SRI International

Final Technical Report • February 28, 1992

A REAL-TIME INTRUSION-DETECTION EXPERT SYSTEM (IDES)

Prepared by:

Teresa F. Lunt, Ann Tamaru, Fred Gilham,
R. Jagannathan, Caveh Jalali, Peter G. Neumann
Computer Science Laboratory

Harold S. Javitz, Information Management and Technology Center
Alfonso Valdes, Applied Electromagnetics and Optics Laboratory
Thomas D. Garvey, Artificial Intelligence Center

SRI Project 6784

Prepared for:

Mr. Robert Kolacki, Code 3421
SPAWAR
Washington, DC 20363-5100

Contract No. N0003S89-C-0050

Abstract

SRI International has designed and developed a real-time intrusion-detection expert system (IDES). IDES is a stand-alone system that observes user behavior on one or more monitored computer systems and flags suspicious events. IDES monitors the activities of individual users, groups, remote hosts and entire systems, and detects suspected security violations, by both insiders and outsiders, as they occur. IDES adaptively learns users' behavior patterns over time and detects behavior that deviates from these patterns. IDES also has a rule-based component that can be used to encode information about known system vulnerabilities and intrusion scenarios. Integrating the two approaches makes IDES a comprehensive system for detecting intrusions as well as misuse by authorized users. IDES has been enhanced to run under GLU, a platform supporting distributed, parallel computation; GLU enhances configuration flexibility and system fault tolerance.

This final report is a deliverable item for work supported by the U.S. Navy, SPAWAR, which funded SRI through U.S. Government Contract No. N00039-89-C-0050.

Contents

List of Figures

Chapter I

Introduction

Existing security mechanisms protect computers and networks from unauthorized use through access controls, such as passwords. However, if these access controls are compromised or can be bypassed, an abuser may gain unauthorized access and thus can cause great damage and disruption to system operation.

Although a computer system's primary defense is its access controls, it is clear from numerous newspaper accounts of break-ins, viruses, and computerized thefts that we cannot rely on access control mechanisms in every case to safeguard against a penetration. or insider attack. Even the most secure systems are vulnerable to abuse by insiders who misuse their privileges, and audit trails may be the only means of detecting authorized but abusive user activity.

Other modes of protection can be devised, however. An intruder is likely to exhibit a behavior pattern that differs markedly from that of a legitimate user. An intruder masquerading as a legitimate user can be detected through observation of this statistically unusual behavior. This idea is the basis for enhancing system security by monitoring system activity and detecting atypical behavior. Such a monitoring system will be capable of detecting intrusions that could not be detected by any other means, for example, intrusions that exploit *unknown* vulnerabilities. In addition, any computer system or network has known vulnerabilities that can be exploited by an intruder. However, it is more efficient to detect intrusions that exploit these known vulnerabilities through the use of explicit expert-system rules than through statistical anomaly detection.

While many computer systems collect audit data, most do not have any capability for automated analysis of that data. Moreover, those systems that do collect audit data generally collect large volumes of data that are not necessarily security relevant. Thus, for security analysis, a security officer (SO) must wade through stacks of printed output of audit data. Besides the pure tedium of this task, the sheer volume of the data makes it impossible for the security officer to detect suspicious activity that does not conform to a handful of obvious intrusion scenarios. Thus, the capability for automated security analysis of audit trails is needed.

The Intrusion-Detection Expert System (IDES) is theresult of research that started in the Computer Science Laboratory at SRI International in the early 1980s and that led to a series of IDES prototypes. The current prototype, described in this final report, is designed to operate in real time to detect intrusions as they occur. IDES is a comprehensive system that uses innovative statistical algorithms for anomaly detection, as well as an expert system that encodes known intrusion scenarios [l, 2, 3, 4, 5, 6, 7, 8, 9]. One version of the prototype is running at SRI and monitoring a network of Sun workstations. Another version is running at the FBI and is monitoring an IBM mainframe.

We have also conducted an SRI IR&D project to investigate the application of model-based reasoning to intrusion detection [10].

1.1 Earlier Work

One of the earliest works on intrusion detection was a study by Jim Anderson [11], who suggested ways of analyzing computer system audit data. His methods use data that are collected for other reasons (e.g., performance analysis) and were designed for "batch" mode processing; that is, a day's worth of audit data would be analyzed at the end of the day.

Subsequent to Anderson's study, early work focused on developing procedures and algorithms for automating the offline security analysis of audit trails. The aim of such algorithms and procedures was to provide automated tools to help the security officer in his or her daily assessment of the previous day's computer system activity. One of these projects, conducted at SRI, used existing audit trails and studied possible approaches for building automated tools for audit trail security analysis [12]. This work involved performing an extensive statistical analysis on audit data from IBM systems running MVS and VM. The intent of the study was to develop analytical statistical techniques for screening computer system accounting data to detect user behavior indicative of intrusions. One result of this work was the development of a high-speed algorithm that could accurately discriminate among users, based on their behavior profiles.

Another such project, led by Teresa Lunt at Sytek [13], considered building special security audit trails and studied possible approaches for their automated analysis. These projects provided the first experimental evidence that users could be distinguished from one another through their patterns of use of the computer system [12], and that user behavior characteristics could be found that could be used to discriminate between normal user behavior and a variety of simulated intrusions [14, 15, 16, 17, 18, 19].

In addition, the Sytek work sought to provide a feasibility demonstration for a tool that would rank the detected unusual activity by its suspiciousness [13]. Sytek's work was guided by concepts from pattern recognition theory. Each user session was recognized as normal or intrusive, based on patterns formed by the individual records on the audit trail for that session. The Sytek study defined several audit record *features* as functions of the audit record fields. For each user, expected values for the features were determined through a process called *training* (that is, for each feature, the set or range of values was determined from the audit data). The study then tested the features for their ability to discriminate between normal sessions and sessions containing staged intrusions. Features that successfully detected the staged intrusions were combined to create for each user a *user profile* - the set of normal values for each feature.

The evidence of the early Sytek and SRI studies was the basis for SRI's real-time intrusion-detection system, that is, a system that can continuously monitor user behavior and detect suspicious behavior as it occurs. This system, IDES, is based on two approaches: (1) intrusions, whether successful or attempted, can be detected by flagging departures from historically established norms of behavior for individual users, and (2) known intrusion scenarios, known system vulnerabilities, and other violations of a system's intended security policy (i.e., *a priori* definition of what is to be considered suspicious) are best detected through use of an expert-system rule base.

Largely as a result of the pioneering work at SRI, several other intrusion-detection projects are under way at other institutions. For a survey of these, see [20].

1.2 IDES Overview

IDES runs independently on its own hardware (Sun workstations) and processes audit data received from one or more target systems via a network. IDES is intended to provide a system-independent mechanism for real-time detection of security violations, whether they are initiated by outsiders who attempt to break into a system or by insiders who attempt to misuse their privileges. IDES is based on the intrusion-detection model developed at SRI [21, 22]. This model is independent of any particular target system, application environment, level of audit data (e.g., user level or network level), system vulnerability, or type of intrusion, thereby providing a framework for a general-purpose intrusion-detection system using real-time analysis of audit data.

The IDES prototype determines whether user behavior as reported in the audit data is normal with respect to past or acceptable behavior as represented by a user's historical profile of activity. The IDES prototype continually updates the historical profiles over time, using the reported audit data to learn the expected behavior of users of the target systems. IDES raises an alarm when a user's current observed activity deviates significantly from the user's historical profile. The prototype provides mechanisms for summarizing and reporting security violations as well as for detecting intrusions that cannot be detected by the access controls (e.g., because they circumvent the controls or exploit a deficiency in the system or its security mechanisms).

IDES employs several approaches toward detecting suspicious behavior. IDES attempts to detect masqueraders by observing departures from established patterns of use for individuals. It does this by keeping statistical profiles of past user behavior. IDES also includes expert-system rules that characterize certain types of intrusions. IDES raises an alarm if observed activity matches any of its encoded intrusion scenarios.

IDES also monitors certain system-wide parameters, such as CPU usage and failed login attempts, and compares these with what has been historically established as "usual" or normal for that facility. The IDES security officer interface maintains a continuous display of various indicators of user behavior on the monitored system. When IDES detects an anomaly, it sends to the screen a message indicating the cause of the anomaly.

Each target system must install a facility to collect the relevant audit data and put them into IDES's generic audit record format. We have developed a flexible format for the audit records that IDES expects to receive from the target system and a protocol for the transmission of audit records from the target system. The IDES protocol and its audit record format are system independent; our intent is that IDES can be used to monitor different systems without fundamental alteration.

IDES monitors target system activity as it is recorded in audit records generated by the target system. IDES maintains profiles for subjects. A profile is a description of a subject's normal (i.e., expected) behavior with respect to a set of intrusion-detection measures. The subjects profiled by IDES are users, remote hosts, and target systems. Profiling remote hosts allows IDES to detect unusual activity originating from a particular outside system, whether or not the user activity seems abnormal. Profiling a target system enables IDES to detect system-wide deviations in behavior. Again, such deviations may not be attributable to a single user. For example, the number of system-wide login attempts may be indicative of an intrusion attempt, although they might not all be attributable to a single user ID. We also plan to profile groups of subjects, which enables IDES to detect when the behavior of an individual member of a group deviates from the overall average behavior of the group. For example, IDES can detect when a secretary is not behaving like a "typical" secretary.

The profiles are updated daily, based on the observed behavior of the users of the target system. Thus, IDES adaptively learns subjects' behavior patterns; as users alter their behavior, the profiles change.

IDES also includes an expert-system component. The expert system contains rules that describe suspicious behavior that is based on knowledge of past intrusions, known system vulnerabilities, or the installation-specific security policy. The rules describe suspicious behavior that is independent of whether a user is deviating from past behavior patterns. Thus, IDES is also sensitive to known or posited intrusion scenarios that may not be easily detectable as deviations from past behavior. The rules can be used to encode information about known system vulnerabilities and reported attack scenarios, as well as intuition about suspicious behavior. Audit data from the monitored system are matched against these rules to determine whether the behavior is suspicious. Although it is not reasonable to expect that all possible intrusion scenarios can be foreseen and encoded in IDES's rule base, the combination of the rule-based component and the statistically based component should have greater detective power than either component by itself.

1.3 Progress

The patterns of use for users of Sun workstations are different from patterns of use for users of a centralized computing resource using computer terminals. Sun users typically have several windows open simultaneously, many of which may be running UNIX shells independent of the other windows. Unlike the TOPS-20 environment that the IDES prototype monitored previously, there is no concept of user session, because users may open or close UNIX shells or other windows without logging in or out, and may leave these windows activated for extended periods (days or weeks) without ever logging out of the workstation. A user typically has "superuser" privileges on his own workstation, but usually does not have such privileges on other workstations. Users in a networked Sun facility have more remote logins, remote procedure calls, and file transfers than users of a centralized computer. In addition, patterns of abuse may differ significantly because users typically have to remotely log in to another workstation or use remote procedure calls to access sensitive resources, which are more likely to be associated with a machine other than the user's own. Thus, we have included suitable measures of behavior, for users of a networked Sun facility, that capture any given user's distinctive behavior pattern and that are also useful in discriminating between normal and potentially abusive behavior. We have made modifications to IDES to enable it to integrate data from numerous sources for any given user, allowing us to track the activity of a user who is remotely accessing other workstations or who moves from workstation to workstation.

Based on preliminary experiments and tests using IDES on the TOPS-20 target system, we found many areas in which IDES's statistical algorithms could be improved. We tested an earlier version (Release 1) of our statistical algorithms using data collected from our now-retired TOPS-20 machine. (Release 1 was the "improved" version that monitored TOPS-20. It was our second TOPS-20 prototype; the first was a much simpler version delivered on tape to SPAWAR in 1988.) The Release 1 algorithms made certain assumptions about the way usage patterns would be reflected in audit data. For example, in many cases we assumed that the data were multivariate normally distributed after a suitable transform, and thus we had simple procedures that merely looked for variances greater than *N* standard deviations from the mean. After testing these algorithms against the TOPS-20 data, we found that these assumptions resulted in an unacceptably high false alarm rate. As a result, we decided to redo the algorithms to remove some of these assumptions. In the process, we have completely redesigned and reimplemented the statistical component of IDES to accommodate the new algorithms and to make this component easier to modify in the future.

We have completely redesigned IDES, making it highly modular. Previous versions of IDES were much simpler and involved a programming staff of only one or two people. The current version of IDES is by comparison quite complex. It includes the capability of monitoring heterogeneous target machines (e.g., UNIX and IBM mainframe) simultaneously, as well as the capability to simultaneously monitor multiple homogeneous targets. It is also capable of monitoring several sources of audit data from any single machine. It is intended to be extensible, in that it will be easy to add additional analysis components. It uses more robust statistical algorithms, an expert- system component, and a more sophisticated user interface. To support the development of this new version, the IDES team has grown to a size of eight people. The original design had evolved by accretion. We felt that it was time to start over and develop a modular design that would

1. Facilitate communication among team members
2. Isolate a given programmer's responsibility to a clearly defined system layer or component
3. Remove low-level dependencies from high-level IDES functionality
4. Provide a framework for the gradual removal of dependence on a database management system
5. Allow enforcement of proven software engineering practices

We have also put a substantial amount of effort into the IDES security officer interface. We have developed a library of graphical routines that can be used to facilitate quick assembly of a graphical user interface. We have moved from the Sun windowing system (SunViews) to the X Window System, which is emerging as a *de facto* standard windowing system.

We have reassessed our use of the ORACLE database management system in IDES. After reviewing the new design approach and running several benchmark experiments, we have come to believe that there is a mismatch between the requirements of the online IDES processor and the capabilities of ORACLE. For this reason we have decided to stop using ORACLE for the online IDES processor. There is a possibility that ORACLE or another database system can be used for the purposes of providing query capability, both for the security officer, using "canned" queries, and for the data analyst, who will be able to make ad-hoc queries over the audit data.

Our primary consideration in the decision to abandon ORACLE for the online IDES processor was that ORACLE did not allow us to retrieve data fast enough. We felt that there was a lot of mechanism in ORACLE that we did not need, and this was the major factor in the poor performance. We found that using a UNIX file-based implementation provided adequate performance.

One concern that we had was the issue of concurrent access to the database. ORACLE provides a very general mechanism of row-level locking, allowing concurrent access by both reading and writing processes. We originally planned to make use of this facility, and when we decided not to use ORACLE, the question arose as to how much effort we would have to spend implementing our own concurrency mechanism. By analyzing the patterns of access, we discovered that we would, in the worst case, have only read-write concurrency, and never write-write concurrency. This worst-case situation would occur only once a day, at profile update time. We also found that reader-reader concurrency would be infrequent (occurring only when a security officer or data analyst wanted to view a subject's profile on the fly). On this basis we felt that a caching process coupled with a simple protocol to update the cache as needed could mediate access to user profile data.

We have also integrated an expert-system tool into IDES and have developed an expert rule base for IDES. The rule base encodes known intrusion scenarios and known system vulnerabilities. The tool provides a rule-specification language, a translator that produces the actual expert-system code, and run-time libraries to support all the capabilities of the expert system. It also provides the ability to create a debugging version of the expert system. This tool was developed in house. The rule debugging facility helps the person who is developing the rule base to test and debug the candidate rules. We envision that the IDES security officer will be adding and modifying rules in the IDES rule base and will make extensive use of the rule debugging facility. A rule is analogous to a software procedure or subroutine, and must be tested and debugged to ensure that it behaves as expected under a variety of input data conditions.

We have implemented the expert-system portion of IDES, and we are analyzing data from the FBI, using a rule base we are developing in conjunction with the FBI that is specific to its application. In addition, we have developed expert-system rules specific to Sun UNIX.

In the process of moving toward monitoring a network of Sun workstations, we evaluated the audit data collected by the C2 Sun UNIX system, and we made some changes to the specific data collected so as to obtain the data we felt were most meaningful while significantly reducing the amount of data to be analyzed.

We have developed a new core technology, which we call GLU (Granular Lucid), that will allow IDES to be easily scalable to many processors. From a design standpoint, GLU allows us to exploit the natural parallelism present in the IDES processing of audit data, and encourages a highly modular design. GLU also gives IDES increased fault tolerance, because it automatically reschedules to another processor tasks that were active on a processor that failed. We have tested GLU by implementing a portion of the IDES software on it, and have demonstrated its ability to transparently distribute IDES processing among several Sun workstations and to redistribute IDES processing when one of the workstations goes down. We plan to use GLU to allow us to easily migrate to the Next Generation IDES.

Chapter 2

The IDES Design Mode1

The original version of IDES was designed and developed to support a single target host. We have redesigned its architecture so that it can be applied to any number of heterogeneous target systems, as well as provide a platform for experimental intrusion-detection techniques and various implementation methods.

The design model we use is divided into four parts: the target system domain, the realm interface, the processing engine, and the user interface. Figure 2.1 is a diagram of the model.

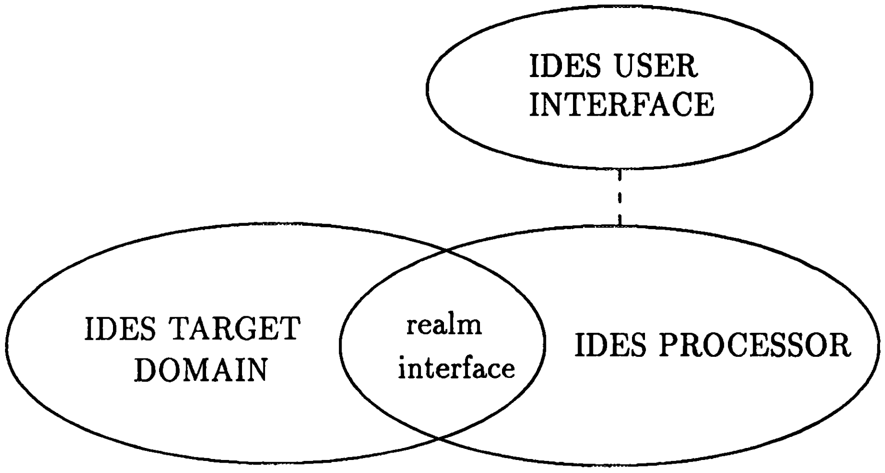


Figure 2.1: IDES Design Model

In the IDES environment, a *realm* is a group of similar target machines that are being monitored; “similar” means that the machines produce audit trails that have the same format. For example, in our prototype, the network of Sun workstations that we monitor consists of one realm type. In addition to providing the target audit trails, each realm interface may also support some authentication of the audit data to verify the source from which it came.

The IDES *target domain* consists of a set of realms that are being monitored for anomalous activity. Because there may be a number of monitored hosts, there can be more than one centralized point of audit data collection and collation for a set of target systems as well as for realms.

The IDES *processor* is the computing environment that is responsible for the analysis of the information acquired from the IDES domain. IDES audit records are passed to the various analytical engines, also known as *event subsystems.* All IDES event subsystems process events on a per-audit record basis (or groups thereof). Currently, we have implemented two anomaly-detection event subsystems, one based upon statistical methods and the other a rule-based system. However, the design model does not limit subsystems to provide only detection of abnormal behavior (e.g., a subsystem could be designed to gather statistics for general target-system usage).

The IDES *realm interface* is the bridge that connects the IDES domain and the IDES analytical components. This interface has two pieces: the part that resides within the target system (the *realm client)* and the other one local to the IDES processor (the *realm server).* Because various realm types have their own audit trail formats, a different realm interface must be developed for each realm type. Together, these two components are responsible for collecting the target system activities for the IDES analytical engines, and for filtering out any realm-private data and setting them aside for specific uses as needed.

For the sake of discussion, we will assume that the realm clients and server exist on different systems on the same network, although the architecture is not restricted to this configuration. Currently, each realm client converts audit records into a generic IDES audit record format. Using a simple data transfer protocol, each realm client establishes a connection with the realm server whereby the target system's audit information is transported across the network. The realm server stores the data, which it makes available to the event subsystems as they indicate that they are ready for it. Note that there is no definite distribution of functional responsibility between the realm server and its corresponding clients, as the implementation may vary according to the features of the target system. For example, a particular implementation is free to perform the conversion of audit records into IDES format on the IDES machine itself if this is desirable for performance reasons.

Two diagrams show how these three parts of the model are related to each other. Figure 2.2 shows a sample IDES domain containing four realms. Similar realm types may share a single realm server, as indicated in the diagram with realm B using realm A's server. Realms C and D are drawn as individual realms, having their own sets of realm server and client processes.

Next, Figure 2.3 shows the corresponding configuration for the IDES processing environment. The realm clients pass processed audit information to the event subsystems simultaneously. The diagram also shows that IDES subsystems may be organized as "phases." For example, all anomalies, regardless of severity, may be diagnosed during one phase, and then filtered at subsequent phases until the degree of abnormality is significant to the IDES end user (usually a security officer).

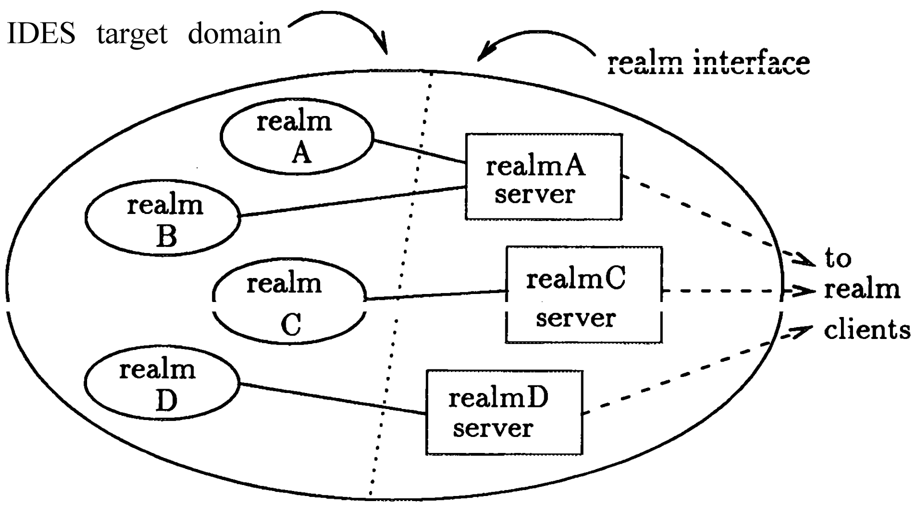


Figure 2.2: IDES Domain and Realm Interface

Note that these diagrams represent a more complex organization of an IDES environment; our current prototype is a simpler instantiation of this model, using only one realm type.

The IDES *user interface* component allows the user to view any of the information created and processed within the IDES system (i.e., the realm interface and the IDES event subsystems), as well as to observe any component's status and activity. Note that the IDES domain does not encompass the IDES user interface; we have elected to keep realm-specific data independent from the core IDES user interface, as they are generally not relevant to the basic IDES data processing. Details on the user interface design are discussed in Chapter 7.

Figure 2.4 puts the pieces of the entire IDES environment together in one diagram. It shows the data processed by each component. The diagram also depicts where the IDES user interfaces are incorporated into the system.

The IDES design model naturally lends itself to be engineered as a modular system, where the implementation of each component can be changed without requiring major

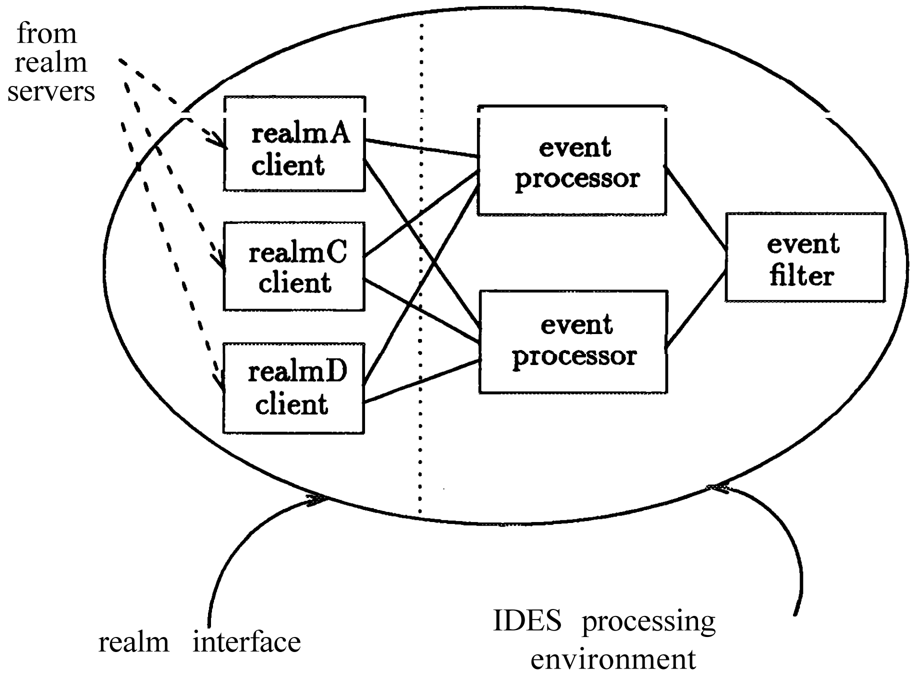


Figure 2.3: IDES Realm Interface and Processing Environment

changes to the overall system. It essentially allows us to separate the infrastructure of IDES (methods of implementation) from the core functionality of the system (anomaly detection).

