

3 Previous Work 27

3.1 Statistical Methods 27

3.1.1 Haystack 29

3.1.2 IDES 30

3.2 Rule Based Expert Systems 31

3.2.1 Signature Analysis 32

3.2.2 MIDAS 33

3.2.3 Wisdom & Sense 35

3.2.4 ComputerWatch Audit Trail Analysis Tool 36

3.2.5 Information Security Officer's Assistant (ISOA) 36

3.3 Machine Learning 37
3.4 Other Systems 38
3.5 Audit Trail Browsing 38

4 The Audit Workbench 41
4.1 Architecture 41
4.2 Research Goal 44
4.3 Rationale 45
4.3.1 Utilizing Multiple Views 45
4.3.2 Combining the Stronger Pieces of Different Auditing Systems 46
4.3.3 Portability of Analysis Algorithms . . 46
4.3.4 Testbed for Intrusion Detection Systems 47
4.3.5 Extensible Audit Trails 48
4.3.6 Experimenting With Hard-to-Duplicate Conditions 48
4.3.7 Ease of Data Collection 48
4.4 Development and Testing 49

5 Towards More Effective Audit Browsing 52
5.1 Questions to Ask of Audit Trails 52
5.2 Object-Based Analysis 54
5.3 Applicability of Relational Databases to Audit Browsing 58
5.4 Model #1, A UNIX Specific Model 59

5.4.1 A Worked Example 65
5.4.2 Advantages of this Model 67
5.4.3 Disadvantages of this Model 69
5.5 Model #2, A More General Model 70
5.5.1 Advantages of this Model 75
5.5.2 Disadvantages of this Model 76
5.6 Answering the Questions 77
5.7 The Downfall of Slicing 80
5.8 Experiences with BSM 84

6 Future Work 90

7 Conclusions 92

8 Acknowledgements 94

9 References 95

10 Appendix 1 – Output of "ab" 99

1 Introduction

Ever since man began to engage in trade, he has looked for ways to keep records of his transactions. With the advent of modern computing machinery, there has been an increase in the number of transactions by several orders of magnitude because of the relative ease in which those transactions can be conducted. Due to computing's high cost, accounting mechanisms were placed on systems to track resources used. Users could be held accountable for actions, and billed accordingly. As the data being processed became more sensitive and expensive, the need for security became critical. Unfortunately, many early systems tried to apply techniques from the accounting analysis domain into security. Experience shows that many of these techniques are not easily adaptable [Picc87].

Computer security issues have become extremely important in many organizations. Virtually every day we witness news stories of an invaded government machine, a person who was harassed because of certain information maintained in a computer, or in which some company's accounting data was destroyed by a malicious program. With our increasing dependence on computers, the inter-networking of systems, and the increasing technological base of expertise, the need for security has never been greater.

Computer security can take forms from preventive to post-mortem. Access control mechanisms, such as passwords and file permissions, restrict users from making illegal accesses to system resources, and are routinely placed on systems. However, as history shows, every major system has flaws. Even a supposedly secure system can be invaded by an unauthorized person masquerading as a legitimate user, or by a legitimate user abusing his privileges. In addition, one cannot ignore programmatic flaws in the implementation of the system.

Accepting the premise that systems are not necessarily secure, we immediately see the need for documenting user actions. Auditing systems are one of the many tools available for post-mortem analysis. There are many problems with current models of auditing, largely due to the huge number of records recorded. (Other problems will be highlighted in later chapters.) Analysis of audit data has traditionally been done by hand; the work is tedious and highly error prone. The auditor must be highly trained in security issues, and know where to look. For most situations, a thorough analysis is impossible. Several automated audit analysis systems have been prototyped, but many administrators feel that the "best way" to analyze audit data has yet to emerge. Today's approaches still require extensive human interaction.

Most modern Intrusion Detection Systems (IDS) use audit data as the primary source of information. IDSs are based on two premises: users tend to repeat their behavior over time, and intruders have distinct methods of operation, generally using distinct sequences of events in order to penetrate a machine. Most IDSs research has been focused either on defining normal use (anomaly detection), or on developing rules that indicate intrusive behavior (misuse detection). Methodologies for examining audit data have been underdeveloped in lieu of developing IDSs.

In this thesis, I begin by presenting an overview of auditing for security purposes: theory, definitions, and some current issues. Next I discuss some of the previous work in automated audit systems, followed by an introduction to a project underway at the University of California at Davis called the Audit Workbench. This section is followed by some research into models for a generalized audit trail browser, which may eventually be incorporated into the Audit Workbench. Finally we explore the use of a debugging technique called slicing which naturally lends itself to audit trail browsing.

Although the debugging technique didn't work well in practice, the other results of this research were very encouraging. Following the construction of a prototype audit browser, I was able to trace through a series of simulated attacks with little difficulty. This work led to

a more general idea for a browser, which is presented as a proposal. After these initial experiences, the construction of an audit browser would be very straightforward, and could make the SSO's job of browsing audit trails much easier.

2 Background

[Webs86] defines an audit as a formal examination of an organization's or individual's accounts or financial situation. Auditing is currently used as a tool for several purposes as listed in [Bish89]:

> to restore file systems and databases to known states after crashes,
>
> for security purposes, such as documenting intrusions, assessing damage or information disclosed,
>
> for electronic funds transfer systems used in banking,
>
> and for other types of data processing.

In this research we focus primarily on auditing for security purposes. We can modify Webster's definition of audit to be "a formal examination of records to analyze and document action within a computer system."

In the 1980's, the United States Department of Defense published a series of computer security guidelines known as the "Rainbow" series. In particular, [NCSC85] (or the "Orange Book"), and [NCSC88] (or the "Tan Book") describe a set of guidelines to which computer systems must adhere to in order to be rated at a certain level of security. In sections relating to accountability, the specifications are clear. Computer systems must document all security related incidents, and keep those records available for later

review. The product of such record-keeping is called an "audit trail," and is "a set of records that collectively provide documentary evidence of processing used to aid in tracing from original transactions forward to related records and reports, and/or backwards from records and reports to their component source transactions" [NCSC85, NCSC88].

Most auditing systems follow the model of subjects, objects, and actions, as highlighted in [Denn86]. Subjects are generally the users of a system (but could be the systems themselves), those entities which initiate actions upon one or more objects. Objects are entities such as files, processes, or devices, and are managed by the system. Actions are the events which change the model state between the subjects and objects. Examples include file actions, process controls, or machine parameter modifications.

2.1 Steps in Audit Trail Analysis

When we refer to the entire process of auditing, several distinct events are required:

1) determine which types of audit data to collect,
2) collect that information,
3) reduce the information to a manageable size by eliminating useless information,
4) analyze the reduced data,

5) notify the proper authorities of the results, and

6) take appropriate actions (see Figure 1).

[Bish89] makes the distinction between logging and auditing. Logging is the process of creating the records, while auditing is the actual review. In the above list of events, logging consists of steps 1 and 2, while the remaining steps make up the auditing stage. I will follow his suggestions, and also use the term "auditing systems" for systems that perform the combined action.

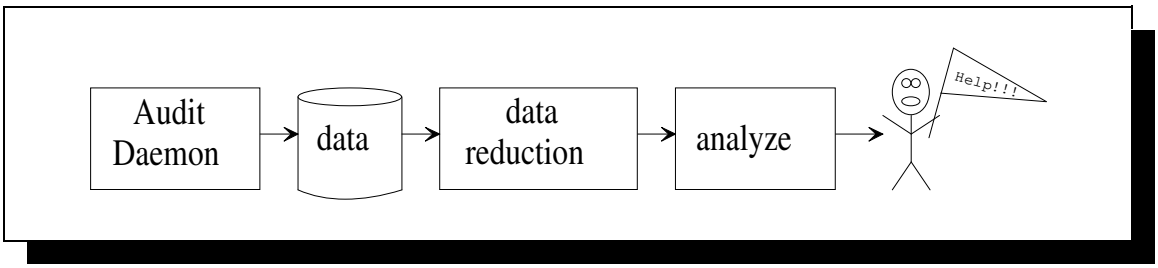

Figure 1 Steps in Audit Analysis.

Most current research has focused on steps 4 and 5. Most IDSs attempt to automatically reduce the audit data which the SSO must inspect. They scan for "interesting" events, patterns of events, and behavior out of the ordinary, and then they summarize the findings.

We also should make the distinction between "passive" and "active" auditing [Bony81]. Passive auditing assumes that the logs will be available for inspection, but does not assume that any such analysis will take place unless there is reason. Active auditing differs by attempting to determine if the audit log contains suspicious events as

near in time to the actual occurrence as possible. If such events exist, the audit program or the SSO can take immediate action. Options include modifying the level of auditing on a user or file, removing a user from the system, in extreme cases shutting down the system, etc.

2.2 Purposes of Auditing

According to [NCSC88], the purposes of the audit mechanism are fivefold, viz. they must:

1) allow the review of patterns of accesses to objects, provide on demand the access histories of specific processes and individuals, and allow the use of the various protection mechanisms supported by the system [Glig85],
2) allow discovery of both user's and outsider's repeated attempts to bypass the protection mechanisms,
3) allow discovery of use of privileges that may occur when a user assumes a functionality with privileges greater than his own,
4) act as a deterrent against perpetrators' habitual attempts to bypass the system protection mechanisms, and

5) supply an additional form of user assurance that attempts to bypass the protection mechanisms are recorded and discovered [Glig85].

Unfortunately, a large gap has developed between the original design goals and the resulting implementations.

The first goal has been the most underdeveloped. The auditing systems do allow for logging the activities; however, better tools for analyzing the data need to be developed. Security personnel have continually voiced the need for audit browsers, tools which provide easy observation of all events and their interconnections. Logging mechanisms generate single records as events occur, with no attempt made to link related records.

System vendors such as Sun Microsystems have stated their job is only to provide the mechanisms for producing audit trails [Wetm92a]. (However, new initiatives taken by Sun may reverse this trend. Increasingly, management has begun to see the advantages to providing additional tools to help in the system administrator's role.) But for now, further analysis remains the responsibility of the customer. Research institutions seem primarily interested in developing the very general models of security. Unfortunately, most development efforts have focused on other areas such as intrusion and anomaly detection, and very little on methodologies for inspecting auditing trails.

Until better mechanisms for browsing and intrusion detection are developed, the fourth and fifth goals will not be accomplished. Most sites do not examine their logs, and of those who do, many only scan for the most suspicious users based on a specified criterion. If malicious users know what sequences of actions will generate warnings, they can alter their mode of operation and still obtain their desired goal.

An additional question to consider is the interrelationship between the auditing communities and those they seek to hinder. Hackers will not heed the deterrent of an auditing system, given the combined lack of audit inspections and applicable legislation. This is the so-called social gap between social policies and actual human behavior [Neum88]. Prosecution is expensive, and usually undertaken only in extreme circumstances. In addition, previous laws were not conceived to cover the types of computing currently in use. Until stricter and more precise laws are enacted regarding computer usage, and unless there is a belief on the existence of an effective auditing mechanism and a significant risk of getting caught, intruders will continue to operate in relative freedom.

2.3 Auditing for Intrusions

The first published work on intrusion detection tried to establish different classes of threats, and how they might be detected using audit data [Ande80]. Table 1 summarizes this work:

	Penetrator Not Authorized to Use Data/Program Resource	Penetrator Authorized to Use Data/Program Resource
Penetrator Not Authorized Use of Computer	Case A: External Penetration	none
Penetrator Authorized Use of Computer	Case B: Internal Penetration	Case C: Misfeasance

Table 1 Defining Intrusions.

External penetrators are those users not authorized to use a computer system. There are many ways by which this type of user can gain access: wiretaps (monitoring a network to learn login/password combinations), trial and error password guessing, or obtaining a valid user's password by various means. Accepting that wiretaps and events outside the computer's realm are not generally auditable, trial and error attacks can be detected by an

abnormal amount of unauthorized activity, in the form of repeated failed login attempts.

Internal penetration is generally more frequent than external penetration, according to [Ande80]. The three types of users in this class are the masquerader, the legitimate user, the clandestine user.

The masquerader is a user who impersonates another user. The problem with detecting a masquerader is that there are no particular features which can provide conclusive evidence of masquerading. Once identified to a system, all actions are performed on behalf of that user. Anderson suggested that the masquerader be detected by developing models of what behavior is "normal" for a user, and compare previous histories with current observations. Of course this requires a definition of what is "normal," something the intrusion detection community has been attempting to define in the ensuing years since Anderson's report.

The legitimate user is one who has access to the computer and the data but abuses his privileges. This makes the "abnormal" use by a legitimate user harder to detect than that of a masquerader. Again Anderson suggests that a comparison of historical patterns be used to look for abnormal use. Anderson stated that in some instances, some misuse simply cannot be caught. A statistical method is probably not feasible.

A clandestine user is one that can change the level of auditing such that he may operate without leaving a trace. The clandestine user is the most difficult to detect. Anderson suggests that in cases where the probability of a clandestine user is high, one should add additional levels of auditing, including monitoring CPU and memory usage, secondary storage and so on. Another suggestion is to look for changes in the Operating System files, perhaps with a static analysis tool such as COPS [Farm91] or Lawrence Livermore's Security Profile Inspector [Bart92]. Such users have been known to modify system files and programs such as "login" to allow use of a system without detection.

2.4 Audit Trail Formats

Audit trails have been implemented in a variety of ways. Two major styles are in use, delineated by the type of information contained in each record. The self-contained style, used in the Sun Microsystems' Basic Security Module (BSM) [SUN91] requires that each record list all the "interesting" properties of all subjects and objects involved. For example, a file access record may contain the file number, the filename, the actual and effective user name, the group name, the time, the attributes of the file, any return codes, and so on. This method is quite useful when browsing the audit trails manually. The more the

amount of data contained in the audit trails, the less is the information the audit analysis package has to remember, thereby meaning that less complex programs are required for audit analysis. However, this method generates a large amount of data, namely because the information contained in repeated references to this file will be repeated in each audit record. Fortunately, standard compression utilities can offset this bulk. [Sibe88] predicts up to a 8 to 1 compression ratio for highly repetitious data. The tradeoff is classic: ease-of-use versus resource space.

The non-self- contained audit records employed by AT&T follows the opposite path [Dowe90]. By recording only the minimal amount of state change information, a smaller audit trail results. In the example above, a file access record might only contain the time, effective user name, an abbreviated filename, and return codes. All other information, such as the actual user name and file attributes must be obtained from other records in the audit trail or from system tables. If the effective user name and the actual user name were different, an auditor would have to look backwards in the audit trail to find the record containing the user name switch. This requires a good audit browser or a lot of patience, two things missing in current analysis systems!

Since UC Davis uses the Sun BSM package for its Distributed Intrusion Detection System [Snap91a], most of the discussion that follows will be Sun-based.

2.5 Methods of Data Collection

My experience is in software based auditing systems only, but it would not be too difficult to develop hardware based packages as well. These systems could record system performance, I/O usage, external devices, and so on. These packages could be logically disconnected from the systems being monitored, so that even if the monitored machine were compromised, the disjoint system would also have to be compromised in order for the audit data to be modified.

Most software audit packages operate upon entry to system calls, although there are other places where auditing may be performed. For example, in 4.2 BSD, when any system call occurs, control is given to a function called syscall(). syscall() is responsible for determining which function should be used to handle the request, and then it builds the interface between the user and system space. If auditing is enabled, an audit record is created. Control is then transferred to the kernel, which then completes the system call. When control returns, syscall() notes the return values if any; then it updates and writes the audit record before exiting [Picc87].