The IDES system is currently composed of the following functional components:

- Realm Interface
- Statistical Anomaly Detector
- Expert System Anomaly Detector
- User Interface

Each of these components is implemented as an individual process, so as to exploit distributed and parallel processing techniques to provide intrusion detection in as close to real time as possible. Details on each component are described in subsequent sections of this report.

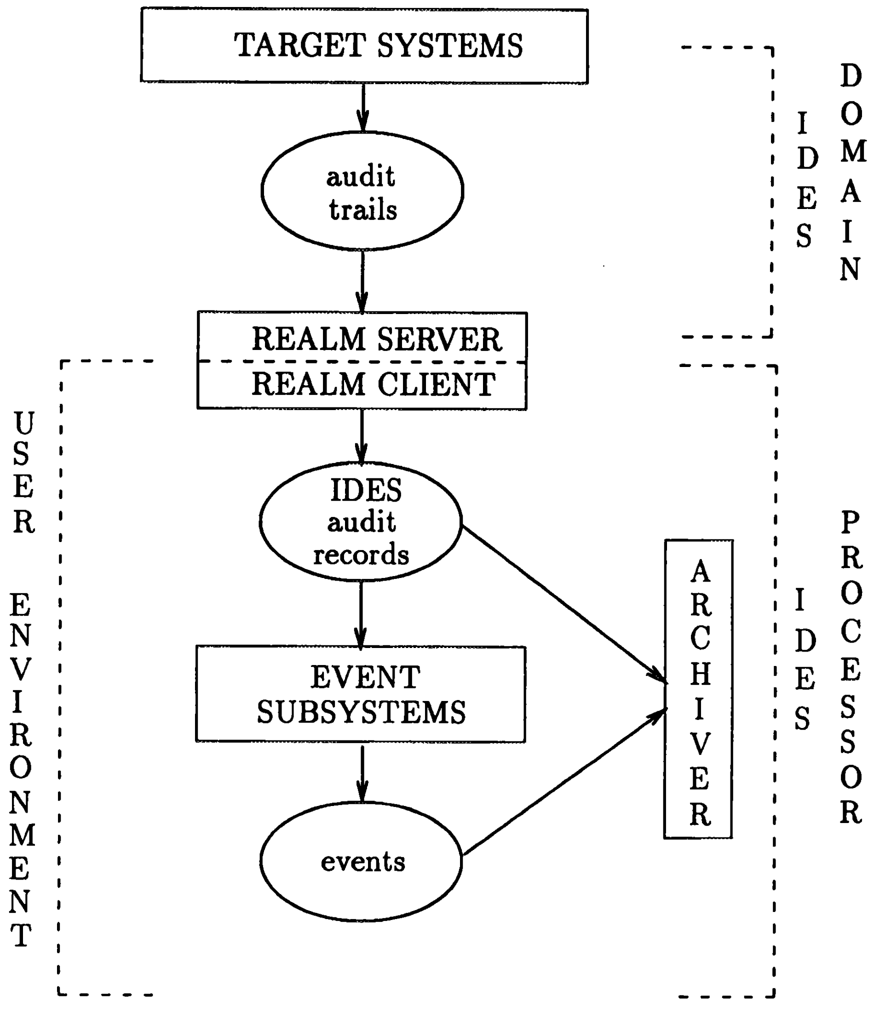


Figure 2.4: IDES General Functional Diagram

Chapter 3

The Audit Data

The audit data described in this chapter are to the "raw" audit information supplied by the target system. In particular, we discuss the audit data collected for IDES from a Sun UNIX target system environment.

3.1 Types of Audit Data

Our intent in audit data collection is to gather as much information about a running system as is practical. Typically, this information falls into four general categories:

- *File Access.* Includes operations on files and directories such as reads, writes, creations, deletions, and access list modifications.
- *System Access.* Includes logins, logouts, invocation/termination of superuser privileges, and so forth. In general, any activity that requires a password falls into this category.
- *Resource Consumption.* Includes CPU, I/O, and memory usage. We are currently able to collect this information on a per-process basis, although it is available only after a process has terminated. It is possible to obtain this information at some arbitrary interval from an executing process; however, this would involve extra programming effort and can increase the load on the target system, because it requires polling the processes and scanning various kernel data structures with every poll. For these reasons, we have not tried to acquire resource consumption information for executing processes.
- *Process Creation/Command Invocation.* Indicates the creation of a process. This information is usually available after invoking a command.

IDES analytical units are capable of handling other potential types of data, but currently we do not monitor for them because substantial Sun source code modification would be required or because the collection of such data would encumber the target system itself, resulting in significant degradation of performance. These types of data include

- *Electronic Mail Traffic.* This would require monitoring the source and destination address of all incoming and outgoing mail on a user-by-user basis.
- *Dynamic Resource Usage.* This would require monitoring the resource consumption (CPU usage) of every process on the system to determine its resource usage pattern while it is executing. UNIX generally logs this information only when a process exits.

We are considering the possibility of monitoring the target system for these potential data for future versions of IDES, and we hope to find a way to collect such data without significantly affecting the performance of the target system.

3.2 Generation of Audit Data on Sun UNIX

To provide the IDES analytical components with enough information to perform adequate intrusion-detection analysis, we had to gather audit data from several different sources on the Sun environment. These sources include

- SunOS 4.0 standard auditing system
- Sun C2 security auditing package[1]
- UNIX accounting system

The standard SunOS 4.0 system collects all audit information in log files. Each host writes to exactly one audit file, uniquely defined for that host. These files grow in size as new audit records are appended to them. Furthermore, at selected intervals (set by the Sun system administrator), the system closes an audit file and begins writing to a new one. Previously used audit files remain until they are deleted manually by the system administrator (or by a privileged automated process).

The audit data for IDES are also obtained from the Sun C2 security package. With systems running version SunOS 4.0 and later, the target system can be configured with

[1] "C2" refers to evaluation class C2 of the *Department of Defense Trusted Computer System Evaluation Criteria [23]*

this package, enabling each target machine to record selected system calls into an audit log file.

However, the information obtained solely from these two auditing systems is not sufficient for IDES intrusion-detection analysis. For example, in order to obtain CPU usage statistics, we must also poll various UNIX accounting files. At present, only one such file is being monitored (*/var/adm/acct* for Sun 3 workstations or */var/adm/pacct* for Sun SPARCstations). In the future, it may be profitable to monitor other files as well. For example, one such source will provide a detailed record of all mail exchanges for a system and thus for each of its users.

3.3 Sun Source Code Bug Fixes

While it was our intent to avoid changing source code not under our control, such as SunOS functions, several instances were encountered where we felt changes had to be made in order to provide sufficient data for IDES intrusion-detection analysis. In most cases, these were bug fixes to SunOS that were necessary in order to obtain the information specified in Sun's documentation. This had to be done by CSL because Sun could not accommodate our needs in time for us to get a working IDES system available for the project. The following list shows the SunOS source files that were modified and the bugs that were fixed:

1. *bin/login.c.* This is the basic login mechanism that prompts the user for an account name and a password. The bugs corrected were as follows:

   - The C2 audit record was filled in with the user name used to invoke */bin/login* instead of the command "login". This made it impossible to identify the audit record. The audit record is now filled in with the proper information.
   - The name of the physical port reported was incorrect or missing, and was fixed.
   - The user name for an attempted (failed) login was not reported if this name did not appear in the password file. This name is useful for detecting login attacks, and hence was added to the code.

2. *bin/passwd.c.* This is the UNIX password changer for local passwords. The C2 audit record was filled with the user name in the type field instead of the string "passwd". This made it impossible to identify the audit record. The audit record is now filled in with the proper information.

3. *bin/su.c.* The *su* command allows users to change their user IDs if they know the passwords of the user IDs they wish to assume. The C2 audit record was filled

with the name used to invoke */bin/su* instead of "su". This made it impossible to identify the audit record. The audit record is now filled in with the proper information.

4. *usr.etc/rexd/unix_1ogin.c. Rexd* is a daemon that allows the remote execution of commands using the "on" command. The audit flag of the remote process was incorrectly changed to full auditing, and this was fixed.

5. *usr.etc/rpc.mountd.c. Mountd* is part of the NFS service and provides the initial contact for clients to mount directories using NFS. C2 auditing is set to "off" by default when a process starts at system boot time. We changed this to enable C2 logging by default instead. We also added C2 and *syslog* auditing of failed mount requests, as originally such requests were not logged. Now, any failed attempts of mounting a directory are logged using both the *syslog(3)* and C2 audit mechanisms.

6. *usr.etc/in.ftpd/ftpd.c. Ftpd* is the server part of the ftp service and is responsible for providing access to local files for a remote user. We added the capability of logging C2 audit records for logins and attempted logins that failed because of a bad password.

7. *usr.etc/rpc.pwdauthd.c. Pwdauthd* performs password checking on C2 systems because the encrypted passwords are hidden on C2 systems. *Authd* now returns "success" only if the password check request originated on the local machine; otherwise, "failure" is returned, and a l-second time delay occurs before the next request is processed. Failure is returned so that the cracker will never know if he has guessed a password. The l-second delay is intended to prevent outsiders from loading our CPU by generating a large number of requests. In addition, logging and C2 auditing were added.

The following problems with C2 auditing that could not be resolved with only minimal changes to the SunOS source code:

1. *Yppasswdd* seems to be started before auditing is enabled when a system boots; thus, it is not audited. This is actually a problem with the *audit-uid* and *audit-state* values. These seem to be set to a "disabled" value at boot time, so daemon processes are not audited unless they explicitly set these values.

2. During audit file switching, */usr/etc/security/audit_data* seems to be updated before the new audit file is created and after the trailer mark has been placed in the old audit file. This created an annoying problem for applications that must read these files as a continuous stream of audit records over time.

Chapter 4

The Realm Interface

The IDES realm interface defines the interface between IDES and the target systems to be monitored. It is assumed that each target system is somehow capable of providing "raw" audit data to IDES. The realm interface is responsible for accepting the raw data, converting them to a standard IDES audit record format, and temporarily storing them (if necessary) until the IDES analytical components can process them. The realm interface consists of two main components: a target system component (called *agen)* and an IDES component (called *arpool).* The general functionality of these components is described in the first few sections, followed by implementation details.

4.1 IDES Audit Record Generator *(agen)*

Agen is a utility that resides on the target machine (the one being monitored). It collects raw audit data from several sources on the target machine (see Section 3.2) and translates the system's native audit record format into a canonical format that IDES can process. For example, IDES has a notion of a user logging into a machine, and needs to know that such an event has occurred. It is up to *agen* to decide what constitutes a login on a particular target machine. In the UNIX environment, every time a *(username, password)* pair is entered, a login has occurred. This could be through either a login from a terminal, a remote login (rlogin) over the network, or perhaps by means of an FTP session.

Currently, each audit file is polled by *agen* at discrete intervals to see if the target system has added audit data to it since the last poll. Each time new records are discovered, *agen* reads batches of unprocessed audit records from the file, preprocesses them into IDES format, and sends them to *arpool.* If no new records have been added, *agen* "sleeps" for a short time before polling the files again.

This scheme provides several benefits. First, the target system automatically provides the buffering of unprocessed audit records. Second, natural grouping (batching) of audit records can take place. If IDES should fall behind in processing audit records, then many new audit records can accumulate by the time the log files are polled again. The software recognizes this situation and compensates by reading a group of audit records the next time the log files are polled. In doing so, the average overhead of processing each audit record is reduced, thus allowing IDES to work more efficiently in times of high demand, while still providing good response times when very few audit records need to be processed.

Some effort is also made to sort audit records by time. Due to the unpredictability of UNIX scheduling, it is conceivable that one source of audit records may grow more quickly and appear to be "ahead" of another source in that it contains audit records that are newer than those in other files. By merge-sorting audit records from different sources, we can greatly limit this phenomenon.

Agen can be considered the system-dependent part of the realm interface. It is sufficient to rewrite only the *agen* component to be able to monitor a new type of target system. There is considerable flexibility in its implementation. For example, under UNIX, it is desirable and practical to monitor and report audit records in real time, and hence *agen* should be implemented so that IDES audit records are passed to the IDES processing environment as events occur on the target system. However, it is also possible to pass audit records in batch mode, where previously collected audit data are processed by *agen* and passed on to the IDES side of the realm interface. The latter is suitable for replaying audit data for backtracking analysis, or for IDES experimentation.

4.2 Audit Record Pool *(arpool)*

The interface component of IDES to which monitored machines send their audit records is the *arpool* process (also known as the *envelope).* It is the clearing house for IDES audit records. *Arpool* resides on the server side of the realm interface, that is, its purpose is to accept IDES-formatted audit records from multiple target machines and serialize them into a single stream. It then does a little further processing of the data, such as timestamping each IDES audit record and assigning a unique sequence number to each one. Only one *arpool* process is required for a single IDES system.

The purpose for creating a single stream of IDES audit records is to allow IDES to repartition the input stream in any way it chooses to achieve the best parallelism in processing the records. In particular, GLU is capable of parallelizing the processing of this input stream (see Chapter 8). Furthermore, in a networked target environment, a user may be active on several machines, and thus it is likely for IDES to receive audit records for the same user from different target machines. To relate all the activities to that one user, it is necessary to merge audit records from multiple sources into a single stream.

Arpool also functions as a temporary store (cache) of audit records waiting to be processed by IDES. Since the collection and preprocessing of audit data is much faster than the speed with which the analytical components can process the data, *arpool* can regulate the audit data traffic. This feature also provides some degree of data integrity, as it can store a substantial amount of IDES audit information in case one or more of the IDES analytical components is temporarily down and unable to process the incoming records.

4.3 IDES Audit Record Design

The IDES audit record format is subject to several design considerations. First, it must be general enough to be able to represent all possible kinds of events for the system(s) being monitored. Second, it should be in a machine's most efficient data representation to allow processing with a minimal amount of representation overhead. For example, if IDES needs to do numerical operations on integers, these values should be represented as integers and not as ASCII strings. Likewise, the byte conversion issue should be resolved before the audit record is sent to IDES. In other words, if the target machine uses a native representation different from that used by the IDES processor, then the target machine should do the data conversion. Third, the format should be standardized so that IDES can accept input from a variety of machine types without performing any data conversion. Ideally, audit records should be formatted only once. Since the IDES audit record is always crafted from a system's native audit record format, it makes sense to generate the IDES audit record at the source (target system), so that no further data conversions are necessary once the audit record has been accepted by the IDES processing environment.

IDES audit records are classified by action types. There are about 30 different action types, such as file reads, file writes, and logins. These actions enumerate the types of events that IDES has been designed to monitor. They are predetermined and cannot be changed without modifying IDES itself. However, in order to accommodate the possibly limited auditing capabilities of different target systems, there is considerable flexibility in qualifying each IDES action.

Each IDES audit record consists of an action type with several fields parameterizing that action. Some of these fields are always defined, such as the timestamp of the audit record, a subject ID, and target hostname. This mandatory collection of data forms the fixed part of the IDES audit record. There is also a variable part of the IDES audit record that contains information such as file names, directory names, or other appropriate information, depending on the action type.

The functionality of IDES is only as good as the information it receives. Thus, it is desirable to supply as much information as possible with each audit record. IDES recognizes the fact that not all target systems can provide all the information that IDES would ideally like to receive, so it is reasonable for IDES to ignore some fields in the IDES audit record if the information is not generated by the target system.

As a rule of thumb, all fields in the IDES audit record that can be filled with some relevant information should be filled. If no data are available for some fields to be filled, then these fields should be set to some distinct default value (either 0 or a null string, as the case may be). For example, UNIX is not capable of generating the CPU-time used by a system call; thus, the CPU utilization field in the audit records is set to 0, whereas other target systems that do maintain such information would set this field to the appropriate value.

On the other hand, not all detectable events need be reported to IDES. For example, under SunOS with C2 enabled, we have chosen to not report *stat(2)* system calls to IDES for two reasons. First, these system calls occur very frequently and are usually redundant, and are thus not especially useful for intrusion detection. Second, the volume of *stat(2)* calls is so large that considerable performance benefits are realized by ignoring them.

4.4 Implementation

The following subsections discuss the implementation details of the realm interface.

4.4.1 Communication between *Agen* and *Arpool*

Arpool accepts audit records using SUNRPC. Our use of SUNRPC expands on the simple RPC paradigm by allowing the RPC server to accept multiple remote procedures before replying to any of them. This enables us to easily implement flow control in *arpool.*

Agen and *arpool* communicate with each other using SUN's RPC mechanism. This allows the implementation of a client/server model for communication between *arpool* and *agen* with *arpool* as the server. We have chosen this protocol for transferring audit records to IDES for its generality of implementation. Using RPC, many of the implementation details of data transfer have been hidden, leaving us with a relatively high-level abstraction for data transfer. SUN's RPC mechanism supports XDR (external data representation), which attempts to solve problems that might otherwise have been introduced by communication between heterogeneous machines.

However, the full XDR capabilities of SUNRPC have not been exploited because of certain limitations of the RPC language. In particular, some of the C language constructs used by IDES cannot be parsed by *rpcgen,* the RPC/XDR compiler. It was necessary to create a separate header file defining the IDES audit record in a way that *rpcgen* could understand. Since this would mean that the same data structures must be defined in two places without any way to verify their consistency, we are not currently doing this. At present, we support the Sun 3 (mc680*x*0) and Sun 4 (SPARC) architectures.

We have enhanced the standard RPC mechanism by allowing the RPC server process *(arpool* in this case) to block a client *(agen)* while continuing to service other requests, then unblock a previously blocked client. This all happens transparently to the client; the client may merely notice that the RPC took a long time. *Arpool* uses this scheme for flow control. When an *agen* process produces audit records much faster than IDES can process them, that *agen* process is blocked until IDES's analytical components can catch up. The blocking takes place by delaying the reply phase of the RPC, and thus from the client's point of view, the RPC merely took a little longer than its usual transaction time. The advantage of this scheme is that "producers" can be blocked, while the IDES "consumers" (analytical units) can continue to be served.

Arpool is capable of noticing the termination of its clients. All communication is performed using RPC over a TCP/IP channel. Failure of notification is obtained from the state of this TCP channel. If the channel closes unexpectedly, then *arpool* assumes that the remote process has terminated because of the death of that process or a machine crash. The actual reason for the channel failure is not particularly important as long as *arpool* can detect that the client has gone down.

Arpool does very little special processing upon the death of a client; however, a pseudo audit record is created to indicate the abnormal termination of an *agen* process. Typically, *arpool* removes any internal references that it might have had to that client so that all further processing can continue as if that client had never existed. When a previously failed client reconnects to *arpool,* it is treated as a new client since no relation is assumed with any of its previous incarnations.

It is conceivable and possibly even tempting to have *arpool* restart any of its clients that have terminated unexpectedly. However, it is probably the case that if a client fails once, it will fail repeatedly until the actual problem has been resolved. When the problem has been resolved, one can manually restart the failed client. If a machine crashes, clients such as an *agen* process should be started automatically as part of the boot process, thus ensuring that audit records are always reported to IDES.

4.4.2 IDES Audit Record Format

Action Types

We have defined the following action types for IDES intrusion-detection analysis:

- *IA_VOID*. This is a no-op.
- *IA_ACCESS*. The specified file was referenced (without reading/writing it).
- *IA_WRITE*. The file was opened for writing or was written.
- *IA_READ*. The file was opened for reading or was read.
- *IA_DELETE*. The specified file was deleted.
- *IA_CREATE*. The specified file was created.
- *IA_RMDIR*. The specified directory was deleted.
- *IA_XHMOD*. The access modes of the specified file have been changed.
- *IA_EXEC*. The specified command has been invoked.
- *IA_XHOWN*. The ownership of the specified object (file) has been changed.
- *IA_LINK*. A link has been created to the specified file.
- *IA_CHDIR*. The working directory has been changed.
- *IA_RENAME*. A file has been renamed.
- *IA_MKDIR*. A directory was created.
- *IA_LOGIN*. The specified user logged in.
- *IA_BAD_LOGIN*. The specified user tried unsuccessfully to log in.
- *IA_SU*. Superuser privileges were invoked.
- *IA_BAD_SU*. An attempt was made to invoke superuser privileges.
- *IA_RESOURCE*. Resources (memory, IO, CPU time) have been consumed.
- *IA_LOGOUT*. The specified user has logged out.
- *IA_UNCAT*. All other unspecified actions.
- *IA_RSH*. Remote shell invocation.
- *IA_BAD_RSH*. Denied remote shell.
- *IA_PASSWD*. Password change.
- *IA_RMOUNT*. Remote file system mount request (Network File Server).
- *IA_BAD_RMOUNT*. Denied mount request.

- *IA_PASSWD_AUTH.* Password confirmed.
- *IA_BAD_PASSWD_AUTH.* Password confirmation denied.
- *IA_DISCON. Agen* disconnected from *arpool* (pseudo-record).

IDES Audit Record Structure

Below is the C structure that represents an IDES audit record. As mentioned in Section 4.3, the record has a fixed section and a variable section. *ides_audit_block* contains the fields that are mandatory for each audit record, and *ides_audit_header* contains the variable data.

```
struct ides_audit_header (
        unsigned long           tseq;
        char                    hostname[32];
        char                    remotehost[32];
        char                    ttyname[16];
        char                    cmd[l8];
        char                    _pad1[21;
        char                    jobname[l6];
        enum ides_audit_action  action;
        time_t                  time;

        /* this section is unix specific */
        long                    syscall;
        unsigned long           event;
        long                    errno;
        long                    rval;
        long                    pid;

        /* unix: these fields are 0 in most (but not all) cases */
        struct resource_info    resource;

        enum ides_audit_type    act_type;
        long                    subjtype;
        char                    uname[IDES,UNAME,LEN];
        char                    auname[IDES,UNAME,LEN];
        char                    ouname[IDES,UIVAME,LEIV];
        long                    arglen;
};
```

A description of each of the above variables in the *ides_audit_header* structure are defined as follows:

- *tseq.* Target sequence number, which uniquely identifies this audit record within the set of audit records received from the same target. It is set by *agen.*
- *hostname.* Hostname of target machine.
- *remotehost.* If a remote host was involved, this field represents its name.
- *ttynume.* The terminal port/line used.
- *cmd.* The command invoked to cause this audit record.
- *jobname.* Batch job name (used for TOPS-20, FOIMS).
- *action.* The action that caused this audit record.
- *time.* The target timestamp. Obtained from raw audit data source or set by *agen* if raw audit data has no timestamp.
- *syscall.* The syscall number causing audit record.
- *event.* The C2 audit event mask (SunOS specific)
- *errno.* System error code.
- *rval.* System return code.
- *pid.* Process ID responsible for this audit record.
- *resource.mem.* Amount of memory used by this action.
- *resource.io.* I/O (disk) utilization of this action.
- *resource.cpu.* CPU time used by this action.
- *act_type.* Indicates whether audit record was generated by a batch job or interactive process. UNIX is always interactive.
- *subjtype.* Currently not used.
- *uname.* Name of user (subject) performing action.
- *auname.* User (subject) name being audited (e.g., this does not change when *uname* changes due to a superuser command).
- *ouname.* Other user name (e.g., new user ID set when changing from one user to another).

- *arglen.* Length of variable part of record (placed in *ides_audit_block.arg_un)*

The structure definition of *ides_audit_block* is

```
typedef struct {
        long                    ab_size;
        aud_type                ab_type;

        unsigned long           rseq;
        time_t                  rectime;

        ides-audit-header       ah;

        union ab_args {
                char                    maxbuf[AUP_USER];
                ides_path_desc          _ipd[2];
#define ab_path0        _ipd[0].path
#define ab_path1        _ipd[l].path
        } arg_un;
} ides_audit_block;
```

The above variables in the *ides_audit_block* structure are defined as follows:

- *ab_size.* Actual byte length of ides-audit-block, suitable for *malloc()*ing or *bcopy()* ing such a structure.
- *ab_type.* Audit record type. Obsolete, and no longer used.
- *rseq.* Audit record realm sequence number. This number is assigned by *arpool,* and uniquely identifies an audit record across all target systems.
- *rectime.* Timestamp generated by *arpool* indicating the time when the audit record was accepted by IDES.
- *ah.* See description of *ides_audit_header.*
- *ab_path0.* File name (if applicable).
- *ab_pathl.* Second file name (if applicable).