There are two major models for auditing, viz. application level and kernel level, and each has different properties. Application level auditing can drastically reduce the volume of audit data and make it easier to comprehend a user's intentions. (This is because the actual intent of the action is captured in the data, rather than a series of lower level events which make up the action. A SSO is not required to deduce from the increased number of events what the actual intent was.) For example, if a user employed an editor to modify a file, a system call level auditor would report all the temporary and system files accessed, all the supporting processes that were invoked, etc. This extra data only obfuscates the user's intentions [Picc87]. An application level auditor need only report that an editor was used to modify a specific file, and perhaps the actual data that was modified. However, this requires that the application program be trusted to properly report actions, which opens up the possibility of many vulnerabilities not being reported. This also requires that the application program provide its own auditing subsystem, which can drastically increase program complexity and impact performance.

The other place where auditing may occur is at the kernel level. Sometimes it is necessary for auditing to be performed at this level: namely, when system calls don't return (as with reboot()), or in cases where calls cannot be

audited after returning (modifying the audit state). Auditing at this level is perhaps the most trustworthy, as normal user programs should not be allowed to change kernel code.

There are advantages to having auditing done at various levels at the same time, but there are limitations as well. Auditing code must be placed throughout the system: potentially in the kernel, application, and system call interface. Additionally, two or more audit records may be written for a single event. This could make the auditor's job more difficult in that he has to correlate the two event records as being the same event.

2.6 Determining What to Audit

When a user logs into a computer system, an audit state is established for that user. This state is used by the logging system to decide which events to write into the log. Depending on the resources available and the interests of the SSO, this state may be all encompassing or very minimal and can dynamically change as required. ([NCSC88] calls this "pre-selection of auditable events.") Depending on the needs of the SSO and the resources available, the logging system may record successful and/or failed accesses to objects. The SSO can set this state by a series of software switches, generally based on the type of auditing required.

[Picc87] has estimated that a Compartmented Mode Workstation (CMW) can generate up to 135 megabytes/user/day. Obviously, unless significant system resources are available, logging must be performed selectively.

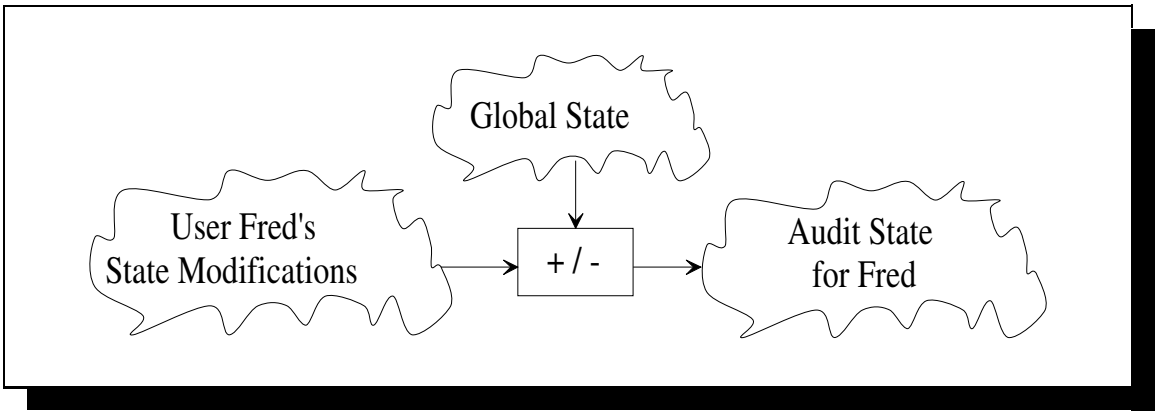

Figure 2 Establishing a User State.

In BSM auditing, a series of steps determine the audit state for a user. In Figure 2, a global default state is set by the SSO, and flags are then modified on a per-user basis to arrive at a user's audit state. This state is then used by the application program. When an event occurs which is auditable, the program then compares the audit state of the user with the type of audit record: if there is a match as in Figure 3, the audit daemon will be instructed to output the record to a file, otherwise the daemon will wait for the next event.

The advantages of performing data reduction at run time is that only records which interest the SSO are kept, and the number of audit records is reduced. However, the SSO may decide at a later time that the records which were not

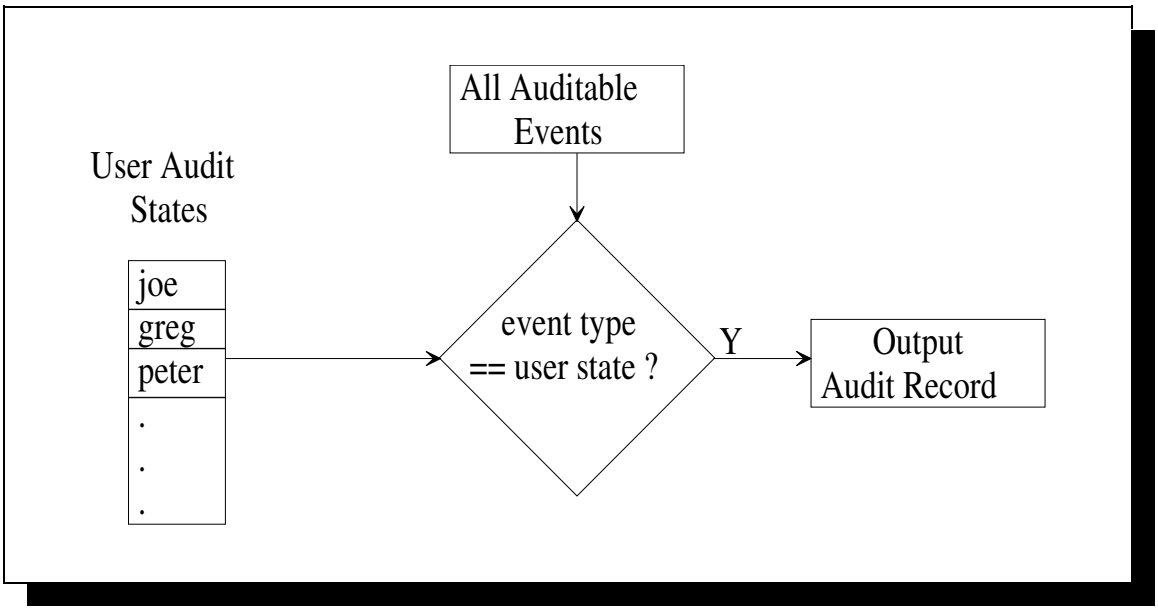

Figure 3 Actions of the Audit Daemon.

collected are now needed. If the records were not generated, that data is irretrievable. The SSO must decide where to balance the tradeoff.

2.7 Examples of Sun BSM Audit Data

As previously mentioned, the Sun BSM audit trails are self-contained. Each record consists of several tokens, either control, data, or modifier, bracketed between a header and a trailer token. Each audit record is the result of an action, and as such its type is defined by the action. Each audit record type has a defined series of subtokens. Each subtoken may either be another or series of subtokens, or a series of bytes.

There are a variety of token types. The more common ones are shown below:

header token:	provides information about the event type, date and time,
trailer token:	delineates the audit record,
path token:	contains information about the path used to access the object,
attribute token:	contains attributes of the object accessed, e.g., types and device numbers,
process token:	contains information about the process which accessed the object,
argument token:	contains the arguments to the action, and
return token:	contains any return values which were returned to the calling program.

Sun stores the audit data in a binary format, with items such as user names, token types, and times in their respective integer notation. This conserves space, but it is impossible for a human to interpret. To achieve this purpose, Sun provides a tool called **praudit** to translate the raw data into a human-readable form.

The following will help to illustrate the format of the BSM audit trail, and to demonstrate praudit. In the first example, user frincke was running process 624, and created (forked) a new process 1352.

```
Ex. 1)    header,53,fork(2): process creation,
               Tue Sep 29 15:34:27 1992, + 170000 msec
          argument,0,1352,child PID
          process,frincke,frincke,frincke,staff,624
```

```
return,Error 0,1352
trailer,53
```

As mentioned, the entire audit record consists of several tokens, delineated by a header and a trailer. The fields in each token are as follows:

header token:	size in bytes of the binary audit record, record type, and time and date of access.
argument token:	"0" represents the first argument (additional arguments have increasing values), child process number, text explanation of this argument token,
process token:	audit, real, and effective user names, group name, and process number,
return token:	error number if the action generated one, and return value of the system call, and
trailer token:	record size in the binary audit data.

The two identical numbers in the header and trailer represent how many bytes this records takes in the raw audit trail file. These numbers can be used by an analysis program to skip over the audit record if needed. Each BSM audit record normally range from tens to hundreds of bytes, with tests conducted at UC Davis showing an average of around ninety.

In the next example frincke's new process executed the /usr/bin/cat program:

```
Ex. 2)    header,95,execve(2):,
               Tue Sep 29 15:34:27 1992, + 180000 msec
          path,/,/usr,/usr/bin/cat
          attribute,100755,root,staff,1798,2467,5952
          process,frincke,frincke,frincke,staff,1352
          return,Error 0,0
          trailer,95
```

Because the execution audit format is different from the fork format above, some tokens are replaced by others:

path token:	user's conceptual root directory as set by the chroot(2) system call, user's current directory, and filename, and
attribute token:	file permissions (10755), the owner and group names, and file system numbers (inodes, devices, etc.).

In the third example, /usr/bin/cat (process #1352) read a file called /home/frincke/robin:

```
Ex. 3)    header,101,open(2):read,
               Tue Sep 29 15:34:23 1992, + 420000 msec
          path,/,/home/frincke,/home/frincke/robin
          attribute,100644,frincke,staff,1802,1056,224
          process,frincke,frincke,frincke,staff,1352
          return,Error 0,4
          trailer,101
```

Sun's BSM was actually a port of the auditing code from the Compartmented Mode Workstation (CMW) project, which in turn was adapted from Sun's Multi-Level Security (MLS) project. Therefore, a similar format was also used in each of the respective formats. [Sibe88] lists the design goals that were met using the token format, namely the expendability by third party vendors of the audit trail event and token types as well as the minimal changes to any existing or future audit analysis software. Because the

audit tokens are self-contained, an analysis program can work regardless of the auditing style of the system generating the messages [Sibe88].

2.8 Problems with Audit Analysis

In addition to the problems discussed earlier, the major obstacle in developing effective analysis tools is the copious amounts of data that logging mechanisms generate. Most of the data are uninteresting for normal analysis by the SSO; the problem lies in identifying and eliminating such data and focusing on the suspicious data, akin to the proverbial "Needle in the Haystack" problem. Several tools have been developed based on statistical methods, expert systems, and machine learning, but each method has limitations which seriously hinder its effectiveness.

Analysis and tools related to the interconnection of audit entities is the major focus of this thesis. By developing "dependency threads" within the audit data that logically link the entities, we can have a much clearer picture of the events in a system and how they interact. Not only can we examine users and system resources to identify at what point they become suspicious, but we can also examine the events that lead up to the suspiciousness. Preliminary trials with an entity interconnection examiner have proven very successful.

Another problem area being explored in a project at UC Davis is the identification of actions which violate security policies. When organizations develop security policies, the results are normally abstract English sentences, with no thought given as to how to audit such actions. From Columbia University, we have:

> "Each user is responsible for insuring that his/her use of the computing facility does not interfere with other users or with proper function of the system" [Colu92].

Or from Rice University, we have:

> "It is expected that all users of University computing resources will use the facilities at their disposal in a manner that is ethical, legal, and responsible. Exercise etiquette and common sense when using the academic computing resources" [Rice92].

As a result, it becomes a challenge to determine if actions violate policy. If we could develop and specify a policy language, we could take a big step to determine what kind of audit data is required for effective auditing.

As mentioned before, most logging mechanisms operate at the system call level. A main obstacle to rapid prototyping of audit analysis algorithms seems to be the absence of high-level abstraction. We must bridge the gap between the very low-level, OS specific audit trail data and the fairly high level abstract concepts used in analysis algorithms. By converting system-specific data into appropriate high-

level abstractions that are easy to analyze, new analysis algorithms can be prototyped with less effort.

By employing abstractions, we can tackle another problem that plagues audit systems, viz. portability. Audit trails frequently contain many system-specific details in their data. If such system-specific details are allowed to permeate the analysis, the details usually become inextricably embedded into the analysis algorithms causing the audit analysis algorithms, auditor, and operating system to become inseparable. This tight coupling hinders portability amongst different auditors, and makes comparison and evaluation of audit analysis algorithms developed using different auditors very difficult. Furthermore, an algorithm developed under a specific auditor usually inherits all the strengths and weaknesses of that particular auditor or OS. The contribution from a powerful auditor or the handicap of a weak one can significantly influence the performance or effectiveness of an audit analysis algorithm.

Another problem of audit trails results from the way audit trails were originally designed. In the early days of computing, a system was its own universe, with all of its resources centrally located. With the advent of mass interconnections and shared resources, actions are generally audited on the machine which initiated the request. If we were to ask for a list of modifications on a file, we must poll all machines which have write access to a filesystem.

(Note that this approach is not acceptable if the initiating machine is untrusted.) With some of the new architectures being proposed, such as massively parallel CPUs and distributed computing bases, new methodologies for distributed auditing must be quickly developed.

One last problem is that the audit trails themselves may become the target of the intruder. According to the Orange Book, it is the responsibility of the Trusted Computing Base to protect the audit trails from intruders. However in some systems, notably the Sun BSM, no provisions are made for protecting the files once the intruder becomes root. (Sun's CMW does attempt to protect the audit trails by separating the roles of the system administration team.) Methodologies for protecting the audit trails will not be discussed in this thesis.

In addition to the issues raised above, there are a variety of others, such as types of data to collect, storage methods, protection, and standards, which are very important and necessary to the auditing process. They are beyond the scope of this thesis.

3 Previous Work

There has been a lot of work done in the area of automated audit analysis, mainly for intrusion detection purposes. The focus of these systems was to make broad judgements about the actions observed, and pass that information to the SSO for further analysis. These tools do not assist in browsing raw audit trails; they only help to determine which data to examine, which also is far from a definitive solution. The three main approaches are statistical, rule-based expert systems, and machine learning.

3.1 Statistical Methods

Automated statistical systems such as SRI's IDES [Denn87, Javi91] and Haystack Laboratory's Haystack [Smah88] focus primarily on defining characteristics of a normal user or group, which generally involves a period of training; then they employ statistical measures to determine if a current user's characteristics match his previously observed behavior. In recent literature, these approaches have been called "anomaly detection," meaning that the systems are attempting to identify patterns of usage which are different from the expected set. Attributes used in statistical

methods include the number of times a user looks at files outside his own directories, the amount of CPU and I/O usage, how many times a user listed a directory, what time a user logged in, etc.

An anomaly reported by such a system is not conclusive evidence of an intrusion, only that the current actions fall outside of normal boundaries. It is the responsibility of the SSO to examine the raw audit records to determine if indeed the events are indicative of an attack.

These systems rely on the premise that there will be statistically significant differences between user behaviors. This idea was first proposed by Denning [Denn86], but has yet to be verified experimentally. (IDES [Javi91] was the best attempt at such a system.) In general computing environments, for this type of anomaly detection it is particularly difficult to define the "normal" user, and what makes a user "suspicious." For this reason, "anomaly detection" seems to be falling out of favor from some members of the intrusion detection community.

For example, independent testing at UC Davis has had difficulty in setting the proper thresholds for Haystack's algorithms in DIDS. And, from individual conversations with Teresa Lunt, the project leader for IDES, IDES has not had a successful record of intrusion detection due to the high number of false conclusions reached [Wetm92b]. Both development efforts have the same problem with defining

exactly what makes a user "suspicious." The ability to accurately discriminate between normal and suspicious behavior highly depends on how widely the user's behavior varies.