4.5 Linking to IDES Processing Units

The IDES processing units are considered clients of *arpool.* Communication between these clients is RPC-based. Each IDES client can request audit records from *arpool* using RPC. Typically, a client would ask for audit records, process them, and then delete the processed records from *arpool.*

Audit records can be requested from *arpool* one at a time or in batches. For efficiency, it is best to request audit records in batches. Each request for audit records can be qualified with a bit-mask to select only certain categories of audit records. At present, there are three categories of audit records: user, system, and remote host.

Once an audit record has been requested by a client, arpool maintains a copy of that audit record until it is explicitly deleted by that same client.

The procedural interface to *arpool* is declared in `ides/arpool_stubs.h` with corresponding data structures being defined in `ides/arpoolrpc.h.` The procedures documented in the following subsections are simple stub routines that utilize the RPC stub routines automatically generated by *rpcgen(1)* from `ides/arpool_rpc.x.`

4.5.1 Preferred Calls

Using the latest version of the *arpool* stub routines, listed below, is the preferred method of obtaining IDES audit records because of its flexibility and performance, although older met hods are still compatible with the current version of IDES.

extern void **call-arpool_sethostname** (char *hostname)

This function sets up the host from which the IDES client will obtain audit records. *hostname* is the remote host to connect to.

extern int **call_arpool_nr** (int nrec, ides_audit_block *buf, int size)

This routine sends a collection of one or more audit records to *arpool. Nrec* is the number of records being sent, and *buf* is a pointer to the block of audit records that is the concatenation of one or more audit records. Size represents the total byte length of block(s) being passed. The return value for this function is 0 if successful, -1 otherwise.

extern arpool_gb_res ***call_arpool_get_block** (long count, long mask)

This function requests up to *count* audit records. The selection of audit records is qualified by the selection mask *mask.* Note that each audit record and the *buflist* array in the returned structure must be deallocated using *free(3).* The return value for this function is a set of audit records.

extern int **call_arpool_get_block_cb** (long count,, long mask,
void (*cb)(const ides_audit_block *), long *loseq, long *hiseq)

Requests up to count audit records. The selection of audit records is qualified by the selection mask mask. For each obtained audit record, function **cb* is invoked, with the audit record as its parameter. *Loseq* and *hiseq* are updated with the lowest and highest sequence number, respectively. This is the simplest interface for obtaining audit records. However, it is clearly not as flexible as *call_arpool_get_block().* This function returns 0 if successful, -1 otherwise.

extern int **call_arpool_rm_range** (long loseq, long hiseq)

The specified audit records are deleted. *Loseq* is the starting sequence number, and *hiseq* is the ending sequence number. The range given is inclusive. The function returns 0 if successful, -1 otherwise.

extern arpool_status_res ***call_arpool_status()**

This call returns a structure describing the status of arpool.

4.5.2 Older Versions of Arpool Stub Routines

This subsection lists older versions of the functions defined in subsection 4.5.1. IDES still supports the older versions, and these functions are still used in some of the IDES processing components.

extern arpool_de_count_res ***call_arpool_de_count** (long user, long pos, long count)

The specified audit records are requested, as well as count of its successors. User is the user ID of the audit records being requested, and *Pos* is the timestamp of the audit records requested. The function returns a set of audit records.

extern int **call_arpool_rm_count** (long user, long pos, long count)

The specified audit records are deleted, as well as count of its successors. User is the user ID of the audit records being requested, and *Pos* is the timestamp of the audit records requested. The function returns a set of audit records.

extern int **call_arpool_gd_count** (long icount, long *user, long *pos, long *count)

Returns the *(user, pos)* dimension of the next available audit record, as well as the count of how many more audit records are available. *Icount* is the maximum number of audit records to return, and count contains the number of audit records that are actually being returned. The function returns 0 if successful, -1 otherwise.

extern **arpool_de_res *call_arpool_de** (long user, long pos)

This routine requests the specified audit record. The audit record is specified by the *(user, pos)* dimension. The specified audit record identified by the *(user, pos)* dimension space is deleted. The function returns 0 if successful, -1 otherwise.

extern int **call_arpool_gd** (long *user, long *pos)

The *(user, pos)* dimension of the next audit record is returned, as well as the count of how many additional audit records have been received. This call blocks if no audit records are available.

extern int **call_arpool_pd** (long *user, long *pos)

The *(user, pos)* dimension of the next audit record is returned, as well as the count of how many additional audit records have been received. This call does not block, but returns a count of 0 if no audit records are available.

Chapter 5

The Statistical Anomaly Detector

The SRI IDES statistical anomaly detector observes behavior on a monitored computer system and adaptively learns what is normal for subjects. The defined subject types are individual users, groups, remote hosts and the overall system. Observed behavior is flagged as a potential intrusion if it deviates significantly from expected behavior for that subject.

The IDES statistical anomaly detector maintains a statistical subject knowledge base consisting of profiles. A profile is a description of a subject's normal (i.e., expected) behavior with respect to a set of intrusion-detection measures. Profiles are designed to require a minimum amount of storage for historical data and yet record sufficient information that can readily be decoded and interpreted during anomaly detection. Rather than storing all historical audit data, the profiles keep only statistics such as frequency tables, means, and covariances.

The deductive process used by IDES in determining whether behavior is anomalous is based on statistics, controlled by dynamically adjustable parameters, many of which are specific to each subject. Audited activity is described by a vector of intrusion-detection variables, corresponding to the measures recorded in the profiles. Measures can be turned "on" or "off" (i.e., included in the statistical tests), depending on whether they are deemed to be useful for that target system. As each audit record arrives, the relevant profiles are retrieved from the knowledge base and compared with the vector of intrusion-detection variables. If the point in N-space defined by the vector of intrusion-detection variables is sufficiently far from the point defined by the expected values stored in the profiles, then the record is considered anomalous. Thus, IDES evaluates the total usage pattern, not just how the subject behaves with respect to each measure considered singly.

The statistical knowledge base is updated daily using the most recent day's observed behavior of the subjects. Before incorporating the new audit data. into the profiles, the frequency tables, means, and covariances stored in each profile are first aged by multiplying them by an exponential decay factor. Although this factor can be set by the security officer, we believe that a value that reduces the contribution of knowledge by a factor of 2 for every 30 days is appropriate (this is the daily profile half-life). This method of aging has the effect of creating a moving time window for the profile data, so that the expected behavior is influenced most strongly by the most recently observed behavior. Thus, IDES adaptively learns subjects' behavior patterns; as subjects alter their behavior, their corresponding profiles change.

In this chapter, we discuss in detail the specific algorithms used to perform the anomaly detection analysis. We also describe the functional design of the statistical analysis component from an implementation standpoint, and we list the measures used as part of the running system.

5.1 Statistical Algorithms

Details of the IDES statistical algorithms are discussed in this section.

5.1.1 The IDES Score Value

For each audit record generated by a user, the IDES system generates a single test statistic value (the IDES score value, denoted T^2) that summarizes the degree of abnormality in the user's behavior in the near past. Consequently, if the user generates 1000 audit records in a day, there will be 1000 assessments of the abnormality of the user's behavior. Because each assessment is based on the user's behavior in the near past, these assessments are not independent.

Large values for T^2 are indicative of abnormal behavior, and values close to zero are indicative of normal behavior (e.g., behavior consistent with previously observed behavior). For the T^2 statistic, we select one or more "critical" values that are associated with appropriate levels of concern and inform the security officer when these levels are reached or exceeded. For example, T^2 values between 0 and 22.0 might be associated with no concern, values between 22.0 and 28.0 might be associated with a "yellow" alert, and values in excess of 28.0 might be associated with "red" alerts. The critical values are selected so that they have a probabilistic interpretation; for example, we might expect false red alerts only once every 100 days (excluding events such as a change in job status that might trigger a red alert). However, the security officer has the freedom to raise or lower the critical values for each system user, in case there is a need to monitor a particular user's behavior more closely or in case the standard critical values result in too many false alerts for a particular user.

Because the T^2 statistic summarizes behavior over the near past, and sequential values of T^2 are dependent, the T^2 values will slowly trend upward or downward. Once the T^2 statistic is in the red alert zone, it will take a number of audit records before it can return to the yellow or green zone. To avoid inundating the security officer with notification of continued red alerts, we notify the security officer only when a change occurs in the alert status, or when the user has remained in a yellow or red zone for a specific time. In addition, the security officer is able to generate a time plot of the T^2 values for a user and thus assess whether or not the user's T^2 statistic indicates a return to more normal behavior.

5.12 How T^2 Is Formed from Individual Measures

The T^2 statistic is itself a summary judgment of the abnormality of many measures. Suppose that there are *n* such constituent measures, and let us denote these individual measures by S_i, $1 \leq i \leq n$. Let the correlation between S_i and S_k be denoted by C_{ik}, where $C_{ii} = 1.0$. In the previous version of IDES (as described in the May 1990 Interim Report [6]), the T^2 statistic was defined as

$$T^2 = (S_1, S_2, \cdots, S_n)C^{-1}(S_1, S_2, \cdots, S_n)^t$$

where C^{-1} is the inverse of the correlation matrix of the vector $(S_1, S_2, \ldots, S_n$ and $(S_1, S_2 \cdots, S_n)^t$ is the transpose of that vector. When the S_i measures were not correlated, then T_2 simplified to $S_1^2 + S_2^2 + \cdots + S_n^2$, the sum of the squares of the measures. When the correlations were non-zero, then T^2 was a more complicated function that takes into account covariation in the S_i.

In working with previous versions of IDES, we identified three difficulties with the functional form of T^2:

- It was very difficult to determine the contribution of the individual S_i to T^2. The statistic T^2 is a complicated quadratic form in the S_i^2 and the S_iS_j with both positive and negative coefficients.
- The security officer could not specify which of the S_i are most important. The weighting coefficients in the quadratic form are completely determined by the correlation matrix C.
- The T^2 statistic was not as well-behaved as we would like it to be when two measures, say S_i and S_j, are negatively correlated at magnitudes of -0.8 to -1.0. This appears to be a consequence of the fact that the S_i follow a half-normal rather than a standard normal distribution.

As an interim measure to handle these difficulties, in the current version of IDES we have set the off-diagonal elements in the correlation matrix C to zero. This reduces the T^2 statistic to the sum of the squares of the S_i. This may be considered to be a simplified form of the following statistic, which addresses all of the problems mentioned above:

$$T^2 = a_1S_1^2 + a_2S_2^2 + \cdots + a_nS_n^2$$

where the a_i are positive coefficients that are specified by the security officer. With this definition for T^2 (which is currently being implemented, and is considered to be part of the current version of IDES) it is quite obvious how much each S_i contributes to T^2, the security officer can increase the importance of a measure S_i by increasing the value of the corresponding a_i, and the statistic T^2 is well-behaved even when there are large negative correlations between the S_i.

Unfortunately, when the T^2 statistic is a weighted sum of the squares of the S_i, T^2 is no longer sensitive to correlations among the S_i. In the Next Generation IDES we intend to reintroduce this covariation by defining T^2 as follows:

$$T^2 = \sum_{i=1}^{n} a_iS_i^2 + \sum_{i=1}^{n} a_{ij}h(S_i, S_j, C_{ij})$$

where the a_{ij} are positive coefficients specified by the security officer and $h(S_i, S_j, C_{ij})$ is a well behaved function of S_i, S_j, and their correlation C_{ij}, which takes large values when S_i and S_j are not behaving in accordance with their historical correlations. Thus, the new T^2 statistic will accommodate pairwise covariation in the S_i while addressing the problems noted previously.

5.1.3 Types of Individual Measures

The individual S measures each represent, some aspect of behavior. For example, an S measure might represent file accesses, CPU time used, or terminals used to log on. Two S measures might also represent only slightly different ways of examining the same aspect of behavior. For example, both S_i and S_j might represent slightly different, ways of examining file access.

We have found it useful to classify the different types of individual measures in the IDES statistical system into the following four categories:

Activity Intensity Measures. These measures (currently three) track the number of audit records that occur in different time intervals, on the order of 1 minute to 1 hour. These measures can detect bursts of activity or prolonged activity that are abnormal.

Audit Record Distribution Measure. This single measure tracks all activity types that have been generated in the recent past, with the last few hundred audit records having the most influence on the distribution of activity types. For example, we might find that the last 100 audit records contained 25 audit records that indicated that files were accessed, 50 audit records that indicated that CPU time was incremented, 30 audit records that indicated that I/O activity occurred, 10 audit records that indicated activity from a remote host, and so forth. These data are compared to a profile of previous activity (generated over the last few months) to determine whether the distribution of activity types generated in the recent past (e.g., the last few hundred audit records) is unusual or not. (Note that even though we call this measure "audit record distribution" we are not computing a distribution of the types of audit records themselves, but rather we are computing the probability that an audit record will indicate that a particular type of activity is occurring.)

Categorical Measures. These are activity-specific measures for which the outcomes are categories. For example, categorical measures might include the names of files accessed, the ID of the terminals used for logon, and the names of the remote hosts used. The categories that were used in the last 100 to 200 audit records that affected that activity are compared to a historical profile of category usage to determine if recent usage is abnormal.

Ordinal Measures. These are activity-specific measures for which the outcomes are counts. For example, ordinal measures might include CPU time (which counts the number of milliseconds of CPU used) or the amount of I/O. Behavior over the last 100 to 200 audit records that affected that activity is compared to a historical profile of behavior to determine if recent usage is abnormal.

These different categories of measures serve different purposes. The activity intensity measures assure that the volume of activity generated is normal. The audit record distribution measure assures that over the last few hundred audit records generated, the types of actions being generated are normal. The categorical and ordinal measures assure that within a type of activity (for example, an action that involves accessing a file, or an action that involves incrementing CPU time), the behavior over the past few hundred audit records that affect that action is normal.

The activity intensity measures and the audit record distribution measures are relatively recent additions to IDES. They were not available in the previous version of IDES (as described in the interim report [6]). The categorical and ordinal measures were available and continue to be available in all three generations of IDES. However, as described later in this section, there have been changes made in the details of the implementation of the categorical and ordinal measures between the previous and current versions of IDES, and it was these changes that prompted the addition of the activity intensity and audit record distribution measures.

5.1.4 Heuristic Description of the Relationship of S to Q

Each S measure is derived from a corresponding statistic that we will call Q. In fact, each S measure is a transformation of the Q statistic that indicates whether the Q value associated with the current audit record and its near past is unlikely or not.

The transformation of Q to S requires knowledge of the historical distribution of Q. For example, consider an S measure that represents CPU time used. The corresponding Q statistic would also measure CPU time used in the near past. By observing the values of Q over many audit records, and by selecting appropriate intervals for categorizing Q values, we could build a frequency distribution for Q. For example, we might find the following:

- 1% of the Q values are in the interval 0 to 1 (milliseconds)
- 7% are in the interval 1 to 2
- 35% are in the interval 2 to 4
- 18% are in the interval 4 to 8
- 28% are in the interval 8 to 16
- 11% are in the interval 16 to 32

In the previous and current versions of IDES, the S statistic would be a large positive value whenever the Q statistic was in the interval 0 to 1 (because this is a relatively unusual value for Q) or whenever Q was larger than 32 (because this value has not historically occurred). The S statistic would be close to zero whenever Q was in the interval 2 to 4, because these are relatively frequently seen values for Q. The selection of appropriate intervals for categorizing Q is important, and it is better to err on the side of too many intervals than too few. We are currently using 32 intervals for each Q measure, with interval spacing being either linear or geometric. The last interval does not have an upper bound, so that all values of Q belong to some interval.

As explained later in this report, in the Next Generation IDES, the definition of Q will change (for all measures except the activity intensity measures) in such a way that small values of Q are indicative of a recent past that is similar to historical behavior and large values of Q are indicative of a recent past that is not similar to historical behavior. This induces a modification in the transformation of Q to S, so that S is small whenever Q is small, and S is large whenever Q is large. Hence, S can be viewed as a type of rescaling of the magnitude of Q. In the Next Generation IDES, a Q value of 0 to 1 in our example would correspond to a small value for S.

5.1.5 Algorithm for Computing *S* from *Q*

Assume for the moment that we have defined a method for updating the Q value each time a new audit record is received, and that we have defined intervals that we have used to develop a historical frequency distribution for Q. The algorithm for converting individual Q values to S values in the previous and the current version of IDES is as follows:

1. Let P_m denote the relative frequency with which Q belongs to the mth interval. In our example the first interval is 0 to 1 and the corresponding P value (say P_0) equals 1%. There are 32 values for P_m, with $0 \leq m \leq 31$.

2. For the mth interval, let $TPROB_m$ denote the sum of P_m and all other P values that are smaller than or equal to P_m in magnitude. In our previous example, $TPROB$ for the interval of $4 \leq Q \leq 8$ is equal to 18% + 11% + 7% + 1% = 37%.

3. For the mth interval, let s_m be the value such that the probability that a normally distributed variable with mean 0 and variance 1 is larger than s_m, in absolute value equals $TPROB_m$. The value of sm satisfies the equation $P(|N(0,1)| \geq s_m) = TPROB_m$, or $s_m = \Phi^{-1}(1-(\frac{TPROB_m}{2}))$, where Φ is the cumulative distribution function of an $N(0, 1)$ variable. For example, if $TPROB_m$ is 5%, then we set s_m equal to 1.96, and if $TPROB_m$ is equal to 100% then we set s_m equal to 0. We do not allow s_m to be larger than 4.0.

4. Suppose that after processing an audit record we find that the Q value is in the mth interval. Then S is set equal to s_m, the s value corresponding to $TPROB_m$.

In the Next Generation IDES, the transformation of Q to S is slightly simplified by letting $TPROB_m = P_m + P_{m+1} + \cdots + P_{31}$. In our previous example, the $TPROB$ value of the interval $4 \leq Q \leq 8$ is equal to 18% + 28% + 11% = 57%. Thus, in the next generation of IDES, S is a simple mapping of the percentiles of the distribution of Q onto the percentiles of a half-normal distribution.

In practice these algorithms are easy to implement, and the calculations of the s_i values are done only once at update time (usually close to midnight). Each interval for Q is associated with a single s value, and when Q is in that interval, S takes the corresponding s value.

5.1.6 Computing the Frequency Distribution for *Q*

This subsection describes how we compute the frequency distribution for *Q*, which is necessary for transforming *Q* to *S*. All procedures for computing the frequency distribution for *Q* are identical in the previous, current, and Next Generation IDES.

The first step in calculating the historical probability distribution for *Q* is to define bins into which *Q* can be classified. We always use 32 bins (numbered 0 to 31) for a measure *Q*. Let Q_{max} be the maximum value that we ever expect to see for *Q*. This maximum value depends on the particular types of measures being considered. Default values are provided in the IDES system for these maximum values and they should be reset by the security officer if *Q* is in the highest bin more than 1% of the time. The cut points for the 32 bins are defined on either a linear or geometric scale. For example, when a geometric scale is used, bin zero extends from 0 to 1, bin one extends from 1 to $Q_{max}^{\frac{1}{32}}$, bin two extends from $Q_{max}^{\frac{1}{32}}$ to $Q_{max}^{\frac{2}{32}}$, and bin 31 extends from $Q_{max}^{\frac{30}{32}}$ to infinity.

As before, let P_m denote the relative frequency with which *Q* is in the *m*th interval (i.e., bin). Each *Q* statistic is evaluated after each audit record is generated (whether or not the value of *Q* has changed), and therefore P_m is the percentage of all audit records for which *Q* is in the *m*th interval.

The formula for calculating P_m on the *k*th day after initiating IDES monitoring of a subject is:

$$P_{m,k} = \frac{1}{N_k} \sum_{j=1}^{k} W_{m,j} 2^{-b(k-j)}$$

where

- *k* is the number of days that have occurred since the user was first monitored
- *b* is the decay rate for P_m, which determines the half-life of the data used to estimate P_m. We currently recommend a 30-day half-life, corresponding to a *b* value of $\frac{-\log_2(0.5)}{30} = 0.0333$.
- $W_{m,j}$ is the number of audit records on the *j*th day for which *Q* was in the *m*th interval
- N_k is the exponentially weighted total number of audit records that have occurred since the user was first monitored.

The formula for N_k is:

$$N_k = \sum_{j=1}^{k} W_j 2^{-b(k-j)}$$

where W_j is the number of audit records occurring on the *j*th day.

The formula for P_m appears to involve keeping an infinite sum, but the computations are simplified by using the following recursion formulas:

$$P_{m,k} = \frac{2^{-b} P_{m,k-1} N_{k-1} + W_{m,k}}{N_k}$$

$$N_k = 2^{-b} N_{k-1} + W_k$$

In the IDES system we update $P_{m,k}$ and N_k once per day and keep running totals for $W_{m,k}$ and W_k during the day.

5.1.7 Computing the *Q* Statistic for the Activity Intensity Measures

When a subject is first audited, that subject has no history. Consequently, we choose some convenient value to begin the *Q* statistic history. For example, we might let each *Q* statistic be zero, or some value close to the mean value for other subjects that are probably similar.

Each *Q* statistic for activity intensities is updated each time a new audit record is generated. Let us now consider how to update *Q*. Let Q_n be the value for *Q* after the *n*th audit record, and Q_{n+1} be the value for *Q* after the *(n +* 1)st audit record. In both the current and Next Generation IDES, the formula for updating *Q* is:

$$Q_{n+1} = 1 + 2^{-rt} Q_n$$

where

1 The variable *t* represents the time (say in minutes or fractions thereof) that has elapsed between the *n*th and *(n +* 1)st audit records.

- The decay rate r determines the half-life of the measure Q. Large values of r imply that the value of Q will be primarily influenced by the most recent audit records. Small values of the decay rate r imply that Q will be more heavily influenced by audit records in the more distant past. For example, a half-life of 10 minutes corresponds to an r value of $0.10 = \log_2 \frac{0.5}{10.0}$. The security officer may set the half-life of the activity intensity measures at any values that he or she feels are appropriate. We are currently running three activity intensity measures with half-lives of 10, 30, and 60 minutes, respectively.

Q is the sum of audit record activity over the entire past usage, exponentially weighted so that the more current usage has a greater impact on the sum. Q is more a measure of near past behavior than of distant past behavior, even though behavior in the distant past also has some influence on Q. The Q statistic has the important property that it is not necessary to keep extensive information about the past to update Q.

5.1.8 Computing the Q Statistic for the Audit Record Distribution Measure

Each audit record that is generated indicates one or more types of activity that have occurred for a subject. For example, a single audit record may indicate that a file has been accessed, that I/O has occurred, and that these activities occurred from a remote host. The Q statistic for the audit record distribution measure is used to evaluate the degree to which the type of actions in the recent past agree with the distribution of action types in a longer-term profile.

In both the current and Next Generation IDES, the calculation of the audit record measure begins by specifying the types of actions that will be examined. We currently recommend that each categorical and ordinal measure (with some exceptions as noted below) constitute an activity type. For example, if name of file accessed is a categorical measure, then the corresponding activity type would be that any file was accessed. Similarly, if the amount of I/O used is an ordinal measure, then the corresponding activity type would be that any I/O was used. That is, if an audit record would cause a categorical or ordinal measure to be recalculated, then a corresponding activity type should be defined. The exception would be a categorical or ordinal measure that would be affected by any audit record. For example, if an hour of audit record generation is a categorical measure, then every audit record causes this categorical measure to be updated and no purpose is served in defining a corresponding activity type.

In addition to defining activity types based on the categorical and ordinal measures used, it may be useful to define additional activity types. For example, the security officer may want to evaluate the percentage of audit records generated that indicate that the user is logged onto a remote host. If the security officer is not interested in which remote host is being used, then he or she may accomplish this goal in one of two ways. The first way is to establish a categorical measure with two categories: "remote host indicated" and "remote host not indicated." If this approach is used, then there should be no corresponding activity type defined because every audit record is relevant to the calculation of the categorical measure. The second way is to define an activity type of "remote host indicated" to be used by the audit record distribution measure and not to define a categorical measure. These two approaches yield similar (though not totally equivalent) results, but the second approach is computationally less intensive.

Suppose that we have established M activity types. For each action we must calculate a long-term historical relative frequency of occurrence, denoted f_m, for that activity type. For example, suppose that over the last three months, 7% of all audit records have involved file accesses. Then, f_m for the file access activity type would be 0.07. Note that each f_m is between 0 and 1.0 inclusive. The sum of the f_m may be greater than 1.0 because a single audit record may indicate that multiple activity types have occurred.

The algorithm used to compute f_m is essentially the same as that used to calculate P_m and uses the same decay rate. That is, we may write that the value of f_m on the kth day is equal to

$$f_{m,k} = \frac{1}{N_k} \sum_{j=1}^{k} W_{m,j} 2^{-b(k-j)}$$

where N_k, and b are defined as before and $W_{m,j}$ is the number of audit records on the jth day, which indicate that the mth activity type has occurred.

The Q Statistic for Audit Record Distribution in the Current IDES

The current and Next Generation IDES systems use different algorithms for defining the Q statistic for the audit record distribution measure. In the current IDES, Q is defined as

$$Q_{n+1} = \left(\sum_{m=1}^{M} I(n+1, m)(-\log_2 f_m) \right) + 2^{-r} Q_n$$

where

- Q_{n+1} is the value of the Q statistic after the (n+l)st audit record has been received.
- M is the number of defined activity types.

- $I(n + 1,m) = 1.0$ if the $(n + 1)$st audit record indicated that the mth activity type occurred and 0.0 otherwise.

- r is the decay rate for Q, which determines the half-life of the Q measure. For example, a half-life of 100 audit records corresponds to an r value of $- \log_2 \frac{0.5}{100} = 0.01$.

We note that when recent audit records indicate that historically less likely types of activity are now occurring, the Q statistic will increase in magnitude.

The Q Statistic for Audit Record Distribution in the Next Generation IDES

In the Next Generation IDES, we have changed the definition of Q so that we pay attention not only to whether more or less likely types of actions are now occurring, but also to which types of actions are now occurring. The revised definition of Q is

$$Q_n = \sum_{m=1}^{M} \frac{(g_{m,n} - f_m)^2}{V_m}$$

where

- $g_{m,n}$ is the relative frequency with which the mth activity type has occurred in the recent past (which ends at the nth audit record).

- V_m is the approximate variance of the $g_{m,n}$.

That is, Q_n is larger whenever the distribution of activity types in the recent past differs substantially from the historical distribution of activity types, where "substantially" is measured in terms of the statistical variability introduced because the near past contains relatively small (effective) sample size. The value of $g_{m,n}$ is given by the formula

$$g_{m,n} = \frac{1}{N_r} \sum_{j=1}^{n} I(j,m) 2^{-r(n-j)}$$

or by the recursion formula

$$g_{m,n} = 2^{-r} g_{m,n-1} + \frac{I(n,m)}{N_r}$$

where

- j is an index denoting audit record sequence.
- $I(j,m)$ is 1.0 if the jth audit record indicates activity type m has occurred and 0.0 otherwise.
- r is the decay rate for Q that determines the half-life for the Q measure. We have set the half-life to approximately 100 to 200 audit records.
- N_r is the effective sample size for the Q statistic, which is set at its asymptotic value of $\frac{1}{1-2^{-r}}$.

The value of V_m is given by the formula

$$V_m = \frac{f_m(1 - f_m)}{N_r}$$

except that V_m is not allowed to be smaller than $\frac{0.02(0.98)}{N_r}$.

5.1.9 Computing the Q Statistic for Categorical Measures

Categorical measures are those that involve the names of particular resources being used (such as the names of files being accessed, or the location from which logons are attempted) or involve other categorical characteristics of audit records, such as the hour of the day on which the audit record was generated.

In both the current and Next Generation IDES, the method used for computing Q for categorical measures is essentially the same as that previously described for computing Q for the audit record distribution measure. In fact, we may view the activity type measure as a categorical measure. The only difference in the definition for Q is that every audit record results in the recalculation of the audit record distribution measure, whereas for all other categorical measures, Q is updated only when the audit record contains information relevant to the particular measure. For example, the file name accessed measure would be updated only when the audit record concerns a file access. A half-life for the file name accessed measure of 100 audit records would refer to 100 audit records relevant to file names accessed, rather than to the last 100 audit records.

In the previous version of IDES, the form for the Q statistic was very similar to that used in the current version of IDES except that the exponential decay factor for Q depended on the time between appropriate audit records. That is, the formula for Q was

$$Q_{n+1} = \left[\sum_{m=1}^{M} I(n+1, m)(-\log_2 f_m)\right] + 2^{-rt_{n+1}} Q_n$$

where Q_{n+1}, M, and $I(n + 1, m)$ are defined as in the current version of IDES, but

- r is the decay rate for Q, which determines the half-life of the Q measure in seconds.
- t_{n+1} is the number of seconds that have expired between the $(n + 1)$st and nth audit records.

The addition of the factor t in the exponent caused the Q statistic to be sensitive to the volume of activity. When activity volumes per unit time were higher, the Q statistic tended to grow. When activity volume was lower (such as overnight, or directly after lunch), the Q statistic tended to be smaller. We decided that this dependence on activity volume confounded the amount of activity with the normality of the type of activity and that it would be better to separate these aspects of behavior into separate measures. Consequently, we removed the factor t from the exponent and converted the decay rate r to refer to the exponential decay per audit record. In the current version of IDES, this modification allows the Q statistic for categorical measures to examine the normality of a specific type of behavior (such as file accesses) over the last few hundred audit records that contain information about that behavior, regardless of when that activity occurred. The information on activity volume and type was transferred to two new types of measures: the activity intensity measures, which track the number of audit records per unit of time, and the audit record distribution measure, which tracks the type of actions being generated.

5.1.10 Computing the *Q* Statistic for Ordinal Measure

Ordinal measures are those that involve counts of particular resources used (such as CPU time in milliseconds or I/O counts) or some other counting measure (such as the interarrival time in milliseconds of consecutive audit records).

In the current and Next Generation IDES systems, ordinal measures are transformed to categorical measures. For example, consider the ordinal measure of CPU time: Individual audit records arrive that indicate that a non-zero amount of CPU time has been incrementally expended since the last reporting of CPU usage. We might expect that this delta-CPU value would be between 0 and a maximum of 100,000 milliseconds. We define 32 geometrically scaled intervals between 0 and 100,000 milliseconds employing the same procedure as we used for defining intervals for the historical profile for Q (including the convention that the last interval actually extends to infinity). When a delta-CPU value arrives and is classified into interval m, we state that a categorical event m has occurred. We treat the Q measure for CPU time as a categorical measure, where a category is activated whenever an audit record arrives with a delta-CPU value in that category. Thus, the Q measure for CPU time does not directly measure total CPU usage in the near past; rather, it measures whether the distribution of delta-CPU values in the near past is similar to the historical distribution of delta-CPU values. Once the ordinal measure is redefined as a categorical measure as discussed above, the *Q* statistic is calculated in the same fashion as any other categorical measure.

In the previous version of IDES, ordinal measures were not converted to categorical measures. Bather, the definition for *Q* was

$$Q_{n+1} = D_{n+1} + 2^{-rt_{n+1}} Q_n$$

where D_{n+1} is the difference in the ordinal count between the *n*th and *(n* + 1)st audit record. For example, if the measure were CPU time, then D_{n+1} would be the amount of CPU time that was expended between the *n*th and *(n* + 1)st audit record. As with the categorical measures in the previous version of IDES, the ordinal measures were heavily influenced by volume of activity. More importantly, *Q* tended to be a measure of mean behavior per unit of time (e.g., mean CPU usage per unit time). We decided that we would be more likely to detect abnormal usage if we looked for unusual increments of behavior (for example, individual audit records that demonstrate an unusual amount of CPU). As a result, we both removed t_{n+1} from the decay factor and categorized the increments D_{n+1}, yielding the ordinal measures as represented in the current version of IDES.

5.2 Design and Implementation

This section describes the design and implementation of the IDES statistical component.

5.2.1 Functional Architecture

Conceptually, the statistical anomaly detection module of IDES is designed to be similar to that of the first IDES prototype. That is, by using profiles, anomalous behavior is computed by comparing a subject's ongoing activity (which we will refer to as a subject's "current" profile) with his or her past activity. At the end of each day, the subject's accumulated observed behavior for that day is folded into the corresponding historical profile. This enables IDES to adaptively learn subjects' behavior patterns; as subjects alter their behavior, the profiles will change accordingly.

The IDES statistical component is driven by the arrival of audit records. By examining the audit records as they arrive from the target system, IDES is able to determine through a variety of statistical algorithms whether the observed activity is abnormal with respect to the profiles. Anomalous behavior is flagged when such behavior deviates from the subject's normal behavior by a predetermined amount. When an anomaly is detected, an anomaly record is generated and is presented to a security officer and also recorded for further analysis.

Figure 5.1 shows a diagram of the major parts of the statistical component and their relationship to one another. In the following subsections, we describe in detail the functionality of each of these modules.

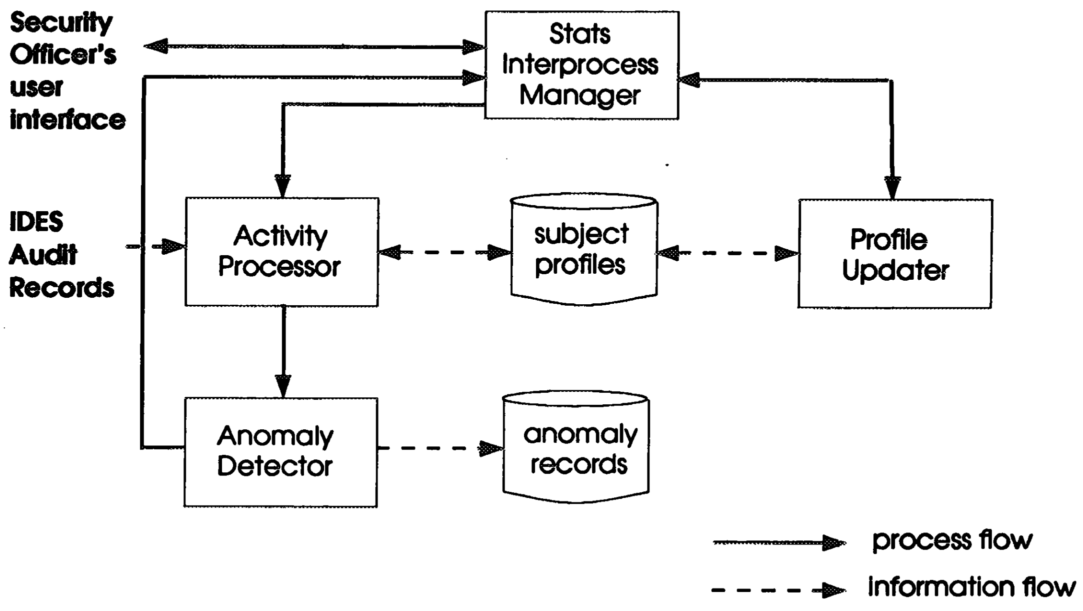


Figure 5.1: Statistical Anomaly Detection Process Unit

Interprocess Manager

This module is the driver of the entire statistical component. All processes (both internal and external) communicate with this module, and it directs the process flow accordingly. When the IDES statistical component starts, the interprocess manager first loads in the appropriate statistical parameters, such as which measures to activate, what values to use for aging factors, classification of commands, and grouping of subjects. Such information is stored in a configuration file that can be modified by authorized users either before or during the execution of the statistical component.

Prior to reading each audit record, the interprocess manager "listens" for messages coming from other IDES processes, such as the expert-system component, the profile updater, or one of the user interface programs. Upon receiving one of these messages, the interprocess manager calls the appropriate process to deal with the message, and then continues on to process the next IDES audit record. Some examples of such messages include a request to dump a subject's current profile cache (usually in order to update its profile, or to display the latest accumulation of statistical data), or a notification from another process (most likely the profile updater) to reload the anomaly detector's profile cache to incorporate more recent data.

In addition, the interprocess manager sends messages to other processes as needed. Currently, the statistical component sends a statistical score for every audit record to the security officer's user interface, as well as an alert message if this score is approaching or exceeds a given threshold value for that subject.

Activity Record Processor

This module receives IDES audit records from the realm interface and transforms the data into a data structure called an *activity vector.* This activity vector is N elements long, where N is the number of measures (active and inactive), and contains observed values for each measure affected by an audit record. It should be noted that not all measures are always touched by each audit record, but this is generally determined by the construction of the target system's auditing mechanism. If a measure is inactive, a zero or null value is inserted into that measure's location in the activity vector. This activity vector is passed on to the anomaly detector for statistical computation.

Anomaly Detector

The anomaly detector is the heart of the statistical engine. This module takes the activity data produced by the activity record processor and calls the routines that support the statistical algorithms used for anomaly detection.

The anomaly detector continually updates the subject's daily accumulation of activity (represented as probability vectors and matrices) with each incoming audit record, and compares those values with the historical profile to determine deviation from expected (normal) behavior. After applying these series of statistical tests (see Section 5.1 for a detailed discussion of the algorithms used), an anomaly record may be produced if an observed activity vector indicates unusual behavior according to the given subject's historical profile. When this happens, a message is passed to the interprocess manager to send to the appropriate external processes, such as the security officer's user interface.

Profile Updater

The profile updater is generally invoked once a day. It takes the day's accumulated activity information (represented as various probability structures) for a particular subject, and folds it into its historical profile after applying some arbitrary aging factor (the default is set to a 30-day half-life). This newly updated profile is subsequently used against the next day's activity for that subject.

The profile updater can be run in two modes:

Parallel. The profile updater can be run as an independent process that "wakes up" at a predetermined time of day (usually midnight) and simultaneously processes the profiles for that day while the anomaly detector module continues to receive audit records from the realm interface. Any incoming audit records for a subject whose profile is currently being updated are temporarily set aside until the update process has completed for that subject.

Sequential. The updater can also be invoked from the interprocess manager, which looks at each audit record timestamp to determine whether or not a new day has occurred. When a new day occurs, the anomaly detector is temporarily suspended while the profile updater process is called for each active subject.

When running in either of the above modes, the profile updater process maintains a simple communication protocol with the interprocess manager to keep from corrupting the subject's profiles while the anomaly detector is running. Even if the updater is running in sequential mode, it is important for the anomaly detector to know when a subject's profile has been modified because certain profile structures (mainly counters) must be reset to zero after the day's activity is incorporated into the subject's historical profile. The anomaly detector caches profiles in memory, and hence this cache needs to be refreshed after a profile update.

The profile updater can also be invoked manually, although this method is rarely used except in cases in which one of the above modes did not correctly start the updater, or for testing and experimentation.

5.2.2 Implementation Details

The statistical component is designed so that its pieces can be implemented as software modules. It is important to provide flexibility within this component so that different techniques for numerical analysis, data storage, and data presentation can be applied without any major redesign or reimplementation of the component.

Earlier in the project, extensive experimentation was done within the data storage layer. Originally a standard DBMS product (ORACLE) was used as the underlying data management software, but rigorous tests indicated a performance bottleneck at this level due to the increased amount of profile data necessary to process and store for the more complex statistical algorithms. While a sophisticated relational DBMS is beneficial for complex data manipulation and query management, it appeared to be more than was needed for the basic data processing used in the statistical analysis. Hence, we opted for a simpler data storage mechanism for the statistical processing, using UNIX flat files with indexing. However, the use of a powerful DBMS product may be desirable for other aspects of IDES, especially where complicated and ad-hoc queries may be required to retrieve processed information for postanalysis (e.g., playback of events or building evidence to support a case).

We also discovered that caching of subject profiles greatly enhanced the performance of data retrieval. By keeping a number' of subjects' profiles in memory during execution, I/O access to the storage area (relatively expensive with respect to other computations within the statistical component) was reduced significantly with caching. A simple LRU (least recently used) caching algorithm is used to determine which profiles should exist in the cache, along with aging the contents of each cache entry with the reception of each new audit record.

The usage of named pipes (a UNIX feature) is common throughout the statistical component, as it is the means of the interprocess communication between each of the internal modules and external processes.

5.2.3 Future Work

Available experimental data were limited to a small sample size during this project period. While the data that we had were sufficient to determine that the algorithms were behaving as expected, we plan to experiment with a wider quantity and variety of data in the future. We recently acquired a large amount of audit data from the FBI, and we have also collected several months worth of UNIX data from our own lab; it is our desire to thoroughly exercise the statistical component, using these data. The next phase of the IDES development will put an emphasis on more rigorous testing of the analytical components, both for anomaly detection accuracy and for system performance and usability.

The statistical algorithms went through several enhancements during the term of this project, and it was discovered that in some cases certain methods were suited more towards a particular type of system or subject activity, or a new type of subject behavior was discovered that the current version of algorithms did not efficiently deal with. In light of this, it would be practical to have the system constructed so that different versions of the statistical component can run simultaneously, using the same set of audit data, to evaluate the performance and accuracy of the various detection algorithms.

We also plan to perform second-order intrusion-detection tests (such as detecting how fast a user's mean and variance change). These trend tests will help evaluate the effectiveness of detecting planned attacks in which a subject may intentionally change his or her behavior gradually to a different normal behavior to do subsequent malicious acts, or to purposely broaden the spectrum of "normal" behavior to include misuse or abuse of the system.

Although we have currently implemented a means to allow a user to dynamically reconfigure various statistical parameters, this feature needs to be considered more carefully with respect to the consequential effects on the running system. For example, when new measures have been turned on or off, a certain amount of time is needed to allow the profiles to adapt to the new configuration or else the anomaly detection component will very likely produce false alarms because of the new influence by the new measure(s), or a lack thereof. The whole issue of dynamic reconfiguration of the system cannot be treated trivially, and thus certain procedures or techniques must be defined and implemented to avoid the corruption of an ongoing IDES system.

During this project, we primarily focused on the user subject type. While the design of IDES is by no means limited to one particular type of subject, we plan to experiment with IDES using other subject types, such as remote hosts, target systems, and local hosts. This effort will require the definition of appropriate measures for these new subjects, and some code enhancements to the activity record processor module of the statistical component.

5.3 Intrusion-Detection Measures

As we discussed previously, IDES determines whether observed behavior as reported in the audit data is normal with respect to past or acceptable behavior characterized by specific intrusion-detection *measures.* A measure is an aspect of a subject's behavior on the target system, and is applied to individual subject activity.

This section lists all the measures we are currently using in the IDES statistical component. Not all measures will apply to all target system types (for example, the measure for window commands would be an irrelevant measure for a target system without any window-management facilities). The statistical anomaly component has a mechanism to inactivate such measures as needed.

5.3.1 User Measures

As described in Section 5.1, the statistical measures have been classified into four groups: *ordinal, categorical, audit record distribution,* and *activity intensity.* Below we list these measures, which can be monitored for a user subject type on the target system according to their classification:

1. **Ordinal**
   - *CPU Usage.* Indicates the amount of CPU usage by a particular user. The value recorded is the delta of CPU seconds between the last audit record and the current one observed for this user.
   - *I/O Usage.* Indicates the amount of I/O usage while the user is on the system. I/O activity can be either disk accesses or characters typed at the terminal, depending on the target system and its capabilities to track either one.
2. **Categorical**
   - *Physical Location of Use.* Records the relative number of times a user connects to the target system from different locations - host name, modem number, or network address.
   - *Mailer Usage.* Records the relative number of times various mailers are used by the user.
   - *Editor Usage.* Records the relative number of times various editors are used by the user.
   - *Compiler Usage.* Records the relative number of times various compilers are used by the user.
   - *Shell Usage.* Indicates the relative number of times various shells and programming environments are invoked by the user.
   - *Window Command Usage.* Indicates the relative number of times different window commands (such as opening a window, closing a window) are invoked by the user. This measure applies only to target systems that support windowing environments.
   - *General Program Usage.* Indicates the relative number of times each program is used by the user. These programs include those that are in the special command measures (e.g., mailers, compilers). The measure is categorized by program names, and it is important that the actual program name, rather than any aliases, is specified by the security officer.
   - *General System Call Usage.* Indicates the relative number of times a system call is invoked by the user, categorized by system call names. System calls are lower-level functions that are generally invoked by higher-level programs (i.e., not directly by the user).

- *Directory Activity.* Records the relative number of times a user does something to a directory, categorized by the type of activity (e.g., create, delete, read, modify).
- *Directory Usage.* Records the relative number of times a user accesses a directory on the system, categorized by directory names.
- *File Activity.* Records the relative number of times a user does something to a file, categorized by the type of activity (e.g., create, delete, read, modify).
- *File Usage.* Records the relative number of times a user accesses a file within the system, except for temporary files, categorized by file names.
- *User IDs Accessed.* Indicates the relative number of times a user changes user ID.
- *System Errors by Type.* Records the relative number of times each type of error occurs.
- *Hourly Audit Record Activity.* Records the relative number of audit records received for each hour, categorized by the hours of the day.
- *Day of Use.* Records the relative number of audit records produced by the user on a daily basis, categorized by the days of the week.
- *Remote Network Activity by Type.* Records the relative number of individual network-related activities performed by the user, categorized by network-activity type (e.g., remote logins, ftp, remote shells).
- *Remote Network Activity by Hosts.* Records the relative number of network-activity records a user produces for each remote host, categorized by hosts.
- *Local Network Activity by Type.* Records the relative number of different types of local network activity performed by the user, categorized by network-activity types.
- *Local Network Activity by Hosts.* Records the relative number of local-network-activity records produced by a user for individual hosts, categorized by hosts.

3. Audit Record Distribution.

For each of the above measures (categorical and ordinal) there is a corresponding category (or activity type) in the audit record distribution measure. For example, the category corresponding to the CPU measure is "audit record indicated that the incremented CPU usage was greater than zero." The category corresponding to the "file usage" measure is "audit record indicated that a permanent file has been read."

In addition, we have added the following category, which is not directly associated with a measure.

- *Temporary File Accessed.* Records if a temporary file was accessed by the user. Temporary files are those created and used during an execution of a program, and removed after program completion.

4. **Activity Intensity Measures.** We currently are using the following three audit record intensity measures (rate of audit records in a given time frame):

    - *Volume per Minute.* Number of audit records processed by a subject in a time period with a l-minute half-life.
    - *Volume per 10 Minutes.* Number of audit records processed by a subject in a time period with a lo-minute half-life.
    - *Volume per Hour.* Number of audit records processed by a subject in a time period with a l-hour half-life.

While our development and testing so far have been primarily on user subject types, IDES is by no means limited to this particular type of subject. Measures for other types, such as target systems and remote hosts, can also be monitored. The next two subsections list a variety of measures that will be used in subsequent versions of IDES for different subject types.

5.32 Target System

These measures describe the behavior of the target system - that is, the system being monitored:

- *CPU Usage (ordinal).* Records the amount of CPU time used on the target system. The "delta" values used are the increments in overall CPU time used by all subjects on the system.

- *I/O Usage (ordinal).* Records the amount of I/O used on the target system. The delta values used are the increments in overall I/O calls used by all subjects on the system.

- *Hourly Bad Login Attempts (categorical).* Records the relative number of bad login attempts made on the target system during each hour, categorized by the hours of the day. This measure is intended to track the times during the day that bad login attempts are usually common/uncommon.

- *Target System Errors by Type (categorical).* Records the relative number of errors of different types made on the target system, categorized by error types.

- *Hourly System Errors (categorical).* Records the relative number of errors that occurred on the target system during each hour, categorized by the hours of the day. It is intended to track the times during the day that system errors are likely to occur.
- *Network Activity by Type (categorical).* Records the number of individual network activity-related audit records produced on the target system, categorized by network activity types.
- *Network Activity by Hosts (categorical).* Records the number of network activity audit records the target system produced for a particular host, categorized by remote hosts.

As in the user subject measures, the audit record distribution measure encapsulates all of the categorical and ordinal measures defined above (does not include activity intensity measures).

5.3.3 Remote Host

Remote host subjects are nonlocal machines that can connect to the target system via network communications. Remote hosts are not necessarily being monitored themselves, but their behavior with respect to the target system can be observed using the following measures:

- *User Accounts from the Remote Host (categorical).* Records the relative number of times a user account is accessed on the target system from the remote host subject, categorized by account names. This measure is intended to track how often an account on the target system is used from some outside host.
- *Activity Type Count from the Remote Host (categorical).* Records the relative number of different types of network activity (as designated by the port number) that originated from the remote host subject, categorized by the network-activity type.
- *Hourly Use from the Remote Host (categorical).* Records the relative number of network-activity audit records received each hour from the remote host subject, categorized by the hours of the day.
- *Hourly Use by Type from the Remote Host (categorical).* Records the relative number of network-activity audit records of each type received each hour from the remote host subject, categorized by the hours of the day and the type of network activity.

- *Bad Login Attempts from the Remote Host (categorical).* Records the relative number of bad login attempts made from the remote host subject. It keeps track of bad login attempts coming in from each particular remote host.

Chapter 6

The IDES Expert System

The rule-based component of IDES evaluates user behavior by evaluating events (audit records) using a set of rules. These rules describe suspicious behavior that is based on knowledge of past intrusions, known system vulnerabilities, and the installation-specific security policy. The user's behavior is analyzed without reference to whether it matches past behavior patterns. In effect, the rules in the rule-based component constitute a minimum standard of behavior for users of the host system. While the statistical anomaly detector attempts to define *normal* behavior for a user, the rule-based component attempts to define *proper* behavior, and to detect any breaches of etiquette.

The IDES rule-based component operates in parallel with the statistical component. It is *loosely coupled* in the sense that the inferences made by the two subsystems are independent. The rule-based detector and statistical anomaly detector share the same source of audit records and produce similar anomaly reports, but the internal processing of the two systems is done in isolation. (In the Next Generation IDES we plan to introduce a resolver to combine and further post-process the outputs of the two components.)