There are additional problems with statistical methods, namely:

they can be computationally expensive,

they can generalize too much and lose specifics,

the algorithms have predetermined metrics. This predetermination is usually done in an ad-hoc manner.

However, only anomaly detection can catch masqueradors, unless the perpetrator takes actions which are easily caught.

3.1.1 Haystack

Haystack [Smah88] is a monitoring system originally designed to collect audit trails from a single Unisys (Sperry) 1100/60 mainframe, and perform the analysis on a personal computer.

The data is first transformed into a Canonical Audit Trail, which is an attempt to define a system independent audit trail. This data is then aggregated into user sessions. These sessions are then analyzed to form session vectors, which categorize a session's activity with respect

to the variables measured. The vectors are then multiplied against a weighted multinomial function with a set of ranges for approximately two dozen features. If a vector's values fall outside of 90% of the normally expected values, a warning is generated. After combining these warnings, Haystack attempts to draw conclusions about this session, such as whether this person is a browser, leaking data, malicious, mobile, paranoid, or a possible security penetrator.

Haystack Laboratories, Inc. has been actively expanding its number of supported platforms. Haystack now runs on a Sun Sparcstation running BSM, and there was an effort to port it to the VAX VMS operating system. Haystack's algorithms were also recently modified from strict session boundaries to handle a real-time monitoring capability, and were incorporated into the UC Davis Distributed Intrusion Detection System (DIDS) [Snap91a].

3.1.2 IDES

IDES grew out of work performed earlier by Sytek International [Syte85], and primarily from a model proposed by Dorothy Denning [Denn86]. The main goal of IDES is to provide a series of tools which could detect many forms of intrusions. IDES does have an expert system component, but its main contribution is the use of complex statistical

computations. It transforms audit records into a N-space user profile, which is compared against previously observed behavior. Like Haystack, if the previous profiles are sufficiently far from current behavior, a warning report is generated.

The original IDES prototype used several types of measures, and unlike Haystack, employed a covariance matrix to account for the interconnectedness of the variables used. IDES has changed much during its development. In the latest report, the covariance matrix was dropped, mainly because of the huge expense in computation [Wetm92c]. In fact, [Wetm92c] found that the statistical methods in IDES were very similar to Haystack's.

3.2 Rule Based Expert Systems

Automated expert systems such as portions of IDES [Lunt89, Garv91], DIDS [Snap91a], Wisdom & Sense (W&S) [Vacc89], and signature analysis [Snap91b] pursue a different approach. Instead of detecting anomalies, these systems attempt "misuse detection" by using a priori rules that are indicative to a human expert of an intrusion. These systems rely on the premise that intruders have distinct methods of operation, and use unique signatures to penetrate systems. Generally, these systems use categorical data (object names, object types, etc.) instead of metric

data (number of times an action was observed) as in statistical systems, because it is relatively easy to write rules for categories of data.

As an example of a rule, one particular vulnerability requires two separate actions: installing a login trojan horse by one user intended for another, followed by the latter's login activity. The expert system scans the resulting audit trails, picking out first the record which corresponds to the installation, and then watching for the user's login. The rules which correspond to such an attack would then be fired, and warnings could be generated. Of course, this warning is only a precursor to further investigation.

The expert system approach has several limitations; mainly, the expert systems are always playing "catch-up," meaning that rules can only be added or updated AFTER new vulnerabilities have been identified, potentially after the damage has already been done. In addition, most systems are hand-crafted, meaning that an expert's knowledge must be extracted, codified, and documented. This process can take a great deal of energy.

3.2.1 Signature Analysis

[Snap91b] explored the idea of applying a concept called signature analysis to intrusion detection. Snapp

proposed a method for utilizing finite state automata to determine whether a specific attack is under way. This method has subsequently been implemented in later versions of Haystack, and is being used in the UC Davis DIDS project.

The method works by using a series of tagged objects. By analyzing accesses to these objects, it advances the state of a finite automaton to represent how far a specific attack has proceeded. When all the requirements for an attack have been observed, a warning can be generated. One advantage of employing this method is that for particularly long signatures, warnings can be generated before the attack has been completed.

For example, suppose user A creates an executable trojan program, and then waits for user B to execute it. Once executed, the program modifies user B's security state (say adds an entry to .rhosts file), allowing A to operate as B. A state machine could be created as in Figure 4. It then becomes a simple matter to traverse the states as the audit actions are observed.

Intuitively, the mechanism holds promise, but currently only simple signatures have been implemented.

3.2.2 MIDAS

MIDAS also attempts to encode a priori rules that define an intrusion [Sebr88]. This system is under

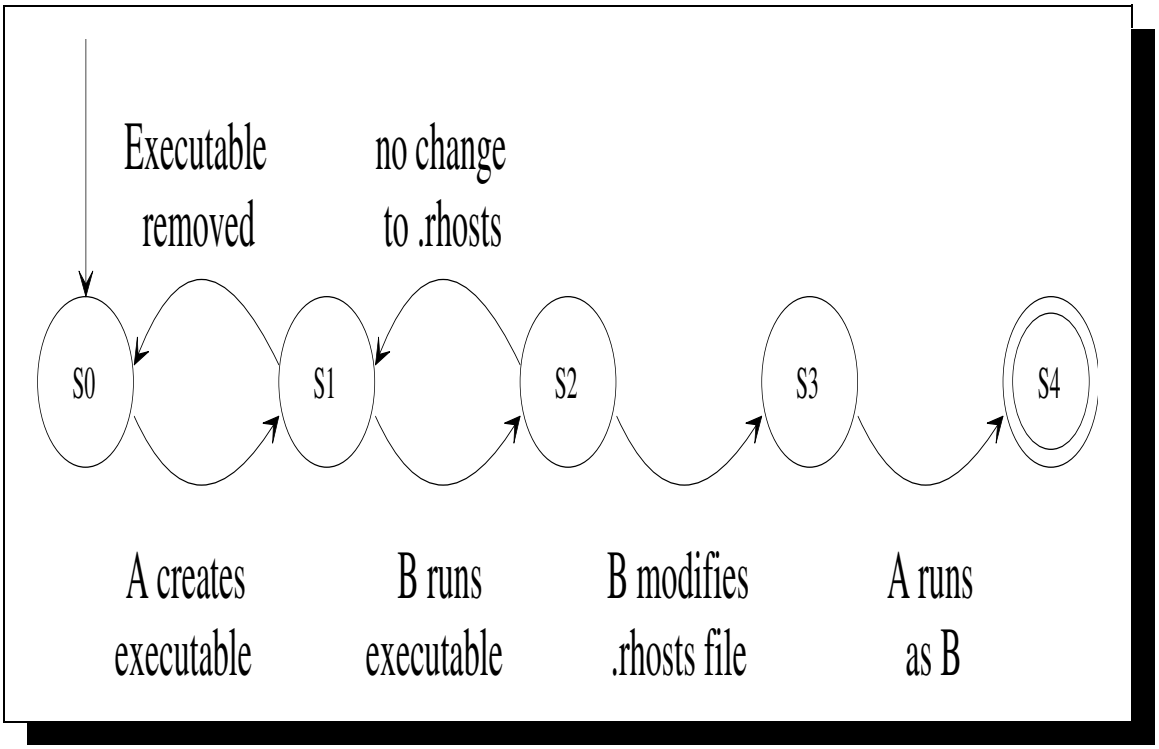

Figure 4 State Machine for Signature Analysis.

development at the National Computer Security Center (NCSC), and is designed to monitor a Multics system. It uses a forward chaining engine and an explanation facility, much like the signature analysis method above. The rules are written in LISP, and can run at 150 inferences per second. At the time the survey paper [Lunt88] was written, approximately 40 rules had been encoded.

MIDAS is also based on Denning's intrusion detection model, and uses four types of rules to detect attacks:

Immediate uses no previous information about past or expected user behavior.

Anomaly	uses statistical user profiles to detect departures from a user's normal profile.
System-wide	maintains system wide parameters to determine what is normal for the system globally.
Sensitive-Path	similar to Snapp's signature analysis, tries to encode the SSO's knowledge to detect attacks in progress.

Specifically, MIDAS attempts to catch the following kinds of attacks: attempted break-ins (by authentication failures, unusual originating hosts, etc.), masqueraders (unusual times, locations, commands, etc.), penetrators (sensitive commands, unauthorized commands, sensitive objects, etc.), misuse (overuse of resources), and trojan horses and viruses which modify system files and programs by unusual execution of predictable commands.

3.2.3 Wisdom & Sense

W&S [Vacc89] is a system under development at Los Alamos National Laboratory. It tries to combine statistical methods, machine learning, and rule-based expert systems. Given a set of training data, W&S tries to create rules regarding the observed behavior. The rules created can be very general or very specific, such as; privileged users do

not work over dial-up lines during normal working hours. According to the literature, the rules are English-like and can be easily modified by a SSO, so that he can quickly incorporate his knowledge of specific attacks. The rule base generated can be quite large, on the order of 10^4 to 10^6 rules, but through optimization each rule can take just 6-7 bytes of memory, and on an inexpensive workstation W&S can fire approximately 20,000 rules per second. This makes it suitable for real-time intrusion detection.

W&S 1.0 and 2.0 were designed for a single host, but W&S 3.0 is designed for a distributed environment, in which several systems are monitored by a single host [McAu90].

3.2.4 ComputerWatch Audit Trail Analysis Tool

AT&T's ComputerWatch tool [Dowe90] was designed to summarize audit trails for the AT&T System V/MLS Operating System. (MLS stands for Multi Level Security, which adds Mandatory Access Control to UNIX. This is similar to Sun's CMW.) Again, a priori rules are applied to the audit data, which give the usual security summaries. An interactive rule editor is also provided which allows the SSO to modify the rules to fit the characteristics of a system. Finally, a SQL-like language is available to produce custom queries.

3.2.5 Information Security Officer's Assistant (ISOA)

ISOA was developed by Planning Research Corporation as a prototype real-time intrusion detection system and network security monitor. Like IDES and Haystack, it uses statistical and expert rules to create warnings. However, unlike the other two systems, ISOA has four levels of auditing modes: real-time (incoming audited record is immediately examined), session (marked by login and logout), intermediate (done at the end of a particular type of action, such as a program run), and high level (to prevent attacks from occuring over multiple logins).

3.3 Machine Learning

The application of machine learning and neural networks [Deba92, Doak92, Goan92, Vacc89] is a relatively new approach to the intrusion detection problem. Machine learning attempts to monitor and learn the normal activities of users. By knowing past events, inductive learning algorithms try to predict later events. This technique differs from the statistical systems above by not predefining the specific characteristics or traits, but rather choosing and weighing data as it becomes available. This approach also differs from Expert Systems in that it does not incorporate specific system vulnerabilities. The whole field of machine learning is still very young, so no major revelations have come from this area yet.

3.4 Other Systems

Most of the above systems use system-generated audit data as input. There are other systems which provide for some analysis without relying on the system logs. Heberlein's NSM [Hebe91] and Clyde-Digital System's Audit [Lunt88] both allow a SSO to collect and analyze data passing through a network or terminal, respectively. They provide a set of warning levels for each session, and tell the SSO why the session was considered suspicious. They are interesting in that they do not rely on the host audit trails for data collection, but rather do it themselves.

3.5 Audit Trail Browsing

Most of the current automated analysis tools remove the user from having to "get dirty" in analyzing raw audit trails. Most of the above tools generate fairly abstract views of systems, so much so that the SSO relies on the conclusions reached. However, getting and working with the raw audit data is exactly what is required when documenting damage, information flow, or change to a system.

The only tool I know of for helping browse audit trails is ComputerWatch for AT&T's MLS [Dowe90]. It performs its analysis by formulating the raw data into eight human-readable database files. The exact contents of the files

were not apparent in [Dowe90], but they are grouped as follows:

exec.tab	Process execution information
fork.tab	Process fork/exit information
alias.tab	files accesses and link information
ipc.tab	interprocess communication information
syscall.tab	system call failure information
uli.tab	user level information (logins, suid's)
io.tab	all read/write success/failure information
other.tab	everything else (mounts, kills, etc.)

ComputerWatch approaches modelling as a relational database problem. The reason for selecting these formats is based on the combinations of data items most frequently referenced together. From the paper, it appears that the database files do not try to link threads within the data, they only reformat audit records into database records. Navigating a path through the data must be done as a series of SQL database management system queries.

Also, it appears that the size required for all the database files may become quite large. Say a process opened a file. The execution file exec.tab would have the corresponding record, as well as alias.tab (for recording file accesses) and io.tab (for read/write information). By using the relational database format, information will be

repeated throughout the database files, adding to their size.

4 The Audit Workbench

In 1992, Chris Wee and I developed the idea for an "Auditing Workbench." The idea was to build a "workbench" or "sandbox" to experiment with auditors and audit analysis systems. The idea has generated much interest, and a new auditing research group has been formed at UC Davis. The original scope of this project has grown, and now includes many diverse aspects of auditing. But one of the primary goals for the workbench was to aid in browsing audit trails. Below, we shall examine the original design and goals of the workbench.

4.1 Architecture

Our standard model of audit analysis consists of four separate processes: data collection, management, reduction, and analysis. Correspondingly, there are four layers to the Audit Workbench shown in Figure 5, each component of which represents a different layer of abstraction:

Audit Generators — responsible for the creation of audit data,

Audit Management — responsible for the parsing, translation and management of audit data,

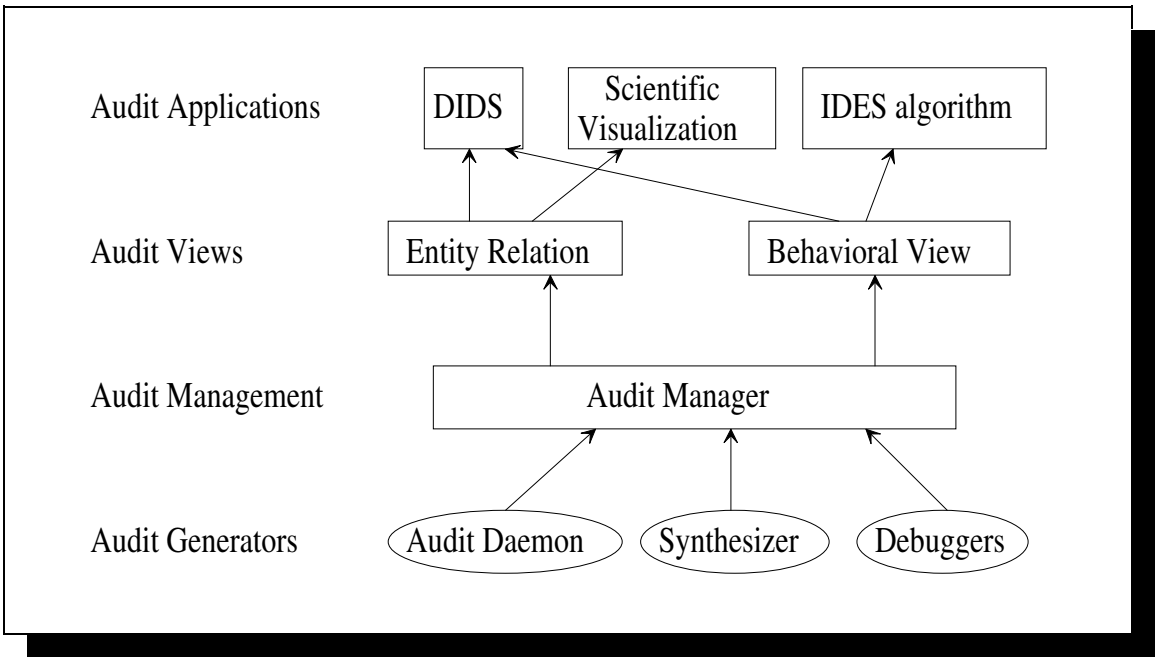

Figure 5 Audit Workbench Architecture.