The knowledge base of the rule-based component contains information about known system vulnerabilities and reported attack scenarios, as well as our intuitions about suspicious behavior. We have obtained information about known system vulnerabilities through discussions with members of CERT (Computer Emergency Response Team) at Carnegie Mellon University's Software Engineering Institute and other members of the computer security community. Not all of the information we obtained in this fashion has been incorporated into the rule base; see below for a discussion of our experiences in this area.

The rule-based component can be vulnerable to system deficiencies and intrusion scenarios that are not in its rule base. For this reason, it is important to update the system as these vulnerabilities come to light. Apart from this, our expectation is that the statistical component is likely to find new intrusion scenarios unusual enough to trigger an anomaly.

IDES is thus a system containing complementary approaches, each of which helps cover the vulnerabilities of the other.

The IDES rule-based detection component is a rule-based, forward-chaining system using the Rete algorithm. By 'forward-chaining' we mean that the system is driven by the facts that are put into the knowledge base, as opposed to being driven by goals that the user states. When a new fact appears, the system makes all the deductions it can about that fact. This way of driving deduction fits in well with an event-driven system like IDES. Each IDES audit event becomes a fact that gets asserted into the knowledge base. This assertion causes the rules to be applied to the new fact and whatever other facts are in the knowledge base.

We produce the executable system by writing a rule base that is translated into C language code using the PBEST (Production-Based Expert System Tool) translator. The syntax for the rule-specification language is relatively easy to master; the use of a translator instead of an interpreter improves the performance of the resulting system.

We have included a draft manual for PBEST as an appendix to this report. This manual describes the tool in more detail, including information on how it works, how to write rule bases for it, and how to incorporate the resulting rule-based engines into larger programs.

Early development effort on the rule-based component focused on the following areas:

Interactive Interface. PBEST can optionally produce an expert system with a window-based interactive interface. This interface uses the IDES User Interface Library, which is based on the X Window System. The interactive interface currently supports free running, single stepping, breakpoints, tracing, and assertion and negation of facts. It maintains a display of the current state of the knowledge base during execution. It also gives information about the conflict-resolution mechanism, showing which rules can fire and which facts make the rule eligible to fire. The interface lists the firable rules in the order of firing priority.

Memory Management. We have incorporated a garbage-collecting memory allocator into PBEST. Besides simplifying the memory management problem, this memory allocator helps the overall performance of IDES by bounding the working set of the rule-based component process and limiting fragmentation. That is, we are able to specify that the memory allocator should create a memory area of a certain size and use only that memory until it can no longer reclaim enough memory by garbage collection to continue. The allocator will then expand its memory area. We have found that in practice the rule-based components we have made to process FBI data will run forever (e.g., process millions of audit records) without expanding a l-megabyte heap. By tuning the initial heap size, we can find a good tradeoff between time delays due to garbage collection and time delays due to paging. In addition, since the garbage collector uses a specific area of memory for its heap, it will not scatter small areas of free memory throughout the process address space in the course of asserting and negating facts.

GLU Implementation. We have brought up a version of the rule-based component that will run under GLU. Chapter 8, describing the IDES version of GLU, discusses our work in this area.

Rule Base. Most of our early effort in developing a rule base had gone into creating one for the FBI's IDES system.

Recent effort on the rule-based component has involved creating a UNIX-specific rule base, exercising it, and modifying it in light of our experiences. We currently have 42 rules, of which six are used for interface purposes. The primary areas we have been able to monitor with this component are logins, user privilege, and file access.

The rule-based component attempts to detect programmed login attacks by keeping track of bad login attempts and creating alerts when several bad login attempts occur without an intervening successful login attempt. The rule base also distinguishes between remote and local logins, creating low-level alerts when it sees remote logins. Several rules also create alerts when remote users do certain things. For example, the rule base attempts to detect *leapfroggers.* A leapfrogger is a remote user who uses a given system to access other systems with the intent of concealing his or her identity from system administrators who want to find out who it was that was really accessing their systems. The system also attempts to detect users outside of the local network who attempt to mount filesystems residing on the local system.

The rule base also incorporates rules that attempt to detect when a user gains unauthorized privileges. It does this by comparing the audit user ID, which is guaranteed by the auditing system to remain constant for the life of the session, with the current user ID, which can change through the use of 'setuid' system calls. Usually, a user who suddenly starts executing programs with a current user ID different from the audit user ID has obtained unauthorized privileges; the rule base notices this and creates an alert.

Currently, the rule base monitors some types of file access, such as modifying the automatically executed shell scripts belonging to another user. These shell scripts get executed whenever the user logs in and can thereby be used to plant Trojan horse programs. In its present state, the rule base creates an alert whenever a user creates or modifies any such shell scripts in directories other than his or her own.

As mentioned above, we have been unable to make use of all the information we have obtained about UNIX system vulnerabilities. This is primarily due to shortcomings in the auditing system. We have discovered shortcomings that are pertinent to the rule-based component:

Command-line Arguments. The auditing system does not report, the arguments that are given to a program when it is invoked. This prevents us from detecting several types of vulnerabilities. For example, there is a vulnerability in some versions of the `sendmail` program that can be exploited when it is given `-C` as a command-line argument. In addition, the `find` command can be used in many ways to attempt to compromise security; none of these methods can be detected without looking at the command-line arguments given to the `find` command when it is executed.

Program Termination. The auditing system does not report process termination. This means that we cannot determine when a program stops running. If we attempt to do deduction on a per-program basis, we need to know when the program has terminated and the information about the program can be discarded.

It turns out that we can obtain program termination information from accounting files on the system. However, there are two problems with using this source of information. First, due to race conditions we often receive the report of the program's termination before getting the report of the program's execution, especially for small programs that execute rapidly. Handling this properly will require extra cumbersome mechanism in the rule base. Second, the accounting files do not identify the termination of shell scripts. When a shell script executes, the auditing system reports the invocation of the shell script, but the accounting system reports the termination of the shell under which the shell script was run. For example, the auditing system will report the execution of the command `/bin/arch.` When it terminates, the accounting system will report the termination of the program `sh.` Since many shell scripts run under `sh,` there is no easy way to figure out which shell script is terminating. In general we have found it very difficult to correlate the audit information with the accounting information. This has the effect both of making it hard to detect intrusions (since certain kinds of shell scripts can be a major vulnerability) and of causing the knowledge base to grow, since it is hard to write rules to remove unneeded facts that work for every case.

We have run the rule-based component extensively. We have used it as part of IDES to monitor three workstations running full Sun C2 auditing along with the news/mail/ftp/dialin server auditing just accounting (program execution) and logins/logouts. We have seen no problems with throughput running on a Sun 3-260 workstation. The realm interface causes the two components to run at the speed of the slowest component; the rule-based component, which currently does no floating-point arithmetic, is about ten times faster than the statistical component and thus is not the limiting factor on throughput. Its speed is proportional to the number of rules. There is therefore a lot of room for expansion, both of rules and monitored workstations, even running on Sun 3 hardware. Our major limiting factor in running IDES has not been throughput on the IDES machine; it has been finding people willing to be monitored.

The rule-based component does detect what it is programmed to detect. It produces alerts that are indications that, for example, a user has obtained root privilege or modified login scripts. We have also detected unusual events, such as remote logins to our ftp server for very short periods of time (less than 10 seconds). Major shortcomings have been in the area of scope. We are unable to detect some intrusions because the auditing system does not produce the necessary information. We are unable to detect others because we do not yet know how they are done.

In the future, we expect to overcome the auditing system problems either by cooperating with the vendor to enhance the system or by modifying it on our own. We expect to continue our knowledge-engineering efforts to extend the ability of the rule-based component to detect new intrusions. We also plan to make the following enhancements to PBEST:

- Adding and removing rules while the system is running
- Backward chaining
- Better performance profiling
- Improved programming environment
- Improved run-time user interface for noninteractive systems.

Chapter 7

The IDES User Interface

This chapter describes the design and implementation of the user interface component of IDES. In particular, two types of user interfaces have been implemented: the security officer's interface, which assimilates the output of the anomaly detector units, and a set of analyst tools used for experimentation and testing of the IDES statistical component.

7.1 Design Concepts

The goal of the IDES user interface is to satisfy several types of users whose responsibilities vary from high-level analysis of abnormal behavior to the maintenance of the system. We have identified these types as follows.

- Users who need to be informed of any abnormal behavior on a targeted system in as close to real time as possible
- Users who must rapidly assess the severity of an anomalous condition and act accordingly
- Users who can define and/or modify the characteristics of normal behavior for a particular subject or group of subjects; these would include definition of statistical measures and expert-system rules
- Users who may want to make random queries to the data, perhaps to validate different intrusion-detection methods, or for troubleshooting the system
- Users who may wish to do a historical analysis of events, for example, to build a case against a misuser of the target system
- Users who must install and maintain the IDES system.

The design provides a user environment that incorporates some of the innovative concepts of information display, and represents an integrated and consistent view of the IDES information. This section presents some of our ideas that were implemented.

7.1.1 IDES User Environment

IDES users are categorized into the following three classes:

Security Officers (SOs) are users who are normally watching out for target system intrusions, that is, they are considered the IDES "end users." They are interested in unusual events that occur on the target systems, and examine the IDES processed data to determine whether or not a given set of observations is truly abnormal. They can also be responsible for maintaining any user profiles, such as vacation or workhour schedules, or any other information that might contribute to the analysis of abnormal system behavior.

Data Analysts (DAs) understand the methods by which intrusion detection is performed. DAs are interested in either the statistical analysis of IDES or the expert-system rules used to detect abnormal behavior on the target system (or both). Like the SOs, these users want to be notified if an intrusion has occurred, but their focus is on how the intrusion was discovered as opposed to *what* the intrusion was.

System Administrators (SAs) are users who are responsible for the system's maintenance and performance, constantly making sure that the IDES components are running as required. They must also understand how the system operates so that they can tune it as needed for efficient processing.

It should be noted that the roles of each user class are not rigid; SOs can exercise system administrative duties, and vice versa, and often DAs may be considered specialized SOs. The classification is mainly defined for the purposes of design discussion. Nevertheless, the system can be configured with default access privileges to reflect the principle of separation of duties.

While the information available for each class may be different, the data presentations should be similar in format to maintain consistency throughout the environment. This is done by providing a library of display functions used by each user interface component (see Section 7.3).

There are two ways to invoke the IDES user environment. The IDES machine may be configured such that upon logging in, the user is immediately brought into the IDES environment. Alternatively, the IDES user environment may be invoked as a command executed from another environment, such as from a window management system or from the operating system itself. In either case, the user may be requested to supply a login name and password before gaining access to the IDES environment.

The user interface is window based and equipped with menus and mouse control for command selection. It is also available from remote sites, as the window system facility used allows access and invocation of graphical displays from across the network.

To accommodate the classification of user types, the IDES user environment is organized into three areas:

IDES processed data consists of all the information that supports the SO and SA duties. These data include audit records, anomalies, user status, and privileges. This information can be shown in a variety of ways (e.g., graphs, text, lists).

System administration tools support tasks ranging from the installation and configuration of IDES to the regular maintenance of the system. The SA is typically concerned with the running status of IDES, and needs to be aware of any problems that may affect the performance of the system. The IDES environment provides various tools to facilitate this role.

General OS functions are initially installed with all IDES environments. These include processes such as a terminal window environment to invoke regular operating system commands and receive system messages, an electronic mail facility, screen locking, system logout, and so forth. As the IDES user becomes familiar with the operating system environment, this list of commands may be expanded to include other functions and applications. However, if the IDES environment is to be isolated from other types of computer work (i.e., the computer is an IDES-dedicated machine), then additional functionality should be added with discretion.

Section 7.2 describes the types of data each user class may be interested in, and how such information is currently displayed in IDES. Again, it is important to note that the classification is not rigid, and that any particular user may fall into more than one group.

7.1.2 Display Objects

To maintain consistency within the IDES user environment, it is useful to define a set of standard data presentation formats so that the user can view and manipulate data in a similar fashion regardless of what information is being displayed.

For example, the mouse buttons should invoke the same functions throughout the entire user environment. A sample arrangement would be to have the right mouse button bring up a menu with options to be invoked, while the left button selects items that are already displayed on the screen, or confirms a selection that has been made. The middle button might be used to activate window commands, such as moving, resizing, and closing the current window.

Likewise, data displays must be uniform throughout the user interface. Commands and data results for either system-related information or IDES-specific data should be presented in the same manner.

Data views are static presentations of IDES information, and can be invoked at any given time. Information is presented as a "snapshot" of the processed data at a specified moment or period of time. To provide a consistent user interface environment, a standard window format is used to display information. Each window contains the following four components:

The Selection Criteria subwindow is the section of the data view window that is reserved for any search parameters the user may want to specify. Some selection criteria may be mandatory, such as an ID number or a category type; these are labeled so that the user can clearly see that such information must be provided before executing the request (such as boldfacing or highlighting).

The Commands subwindow contains various options pertaining to the information being' requested. These functions include commands like *HELP, EXECUTE, PRINT,* and *QUIT.* These commands are displayed as items that can be selected with the mouse. If a request takes a few minutes (or even seconds) to execute, then the command selection is shaded or blacked out to indicate that the request is still in progress so as not to mislead the user into thinking that his request was not acknowledged by the system.

The Display subwindow presents the information as specified by the selection criteria in a list format. This window is scrollable up or down, left or right. In addition, the items in the displayed list in this subwindow are also selectable for detailed information. This is done by picking an item in the displayed list with the mouse button, and then selecting one of the command options to display appropriate parameter information. The detailed information is either displayed in the same data view window, or a new window is brought up.

The Messages subwindow provides a place for status messages to appear, whether produced by the execution of the data view request or from the system itself. Invocation error messages (such as not specifying enough selection criteria) are also displayed in this window.

Users may often want to see IDES information presented dynamically (i.e., in real time), especially if they wish to track a potential intrusion closely. These interfaces are referred to as monitors. Data monitors are typically graphical in nature, as it is quicker to note unusual events with visual aids as opposed to reading and interpreting text. The following are some samples of graphical presentations that might be used throughout the IDES user interface:

Plotted graphs are graphs of events from a given time to the current time, and are updated frequently (every minute or less). They indicate trends of activity, and are useful in tracking down potential anomalous behavior.

Meters indicate the level of activity in the system. They are usually measured in percentage units; that is, if an activity reaches a maximum allowable state, the meter will be marked at the l00-percent level. These data monitors help keep track of things such as processing rates and data storage usage.

Flashing icons are very effective visual aids to indicate a critical situation or event. They may be supplemented with audio mechanisms (e.g., beeps, buzzes) to emphasize the importance of an anomalous condition. Such monitors are used primarily for events that occur relatively infrequently, but when they do, the user must be notified as quickly as possible.

Data monitors are not limited to graphical displays. Textual presentations are also utilized, especially for occasions in which messages are produced to explain a situation in more detail:

Dynamic lists are similar to those lists presented in data views, except that the list scrolls (or redisplays) automatically as the process receives new information.

Message displays are the textual complements to flashing icons. Coupled with a flashing icon, message displays can provide more meaningful information to the user when something unusual occurs.

It should be mentioned that data monitors may potentially slow down the performance of the IDES processing, although usually not significantly. This is because each monitor runs as its own process, and must request data at frequent intervals to maintain a real-time status. (Alternatively, a triggering function could be used to signal these monitor processes to notify them of newly arrived information.) Data monitors do not allow additional querying (i.e., the user cannot pick an item from a dynamic list for detailed information). If a user wishes to elaborate on a particular piece of information, then the data views are to be used.

Ad-hoc queries must also be addressed. These are requests for random data that do not have a predetermined method of execution. This type of query is best supported by a fourth-generation language (such as SQL) that is generally provided by a standard database management system. However, integrating such a system into the user interface would require the users to learn how to write SQL queries, or to use whatever forms interface is supported by the DBMS, hence increasing the difficulty of maintaining a consistent user interface overall. Ideas on a more sophisticated interface for ad-hoc querying are still being sought.

7.2 User Interface Components

This section describes in detail the different user interface components that are built within IDES. We have built a security officer's user interface, which integrates the output from the statistical and expert system components. We have also implemented a series of Data Analyst tools for the statistical anomaly detector, primarily used by the statisticians for algorithm analysis and experimentation.

7.2.1 Security Officer's User Interface

The security officer must have information ready at hand to quickly track down abnormal system behavior once it has been detected. The SO's primary concern is the occurrence of target system intrusions. Once an anomaly is detected and confirmed by the appropriate analytical engines in the IDES processing environment, the SO is notified by a flashing graphical object and/or a textual message displayed in a designated section of the screen.

After recognizing the intrusion, the SO will generally want to track down the source of the abnormality. Important information for this purpose might include the host on which the intrusion occurred (in a multitarget environment), who/what the subject is, and what other entities were affected. A window can be brought up to show a list of anomalies that have been discovered by the system and an indication of the level of severity of each anomalous record. By picking a particular anomaly in the list the SO may be able to trace the path back to the original audit record(s) from whence the event propagated. In addition, the SO may be able to examine other information that contributed to the anomalous event.

The SO should be able to define various criteria for a set of audit records to be displayed, such as the subject type (user, target system, remote host) and identification number, time period in which they were recorded, and records received from a particular host.

The SO may also wish to actively monitor a particular subject (due to observation or notification of previous suspicious behavior). An activity monitor can be used to dynamically display a list of activities (perhaps a permuted representation of each relevant audit record) for that subject as each audit record gets processed and incorporated into the IDES processing environment.

The IDES security officer user interface (SOUI) is a real-time display that allows the SO to monitor active users on the target system, and at the same time provides alerts to signify anomalous events. Such events are either known security violations that have been traced by the expert system, or a significant threshold of abnormal behavior as detected by the statistical analysis component. The SOUI also provides a means for the SO to closely examine and analyze suspect behavior by retrieving a range of audit records (for a particular user) that potentially contributed to the abnormal/illegal use of the target system.

Main SOUI Window

The main window of the SOUI displays a list of active subjects. When a new subject enters into the target system, its name appears in the window, as does a host name if the subject is remotely logged in to the target system. The sequence number and timestamp of the most recent target audit record for each subject is constantly updated and displayed in this window, giving the SO an indication as to how much and how recently a subject has been active on the target system.

Also displayed in the main window is a statistical score value sent from the statistical anomaly detector (see Chapter 5 for details on what this score value means), and it is also updated regularly.

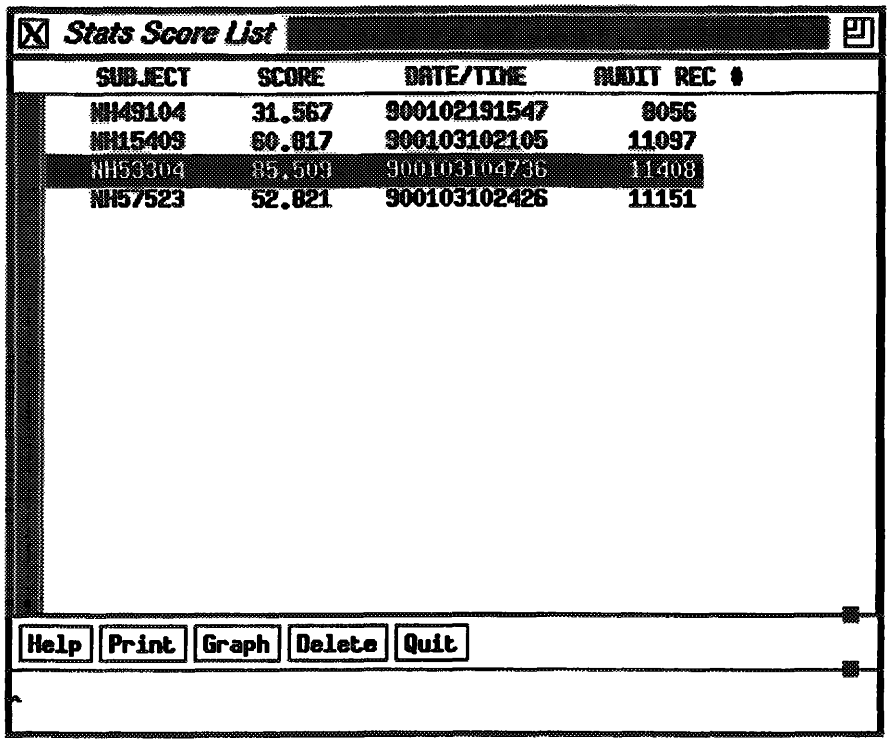
The SO may delete any subject from the main SOUI window, especially as the list of subjects grows to a large number and certain subjects may no longer be active. This may be done by selecting the subject in the list with the mouse to highlight that entry, and then selecting the DELETE option in the command box of the window. However, if a graph window is currently being displayed for this subject, the name will not be deleted from the list. Figure 7.1 shows a sample display of the main SOUI window.

Finally, the bottom-most view of the SOUI main window is reserved for informational messages to the SO. Typical messages are operational errors, such as the SO's trying to delete a subject from the list while its anomaly graph is still being displayed, or miscellaneous status data, such as a subject's logging in to the target system from a remote host (message sent by the expert system).

Online help is available at each window level, and can be selected by clicking on the HELP button in the command box.

Anomaly Graph Window

The SOUI has a subwindow that can be displayed to view the trend of statistical scores that have been produced by each subject. As each audit record is processed for a subject, a score value is produced, and this score value is plotted in real time onto a graph subdivided into four quadrants. These quadrants currently represent the log base 10 value of the scores; that is, the first quadrant (bottom level) represents values between



Stats Score List

SUBJECT	SCORE	DATE/TIME	AUDIT REC #
NH49104	31.567	900102191547	8056
NH15409	60.817	900103102105	11097
NH53304	85.509	900103104736	11408
NH57523	52.821	900103102426	11151

Figure 7.1: SOUI Main Window

0 and 10.0, the second quadrant represents values between 10.0 and 100.0, and so forth. The intent of this subdivision of values is to distinguish "yellow zone" scores from "red zone" ones, allowing the SO to easily view when a subject is approaching a critical level of abnormal behavior (as indicated by the score values). A sample display of the anomaly graph window is shown in Figure 7.2. The score ranges within these zones still need to be determined as more experiments are done with the statistics. As a future enhancement, the zone definitions will be made to be flexible and configurable.

The SO can also select any portion of the displayed graph by using the mouse. If the audit record archiver process is turned on, then another display window containing a list of audit records pertaining to that selected portion of the graph will be shown. This enables the SO to trace a portion of a subject's activity that contributed to the score graph indicated.

Other features at this window level include printing a hardcopy of the graph, and a refresh option to start the graph.

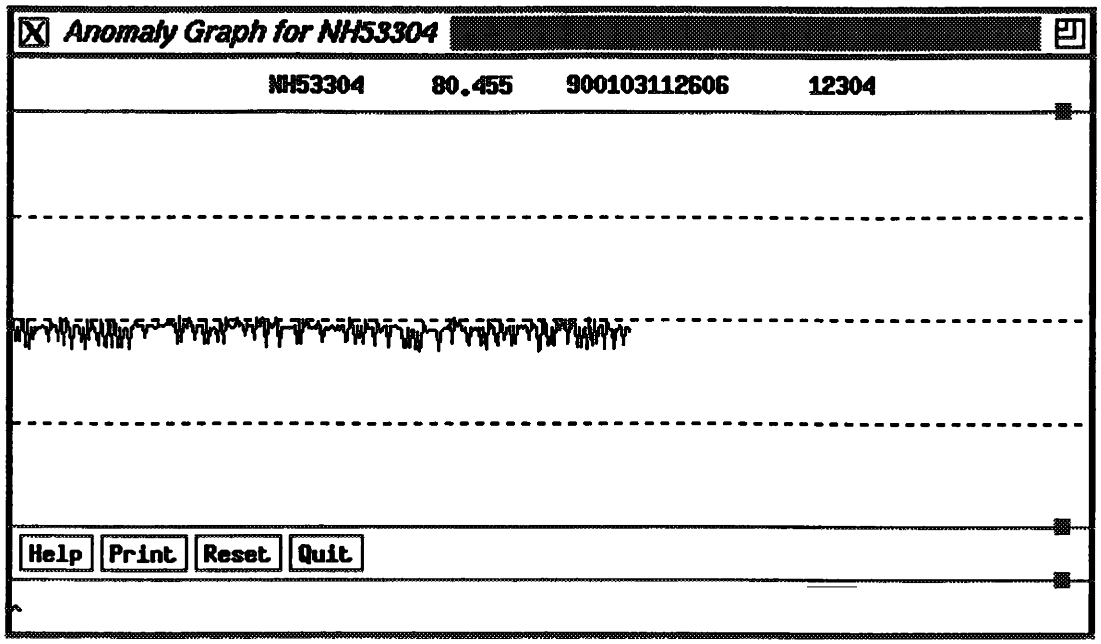


Figure 7.2: SOUI Anomaly Graph Window

Alert Windows

Alert windows are currently divided into two criticality levels. Level 0 is the most critical level, and takes up the top half of the alert window. Level-1 alerts are usually just warnings, letting the security officer know that something unusual may happen.

These alert windows appear automatically, with a flashing background to attract the SO's attention. The SO must acknowledge the alert window for the flashing to discontinue. This is done by clicking on the ACKNOWLEDGE button in the command box. This window remains in existence, and each time a new alert comes in, the flashing occurs.

Alert windows are triggered when one of the following events occurs:

- The score value for a subject reaches a specified threshold value. When the score value approaches the red zone, then the warning level of the alert window flashes. When the score value reaches or exceeds the threshold value, the critical section of the alert window is triggered.
- A subject is traced to a leap frog event - that is, when a user has logged in to one host to log in to another host. This is a warning-level message only.
- A subject has touched his .login file, which triggers a critical-level alert.

Future plans for the SOUI will include the following features.

- More detail on the cause of an alert
- Display of archived records for the statistics
- Journal of anomalous events, which would involve standardizing the format that represents anomalies.

7.22 Data Analyst Information

A Data Analyst (DA) is generally interested in the anomaly-detection components of IDES (i.e., the statistical module and the expert system).

Statistical profiles retain accumulated information about a subject's behavior on the target system. The DA who is interested in the statistical analysis of anomaly detection can view such information. These data include measures that are actively being monitored for each subject, various statistical data for both discrete and continuous measures, and a set of current (for that day) and historical (accumulated over a period of time) profiles of a subject's behavior.

Some examples of expert-system information that might be interesting to the DA are the rules that were applied to detect anomaly and backtracking of rule path followed by the system. Some of this capability is described in detail in Chapter 6.

Data analysts may also want to replay some data to test the accuracy of the anomaly-detection units. Hence, the tools used by the SO users may also be used by the DAs to configure and tweak at various parameters for experimentation.

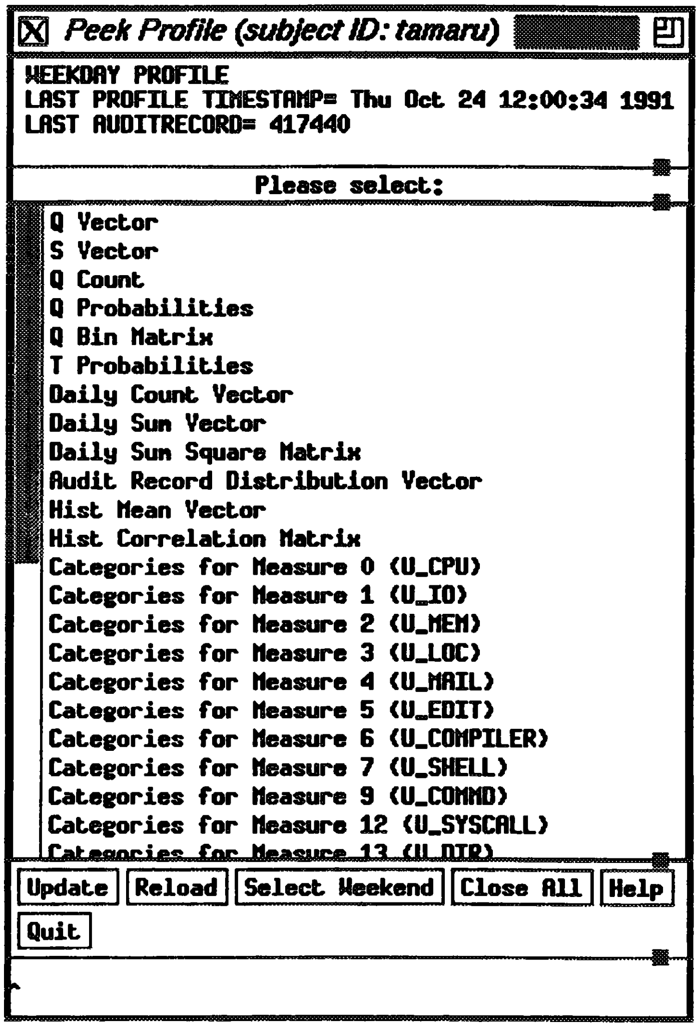
We have implemented three such user interfaces for the data analyst, in particular for the statistical component. Details are described in the following subsections.

Peek

The *peek* program is a tool used for the analysis of subject profiles. Most of the information presented is meaningful only to data analysts (particularly statisticians) with comprehensive knowledge of the algorithms used in the IDES statistical component, and is generally not much use to the general security officer. See Figure 7.3 for a sample *peek* display. *Peek* has been used primarily for algorithm debugging and testing.

The main window of *peek* shows a list of subjects whose profiles are available for browsing. Each subject has two sets of profiles, one for normal work usage (weekdays) and another for weekend/holiday use; either one of these profiles may be selected for viewing. The subject profiles are represented as a variety of probability vectors and matrices, and are displayed in this manner. Again, unless one is familiar with the algorithms used for statistical anomaly detection, most of this information may seem rather cryptic to the average user.

Peek: can run interactively with the active IDES statistical component by sending signals (propagated by the user) to the statistical unit to request dumps of the latest profile information that has been accumulated in the process cache.



```
WEEKDAY PROFILE
LAST PROFILE TIMESTAMP= Thu Oct 24 12:00:34 1991
LAST AUDITRECORD= 417440
```

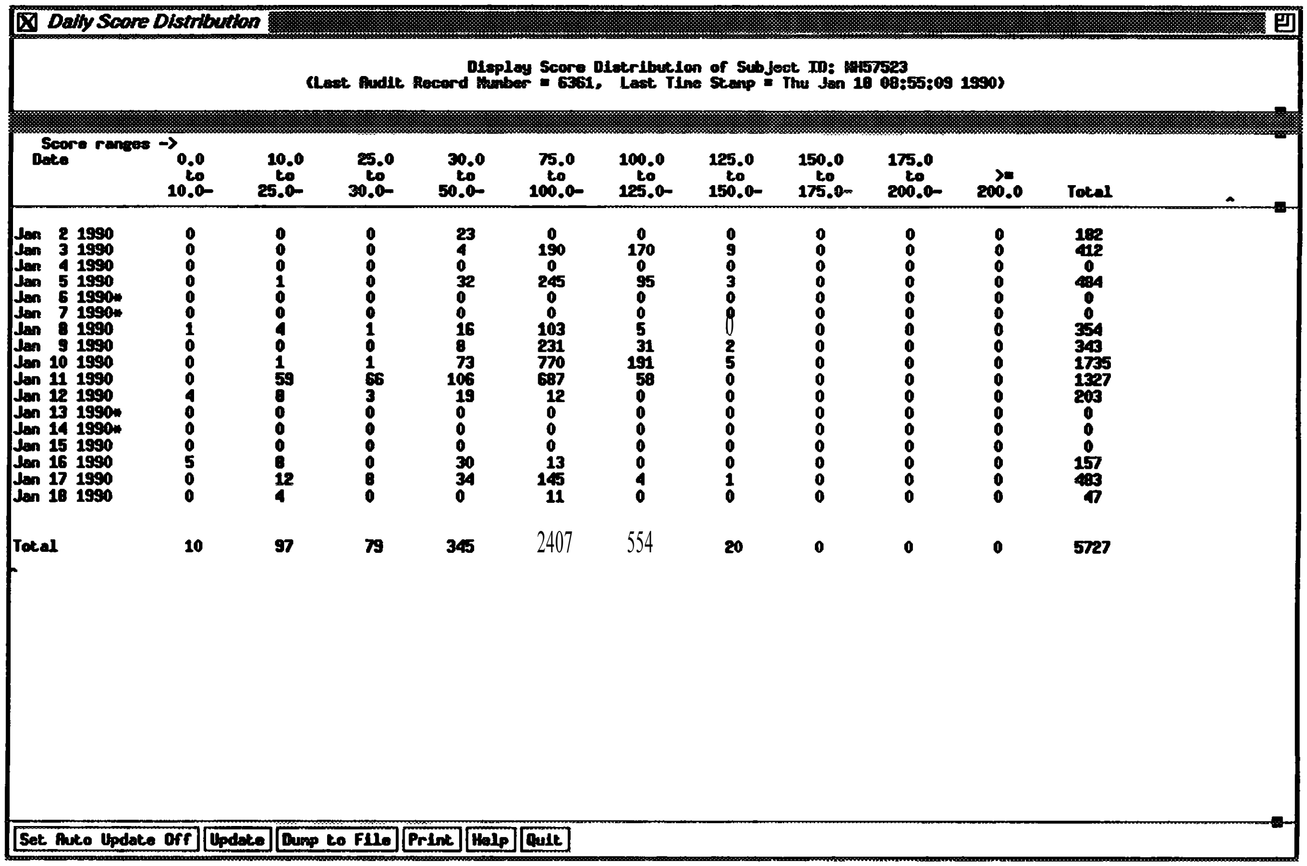
```
Q Vector
S Vector
Q Count
Q Probabilities
Q Bin Matrix
T Probabilities
Daily Count Vector
Daily Sum Vector
Daily Sum Square Matrix
Audit Record Distribution Vector
Hist Mean Vector
Hist Correlation Matrix
Categories for Measure 0 (U_CPU)
Categories for Measure 1 (U_IO)
Categories for Measure 2 (U_MEM)
Categories for Measure 3 (U_LOC)
Categories for Measure 4 (U_MAIL)
Categories for Measure 5 (U_EDIT)
Categories for Measure 6 (U_COMPILER)
Categories for Measure 7 (U_SHELL)
Categories for Measure 9 (U_COMMD)
Categories for Measure 12 (U_SYSCALL)
Categories for Measure 13 (U_DIR)
```

Figure 7.3: Peek Selection Window

Scdist

The *scdist* program collects the cumulative score values for each subject and sorts them into range bins. It is like a spreadsheet of score information, enabling the DA to view a summary of the score trends produced by a particular subject. This tool is used often for experimentation of the statistical component, to see if the tweaking of various statistical parameters will affect the value of the T^2 scores. Figure 7.4 shows an example of an *scdist* display.

The ranges of the score values can be modified to be larger or smaller, depending on how fine the user wishes to view the distribution. *Scdist* can be run in automated mode, where the score distribution display is updated every 10 seconds. A snapshot of the current distribution can be taken any time, and either be saved into an ASCII formatted file or sent directly out to a printer.



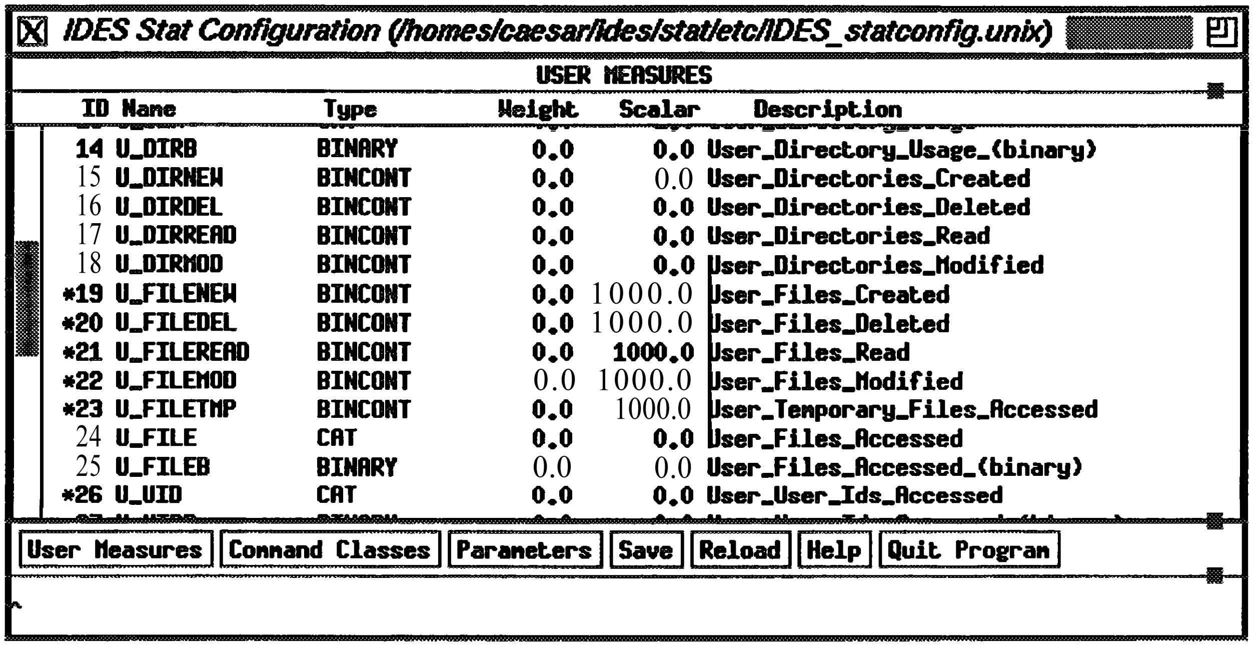
```
                    Display Score Distribution of Subject ID: WH57523
        (Last Audit Record Number = 6361,  Last Time Stamp = Thu Jan 18 08:55:09 1990)

 Score ranges ->
 Date          0.0     10.0     25.0     30.0     75.0    100.0    125.0    150.0    175.0
                to       to       to       to       to       to       to       to       to      >=
              10.0-    25.0-    30.0-    50.0-   100.0-   125.0-   150.0-   175.0-   200.0-   200.0     Total

Jan  2 1990     0        0        0       23        0        0        0        0        0        0       182
Jan  3 1990     0        0        0        4      190      170        9        0        0        0       412
Jan  4 1990     0        0        0        0        0        0        0        0        0        0         0
Jan  5 1990     0        1        0       32      245       95        3        0        0        0       484
Jan  6 1990*    0        0        0        0        0        0        0        0        0        0         0
Jan  7 1990*    0        0        0        0        0        0        0        0        0        0         0
Jan  8 1990     1        4        1       16      103        5        0        0        0        0       354
Jan  9 1990     0        0        0        8      231       31        2        0        0        0       343
Jan 10 1990     0        1        1       73      770      191        5        0        0        0      1735
Jan 11 1990     0       59       66      106      687       58        0        0        0        0      1327
Jan 12 1990     4        8        3       19       12        0        0        0        0        0       203
Jan 13 1990*    0        0        0        0        0        0        0        0        0        0         0
Jan 14 1990*    0        0        0        0        0        0        0        0        0        0         0
Jan 15 1990     0        0        0        0        0        0        0        0        0        0         0
Jan 16 1990     5        8        0       30       13        0        0        0        0        0       157
Jan 17 1990     0       12        8       34      145        4        1        0        0        0       483
Jan 18 1990     0        4        0        0       11        0        0        0        0        0        47

Total          10       97       79      345     2407      554       20        0        0        0      5727
```

Figure 7.4: Scdist Display Window

Statconfig

The statistical anomaly detector runs based upon a particular configuration of statistical parameters. This information can be modified by the SO or DA via the statistics configuration program *(statconfig),* which is a graphical interface to the configuration file.

Three types of parameters are currently used by the statistical program: *user measures, command classes* and *system parameters.* Once modified, these parameters are used in



```
IDES Stat Configuration (/homes/caesar/ides/stat/etc/IDES_statconfig.unix)
                               USER MEASURES
   ID Name           Type         Weight   Scalar   Description
   14 U_DIRB         BINARY         0.0       0.0 User_Directory_Usage_(binary)
   15 U_DIRNEW       BINCONT        0.0       0.0 User_Directories_Created
   16 U_DIRDEL       BINCONT        0.0       0.0 User_Directories_Deleted
   17 U_DIRREAD      BINCONT        0.0       0.0 User_Directories_Read
   18 U_DIRMOD       BINCONT        0.0       0.0 User_Directories_Modified
  *19 U_FILENEW      BINCONT        0.0    1000.0 User_Files_Created
  *20 U_FILEDEL      BINCONT        0.0    1000.0 User_Files_Deleted
  *21 U_FILEREAD     BINCONT        0.0    1000.0 User_Files_Read
  *22 U_FILEMOD      BINCONT        0.0    1000.0 User_Files_Modified
  *23 U_FILETMP      BINCONT        0.0    1000.0 User_Temporary_Files_Accessed
   24 U_FILE         CAT            0.0       0.0 User_Files_Accessed
   25 U_FILEB        BINARY         0.0       0.0 User_Files_Accessed_(binary)
  *26 U_UID          CAT            0.0       0.0 User_User_Ids_Accessed
```

Figure 7.5: Statconfig Program (measure modifications)

the next invocation of the statistical component.

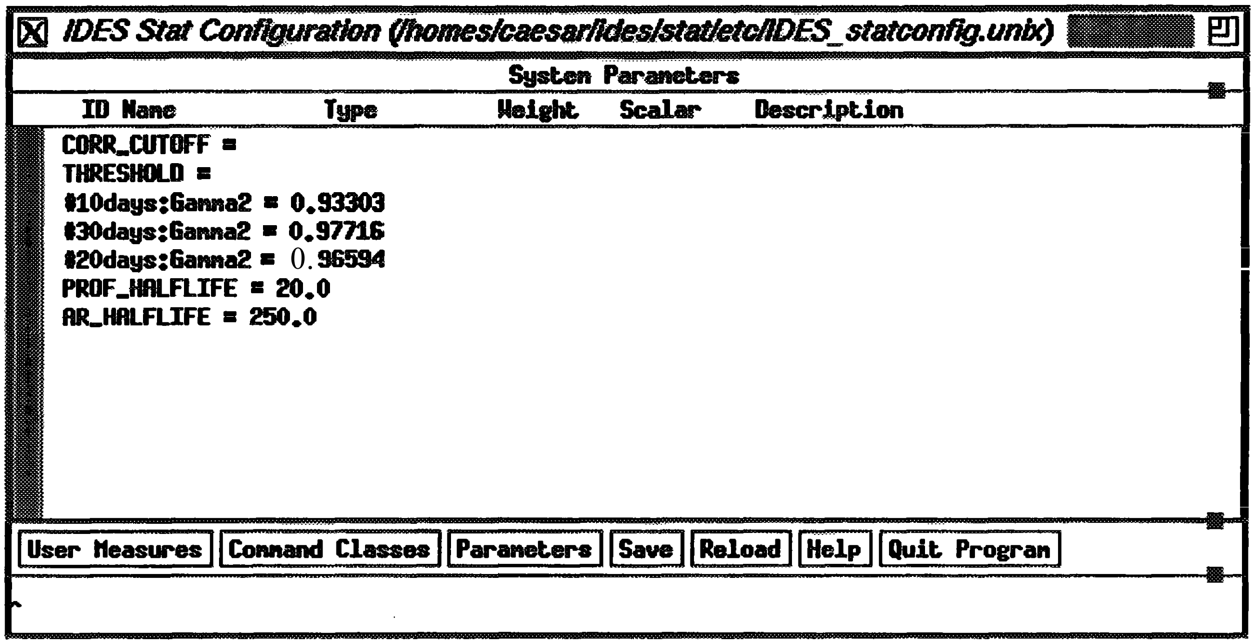
The configuration file contains a list of all possible measures that can be used by the statistical component (see Chapter 5 for a detailed description of these measures). The user can modify a variety of characteristics for any measure, such as activating or deactivating it, reclassifying its type and/or description, or changing its contributory weight or scalar values for the statistical algorithms (these latter items are particular to the statistical algorithms and probably would not be changed directly by the SO). Changes can be made easily by selecting on the appropriate measure and filling in the proper information (see Figure 7.5).

A variety of statistical parameters can be modified by the SO or DA, allowing the statistical component to be tuned for a particular scenario or set of users. Currently, the following items can be modified:

Audit Record Half-life. This parameter determines the half life of audit records processed for each subject.

Profile Half-life. This parameter is the subject profile half-life, represented as a number of days.

Correlation Cutoff. This value is used by the profile updater to determine how close the correlation of measures should be (used only if a correlation matrix is used).



```
CORR_CUTOFF =
THRESHOLD =
#10days:Gamma2 = 0.93303
#30days:Gamma2 = 0.97716
#20days:Gamma2 = 0.96594
PROF_HALFLIFE = 20.0
AR_HALFLIFE = 250.0
```

Figure 7.6: Statconfig program (parameter modifications)

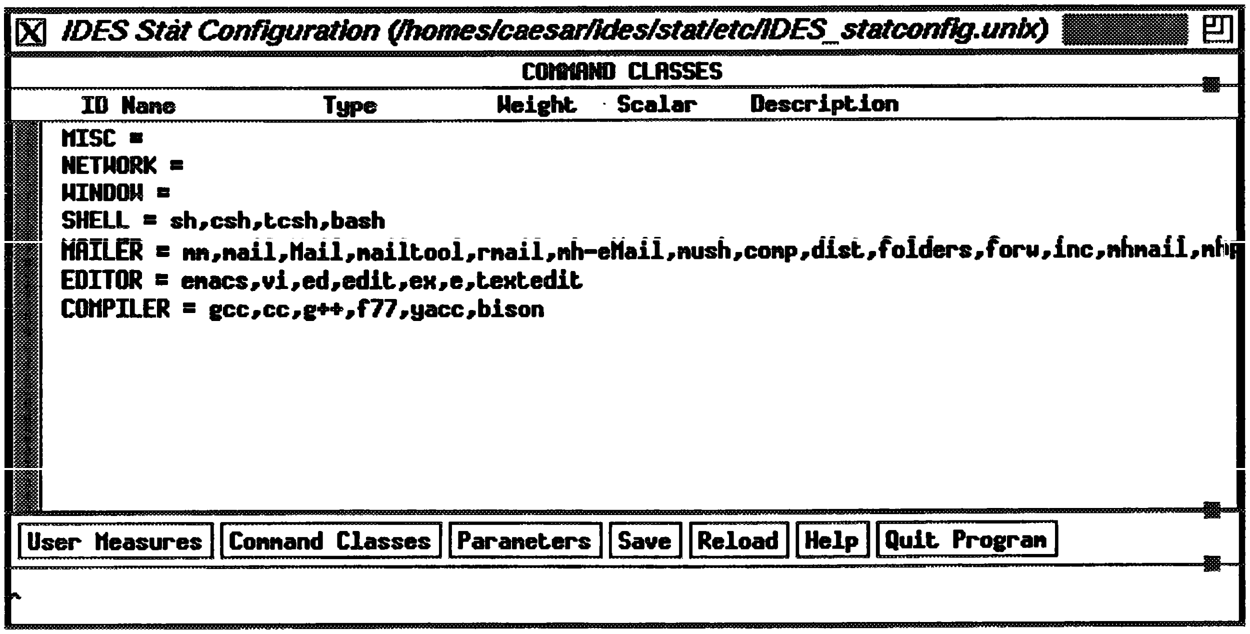
Score Threshold. If this score value is exceeded by any user, an alert is sent to the security officer's user interface. Currently there is one value for all users, but it will be modified in the future to allow different threshold levels for different subjects or subject groups.

Figure 7.6 shows a sample *statconfig* window with which the statistical parameters can be changed.

The configuration file also contains the classification of specific commands, such as mailers and editors. If the statistical measures for any of these special commands are turned on, then a list of commands that fall into specific command classes must be defined. This list can be updated by the security officer at any time using the *statconfig* program (for example, when a new mailer system is installed on the target system). Each item in the list should be separated by a comma. An example of the command classes section of the configuration file is shown in Figure 7.7.

7.2.3 System Administration Data

The IDES system administrator (SA) is presented with a variety of functions that apply to the basic system administration tasks, such as installation, configuration, and general upkeep of the IDES system. While an SA is likely to have some general experience with the operating system functionalities, the tools to aid him or her should present a



```
ID Name            Type         Weight   Scalar    Description
MISC =
NETWORK =
WINDOW =
SHELL = sh,csh,tcsh,bash
MAILER = mm,mail,Mail,mailtool,rmail,mh-eMail,mush,comp,dist,folders,forw,inc,mhmail,mh
EDITOR = emacs,vi,ed,edit,ex,e,textedit
COMPILER = gcc,cc,g++,f77,yacc,bison
```

Figure 7.7: Statconfig Program (command classes)

user interface style consistent with the overall IDES user environment. Procedures are automated to the extent possible and are designed to allow the novice user to maintain the system with a minimal amount of system operating skills.

It is the SA's responsibility to constantly be aware of the system's condition. IDES has several processes running concurrently, and hence tools are provided to show the current state of each process. Some examples of the types of information that might be shown are:

- Number of audit records received from each target
- Number of rejected audit records
- Status of audit record processing in each component
- Processing throughput for each component
- Dates/times when each component was last invoked
- Storage utilization (how full the data storage area is)
- Overall system load
- Locations of components if running in a distributed environment.

SAs must also control the system's functions, such as initializing or terminating IDES processes. Although most of the IDES processes are automatically started either during the installation process or by another process, there are occasions (such as for system maintenance purposes) when any of these must be temporarily halted and later restarted (manual override). There are also situations (although rare) when a process may terminate abnormally and require a manual startup.