Audit Views — responsible for combining lower-level audit trails into higher-level abstractions appropriate for audit applications, and

Audit Applications — which perform the actual analysis, e.g. intrusion detection algorithms, graphical user interfaces, audit browsing, program debuggers, network and system monitors, security analysis, etc.

Any program or module capable of producing audit data (either at the system or the program level) is called an

Audit Generator. These can include audit daemons, audit trail synthesizers, debuggers, accounting systems, and system monitors. We have also envisioned a tool for synthesizing audit trails using an event description language. This would be very useful for experimenting with race conditions, and in situations where actual tests would irrevocably damage a system, such as recursively removing files.

The Management layer would be responsible for performing all of the data manipulation functions, providing mechanisms for the recording, playback, splicing (grafting multiple audit streams into one), compression, storage, and retrieval of audit trails. An open area for research is developing abstract specifications for Generators and Views to automatically create translators that connect Generators and Views.

The Views layer of the Workbench would translate and filter audit trails to reduce the quantity and improve the quality of the audit data. Each audit application may demand to 'view' the system using a different abstract model. It is the responsibility of the View layer to map the incoming audit information onto the abstract model. By converting system specific data into the appropriate high-level easy-to-analyze abstractions, new analysis algorithms can be prototyped with far less effort.

For example, a collection of a user's files and permissions may be abstractly modeled as the security state of a user. Instead of processing audit records that report if a file was read, the analysis algorithm would only receive audit records that specify changes to the security state of the file. This permits the intrusion detection module to only examine what effects this change may have on the security state of the monitored system.

Once the audit data has been reduced and presented to the application layer as a series of views, the application layer programs are free to utilize those views and make higher level inferences and abstractions. The applications are freed from dealing with raw audit information, and can concentrate on higher level abstractions.

4.2 Research Goal

The Audit Workbench will permit us to develop new auditing paradigms and to evaluate audit analysis algorithms independent of specific auditors or operating systems. Additionally, we feel that by using the Audit Workbench, one can prototype new audit analysis algorithms rapidly and easily. The Workbench will allow greater interchangeability between various auditors and algorithms, and ease the tasks of testing, debugging, and evaluating auditing applications. Thus the major design goal is to provide a rich set of tools

for manipulating audit data, which we hope will encourage and expand the usefulness of audit analysis.

4.3 Rationale

We believe that the Audit Workbench would serve as a friendly and robust environment for researching and testing auditing algorithms as well as any other applications that might use auditing of data. The following sections give an overview of some of the perceived benefits of developing such a system.

4.3.1 Utilizing Multiple Views

A natural result of layering is the ability to support multiple Views from the same audit trail, or use multiple sources of audit information to create a View. This flexibility enables us to experiment with much richer models than those permitted by a single auditor. For instance, we could create a very high-level View that tracks user behavior and concurrently draws upon audit trails, environmental sensors, behavioral psychology, and expert systems as generators of audit information. Alternatively, we could provide an intrusion detection algorithm with an object dependency view as well as a session view. This would allow algorithms quick access to objects in the

system, and show how one session is interacting with those objects.

4.3.2 Combining the Stronger Pieces of Different Auditing Systems

With the Workbench, we can tackle another problem which hinders auditing. An algorithm developed under a specific auditor usually inherits all the strengths and weaknesses of the auditor or operating system. The contribution from a powerful auditor or the handicap of a weak one can significantly influence the performance or effectiveness of an algorithm. By being able to utilize the most effective pieces from different auditors, stronger algorithms can be developed.

4.3.3 Portability of Analysis Algorithms

When audit events are described primarily in terms of the abstract logical model, the algorithms need only contain a few system-specific details that are essential to the analysis, making the algorithms more general and portable. Thus, if the algorithms are decoupled from the auditor, we can port the audit analyzer by simply substituting the underlying auditor in the Workbench. This is analogous to writing computer programs that utilize basic services

through an application library rather than accessing the bare machine. Thus, algorithms developed using the Audit Workbench will be more robust and depend less upon the specifics of any particular auditor.

4.3.4 Testbed for Intrusion Detection Systems

The Audit Workbench will also be useful as a standard platform on which to compare different intrusion detection systems. Frequently, algorithms are developed on different operating systems, using different auditors, and it becomes very difficult to distinguish the contribution of a strong auditor or the handicap of a weak one with respect to the performance or the effectiveness of the IDS. There is no level playing field on which to compare intrusion detection algorithms. Instead, we are forced to compare specific implementations of the algorithms, and the side effects of the host-specific auditor cannot be mitigated. The Audit Workbench helps us to concentrate specifically on the IDS algorithms by creating multiple Views from the same audit trails. Thus, we can compare the performance and effectiveness of various algorithms when presented with the same input. Selecting the proper input information, however, is akin to designing good benchmarks and is yet another research issue.

4.3.5 Extensible Audit Trails

Since the Audit Workbench will permit extensible auditing by being able to combine audit data from multiple sources, we can integrate new auditing features with existing auditing packages, and then use the augmented auditor with existing analysis algorithms. For example, there is a project under way at UC Davis to add auditing to Remote Procedure Calls (RPC) and the Network File System (NFS) to the Sun BSM [Choi93]. This will significantly enhance intermachine auditing, as RPCs are not audited.

4.3.6 Experimenting With Hard-to-Duplicate Conditions

Using the Workbench, we hope to be able to synthesize audit trails. A normal trail with specific attacks embedded inside can be used for testing and debugging audit analysis algorithms. These composite trails can help create realistic simulations to evaluate intrusion detection systems. The Workbench can also assist in testing real-time analysis applications by playing back or simulating audit events in a real-time mode.

4.3.7 Ease of Data Collection

Finally, the management facilities of the Audit Workbench would allow audit trails to be manipulated so that archival, splicing, compression, and filtering of audit trails may be performed easily. Many of the management functions are similar to multi-media system controls. The similarity exists because audit trails are sequences through time, not unlike audio or video sequences. Therefore it would be conceptually very easy to press "RECORD" to begin collecting audit data, and to press "PLAY" to simulate the same series of events.

4.4 Development and Testing

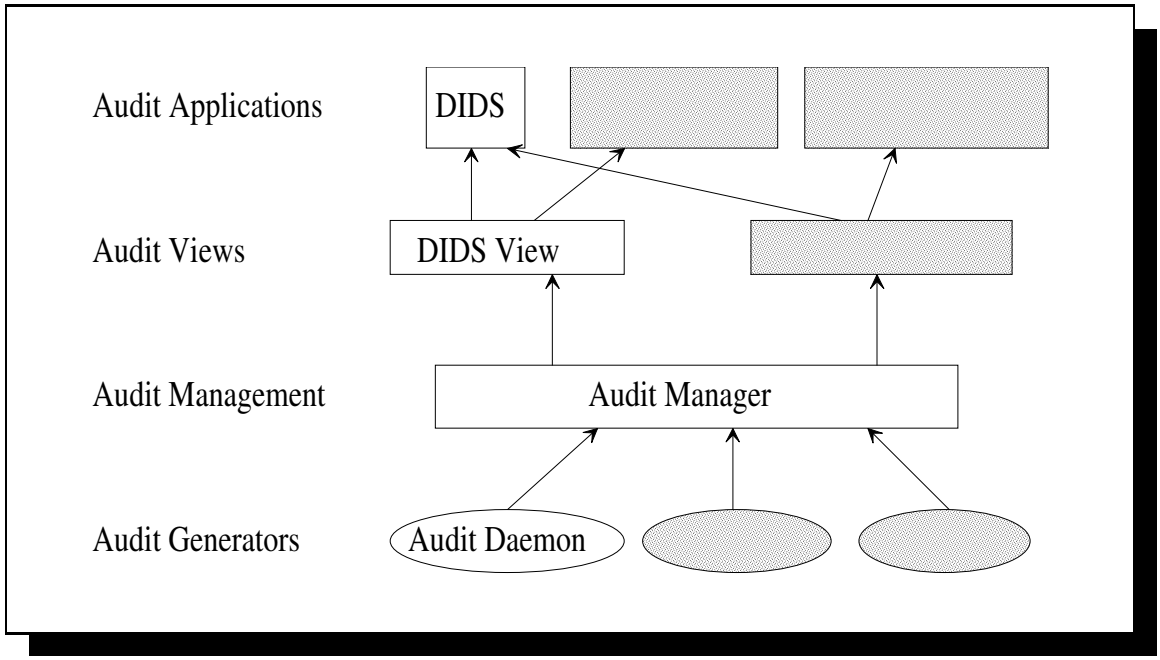

Figure 6 Vertical Testing.

We plan to construct a rapid prototype to explore the feasibility of the Workbench concept, and perform two experiments. The first experiment diagrammed in Figure 6 will integrate a module from each level of the Workbench into a single functioning intrusion detection system. This vertical integration will demonstrate that all the interfaces between each level work as specified. Next, using audit trail generators from two different vendors, e.g., DEC's VMS and Sun Microsystem's Basic Security Module (BSM), we will use the Workbench to translate them into identical Views. The unified View should be constructed such that an IDS algorithm using the View cannot distinguish between the two audit generators. This horizontal layer test in Figure 7 ensures that the layers' concept works with heterogeneous auditing systems.

As an example, UCD's DIDS [Snap91a] is primarily concerned with users, host machines, and their interconnections over a network. Details about the file system, user processes, or activities which are strictly local to the hosts are generally irrelevant to DIDS's analysis algorithms. Unfortunately, the audit trails produced at each host are full of such details and DIDS must filter out most of that noise, but retain enough details to associate users via network connection Ids, user Ids and process Ids. Using the Audit Workbench, we envision creating a DIDS View that describes host machines and

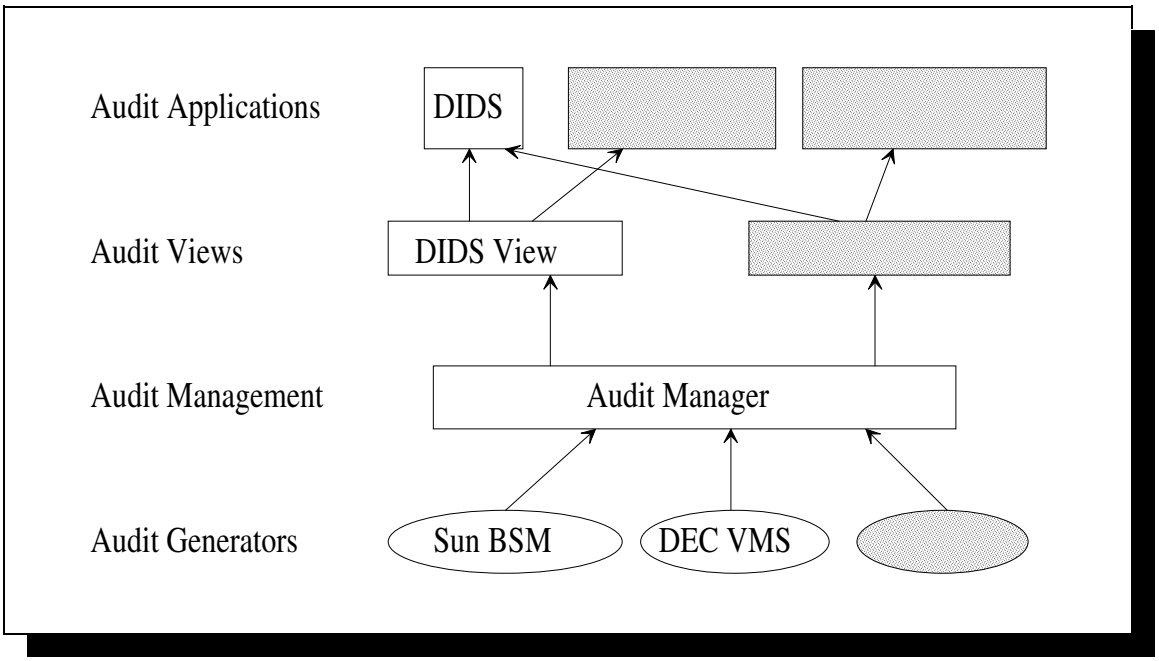

Figure 7 Horizontal Testing.

network connections as the primary entities of the abstract logical model. All other details about processes, user ids, operating system resources, and network protocols are abstracted, combined, or filtered to produce a coherent View consisting only of host machines and users connected between hosts. Thus, the DIDS algorithms can concentrate on the users and hosts as abstract entities without the processing burden of irrelevant information.

5 Towards More Effective Audit Browsing

The focus of this thesis is to develop useful techniques for browsing audit data. As mentioned, most previous intrusion detection research has focused on determining what data to examine, and has assumed that the SSO can examine the underlying audit trails to determine if intrusive activity has occurred. Unfortunately, few techniques exist for browsing. In current practice, most SSO's do not have the time to inspect reams of paper, and must rely on the conclusions reached by an IDS. [Dowe88] has proposed the straightforward application of Database Management Systems (DBMS) technology. I propose an alternate solution. While the DBMS and my solutions primarily focus on audit data from single systems, both models could be generalized to handle distributed auditing.

5.1 Questions to Ask of Audit Trails

Before continuing, we must determine what kind of questions will be posed to such audit analysis tools. The questions will be diverse, starting with the simple ones:

1) Who logged in during a specified time period?
2) Who created, read, wrote, or deleted a particular file?

3) What program(s) did a process run?

4) What subprocesses were created by a process? What did they do?

5) Which process is the parent of a process? What did it do?

6) Did someone try to obfuscate the audit trail by copying a well known system file under a different name?

7) How was this accomplished? (What usernames, programs, permissions, etc., were used?)

8) After some particular action, what did a user do?

9) How did a user gain additional permissions?

10) What effects did a file modification have on the system?

11) How did a file's permissions change without the owner's consent?

12) How did information from one file end up in another?

13) Was a vulnerability exploited on this system? If so, what was it? What steps were used?

14) Are there indications of a virus or trojan horse?

This list is not complete, but it does give a flavor for possible questions. Many of the simple requests can be satisfied by performing string matches on the audit data or by using a DBMS query for particular events. However, in

situations where multiple events were required to document the actions, manual browsing can become very difficult.

As an example, I used a well known vulnerability in SunOS to modify the login program. I wanted to examine the resulting audit data to determine what data was written and how it could be used to aid in a SSO investigation. The attack required several steps, beginning with a particular vulnerability in the UNIX program rdist. By exploiting this hole, an interactive shell was created which gave the user full permissions to the system. The person then modified and recompiled the source code for the login program, finally installing the trojan version. The events were all documented by the logging system, but this was a complex attack involving several stages and different types of system resources. Manually piecing together the individual events in the resulting log was very difficult, due to the number of processes, files, and actions involved.

5.2 Object-Based Analysis

The model I develop centers on the concept of an object. An object is any underlying system abstraction we wish to model: for example, users, processes, or files. (This differs from [Denn86], where a separation is made between subjects and objects. For our model, adding subjects into the object class simplifies the analysis.)

Most of the analysis programs described in Section 3 model users as the only objects. They fail to take into account how actions interconnect users, processes, and files.

In the models presented in this thesis, an object is any abstraction which is the source or destination of an action, e.g., a file write would document the action by a process source object onto a destination file object. As the audit trail is parsed, a graph of the object interconnections can be created by first determining which objects are involved, and then creating links between these objects. Note that this method can be easily implemented using the traditional directed graph notations of nodes and edges.

As an example, suppose we must determine a process hierarchy. A corresponding algorithm might follow this format:

```
/*************************************************
 * src = process which created the new process  *
 * dst = the new process                        *
 *************************************************/

while ( x := get_next_process_create_audit_record() )
     dst_obj := find_or_create_object( get_dst(x) );
     src_obj := find_or_create_object( get_src(x) );

     dst_obj.parent := src_obj;
     src_obj.child_list := src_obj.child_list
                               + dst_obj;
}
```

This example is specific to processes, but it shows how easily objects can be interconnected. By adding other types of objects such as files, user and user sessions, devices,

I/O ports, and other machine resources, a complex data structure may result, but because the objects are interconnected in this manner, a hypertext-like browsing system can be applied, thereby allowing easy navigation.

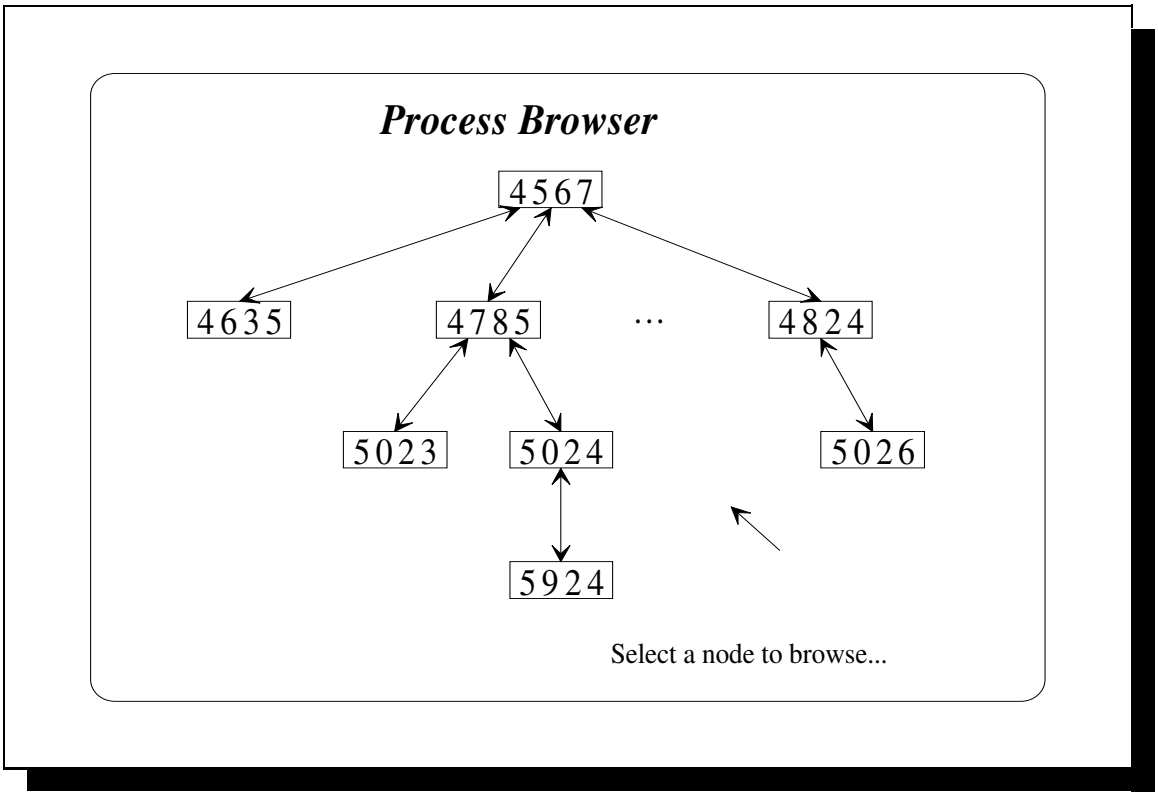

Figure 8 Process Tree.

Although a lot of preprocessing is required, one strong advantage is that once the links are constructed, interactive browsing can be performed rapidly. Since all the interconnections were built during the preprocessing stage, the SSO can immediately see which objects affected or were affected by the current object. The SSO can start at one node, and move to any other node by following the dependency links. In a Graphical User Interface (GUI) as in

Figure 8, following links could be as simple as clicking on paths to follow.

Also, with the proper internal data structures, results of these graphs could be easily incorporated with earlier or future graphs by simply merging the objects and actions involved. Another advantage of object-based analysis is that it can allow the system to be modeled in a manner consistent with the underlying abstractions.

Object-based analysis would also be of assistance in browsing the AT&T style of non-self-contained audit records. These logs have a large header which contains the internal name map describing global characteristics of every object in the system. The remaining audit records simply record the deltas, or changes to the objects. As a result, these audit trails are smaller in the sense that they keep the minimal amount of information for each object, and depend on the user to correlate the header and actions for the objects. Not only will Object-Based Analysis group the actions for each object, but it will also make the header data easily accessible to the analysis tool.

This model has the additional advantage that events on an object over extended periods of real time are compacted to just the list of actions. Suppose a raw audit trail has three million audit records, and only the first and last records involved make references to a particular object. Using a text find utility such as the UNIX "grep" on the raw

audit trails would require that the entire trail be parsed just to find the references. When links are built and maintained, the SSO need only find the object, then display the accesses. The data reduction is handled automatically.

This paradigm might also be useful in signature analysis. A signature is an enumerated list of actions necessary to achieve a particular goal. It might be possible to keep a list of signatures-so-far in each object, and as the next stage of a signature in an object is observed, the signature-so-far can be updated and transmitted to the next object(s) in the sequence. For example, a user modifying someone else's .cshrc file followed by that person logging in might be considered to be a signature of a trojan horse. When the file object for the .cshrc file is consulted, the code could make a note that the owner of the object is not the same as the person initiating the write. The file object could then notify the user object that a subsequent login would trigger a possible trojan horse.

5.3 Applicability of Relational Databases to Audit Browsing

There are some questions that are well suited for a DBMS, but the primary reason for wanting to develop an additional technology is that the interconnections among objects must be hand-crafted using a series of canned or on-

line queries. After each interconnection is discovered, an entirely new query must be formulated and submitted. A lot of time will be spent generating and executing these custom queries. Then, once the results are available, the SSO must decide which results warrant further investigation. Since many of these queries may be red herrings, much time will be wasted.

Additionally, during DBMS table construction, much of the data will be repeated because the DBMS stores each record's data separately. Having to list the name of each object in several records can potentially vastly expand the database files. By using objects, the name needs to be stored only once in the system: each successive reference would simply perform a lookup to find that object.

Finally, trying to get an overall view of a system with all objects and their interconnections displayed simultaneously would be impossible with a DBMS. The queries to establish all possible links would take an unacceptable amount of time.

5.4 Model #1, A UNIX Specific Model

The first model we develop will be UNIX specific. This model expands on the model presented in the Section 5.2 to include files. By adding file objects, we can now ask accountability questions about files, and gain a broader

understanding of system operation than by processes alone. This section will demonstrate two points: firstly, that effective audit browsing is feasible on BSM audit trails, and secondly, that preprocessing can perform a great deal of audit trail reduction.

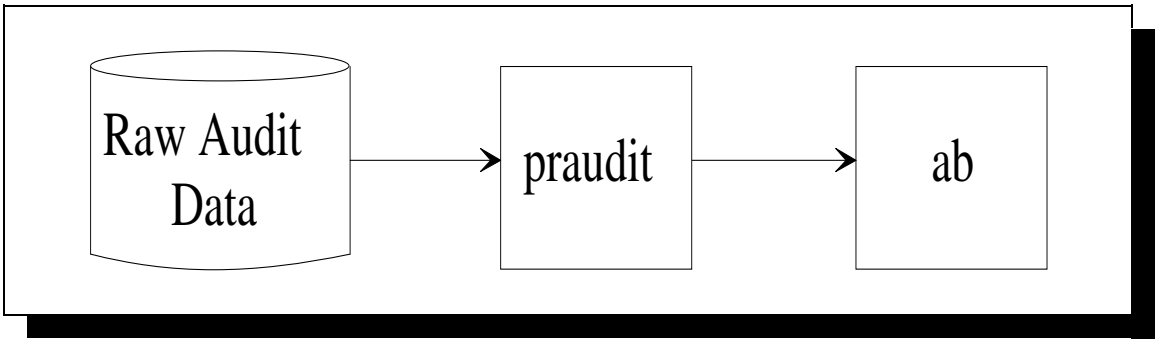

Figure 9 Conceptual Model of the Audit Browser.

To test my two points, a prototype preprocessor called "ab" – audit browser – was constructed in C. Figure 9 diagrams its purpose, viz. to parse the output from Sun's **praudit** command, create objects, and establish interconnections. On a lightly-loaded Sparcsystem 1, "ab" took only 5 minutes to process the data from a 16Mb binary audit file. In comparison, praudit took almost 10 minutes to convert the binary data to the form "ab" eventually used.

For this simple model, I was interested in analyzing only those actions which succeeded. I only wanted to see how the security state of the system actually changed over time, not how users attempted to change the state. Obviously, this method will not be able to detect those attacks which depend on failed accesses, such as doorknob

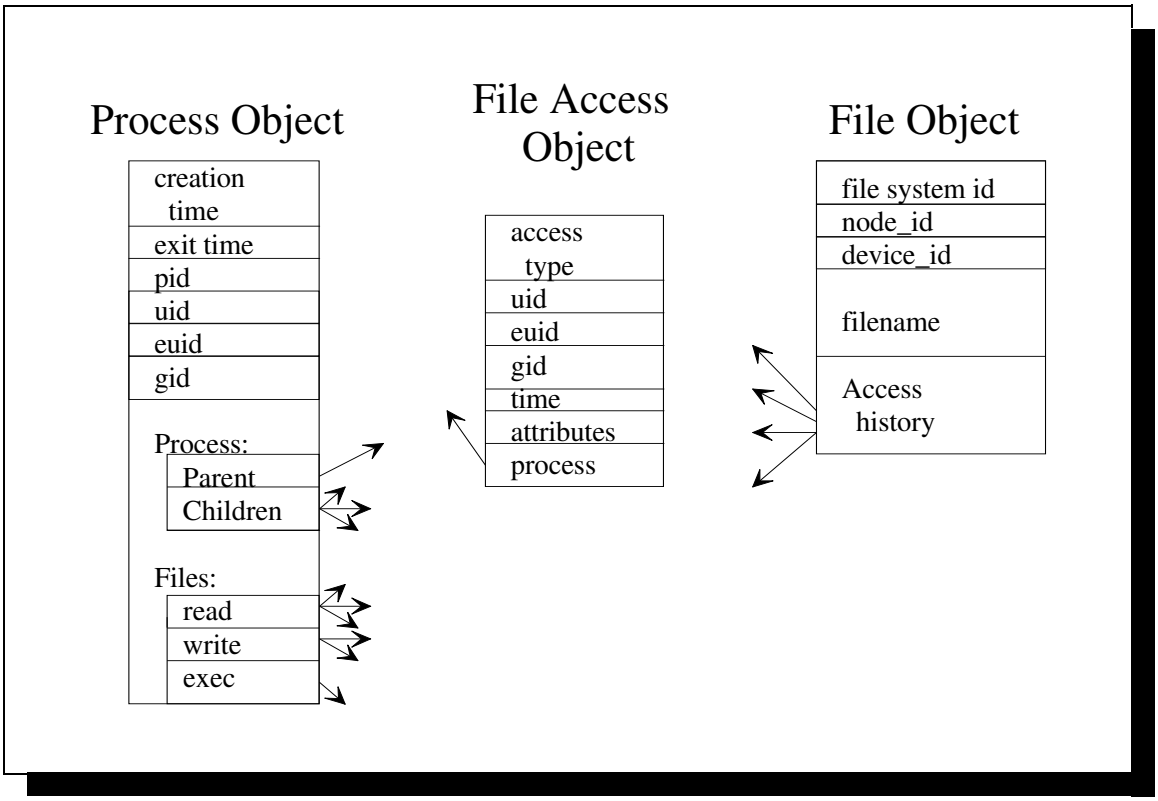

Figure 10 Conceptual Diagram of Objects Used in "ab."

rattling.

Each object contained information useful for audit analysis. In Figure 10, each process object contained the process id, the parent process id, the real and effective user id, the group id, the process creation time, the process exit time, pointers to the children and parent objects, and pointers to file objects for the three main types of file references: read, write, and execute. Each file object contained the file system, node and device ids, the filename, and pointers to the access histories.

Since there were a variety of system calls involving file objects, I used the following heuristics to determine to which group(s) a reference belonged:

1) If the reference caused any portion of the file system to change, the access was considered a write operation. The references included writes, creations, deletions, permission changes, etc. (Note that directory operations were also included in this grouping.)
2) If the reference caused any portion of data to flow from the file system to the processes, the access was considered a read operation.
3) If the reference was caused by a process trying to execute a file object, the access was considered an exec operation.

Note that there are some operations which involved multiple groups, such as a file open for read/write. In these cases, I considered a reference to a file to be both a read and a write.

Finally, I created a third data structure object that held the history of accesses for a file object. The access history object structure contained information such as user and group ids, time of access, file attributes used, and pointers to the process involved. Each file access record in the audit trail caused a file history object to be created; then "ab" filled in the objects with the data

available from the audit record. The associated process object was updated to reflect the new action. Although not specifically shown in Figure 10, this structure was quite complex and was the heart of the linking procedure between processes and files.

The pseudo-code for the main loop for "ab" follows:

```
/************************************************
 * Main driver for modelling.  Get type of audit *
 * record, then take the appropriate action.     *
 ************************************************/

while ( x := get_next_audit_record() ) {
     y := get_type( x );
     case ( y ) of {
          fork           :    create_new_process( x );
          exit           :    exit_process( x );
          exec           :    process_exec( x );
          open_read      :    file_access( x, READ );
          open_write,
          mkdir,
          chown,
          ...            :    file_access( x, WRITE );
          open_read_write:    file_access( x,
                                   READ.OR.WRITE );
          default        :    /* ignore( x ) */
     }
}
browse_objects();
```

"ab" continued to read audit records until the supply was exhausted. After parsing each record, the record's type was determined, and sent to the appropriate handling routine. The routines for forks and exits are similar to the pseudo-code in the previous section. For file references, the pseudo-code follows from expanding the process paradigm to its logical equivalent:

```
/************************************************
```

```
     * x = the parsed audit record                              *
     * access_type = type of access {READ, WRITE, EXEC} *
     ****************************************************
    file_access( x, access_type ) {
         access_obj := create_file_obj();
         access_obj.values := fill_in_audit_data( x );

         process_obj := find_or_create_process_obj( x );
         file_obj := find_or_create_file_obj( x );

         access_obj.process := process_obj;
         file_obj.access_list = file_obj.access_list +
                                access_obj;

         if ( access_type == READ )
              process_obj.read_list :=
                   process_obj.read_list + access_obj;

         if ( access_type == WRITE )
              process_obj.write_list :=
                   process_obj.write_list + access_obj;

         if ( access_type == EXEC )
              process_obj.exec_list :=
                   process_obj.exec_list + access_obj;
    }
```

After the audit data was formed into the model, "ab" then called a display routine which dumped the data in a easy-to-read form. First, all pertinent information such as user ids and group ids for each process object were displayed for each process, followed by the list of file accesses arranged by access type and the list of child nodes. To allow easy observation of the process tree, each child is indented from the parent by a small amount, as is each grandchild and so on. Finally, each file object is displayed followed by its access list.

With this model, several experiments were conducted with actual audit data. In each case, as long as the audit

trails contained the necessary information, "ab" was able to correlate the process and file interconnections. This in turn allowed me to easily browse the audit trails and follow the attack.