The SA may use the archiver process to handle most of the storage hygiene procedures, such as purging old data from the system. Explicit instructions will be provided and automated wherever possible to minimize the number of steps the system administrator needs to take to execute these procedures.

The SA is also responsible for the installation of IDES, and the process should be simple and straightforward. The user interface should carefully guide the installer to ensure the proper sequence of events, and also allow backtracking of steps in case something has been overlooked or cannot be started properly. There should also be an internal checklist of parameters to make sure everything has been installed correctly.

While configuration is part of the installation process, it may also be applied during system execution. Configuration commands should allow the creation and modification of user-definable parameters in IDES (for example, which measures to observe), definition of expert-system rules to be triggered for anomaly detection, setup of location maps for processes residing in a distributed environment, and indication of realm-specific references to processing components.

At this writing, we have implemented a few programs that are currently being used primarily as diagnostic and debugging tools for the realm interface components. As a future task, our plan is to convert these tools into a graphical user interface, thus making it more usable by an SA or SO who will be responsible for installing and configuring the IDES system for a particular environment.

7.3 IDES User Interface Library *(libiui)*

We elected to use the X Window System to implement the IDES user interface. It is portable, has remote viewing capabilities, and provides a wide variety of programming tools to build one's own catalog of viewing objects. A set of IDES data objects was implemented and incorporated into a software library that any IDES component requiring a user interface can use. This is the IDES user interface library, or *libiui.*

Libiui assumes a hierarchical, object-oriented interface to the underlying windowing system. It is based upon the concept of windows and views, with a view being a section of a window that can be manipulated as a unit. A view with more than one child is called a *composite view,* and the location and size of each child is controlled by the parent view.

Some composite views treat all children equally, while others allow the application to specify certain constraints for each child view. Examples of constraints are read/write-only capability, mandatory input, position in a larger view, and boldface or highlighting characteristics.

The following views have been defined as the set of IDES data objects.

Stack views are composite views that stack their children vertically. The children have the same width as the parent view, and their vertical area is divided according to the constraints that have been set for them.

Box views arrange their children in order, from left to right, top to bottom. The box places as many children left to right on a line as will fit, then moves down to the next line.

Scroll views have exactly one child. The child view may be larger than the parent view, which would then allow the user to scroll either vertically or horizontally to see the entire child window. Scroll views are equipped with scroll bars along the sides and bottom of the window.

Label views display a line of text. They are generally used to tag fields or provide headers for columns.

Button views respond to mouse clicks by calling a function passed in by the application.

Toggle views look like button views, but remain highlighted until clicked again.

Question views allow a user to respond to a given question. The user is given a standard mouse-driven text-editing interface to enter in a response.

Multiple Choice views come in two flavors: one presents a list of all the choices simultaneously, while the other shows the choices one at a time (scrolls through the field). In both cases, only one choice can be selected.

List views display lists of information to the user. The user can scroll vertically or horizontally through the list, if the data cover more than the available screen space. Items in the list can be selected and functions invoked as specified by constraints defined by the parent view.

Message views display messages, such as status and error messages, to the user. A scrollbar is included for long-winded text.

Graph views are also called *strip charts,* and visually represent a history of some event(s). Graph views can be displayed in two modes: *display* mode shows a graph that is continuously moving, while *selection* mode is a snapshot representation of a particular moment in time. The user is allowed to pick out a region of the graph in selection mode and invoke a user-defined function.

Meter views may be represented as bars or in “speedometer” style (type of scale with a pointer indication measure amount).

These views have been implemented and are used by the IDES user interface components described in the previous section.

Chapter 8

GLU

One of the main goals of the IDES prototype is to detect and report anomalous activities as they occur despite partial failures and while accommodating changes in the volume of audit data being processed. This chapter describes how this important system goal is realized. First, the specific system requirements for the IDES prototype are identified. Next, we justify the use of a multiprocessing methodology to meet the requirements. We then describe a platform for multiprocessing (called *GLU* for Granular Lucid) and discuss its capabilities. Finally, we present the structure of the IDES prototype on the GLU platform and consider how well it meets the requirements.

8.1 System Requirements

The IDES prototype has to satisfy the following system-level requirements.

1. Detect anomalous activity as, or soon after, it occurs. This is the *soft real-time processing* requirement.
2. Continue to detect anomalous activity despite partial failures of the prototype, albeit with degraded performance. This is the *fault-tolerance* requirement.
3. Accommodate increased amounts of activity by proportional hardware addition without compromising soft real-time processing or fault tolerance. This is the *scalability* requirement.

8.2 The Multiprocessing Approach

An approach to meeting the above requirements is to rely on specialized technology. By choosing a customized system that is sufficiently fast, fault tolerant, and easily expandable, the three requirements can be met. This approach has the following drawbacks.

1. A substantial and nontrivial programming effort would be needed to implement IDES on the chosen system so as to satisfy all three requirements.
2. The peculiarities and limitations of a given system often become evident only after considerable time and cost.

An alternative approach is that of multiprocessing, in which current off-the-shelf technology is used to meet all three requirements. Multiprocessing refers to the use of several communicating autonomous computers to execute a single application such as IDES. With the multiprocessing approach, speed is achieved by exploiting natural implicit parallelism in the activity instead of using fast hardware technology. The multiplicity of processors, storage, and communication components facilitates detection and recovery from failures of some components. Scalability is achievable because a multiprocessing system can be easily expanded and its performance, should scale proportionally.

We have adopted the multiprocessing approach to meet the requirements of the IDES prototype. While the underlying multiprocessing system is simple in its architecture - for example, a network of conventional workstations - the methodology to map an application onto this system is novel. A key aspect of a successful multiprocessing methodology is the extent to which the mechanism and architecture of the methodology are hidden from the applications (in this case the components of the IDES prototype) they support. In our opinion, this not only reduces programming effort and enhances portability, but it enables the IDES prototype to fully meet the requirements.

8.3 GLU: A Software Platform for Multiprocessing

The multiprocessing software platform called *GLU* [24,25,26] allows for development and implementation of IDES on a multiprocessing system such as a network of workstations. The methodology allows the following concerns to be hidden from the application (IDES).

- Multiprocessing system architecture
- Explicit creation and management of parallelism
- Explicit detection and recovery from partial failures.

The platform consists of the programming model and the execution model.

8.3.1 GLU Programming Model

A GLU program is a dataflow graph whose vertices denote functions and whose edges denote data dependencies between functions. A function is either predefined or it is defined by the programmer in a procedural language such as C. Note that issues of how to exploit parallelism and how to react to partial failures are hidden from the GLU program.

Operationally, a GLU program should be viewed as a mapping from a set of input sequences of values to an output sequence of values. Each edge denotes a sequence of values and each vertex denotes a function that consumes its incoming sequences to generate its output sequence of values. Each edge value is referred to by the name of the edge and its *context,* which is the conceptual position of the value in the edge sequence.

The context can be viewed as an n-tuple of integers that uniquely identifies a value of an edge. When the context is n-dimensional, the sequence associated with an edge corresponds to a set of values in an n-dimensional space.

The dataflow graph can be expressed in a simple, equational language as a set of equations. The expressiveness of the language is illustrated by describing the expert system in it (see below and Figure 8.1).

```
// constants
#define BLOCK_SIZE 2000
#define MN_BUNDLE_SIZE 128

// data structure definitions
struct ar_record
        {
          char rec[ar_len];
          struct ar_record *arp_next;
        };
typedef struct ar_record *AUDIT_RECORD;

struct bt_entry
         { AUDIT_RECORD arp_first;
           AUDIT_RECORD arp_last;
           int bsize,csize;
           string uname;
         };
typedef struct bt_entry *AR_BUNDLE;

struct ar_block
        {
          AR_BUNDLE bundle[ mu ];
        };
typedef struct ar_block *AR_BLOCK;

struct filename
        {
          char fn[flen];
        };
typedef struct filename *FILENAME;

// function prototype definitions
local AR_BLOCK get_ar_block( int, int );
local int num_users( AR-BLOCK );
local AR_BUNDLE select_bundle( AR_BLOCK, int );
local int user_name( AR_BUNDLE );
      int es_process( AR_BUNDLE, int );

// GLU program
e where
    e = do_expsys( scatter( blk ) )
        where
          blk = get_ar_block( BLOCK_SIZE, time );
          scatter( block ) = p
          where
            bs = 0 fby bs + num_users( block );
            s = bs upon advance;
            advance = (next time == z);
            z = s + num_users( b );
            b = block upon advance;
            p = select_bundle( b, time-s );
          end;
          do_expsys( bundle ) = score
            where
              user = user_name( bundle );
              score = ( sc @ i ) @.u user
                where
                  index u;
                  i = time upon u == user;
                  user_bundle = bundle wvr u == user;
                  SC = ( first invoke( user_bundle, 0 )
                       fby
                       next invoke( user_bundle, prev sc )
                       )
                       where
                         invoke( ub, sc ) = es_process( ub, time );
                       end;
```

```
                    end;
                end;
            end;
end
```

The GLU program shown above accepts, as input, blocks of audit records from an external source. This way the communication cost is amortized over several records. Procedural function `get_ar_block,` when given block size and block id, returns a block of audit records as a C structure `(AR_BLOCK.)` The block, which may consist of audit records from different users, is divided or scattered into smaller blocks (or bundles) where each bundle consists of audit records from a specific user. This is accomplished using the GLU function `scatter` and C functions `num_users` and `select_bundle.` Each record in each bundle is processed by the expert system (implemented by `es_process)` by invoking the GLU function `do_expsys` on each bundle.

8.32 GLU Execution Model

The GLU platform provides the programmer with important benefits of implicit coarse-grain parallelism, transparent fault tolerance, and architecture independence. These benefits are realized by using a novel model of parallel execution called *eduction.* Eduction corresponds to lazy dynamic dataflow [27]. An important consequence of laziness is that while it shares the greater asynchrony of its eager counterpart, it avoids superfluous computation altogether. This is particularly important since procedural functions with side effects in GLU programs should not be invoked when unnecessary, even if the side effects are benign as far as the computation is concerned.

The most natural way to realize education is by tagged demand-driven execution. Simply speaking, a term in a GLU program is evaluated (at a particular context) only when it is demanded at that context (as identified by the demand's tag). The evaluation consists of two phases: first, simultaneous demands for the constituent subterms at appropriate contexts are propagated and, second, the function associated with the term is applied when the demanded values of each satisfied subterm are available, thereby producing the originally demanded tagged result.

An important aspect of eduction is that it does not recompute the value of a term at a given context (or tag) if that value has been previously demanded. Instead, the tagged value is stored when the initial demand for it is satisfied, and is used to satisfy subsequent demands for the value.

The eduction model exploits two kinds of parallelism. The first kind is functional parallelism, which is exploited in the simultaneous computation of tagged values of subterms demanded to produce a tagged value of a term. The granularity of parallelism is coarse if the subterms correspond to procedural functions.

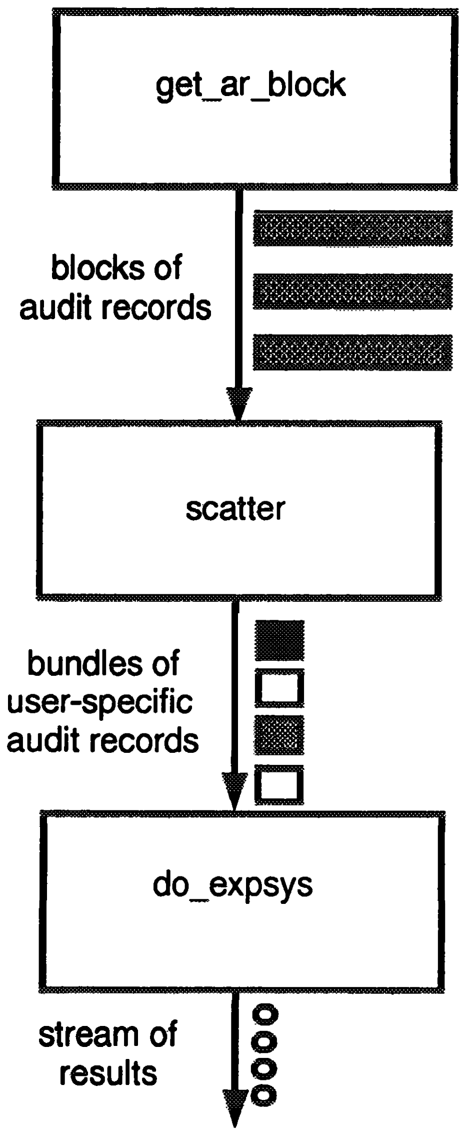


Figure 8.1: IDES Expert System in GLU

Given the program fragment, `f( g( y ), h( z ) ),` if the tagged value `f(...):c` is demanded (`c` is the tag of the value that is sought), demands for `g(...):c` and `h(...):c` would be simultaneously made, which in turn would demand tagged values `y:c` and `z:c.` When the latter become available, functions `g` and `h` can be applied simultaneously, resulting in functional parallelism.

The second kind of parallelism is context parallelism (which loosely corresponds to data parallelism [25]). This is exploited in the simultaneous evaluation of the same term at multiple contexts. As a simple example, with the above program fragment, the term corresponding to `f(...)` can be demanded simultaneously at contexts `c` and `d` and the demanded tagged values of subterms at the two contexts can be computed independently since there is no contextual dependency within the term. Further note that the exploitation of context parallelism can occur independent of the exploitation of the functional parallelism.

The program given in Figure 8.1 exhibits data parallelism in the simultaneous processing of audit record bundles of different users. In other words, while audit records of a given user are required to be processed in order, audit records of independent users can be processed simultaneously. This parallelism is illustrated in Figure 8.2. The parallelism exploited is coarse grain because a bundle (instead of an individual audit record) is used as a unit of audit information If each bundle consists of only one audit record (i.e., fine grain), the cost of communication of the bundle (overhead) will far outstrip any gains made in exploitation of parallelism in simultaneous processing of the individual audit record. Thus, we are currently experimenting with the effect of increasing bundle size on performance.

8.3.3 Recovery from Partial Faults

Another benefit of the execution model is the ability to detect and recover from partial failures. We assume a simple fault model in which a processor either operates correctly or fails and stops. This will appear in our execution model either as an unsatisfied demand or as loss of a previously computed value. In the first case, the unsatisfied demand can be detected by a timeout mechanism and a simple protocol. Recovery would require the demand to be reissued and processed elsewhere (i.e., at another processor). In the second case, while the loss of a previously computed value cannot be detected, it does not matter since the execution model will automatically recompute the demanded value elsewhere.

Detection and recovery from failure is handled entirely by the execution model and does not require programmer participation. However, for correct processing, the execution model assumes that each function is programmed to be idempotent and atomic. Idempotence guarantees that repeated execution of the function with the same values has the same effect and produces the same result. Atomicity guarantees that either none of the

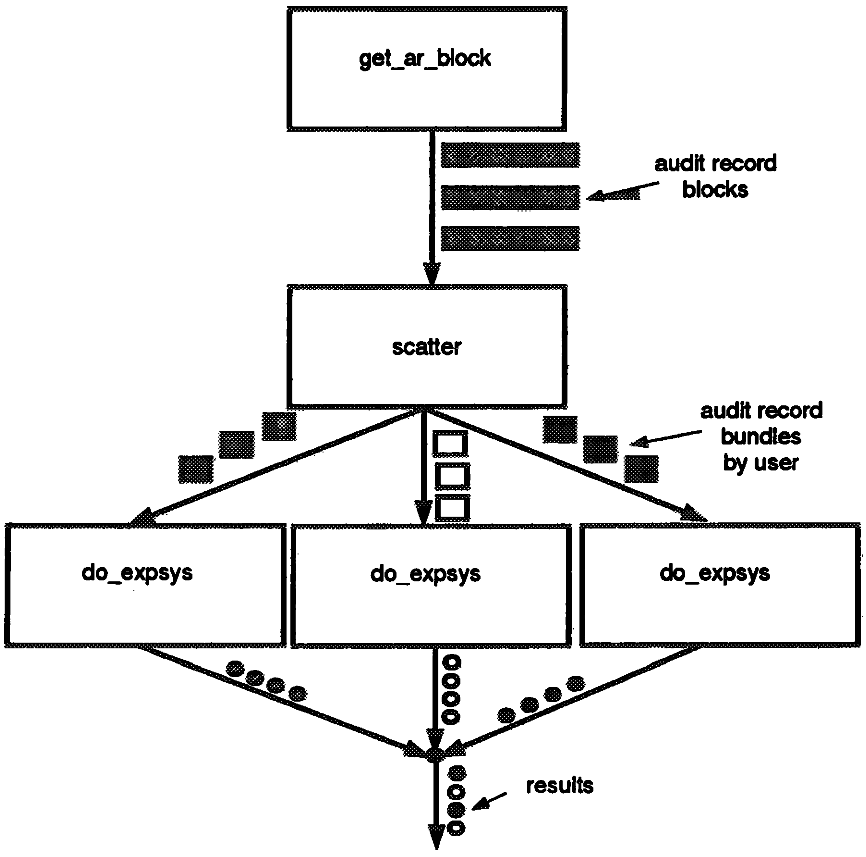


Figure 8.2: Data Parallelism in an Expert System

effects take place and no result is produced, or all of them take place and the result is produced. Functions of the IDES application (as described below) will be idempotent and preserve atomicity.

8.4 Implementation of IDES using GLU

IDES can be viewed as a program that consumes a sequence of audit records and produces a sequence of boolean values, each indicating whether any of the events associated with the corresponding audit record are anomalous. IDES processes each audit record using the rule-based expert subsystem and the historical profile-based statistical subsystem. The analysis of the two subsystems is resolved by the resolver subsystem, which reports whether an audit record signifies one or more anomalies. (Currently, the resolver function is simply a place-holder that returns an anomaly when either the expert system or the statistical component reports anomalies.) The rule-based expert system uses the audit record and the knowledge base associated with the subject specified by the audit record to perform anomaly analysis. For each subject, the statistical subsystem computes the active profile using the audit record and the previous active profile for the subject, and statistically analyzes the active profile with respect to the historical profile. To enhance efficiency of failure recovery, the active and historical profiles are periodically written to stable storage.

The GLU program that implements IDES is given below and an explanation of the program follows.

```
resolve( esp, stp )
where
  esp = do_expsys( bundle );
  stp = do_stats ( bundle );
  bundle = scatter( block );
  block = get_ar_block();
```

The program (shown in Figure 8.3) uses `get_ar_block` to generate audit record blocks that are scattered into bundles by the GLU function `scatter` and processed by both the expert system `(do_expsys)` and the statistical system `(do_stats).` The results from each are resolved using function `resolve.`

The execution of the above program results in two kinds of parallelism being exploited. One kind is in the simultaneous processing of each block by both the expert system `(do_expsys)` and the statistical system `(do_stats).` The other kind is the simultaneous processing of independent bundles by both the expert system and the statistical system.

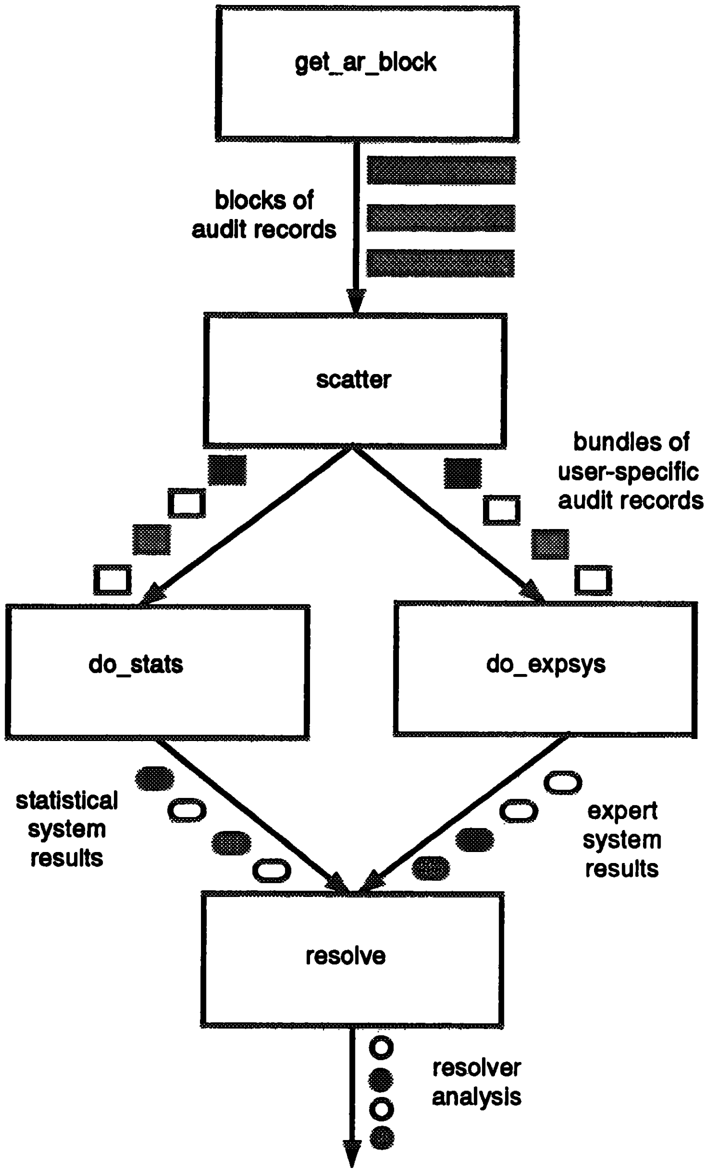


Figure 8.3: IDES in GLU

8.5 Evaluation

We have been successful in expressing both the statistical and expert-system components of IDES using the GLU platform. We qualitatively consider how well the GLU implementation of IDES meets the requirements of soft real-time processing, fault tolerance, and scalability. We also point to other benefits of using the GLU platform.

The soft real-time processing capability is achieved by the ability of GLU to automatically exploit and effectively harness all useful parallelism that is implicit in the IDES application. For example, the implementation would allow audit records from different subjects to be processed in parallel, and within that, the expert and statistical processing can proceed simultaneously. While the execution of a GLU program does entail overhead in the form of demand propagation, this will be easily offset by significant speedup obtained from useful parallelism.

The fault-tolerance requirement is achieved to a good measure because the GLU program for IDES is evaluated using demand-driven execution. Demand-driven execution is a natural and efficient mechanism for detecting failure in computing demanded values and recovery by redemanding and recomputing such values using the available resources.

The scalability requirement is achieved since the details of the architecture are hidden from the GLU program, and the execution model can easily accommodate increases (or decreases) in computing capacity. Essentially, changes in computing capacity need only be accounted for in the load distribution policy in the implementation of the execution model.

GLU has other pertinent benefits. This platform provides a sound basis for software design and development. A GLU program such as IDES is portable since it is independent of the underlying architecture and operating system. The GLU platform also provides efficient mechanisms for data transfer and synchronization, in contrast to the previous version of IDES, which made heavy use of a database system for these purposes.

Chapter 9

Remaining and Proposed Work

The completion of this three-year project is an important milestone in the maturation of the IDES intrusion-detection technology. We have demonstrated the system independence of IDES by its ability to monitor a database-system-based application on an IBM mainframe and a distributed network of Sun workstations running UNIX. A single IDES is in fact capable of monitoring both environments simultaneously, and IDES can be extended to monitor other environments. We are beginning to see the first commercial interest in IDES as a potential product, and we are seeking opportunities for developing customized versions of IDES for particular installations. Our first installation of IDES outside SRI is at FBI Headquarters in Washington D.C., where IDES will monitor the FBI's Field Office Information Management System (FOIMS). We have also given copies of the IDES software to several other U.S. Government agencies. We would like to see IDES and systems based on the IDES technology available for use in sensitive applications in the Federal Government and in private companies working in partnership with Federal agencies.

Such widespread use of IDES-like systems will require that these systems become standard, commercially available products. However, before IDES can easily find wide application or become anything close to a turnkey commercial product, several aspects must be explicitly addressed. These include improvements in the ease of use of IDES, reductions in its false-alarm rate, ensuring its resistance to attack, increasing its scalability, and extending it with respect to network vulnerabilities and attacks. Our intent is to produce a next-generation IDES that significantly extends IDES in each of these areas and to design an architecture for the insertion of the next-generation IDES into a large, complex network of computers. We briefly describe below our plans for each area.

Ease of Use and False Alarm Rate As the statistical component of IDES detects anomalies, IDES reports the anomalies to the IDES security administrator, along with a computation of how far from the expected norm the measured values are and an indicator of the leading five intrusion-detection measures contributing to the anomaly. As the IDES expert-system component detects events that trigger the firing of its rules, IDES reports the events to the IDES security administrator along with an explanation of what rules were fired and why. The number of anomalies reported can be tuned to some extent by raising and lowering IDES's statistical threshold. However, raising the threshold can be done only at the expense of losing some sensitivity to potential intrusions. It is difficult to tune the number of events reported by the expert-system component, because this number does not depend on any threshold or sensitivity factor. Thus, the number of suspicious events reported by IDES will depend primarily on the number of users and systems monitored and on how active those users are.