In addition to the example below, I was also able to follow an attack that relied on the vi editor's ability to save files following a crash. It was a simple matter to follow the objects to see how the file permissions were compromised.

5.4.1 A Worked Example

One of the scenarios I tested was a simulated attack on one of the UC Davis Computer Security Laboratory machines. I created an executable script called change_mod which exploited the rdist vulnerability. rdist then was tricked into changing the permissions on /bin/sh so that by running it, I would gain the permissions of the super-user. The resulting audit trail was then analyzed by "ab." I have included a portion of the output in Appendix 1. To conserve space I have only included a small portion of the output.

Since I knew what to look for, perhaps calling the procedure "easy" is a misnomer. But I will plead temporary ignorance, and lead the reader through my browsing session. Given that the user had masqueraded as root, it was my job to determine how that was accomplished. I first searched

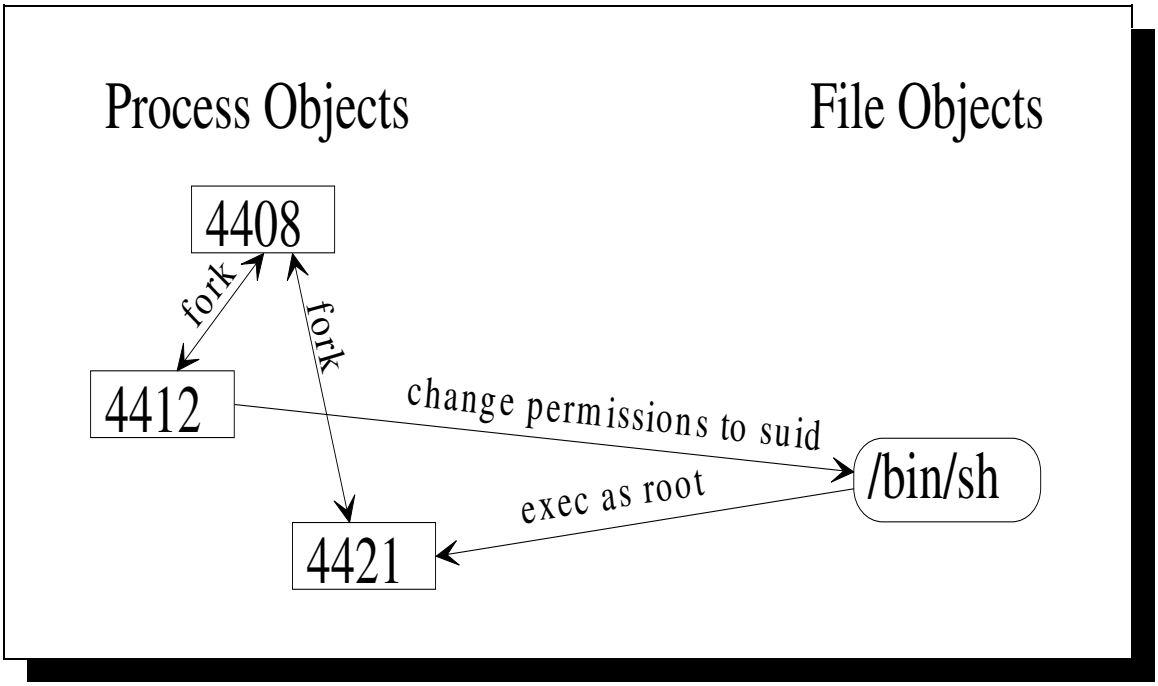

Figure 11 Browsing the rdist attack.

the audit trails by hand to determine which users had an effective user id equal to root. (Unfortunately, in this model I hadn't modelled users as objects, so I had to perform this step by hand.) Once I had the list of possible users, I invoked "ab," and from the full set of objects, I selected the login process for user "insider." I next examined the set of children for this login process. Immediately I noticed that process 4421 was created by running a shell with the user id set to root. (Figure 11 provides a graphical representation of the situation.) This gave me the first evidence that somehow the permissions on /bin/sh had been improperly set. Fearing the worst, I immediately checked all subprocesses of 4421 to d etermine what programs and files had been manipulated as root. In

this case, the only thing the user did was to reset the permissions back to the original setting.

In order to determine how the permissions were set, I next selected the /bin/sh object. By examining the access history, I noticed that the permissions were changed twice, once by process 4412 and once by 4423, a child of 4421. In between the permission changes, I found that only one root shell had been created, and it belonged to the user "insider," which was the process I had just examined. (Apparently no one had time to create another root shell before the window of opportunity closed.) I then selected process 4412 to determine how the permissions were set. By looking at the list of exec'd files, I knew immediately that rdist had either been compromised or that a vulnerability within rdist existed. By checking past accesses to the rdist object, I could find no evidence of any tampering during the time the audit daemon was operational. I finally followed the parent link of this process to process 4408, and determined that the program called change_mod had caused rdist to be executed. I then disabled the "insider" account as a preventive and punitive measure. It took me a week to get my account reactivated!

5.4.2 Advantages of this Model

This UNIX-specific model has a number of advantages, mainly that it presents the SSO with a view which closely models that of the operational system. Further, the SSO is not required to piece together links between files and processes; "ab" does it automatically. Also, it demonstrates proof of concept that an useful automated audit

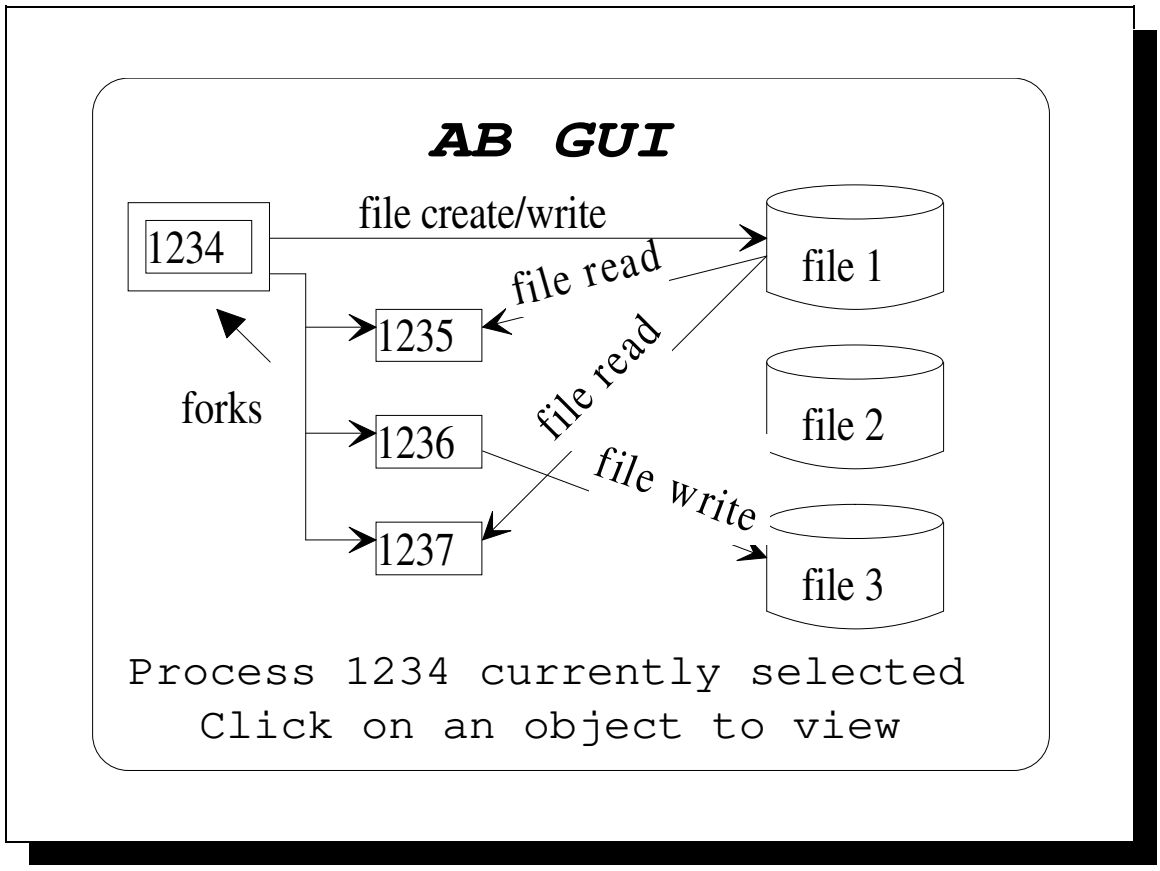

Figure 12 GUI interface to "ab."

browser can be built. The process and file objects are easy to traverse by simply selecting the next object from the list of objects present in the current object, or from a global object selector. If time had allowed, I would have liked to construct a X Windows-based application which would

have displayed the objects and their links together on the screen (see Figure 12). By using a point and click interface, browsing would be far easier than in a text-based environment.

In addition, we have demonstrated a straightforward method for data reduction. By only considering successful actions, only those records which actually change the security state of the system are allowed to enter the analysis. As mentioned, this method will not be able to detect some failure-based attacks. But for documenting how a user was able to change the state of a machine, and how information flow might be detected, the only items of interest are those calls which are successful.

5.4.3 Disadvantages of this Model

Unfortunately, there are a number of disadvantages of this model. Firstly, the model is very specific to BSM UNIX. Although it would have been fairly easy to modify the parser and data structures to accompany the new style of auditing, trying to port "ab" to another Operating System such as DEC's VMS or IBM's MVS, which have different underlying abstractions, would be very difficult. Additionally, some of the object fields were hard to manage, especially those which changed frequently over the life of some objects. There were several instances of filenames

changing, users changing effective user id as they executed certain programs, etc. These problems required that an alternate solution be found, and thus formed the basis for the model described next.

5.5 Model #2, A More General Model

There are a variety of methods from which to view audit records, but the one most intuitively obvious to a SSO would be to follow the model of the system being audited. Unfortunately, while every system has many of the same abstractions such as files and processes, building a general purpose audit browser would be impossible if the specifics of any operating system were allowed to permeate the design. Therefore, in order to keep the model as general as possible, we should design a system which only uses the underlying objects, and keeps a common framework with which to manipulate them.

Consider the following scenario: regardless of the auditor, each audit record represents an action between objects. By specifying which objects may be expected in each audit record and what their interconnections are, we may build a model in the same flavor as our previous UNIX-specific model, while maintaining a much more portable analysis technique.

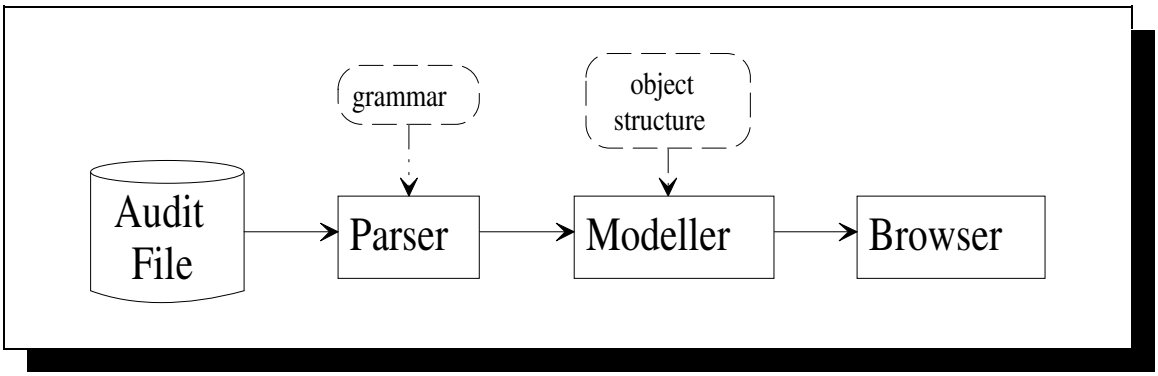

Figure 13 General model of analysis.

The technique follows the same methodology as the previous UNIX model, in that the audit trail is parsed and each record is passed to a modeling component as in Figure 13. However, in this design, instead of being hard-coded, the parsing grammar and the definitions for how object structures are created are now inputs to the system, allowing much more flexible definition of objects.

To determine what objects should be extracted from the audit trails, the audit trail format specification would be manually inspected to determine what information is available. For the interconnections, a table listing the dependencies between the objects would be developed. The dependencies are again determined by inspecting the specification of the audit trail and using the SSO's knowledge of system operations. This table would then be utilized by the parser to extract those objects, and by the modeler to make the interconnects. By changing only this table, this type of browser can be adapted to any kind of logging system. The browser need not know anything about

the underlying abstractions, except that there are objects and interconnections.

For the interconnections, a paradigm of source and destination objects would be quite useful. For example, when presented with the problem of trying to establish information flow from files to users, the original read of a file could be considered a source object, with the user or process involved being the destination. Or for the process creation domain, the parent process might be the source, and the child the destination. This method has the potential to turn the problem of audit browsing into a directed graph problem, a well-examined problem area. A typical query might be: starting at this file object (node) and traversing the links (directed edges), can you reach this user (node)?

As an example of how this table might be constructed, consider a BSM file_open_for_read audit record, of which an example follows (for an explanation of the fields, please see Chapter 2):

```
header,101,open(2):read,
     Tue Sep 29 15:34:23 1992, + 420000 msec
path,/,/home/frincke,/home/frincke/robin
attribute,100644,frincke,staff,1802,1056,224
process,frincke,frincke,frincke,staff,1352
return,Error 0,4
trailer,101
```

The SSO might define the objects to be the file, user, and process associated with the event. Based on this decision, a table of the interconnections is then developed:

(S = source node, D = destination node)

Type/Object	...	file	...	process	child process	user
Open_READ		S		D		
Open_READ		S				D
...						
Process_FORK				S	D	
Process_FORK					D	S

Figure 14 shows diagrammatically how the objects and their interconnections could be represented. The contents of objects will change considerably. In the UNIX-specific model, each object contained several types of information, such as user and group ids. Rather than having to rewrite the object definitions for each logging system, it makes more sense to keep only the information necessary to uniquely identify it, and leave the unnecessary information in the audit trail.

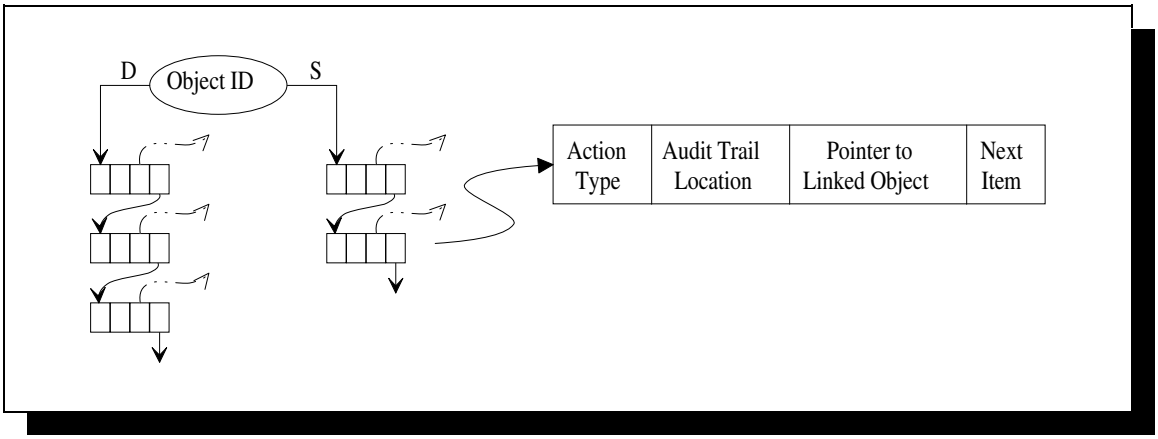

Figure 14 Proposed data Structures for objects.

Additionally, there should be a list of object accesses. Since we have used the example of source and destination accesses, there should be a corresponding list for each. And finally, each accessed object should contain the data found in that record. Unfortunately, since the amount of data necessary could be very large, it would be more efficient to keep a pointer to the corresponding location in the audit trail. A side benefit is that now the SSO can examine the raw audit trails if necessary.

The high-level psuedo-code for this system is very similar to that of the earlier example. However, since most of the information will remain in the audit trail, and the objects and interconnections are much more general, the lower-level code is simplified.

```
while ( x := get_audit_record() ) {
     y := get_type( x );
     foreach ( z := table_entry for y ) {
          src_obj := find_or_create_src_object(z,x);
          dst_obj := find_or_create_dst_object(z,x);
          src_access_obj := create_access_object();
```

```
                dst_access_obj := create_access_object();
                src_obj.access_list := src_obj.access_list +
                     src_access_obj;
                dst_obj.access_list := dst_obj.access_list +
                     dst_access_obj;
                     src_access_obj.action_type := y;
                dst_access_obj.action_type := y;
                     src_access_obj.location :=
                     current_position_in_audit_file
                dst_access_obj.location :=
                     current_position_in_audit_file
                     src_access_obj.obj_ptr := dst_obj;
                dst_access_obj.obj_ptr := src_obj;
           }
     }
     browse_objects();
```

This method has direct applications to the area of modelling and data reduction. By simply changing the model information in the table, an entirely different picture of the system can be generated. Some models might be useful for abstracting information flow, while others are better suited for discovering how a file changed its access permissions. Data reduction could be performed at this stage by simply changing the table to model only the actions of interest. Note that with the proper table, this audit browser could have the same functionality as the UNIX-specific model presented earlier. With enough experience, a wide library of tables could be developed, each suited for a special purpose.

5.5.1 Advantages of this Model

As previously mentioned, there are many advantages of this method. By defining different tables, the system can easily model different objects or kinds of objects. By also changing the grammar, the system becomes highly portable across platforms. The system is not forced to rely on the same underlying abstractions. The only abstraction which must be made is that objects and interconnections exist within the system. The system is less memory intensive since it keeps most of the audit data in the audit trails, and the resulting structures could be used by graph algorithms.

5.5.2 Disadvantages of this Model

The primary disadvantage of this model is that it requires the majority of the audit data to be kept in the audit trails, rather than in memory with the objects. This will alleviate memory requirements in large models, but this method requires that the browser be able to efficiently seek to the required locations. If an object has a multitude of links, it may take the program some time to retrieve all of the data. Additionally, this method requires that the operating system be able to maintain the integrity of the data after the model is built. If the data were to change, the browser could read the wrong information.

5.6 Answering the Questions

At the beginning of this section, we posed some questions which might be asked of audit trail browsers. Therefore it is appropriate to give some guidelines as to how these questions can be answered using this object-based analysis approach.

1) Who logged in during a specified time period?

This type of question would be more suited to a database query, but can be answered using our browser. First, the data must be reduced to include only those records from the time in question. Since records in audit trails are normally stored by time, it should be fairly easy to write a filter. Then once the model has been built, the session objects can be examined for login activity.

2) Who created, read, wrote, or deleted a particular file?

This question can be answered by simply looking at the access history for the file object.

3) What program(s) did a process run?

Similar to 2).

4) What subprocesses were created by a process? What did they do?

The browsing tool could be instructed to show only the process manipulation access histories, from which additional processes can be examined.

5) Which process is the parent of another process? What did it do?

Similar to 4).

6) Did someone try to obfuscate the audit trail by copying portions of a sensitive file under a different name?

7) How was it accomplished? (What usernames, programs, permissions, etc., were used?)

12) How did information from one file end up in another?

To answer 6), 7), and 12), if the file in question was read by a file copy command, it stands to reason that the new file might contain information from the original file. This can be caught very easily by first looking at the file in question and then examining the processes and executed commands. However, if the information was leaked when a process opened a file for reading and another for writing, we cannot be as sure. One process reading the data does not necessarily imply it wrote the same data. Additionally, the process which read the data may not necessarily have done the writing. By some interprocess communication, audited or not, any other process may have performed the write. Unfortunately, there are many ways for processes to communicate, not all of which are currently auditable, e.g. changing system performance metrics which another process could detect and intrepret as data.

8) After some particular action, what did a user do? Again, this involves following the links from a particular point in the model.

9) How did a user gain additional permissions?

11) How did a file change permissions without the owner's consent?

13) Was a vulnerability exploited on this system? If so, what was it? What steps were used?

9), 11), and 13) can be answered by considering the previous example demonstrating how rdist can be subverted into giving root access. The resulting structures were able to detect the change, and the SSO could follow the flow of events to document what happened.

10) What effects did a file modification have on the system?

Many of the questions cannot be answered unless the change had some kind of effect on the audit records. For example, a user modifying the password file to create a new user will only generate file change records. Until the new user logs in, we won't know what change was made. Some of the modification problems would be better handled by static analysis tools such checksumming in COPS [Farm91] and SPI.

14) Are there indications of a virus or trojan horse? If the actions caused by the trojan horse or virus are auditable, it is very likely that the SSO would be able to detect attempted changes to system files using the methods

described. However, some trojan horse attacks do manage to escape auditing, thus frustrating our efforts.

5.7 The Downfall of Slicing

One of the original goals of this thesis was to determine the utility of incorporating a debugging technique called slicing [Weis81, Agra90] into the analysis of audit trails. A goal of slicing is to use dataflow analysis and graph theory to determine all statements in a program or program segment which might affect the value of a variable. The slicer uses these statements to construct a minimal form of the program which still produces the observed behavior.

As an example, it is desirable to find all statements which affect the value of variable Y in statement s10 in the code below.

```
     begin
s1:       read(X);
s2:       if ( X < 0 ) then
s3:            Y := f1(X);
s4:            Z := g1(X);
          else
s5:            if ( X = 0 ) then
s6:                 Y := f2(X);
s7:                 Z := g2(X);
               else
s8:                 Y := f3(X);
s9:                 Z := g3(X);
               endif
          endif
s10:      write(Y);
s11:      write(Z);
     end
```

They are statements s3, s6, and s8. Then we use this set of statements, and add to it all statements which can affect these new nodes, either by a data or a control dependency. This process continues until we develop the set {s5 (because X determines which function is called, affecting the value of Y), s2 (for the same reason), s1 (because the value of X is set here, which affects the outcome of s2 and s5), s3, s6, and s8}. By considering only this minimal subset, we can focus our debugging effort on these statements, because statements s4, s7, s9, and s11 will have no effect on Y.

I had originally hoped that a variation of this method would be useful for analyzing audit trails. Starting with a particular object, we could slice backwards or forwards, only keeping a small subset of references which directly affected or were affected by the objects in the working set. As our slicer moved into the audit trails, if an audit record was found that contained one of the working set objects, the record would be output to the SSO, and the other objects discovered in that record would be added to the working set and slicing would continue.

As an illustrative example, assume that user *joe* logged into a heavily used system, and then changed his user identification to *frank*. User *frank* read some confidential information, and over the course of the next hour, he read some files, changed his user name to *mark*, then created and wrote information into a world readable file called *file1*.

Thirty minutes later, *file1* was downloaded by someone using the file transfer daemon into a machine called cracker. Three days later, the SSO is called to find out how the information was compromised. He may print the voluminous audit trails and step through it by hand (not likely), or call the slicer to slice the audit trail to construct a chain of events starting with the sensitive file access.

The slicer would note that the sensitive file was read by *frank*. By slicing earlier in the data, it would find the user id switch from *joe*. At this point it would know that it can disregard any audit data which do not reference any of three objects, *{sensitive_file*, *frank*, or *joe}*. Now by slicing forward, it would find the user id switch from *mark*, and *mark*'s subsequent creation of *file1*. Our working set now contains {*sensitive_file, joe, frank, mark, file1}*. Note that the SSO only had to provide the initial audit record; while the slicer took care of the data reduction. Note also that this is an interactive process; at each step the SSO must state what objects to add to the working set based on his knowledge of the problem, and instruct the slicer as to which direction to process. I had constructed a small prototype of this slicer, and the results were very encouraging, especially when the results were displayed graphically using a tool called xgrab. (See Figure 15.)

At first glance, the idea seemed plausible. There were several issues to be resolved: namely, which attributes

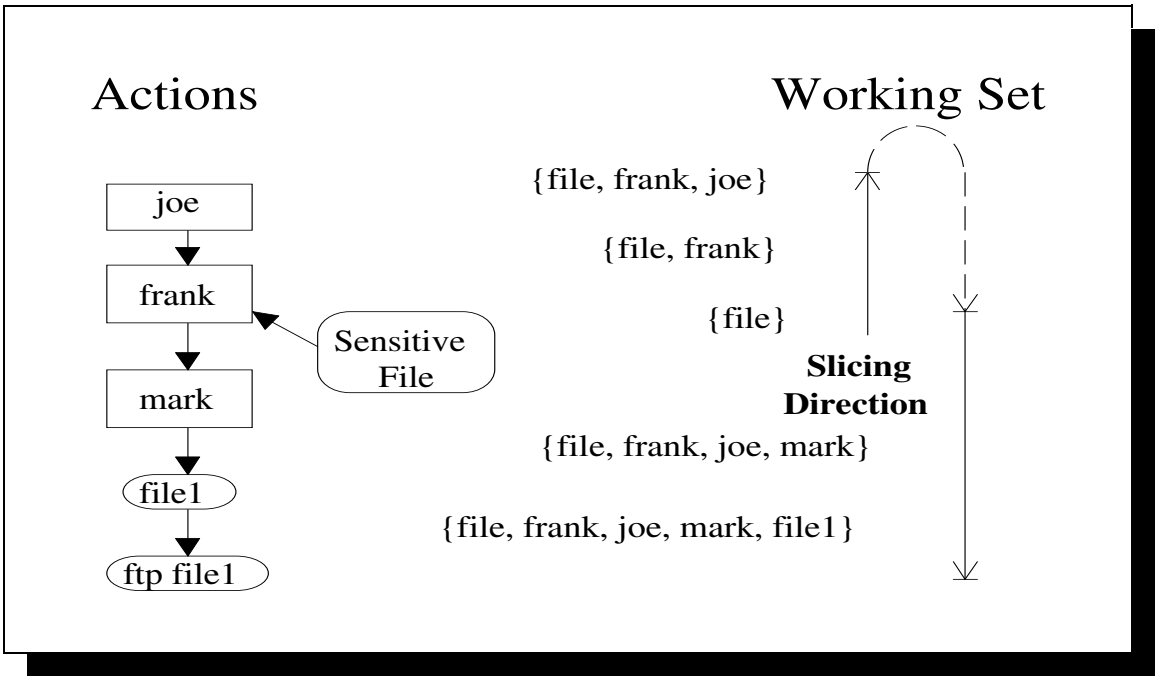

Figure 15 Graphical view of slicing, with working set.

should be used for slicing. Several showed potential, such as users, processes, and files. The attribute problem was the same as the object problem in the earlier object-based models. Some attributes might be of more use for certain kinds of browsing.

While further developing this model, I noticed that this method had a major flaw, which caused me to abandon further research into it. If the slicer started with an object and moved forward, it might pass over objects that are needed later in the analysis. Since those objects are currently not in the working set, they are ignored. After the slicer finds an applicable record and determines those objects should be added to the working set, the slicer must be instructed to move backwards, parsing and analyzing all

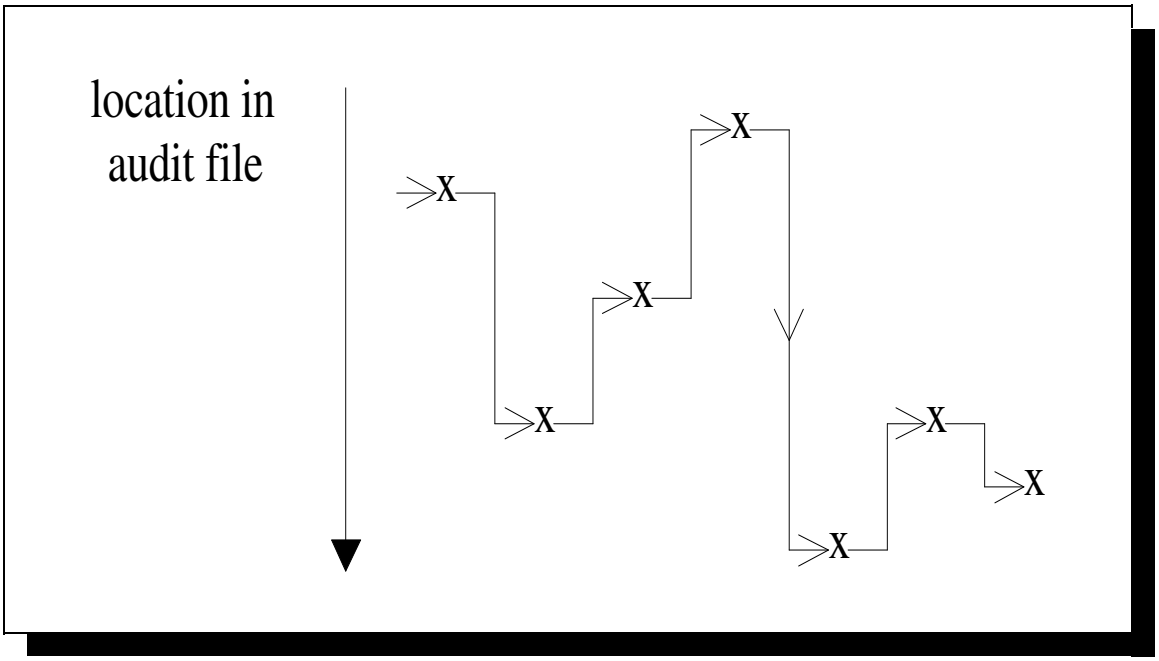

Figure 16 Slicing showing excessive forward and backward motion.

the data it previously examined as in Figure 16. If enough direction changes are necessary, the algorithm becomes terribly inefficient. It might be possible to keep a list of recently seen objects, but this may be of no use, especially if the locality of object references is small. The algorithm does seem to work quite well if the actions occur serially, but this cannot be guaranteed, especially /since BSM sometimes reports records out-of-order.

5.8 Experiences with BSM

Sun's BSM does have a variety of limitations; however it is a relatively complete package. Since the experiments performed for this thesis involved BSM for SunOS 4.1.1, a

brief summary of my experience is in order. The primary limitations are:

1) audit data is sometimes out-of-order,
2) no per-object auditing (other than user),
3) no easy model for distributed auditing,
4) no auditing on some standard OS services,
5) a bug in the output of filenames, and
6) reliance on system configuration between audit collection and analysis.

The first point is well known to Sun [Wetm92d], but should be mentioned. There are some instances in which the audit data is not stored in the same order as the corresponding actions. For example, when a process forks, the audit trail shows the beginning actions of the new process before the process fork is seen. Any audit browser which depends on records being in proper time based order must take this partial ordering problem into account, or the resulting analysis will be invalid. One could overcome this limitation by using a windowing mechanism similar to those used in out-of-order communication protocols.

Sun allows great flexibility for setting the audit state of users, since this is the historical model for collecting data. However, no provisions were made for setting the audit state of other objects. In some cases it would be of great utility to have finer control over objects being audited. In this manner, if the SSO determined that

additional information regarding actions to a particular file is required, the audit daemon could produce that information.

As an example, consider the UNIX password file. Each entry in this file defines an individual user. If a general text editor were used to edit this file, the audit trail would only contain the file open and the processes and programs involved. Of much greater use would be how much data was read or written, or at least the change in the number of lines in the file. There is some work being done at UC Davis to combine static and dynamic analysis tools which could detect these changes, but it would be useful to have the information available directly in the audit data.

Conversely, in other cases some information in audit trails is useless. For example, there are several dynamically linked files which are read before the execution of every program. Unless these files are modified, their inclusion in audit trails is unnecessary. By allowing a per-object preselection audit state to be set, an additional level of audit reduction can be performed.

Another problem area comes from the interconnectivity inherent in intermachine communications, specifically regarding NFS. Auditing is only performed on the system which invoked the request. Frequently, the object accessed resides on a remote file system. Unfortunately, if a SSO tries to determine who accessed a file, he must poll each

audited machine in order to determine accountability. Sun has provided some tools for aggregating audit files, but it seems more straight-forward to perform auditing both on the host machine where an object resides, and on the machine which initiated the request. Additionally, the machine which generated the request may be a machine which is not running BSM, which would preclude all chances of determining responsibility.

In addition to the problems mentioned above, there are several standard service mechanisms which are not audited, and are the focus of ongoing research at several institutions. Inetd and many of the inetd servers, such as finger and remote procedure calls (RPCs) are generally not audited. The requests generated by these servers do invoke auditing once the lower level system calls are invoked. But it would be useful to provide application-level auditing in these areas. It would be of more use to a SSO to know that one machine originated an excessive number of finger requests than knowing the password file was opened multiple times. This data could supplement the information already captured, and would help in analysis by providing higher-level explanations to lower-level actions. Pipes and other interprocess communications mechanisms need to be audited more.

One of the biggest problems noticed during experimentation was BSM's inability to distinguish certain

objects, namely files. The first problem was apparently the result of a bug in the audit daemon. Files with especially long pathnames generated sequences of unknown binary characters. It was not apparent how to get around this limitation, so I switched to file system numbers for object identification. Unfortunately, these numbers were not consistent either, so some of the experiments had to be interpreted manually.

Another potential problem comes from the reliance of system tables remaining the same between audit data collection and analysis. As an example, all references to users are represented in audit trails by the user id assigned in the password file. If a user id number were to change, or if the numbers on the collection machine and analysis machine are different, the SSO might assign responsibility to the wrong individual.

One possible solution would be to include an "object record" the first time an object is seen, along with a cross-reference value which could be used in future references as in the AT&T style of auditing [Dowe90]. This "object record" would include all pertinent information relating to this object, its name, its type, and perhaps an ASCII description of the object. Unfortunately, this would require a penalty in performance in the audit daemon, as the daemon must now maintain this information. This is probably not a valid option. Before an audit browser based on the

techniques presented in this thesis can be developed, this object identification issue must be resolved.

6 Future Work

In the area of audit browsing, much work still needs to be done. The only two methods proposed for browsing are DBMS and Object-Based Analysis. There may yet be another paradigm more appropriate; for following the interconnections of object, Object-Based analysis seems appropriate.

For immediate future work, the most pressing issue is to actually build the proposed generalized audit browser. After the initial success with the UNIX-specific model, I have great confidence that a prototype can be quickly built and effectively used. During the next year, I hope to continue further development of this model and construct the prototype. Once the prototype is built, benchmarks must be established to determine the effectiveness of this system.

There should also be research undertaken to determine if signature analysis could be implemented using these concepts. Since the current method uses individual events to advance the finite state automata, by a careful organization of data structures, the object links could be traversed as a replacement for the automata. By linking the objects in real time much as is done in DIDS, the browsing could be done simultaneously with signature analysis.

This method may also have use for comparing expected versus actual performance for programs. For example, in normal rdist program operation, several processes are created in a very specific order, each with very specific duties. The process running rdist is supposed to read several system files, fork some subservient processes, and then create and change permissions on one temporary file. If the system observed that the create and change operations involved different files, a warning would be generated. By comparing the processes and duties observed with a template of those expected, we should be able to determine if something is amiss. This form of misuse detection would have the additional advantage that the program developers could specify the expected actions of a working program. This method does not require that bugs be exploited before an expert system rule can be generated.

Finally, a closer examination should be made as to how graph algorithms could be incorporated into the analysis. I have hypothesized that such techniques would be useful, but until the object browser has been built, I cannot say for sure.

7 Conclusions

Viewing the media reports of late, computer systems are being compromised with increasing regularity. It is apparent that many of the system restrictions are being circumvented by various means. Many of the intrusions are caused by human factors such as weak or compromised passwords, or by insiders abusing their privileges. When the preventative measures fail in their effectiveness, auditing can be called upon to help document actions. While auditing cannot solve the problem of detecting all intrusions, it has the potential to help the SSO. But until reasonable methods are available for viewing these audit trails, the potential remains untapped.

In this thesis, I have proposed an alternate method to the DBMS method proposed in [Dowe90]. Apparently these are the only two models in existence. This method develops system abstractions into objects, and then attempts to develop interconnective links. A UNIX-specific prototype system was developed, and showed great potential. It allowed a SSO to easily monitor the results of several controlled attacks. A general model was then proposed, with a series of future enhancements. This generalized method seems to be very appropriate for answering many of the

questions an SSO will be asking of audit trails, once generalized browsing becomes widely available.

8 Acknowledgements

I would like to thank the following people for their assistance on this project:

Christopher Wee — for many fruitful discussions, and for being a great source of inspiration.

Steven R. Snapp — for allowing the use of the BSM parser code. Without his help, I would still be coding.

Kristi Williams — for giving the moral encouragement I desperately needed at times.

I would also like to thank the Becky Bace, the National Security Agency, Lawrence Livermore National Laboratory, and the United States Air Force Cryptological Support Center for their support during this project.

9 References

[Agra90] Hiralal Agrawal and Joseph R. Horgan, "Dynamic Program Slicing," Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation, June 1990.

[Ande80] James P. Anderson, "Computer Security Threat Monitoring and Surveillance," Technical Report, James P. Anderson Co., Fort Washington, PA, April 1980.

[Bart92] Tony Bartoletti, "SPI," available from irbus.llnl.gov, Lawrence Livermore National Laboratory, 1992.

[Bish89] Matt Bishop, "A Model of Security Monitoring," Proceedings of the Fifth Annual Computer Security Applications Conference, Tucson, AZ, December 1989.

[Bony81] D. Bonyun, "The Role of a Well Defined Auditing Process in the Enforcement of Privacy Policy and Data Security," Proceedings of the 1982 Symposium on Security and Privacy, April 1981.

[Choi93] Wilson Choi, Master's Thesis, Department of Computer Science, University of California, Davis, CA 95616, 1993.

[Colu92] Columbia University Guidelines for Computer Usage, received from ftp.eff.org: /pub/academic/policies/columbia.edu, October 4, 1992.

[Deba92] Herve' Debar, Monique Becker, and Didier Siboni, "A Neural Network Component for an Intrusion Detection System," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1992.

[Denn86] Dorothy E. Denning, "An Intrusion Detection Model," Proceedings of the IEEE Symposium on Security and Privacy, April 1986.

[Denn87] Dorothy E. Denning, David Edwards, R. Jagannathan, Teresa Lunt, and Peter G. Neumann, "A Prototype IDES - A Real-Time Intrusion Detection Expert

System," Technical Report, SRI International, August 1987.

[Doak92] Justin Edgar Doak, "Intrusion Detection: the Application of Feature Selection, a Comparison of Algorithms, and the Application of a Wide Area Network Analyzer," Master's Thesis, Department of Computer Science, University of California, Davis CA 95616, 1992.

[Dowe90] Cheri Dowell and Paul Ramstedt, "The ComputerWatch Data Reduction Tool," Proceedings of the 13th National Computer Security Conference, Washington, DC, October 1990.

[Farm91] Dan Farmer, "COPS ver 1.04," source available from cert.sei.org, 1991

[Garv91] Thomas D. Garvey and Teresa Lunt, "Model-based Intrusion Detection," Proceedings of the 14th National Computer Security Conference, Washington DC, October 1991.

[Glig85] Virgil D. Gligor, "Guideline for Trusted Facility Management and Audit," Technical Report, University of Maryland, 1985.

[Goan92] Terrance Lee Goan Jr., "Towards a Dynamic System for Accountability and Intrusion Detection in a Network Environment," Master's Thesis, Department of Computer Science, University of California, Davis, CA 95616, 1992.

[Hebe91] Louis Todd Heberlein,"Towards Detecting Intrusions in a Networked Environment," Master's Thesis, Department of Computer Science, University of California, Davis, CA 95616, 1991.

[Javi91] Harold S. Javitz and Al Valdez, "The SRI IDES Statistical Anomaly Detector," Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, May 1991.

[Lunt86] T.F. Lunt, J. van Horne, and L. Halme, "Analysis of Computer System Audit Trails," Sytek Technical Reports TR-85007, Mountain View, CA, May 1986.

[Lunt88] Teresa Lunt, "Automated Audit Trail Analysis and Intrusion Detection: A Survey," Proceedings of the 11th National Computer Security Conference, October 1988.

[Lunt89] Teresa Lunt, R. Jagannathan, Rosanna Lee, Alan Whitehurst, and Sherry Listgarten, "Knowledge-Based Intrusion Detection", Proceedings of the AI Systems in Government Conference, Washington DC, March 1989.

[McAu90] Noelle McAuliffe, Dawn Wolcott, Lorrayne Schaefer, Nancy Kelem, Brian Hubbard, and Theresa Haley, "Is Your Computer Being Misused? A Survey of Current Intrusion Detection Technology," Proceedings of the Sixth Annual Computer Security Applications Conference, Tucson, AZ, December 1990.

[NCSC85] National Computer Security Center, "Department of Defense Trusted Computer System Evaluation Criteria", DoD 5200.28-STD, December 1985.

[NCSC88] National Computer Security Center, "A Guide to Understanding Audit in Trusted Systems," NCSC-TG-001", Version 2, June 1988.

[Neum88] P.G. Neumann, "The Computer-Related Risk of the Year: Computer Abuse", 3rd Annual Conference on Computer Assurance (COMPASS '88), National Bureau of Standards, June 1988.

[Picc87] Jeffrey Piccioto, "The Design of an Effective Auditing Subsystem," Proceedings of the IEEE Symposium on Security and Privacy, April 1987.

[Rice92] Rice University Guidelines for Computer Usage, received from ftp.eff.org: /pub/academic/policies/rice.edu, October 4, 1992.

[Sebr88] Michael M. Sebring, Eric Shellhouse, Mary E. Hanna, and R. Alan Whitehurst, "Expert Systems in Intrusion Detection: A Case Study," Proceedings of the 11th National Computer Security Conference, Washington DC, October 1988.

[Sibe88] W. Olin Sibert, "Auditing in a Distributed System: SunOS MLS Audit Trails," Proceedings of the 11th National Computer Security Conference, October 1988.

[Smah88] Stephen E. Smaha, "Haystack: An Intrusion Detection System," Proceedings of the IEEE Fourth Aerospace Computer Security Applications Conference, Orlando, FL, December 1988.

[Snap91a] Steven R. Snapp, James Brentano, Gihan Dias, Terrance Goan, Louis Todd Heberlein, Che-lin Ho,

Karl Levitt, Biswanath Mukherjee, Stephen Smaha, Tim Grance, Daniel Teal, and Douglas Mansur, "DIDS (Distributed Intrusion Detection System) - Motivation, Architecture, and An Early Prototype", Proceedings of the 14th National Computer Security Conference, Washington DC, October 1991.

[Snap91b] Steven Ray Snapp, "Signature Analysis and Communication Issues in a Distributed Intrusion Detection System," Master's Thesis, Department of Computer Science, University of California, Davis CA 95616, 1991.

[SUN91] Sun Microsystems, Inc., "Installing, Administering, and Using the Basic Security Module", December 1991.

[Syte85] Sytek, Inc. "Analysis of Computer System Audit Trails," Technical Reports 85009, 85012, 85018, 86005, and 86007, Mountain View, CA, 1985-1986.

[Vacc89] H.S. Vaccaro and G.E. Liepins, "Detection of Anomalous Computer Session Activity," Proceedings of the IEEE Symposium on Security and Privacy, May 1989.

[Webs86] Webster's Ninth New Collegiate Dictionary, Merriam-Webster Inc., Springfield, Mass. 1986.

[Weis81] Mark Weiser, "Program Slicing," CH1627-9/81/0000/0439, IEEE, 1981.

[Wetm92a] Brad R. Wetmore, conversations with Gregory Elkinbard of Sun Microsystems during UC Davis' workshop on "Future Directions in Computer Misuse and Anomaly Detection," Davis, CA, April 2, 1992.

[Wetm92b] Brad R. Wetmore, conversations with Teresa Lunt of SRI International during UC Davis' workshop on "Future Directions in Computer Misuse and Anomaly Detection," Davis, CA, April 2, 1992.

[Wetm92c] Brad R. Wetmore, conversations with L. Todd Heberlein of UC Davis, Davis, CA, October 1992.

[Wetm92d] Brad R. Wetmore, conversations with Don Stephenson of Sun Microsystems, Milpitas, CA, July 1992.

10 Appendix 1 – Output of "ab"

```
Processes:

Process:  4408
  began at: Mon Nov  2 23:01:11 1992
  exited at:  Mon Nov  2 23:01:14 1992
  auid = insider  ruid = insider  euid = insider  egid = staff
  ---Files Exec'd---
  execve(2):  /home/insider/change_mod(100700) at Mon Nov  2 23:01:11
1992
  ruid = insider  euid = insider  egid = staff
  ---Files Read---
  open(2):read  /home/insider/change_mod at Mon Nov  2 23:01:11 1992

    ...output deleted...

      Process:  4412
        began at: Mon Nov  2 23:01:12 1992
        exited at: Mon Nov  2 23:01:14 1992
        ruid = insider  euid = insider  egid = staff
        ---Files Exec'd---
        execve(2):  /usr/ucb/rdist(104751) at Mon Nov  2 23:01:12 1992
        ruid = root  euid = insider  egid = staff
        ---Files Written---
        chmod(2):  /usr/bin/sh at Mon Nov  2 23:01:14 1992
        ruid = root  euid = insider  egid = staff

      ...output deleted...

Process:  4421
  began at: Mon Nov  2 23:01:17 1992
  exited at: Mon Nov  2 23:01:32 1992
  ruid = insider  euid = insider  egid = staff
  ---Files Exec'd---
  execve(2):  /usr/bin/sh(104777) at Mon Nov  2 23:01:17 1992
  ruid = root  euid = insider  egid = staff

      ...output deleted...

      Process:  4423
        began at: Mon Nov  2 23:01:29 1992
        exited at: Mon Nov  2 23:01:19 1992
        ruid = root  euid = insider  egid = staff
        ---Files Exec'd---
        execve(2):  /usr/bin/chmod(100755) at Mon Nov  2 23:01:29 1992
        ruid = root  euid = insider  egid = staff
        ---Files Written---
        chmod(2):  /usr/bin/sh at Mon Nov  2 23:01:30 1992

      ...output deleted...

File System:

file_system_id = 1798  node_id = 2441
filename = /usr/bin/sh
Access History:
  chmod(2):  (file attributes 100755) at Mon Nov  2 23:01:14 1992
    process = 4412  ruid = root  euid = insider  egid = staff
```

```
execve(2):  (file attributes 104777) at Mon Nov  2 23:01:17 1992
  process = 4421  ruid = root  euid = insider  egid = staff
chmod(2):  (file attributes 104777) at Mon Nov  2 23:01:30 1992
  process = 4423  ruid = root  euid = insider  egid = staff
```