When IDES is used in a large user community with, for example, thousands of users, it is entirely reasonable for it to be reporting hundreds of anomalies a day. While this is a great improvement over the possibly millions of audit records per day that the auditing system may be generating, it is still a large number of events to be investigated by a security administrator, and may require a staff of several people on every shift. We may safely assume that most of the events reported will be false alarms, explainable after investigation using some information unknown to IDES. For example, the security administrator may rule out some reported suspicious events because he or she knows that the user has just begun a new job, was promoted, moved to a new office, assigned new computer equipment, and so on.

To make IDES more usable, a new intelligent component is necessary that can automatically apply such auxiliary knowledge to rule out "explainable" reported events *before* they are ever shown to the IDES security administrator. This presents the IDES security administrator with even fewer events to investigate, and avoids the waste of time entailed in investigating events that can be explained away. We call this new component of IDES the *resolver.* The resolver could also rank the remaining reported events as to their seriousness, so that the IDES security administrator can make the most effective use of his or her time in tracking down intruders.

Another approach to the problem of false positives is to use a detection component with finer discrimination capabilities. It is likely that a neural net component will be able to make the finer distinctions sought if it is sufficiently trained. While neural nets do not provide explanation facilities, the combination of a neural net, the resolver and the explanation facilities of the other two components could give IDES the best of both worlds: fine discrimination to avoid false positives and the ability to provide useful information about why a true positive is of concern.

The IDES platform supports the concept of interchangeable detection components by specifying a uniform format for audit and anomaly data, and by having a highly modular architecture. These characteristics of IDES allow us to concentrate on engineering the detection components themselves, without needing to modify the superstructure of IDES.

Resistance to Attack For a would-be penetrator attacking a system known to be monitored by IDES, the logical first system component to be disabled would be IDES itself. Thus, IDES itself must be able to resist attacks.

One approach to attack resistance is to implement IDES on a distributed system of autonomous, cooperating processors. To resist attack, the distributed IDES should transparently detect and react to failure or removal of one or more processors while continuing to provide complete functionality even if only one processor is operational.

This fault-tolerant capability can be provided by using the GLU (Granular Lucid) platform to implement IDES. The GLU platform consists of a very high-level programming language that supports coarse-grained parallelism on networked workstations. It assigns responsibility for each part of a computation to a specific processor, thereby making the computation fault tolerant. If part of a computation fails, the GLU system will notice and retry that part of the computation.

The current IDES incorporates GLU technology. However, our focus in this implementation has been on scalability and on gaining experience with the GLU platform itself. Thus, the current implementation is vulnerable if the processor running the interfaces to the GLU environment goes down. An enhanced implementation of IDES in GLU would remove the dependence on a particular processor.

Scalability The amount of processing IDES must do depends on the number and type of events audited, the number of users and systems monitored and their activity, and the type of system being monitored. Even if an IDES installation is initially adequate for a particular target system environment, if that environment grows in size, acquires many new users, or begins to audit many more events, the original processing power selected for the IDES installation may prove inadequate. What is needed is a way to scale IDES as the demands on its processing power increase, without having to scrap its hardware base and purchase an entire new system to support the increased workload.

The GLU platform naturally provides for scalable performance. When IDES is implemented on the GLU platform, increased processing power can be realized by simply adding additional processors to the hardware base. Importantly, this can be done without modifying the IDES application itself and even without stopping the IDES processing.

Detecting Network Intrusions Current computing environments are, more and more, massively networked sets of workstations, servers, mainframes, supercomputers, and special-purpose machines and devices. Focusing on the vulnerabilities of any single host or even any single homogeneous local network of machines may prove inadequate in such a distributed environment. Detecting intruders will require a comprehensive view of a network, possibly extending to other networks connected to the local network. Applying IDES technology to this (possibly heterogeneous) computing environment will require expanding the scope of IDES to include the ability to detect network intrusions as well as intrusions into the individual host machines. To achieve this, the following steps are necessary.

- Build a rule base into IDES that contains specialized knowledge about network vulnerabilities and intrusion scenarios.
- Enable IDES to work with partial information, since in a very large network it is unlikely that IDES will possess complete information about the whole network at all times. This capability will make IDES especially attractive for use in tactical communication networks, which must operate in hostile environments.
- Develop an overall architecture for inserting IDES or a distributed set of component IDES machines into a large, complex network.

Below, we summarize our plans for addressing these requirements. The new capabilities described below will be incorporated into a Next Generation IDES prototype.

- **GLU Platform** - We plan to develop and implement a GLU platform on a distributed system
  - To guarantee highly resistant IDES processing
  - To provide for scalable performance of IDES processing to meet changing needs.
- **Resolver** - We plan to build a resolver component for IDES. The resolver will accept anomaly records from the IDES detection components and perform additional analysis using these records and other sources of data as appropriate. It will then act on the results of these analyses. We will build on the results of a current SRI IR&D project that is investigating the use of model-based reasoning for intrusion detection. We plan to use SRI's Gister[1] evidential reasoning system for the fusion and interpretation of evidence produced by IDES's statistical and expert-system components.
- **Neural Nets** - We plan to build a neural network detection component for IDES.
- **Network Intrusion Detection** - We plan to develop and implement a capability for detecting network intrusions by combining a specialized rule base on network vulnerabilities with intrusion scenarios, develop a capability for the detection of intrusions with partial information, and develop an architecture for the integration of IDES into a large, complex network.

[1]Gister is a trademark of SRI International [28].

- **Large Network Architecture** - We plan to develop an architecture for inserting IDES into a large network, and will we work with a U.S. Government installation (e.g., NOSC) to install IDES in such an environment.
- **Improved Detection Capability** - We will improve the detection capability of the current IDES intrusion-detection components by continued statistical analysis to reduce the false alarm rate, inclusion of trend tests into the statistical component, and an improved rule base for the expert-system component.
- **IDES Experimentation** - We plan to perform experiments using the installed IDES at FBI Headquarters in Washington D.C. to determine the effectiveness of the intrusion-detection capability.

We describe these plans in more detail in the following sections.

9.1 Scalability and Resistance to Attack - The GLU Platform

Our new core technology, GLU, can be used to allow IDES to be easily scaled to many processors and, more important, to allow IDES to resist attack from an intruder. This is extremely important for any intrusion-detection system, because we envision that an intelligent intruder is likely to use an attack strategy in which the intruder attempts to disable IDES before mounting an attack on the system IDES was intended to protect.

GLU will allow IDES to be resistant to attack by distributing processing over a distributed set of workstations. GLU does this in such a way that if one of the machines it is using fails for some reason, it will automatically move processing to another machine without losing any information. If the intruder is informed enough to know that IDES is distributed, he or she may continue to attack the other IDES component machines. As this occurs, GLU will continue to move the IDES processing from machine to machine, without losing any information or any audit records. As the disabled machines reboot and become available again, GLU will automatically distribute IDES processing to these machines. In this way, disabling IDES becomes more than just a matter of disabling one machine. IDES will continue to run as long as any of its machines are available.

In addition, from a design standpoint, GLU allows us to exploit the natural parallelism present in the IDES processing of audit data. By requiring that a program be expressed as a graph with functional, stateless nodes, GLU encourages a highly modular design with very loosely coupled modules.

We have tested GLU by implementing a portion of the IDES software on it. We have demonstrated its ability to transparently distribute IDES processing among several Sun workstations and to redistribute IDES processing when one of the workstations goes down. GLU is the foundation for the Next Generation IDES.

The current prototype embodies our first attempt at incorporating GLU into IDES. The result is something of a compromise in terms of both performance and robustness. For example, the current design still has a weak link - the processor that the target systems send the audit records to. If this processor fails, the whole of IDES will go down. We would like to eliminate this weak link. We would also like to address the performance concerns in the interprocess communication mechanism.

The current platform, which needs to be enhanced to meet the requirements of high resistance to attack and scalable performance, offers a limited degree of resistance by detecting and recovering from the failure of processors other than the main (weak link) processor. However, IDES can be rendered ineffective in two ways:

- By causing the weak-link processor to fail
- By arbitrarily slowing down any processor.

Failure of the weak-link processor will cause audit records from the target systems to be lost and no IDES processing to be performed. This can be compensated for by limited replication, whereby each audit record is redundantly received and processed. Thus, failure of any one processor will not cause IDES to cease operation.

Arbitrarily slowing down any processor can slow the IDES processing at that processor enough to make it ineffective. However, the current GLU platform will not react to the slowdown, because the processor is still operational. The platform needs to be enhanced so that processing of each audit record is conducted redundantly and simultaneously in several other processors. The result of the processing can then be determined as soon as the fastest processor has completed processing, so that arbitrary slowdown of a processor will not cause IDES as a whole to be ineffective.

With the current GLU platform, scalable processing is possible since it automatically exploits and effectively harnesses all useful parallelism that is implicit in the IDES application. However, to be fully scalable, the GLU platform needs to transparently accommodate increases (or decreases) in processing power without user intervention or temporary stoppage of IDES processing.

9.2 The Resolver

Neither the statistical nor the expert-system components of IDES are capable of understanding the causes of anomalies they may detect. This is left to the system security administrator. Because of this, in order to guard against excessive false alarms, detection thresholds must be set rather conservatively.

To improve the performance of IDES in this regard, we plan to develop a model-based *resolver,* which would review the conclusions of the two independent IDES components (the statistical and expert-system components), in the context of more global information in order to determine whether to raise an alarm. The resolver would consider more complex explanations for apparent anomalies, thereby reducing the false-positive rate of anomaly reports as well as eliminating the possibility of multiple alarms due to the same suspicious behavior's being reported by both the rule-based system and the statistical detector.

Furthermore, the resolver would be able to reason about the potential severity of the anomaly based on an understanding of the alarm in the context of the overall session. This could often lead to the conclusion that unusual behavior is not a concern because it does not have any security ramifications.

In ambiguous situations, the resolver could request that additional information be collected or accessed in order to determine the likelihood of an intrusion. This would allow the amount of audit data collected to be determined by the particular situation.

The resolver will be an intelligent, model-based reasoning component that will make its decisions based on information from the other detection components in the context of a larger, unified knowledge base. We plan to implement the resolver using the evidential reasoning technology pioneered by SRI's Artificial Intelligence Center.

The resolver would consist of the following components.

- A deduction engine that analyzes the inputs to the resolver
- A database component, giving the resolver access to volatile data
- A user-interface component, allowing users to interact with it on different levels (a security watch officer could input calendar-type information while a security administrator could modify its inference rules)
- A communication component, allowing the resolver to act on the conclusions it reaches.

9.2.1 Resolver Models and Other Data Sources

The concept of the resolver extends the IDES paradigm to include specific models of proscribed activities along with their associated observables. The role of the resolver will be to determine the likelihood of an intrusion based on the combination of evidence for and against it as determined by the available intrusion scenario models. The intrusion scenarios are expected to vary for different types of intruders and for different systems.

The resolver would be activated in response to an anomaly signaled by either the statistical detector or the expert-system component. The information leading to the anomaly would be combined with other available information (such as the current schedules of valid users) and other indications of anomalies in order to determine the likelihood that the anomalous behavior is truly due to an intruder. At this point, the resolver may request additional information in order to reach a conclusion.

The resolver will require a knowledge base consisting of a set of "scenario models" of typical intrusion activities. These models will be specified in terms of the sequences of user behavior that constitute the scenario. For example, one scenario might represent a programmed password attack. This scenario would describe the typical steps necessary to carry out the attack, expressed in terms of the specific user behavior involved.

The typical "observables" associated with specific behaviors (for example, "accessed the password file") would be linked to the behaviors in such a way that evidence about the observables can be related to particular activities. Finally, the available audit data would be related to the observables (e.g., "file accesses to etc/passwd" is linked to the observable "accessed the password file"). In this way, audit data may be linked to the particular patterns of behavior represented by the models.

Information concerning users, such as changes in user status, new users, terminated users, users on vacation, changed job assignments, user locations, and so forth will be useful for the resolver. Additional information about files, directories, devices, and authorizations will also be of value.

9.22 Resolver Operation

In effect, new anomaly data will cause the resolver to create instances of its models that are implied by those data. In some cases, the degree to which the data (possibly in conjunction with earlier data) imply certain intrusion scenarios may be sufficient to raise an immediate alarm; in other cases this evidence may be inconclusive. If the evidence is inconclusive, the resolver may choose to suspend the decision until further evidence is available, or it may request additional information. The resolver will have the ability to carry out sensitivity analyses over its potential sources of information in order to determine which sources might have the greatest impact on its conclusions. These sources may represent audit data that are not collected normally or that are collected but not evaluated in the normal operation of the system.

This mapping of aspects of user behavior to how the behavior will show up in the audit data must exhibit properties that differentiate the particular behavior of concern from other innocuous activities. These distinguishing properties should have the following characteristics.

- They should be easily recognized, so that they can be readily detected.
- They should be clearly associated with the behavior in question.
- They should allow intrusive behavior to be distinguished from normal behavior.

9.2.3 Evidential Reasoning

We have examined the feasibility of using SRI's Gister evidential reasoning system for the fusion and interpretation of evidence for hypothesized intrusions. Gister grew out of earlier work, sponsored jointly by SPAWAR and DARPA, aimed at developing new techniques for Naval intelligence analysis. We will implement a model-based intrusion-detection capability, based on Gister, for inclusion within IDES.

The goal of evidential reasoning is to assess the effect of all available pieces of evidence upon a hypothesis, by making use of domain-specific knowledge. The first step in applying evidential reasoning to a given problem is to delimit a propositional space of possible situations. Within the theory of belief functions, this propositional space is called the *frame of discernment.* A frame of discernment delimits a set of possible situations, exactly one of which is true at any one time. Once a frame of discernment has been established, propositional statements can be represented by subsets of elements from the frame corresponding to those situations for which the statements are true. Bodies of evidence are expressed as probabilistic opinions about the partial truth or falsity of propositional statements relative to a frame. Belief assigned to a nonatomic subset explicitly represents a lack of information sufficient to enable more precise distribution. This allows belief to be attributed to statements whose granularity is appropriate to the available evidence.

In evidential reasoning, domain-specific knowledge is defined in terms of *compatibility* relations that relate one frame of discernment to another. A compatibility relation simply describes which elements from the two frames can simultaneously be true.

Evidential reasoning provides a number of formal operations for assessing evidence, including:

1. **Fusion** - to determine a consensus from several bodies of evidence obtained from independent sources. Fusion is accomplished through Dempster's rule of combination.
2. **Translation** - to determine the impact of a body of evidence upon elements of a related frame of discernment.

3. **Projection** - to determine the impact of a body of evidence at some future (or past) point in time.
4. **Discounting** - to adjust a body of evidence to account for the credibility of its source.

Several other evidential operations have been defined and are described elsewhere [28].

Independent opinions are expressed by multiple bodies of evidence. The evidential reasoning approach focuses on a body of evidence, which describes a meaningful collection of interrelated beliefs, as the primitive representation. This allows a system to reason about evidence pertaining to a model consisting of multiple, interrelated activities. In contrast, all other such technologies focus on individual propositions. Previous applications of Gister include intelligence processing, military situation assessment, medical diagnosis, and acoustic and electronic signal processing.

9.2.4 Benefits of Model-Based Reasoning

The potential benefits of using a model-based resolver within IDES are manyfold, including the following:

- More data can be processed, because the technology will allow the selective narrowing of the focus of the relevant data. Thus, at any given time, only a small part of the data collected may need to be examined in detail.
- More intuitive explanations of what is being detected can be generated, because the events flagged can be related to the defined intrusion scenarios.
- The system can predict what the intruder's next action will be, based on the defined intrusion models. Such predictions can be used to verify an intrusion hypothesis, to take preventive action, or to determine which data to look for next.

With the model-based reasoning approach, the descriptions of intrusion scenarios allow the intrusion-detection system to focus its attention on the data likely to be of most' utility at the moment. The models can be used to examine only the data most relevant to detecting intrusions. In effect, we can narrow the field of view to optimize the data that have to be analyzed. This is analogous to pointing and tuning a sensor to optimize performance.

If the stream of audit data contains a significant number of intrusions in comparison with the total volume of audit data (i.e., there is a large signal-to-noise ratio), then an approach in which all the incoming data are examined and analyzed can be successful. However, if the number of intrusions is very small in comparison with the total volume of audit data (a small signal-to-noise ratio), then the amount of data to be examined can quickly overwhelm the intrusion-detection system. The system will be drawing very many conclusions, most of which will be dead ends. In this case, a more efficient approach would be to examine only the specific data in the audit data stream that are relevant at the moment. Thus, we can, in effect, increase the signal-to-noise ratio in particular areas by looking only in those areas. This top-down approach to data analysis will be more efficient in the intrusion-detection domain, where the signal-to-noise ratio is extremely small.

With the model-based reasoning approach, the models of intrusion can be used to decide what specific data should be examined next. These models allow the system to predict the action of an intruder who is following a particular scenario. This in turn allows the system to determine specifically which audit data to be concerned with. If the relevant data do not occur in the audit trail, then the scenario under consideration is probably not occurring. If the system does detect what it was looking for, then it predicts the next step and will then examine only data specifically relevant to confirming the hypothesis of the posited intrusion, and so on until a conclusion is reached. Thus, a model-based system reacts to the situation, using only those data most appropriate to the given situation and context.

By providing the means to consider more complex, global explanations of anomalies, the resolver should increase the likelihood of catching actual intrusions, reduce the number of false alarms, and improve the overall effectiveness of both the statistical component and the expert-system component of IDES. Furthermore, by reducing the amount of data to be considered routinely, the overall performance of IDES should be improved.

9.3 Monitoring Network Traffic

We also plan to extend the theoretical basis for IDES to enable the development of an IDES that could monitor traffic in tactical communication networks to detect suspicious activity. The required extensions are twofold:

- IDES's statistical algorithms are based on the probabilities of occurrence of the events it observes. In a network, however, IDES would not be able to operate with global knowledge. Thus, IDES's algorithms must be extended to give meaningful results when some information is missing and probabilities can only be estimated.
- IDES's rule base has been designed for detecting suspicious user activity. To use IDES to monitor network traffic, where user data are not available, we must create a rule base with a set of rules specific to the domain of detecting suspicious network traffic patterns.

In addition to establishing the theoretical foundations for these extensions to IDES, we plan to develop candidate architectures for incorporating one or more IDES into the network topology. So far, IDES has assumed centralized control and is thus appropriate for systems with high channel capacity and low transfer delay. Tactical networks operating under stress are expected to have neither. Moreover, tactical networks can become partitioned, so that an IDES with centralized control would not have global knowledge. Thus, a distributed approach to detecting anomalous behavior is more appropriate for tactical communication networks. We envision that IDES in this context would consist of a set of loosely coupled IDES machines (see Section 9.5).

9.4 A Neural Net Component for IDES

Matching a subject's observed behavior to a model of the subject's past behavior is difficult, since subject behavior can be very complex. False alarms can result from invalid assumptions about the distribution of the audit data made by the statistical algorithms. Missed detections can result from the inability to discriminate intrusive behavior from normal behavior on a purely statistical basis. Our use of a second, rule-based approach is an attempt to help fill some of the gaps in the statistical approach. In this approach, an event can trigger a rule apart from any consideration of whether the event is normal for the subject. Thus, intrusion scenarios that may not be anomalous for the subject (because the intruder has trained the system to see the behavior as normal) can be detected by appropriate rules.

However, the rule-based approach also has limitations. Writing such a rule-based system is a knowledge-engineering problem, and the resulting expert system will be no better than the knowledge and the reasoning principles it incorporates. An intrusion scenario that does not trigger a rule will not be detected by the rule-based approach. Besides, maintaining a complex rule-based system can be as difficult as maintaining any other piece of software of comparable magnitude, especially if the system depends heavily on procedural extensions such as rule ranking and deleting facts.

To address these concerns, we have considered the use of neural nets as another detection component in IDES.

9.4.1 Neural Networks

A neural network, considered abstractly, is a machine for transforming inputs to outputs by the action of a large set of simple, highly connected elements. The particular function computed in the transformation is determined by the characteristics of the elements and the connectivity and strengths of the interconnections. Several canonical structures have been studied and used for practical applications. Neural networks perform either *classi-*

fication functions (e.g., pattern recognition), or *optimization functions* (e.g., minimizing some function over a set of state variables). In the following discussion, only classification is considered.

The particular transformation function computed by the network may be determined by adjusting the weights of the connections between the elements. Adjustment of weights is analogous to the writing of a program for general-purpose computation. It may be accomplished either under supervision, that is, the correctness of a network response to data is decided externally, or without supervision, that is, functions evolve according to a built-in decision rule.

For the classification application, the network output should correctly identify the class to which the input data belong. For this goal, the network would be trained using sets of sample data that exemplify the various classes. In general, the effect of new sample values may be to modify or amalgamate existing classes or to create a new class. In the intrusion-detection application, a network would be trained to recognize the particular pattern of behavior for individual users, so that when a user claims to be a particular person, the neural network for that user would determine if the behavior conforms to the named user's pattern of usage.

9.4.2 Uses of Neural Networks in IDES

We envision that neural networks could address the following problems in IDES.

- *The need for accurate statistical distributions.* Statistical methods sometimes depend on some assumptions about the underlying distributions of subject behavior, such as Gaussian distribution of deviations from a norm. These assumptions may not be valid and can lead to a high false-alarm rate. Neural networks do not require such assumptions; a neural network approach will have the effect of relaxing these assumptions on the data distribution.

- *Difficulty in evaluating detection measures.* We selected the current set of intrusion-detection measures for IDES on the basis of our intuition and experience. We do not know how effective these measures are for characterizing subject behavior, whether for subjects in general, or for any particular subject. A measure may seem to be ineffective when considered for all subjects, but may be useful for some particular subject. A neural network can serve as a tool to help us evaluate the effectiveness of various sets of measures.

- *High cost of algorithm development.* The development time for devising new statistical algorithms and building new software is significant. It is costly to reconstruct the statistical algorithms and to rebuild the software implementing them. We may remove assumptions that are invalid for the audit data we are using, but we may find that we have to modify the algorithms yet again when we apply them to a new user community with different behavior characteristics. Neural network simulators are easier to modify; we envision using a "plug-compatible" approach where one simulator can be replaced by another using a different methodology with minimal effort.

- *Difficulty in scaling.* New problems are anticipated in applying IDES to very large communities, for example, thousands of users. Methods are needed for assigning individuals to groups on the basis of similarity of behavior, so that group profiles may be maintained instead of a profile for each user. Our current thinking is that users will be grouped manually (by a security administrator) according to job title, shift, responsibilities, and so forth. These methods may be inadequate. We envision using a neural network to classify users according to their actual observed behavior, thus making group monitoring more effective.

We do not feel that a neural network can simply replace the IDES statistical component. The most important reason for this is that IDES's statistical component provides information as to which measures contributed to considering an event to be anomalous. Finding ways to get explanatory information out of neural networks is currently a research issue.

Our approach would be to build a prototype intrusion-detection component using a neural network simulator. This component would run in parallel with the statistical and expert-system components of IDES. Its output would also be reported to the resolver.

9.5 Large Network Architecture

There are many issues to be addressed when considering the use of IDES to monitor a large complex network. Such networks many be composed of hundreds or even thousands of computers, from personal computers to workstations to large timesharing systems, and may have gateways to several larger external networks. The monitored network may itself be subdivided into smaller internal networks interconnected by internal gateways or bridges. The issues to be considered include the following.

- How best to integrate IDES into the desired environment
- How to distribute IDES processing over the network
- Which audit data to select for forwarding to IDES from the target environment
- The effect of forwarding audit data to IDES on network performance and capacity

- Which are the most appropriate audit data collection and storage points in the target environment
- The most effective manner for getting the selected audit data to IDES.

We will examine these issues in the context of a particular U.S. Government installation to be selected jointly by SPAWAR and SRI. NOSC is a likely candidate for this.

9.6 Improved Detection Capability

In addition to adding new detection components to IDES, as described above, we will also continue to improve the detection capability of the current IDES detection components - the statistical component and the expert-system component. Our approach is as follows.

- Perform statistical analysis to reduce the false alarm rate while remaining sensitive to suspicious behavior. We will revise the statistical algorithms so as to increase detection strength.
- Perform statistical analysis to find which measures (or combinations of measures) are the best discriminators of intrusive behavior. This analysis will be performed through experimentation, by activating or deactivating certain profiles, monitoring different mixes of users, and adjusting system parameters.
- Stage intrusions in an effort to determine which measures are the most effective in detecting intrusions - that is, which have the highest true-positive rate while maintaining an acceptable false-positive rate. Based on this analysis, we may also implement additional intrusion-detection measures.
- Make IDES statistical tests parameterizable, so that
    1. Some measures can be deactivated for some subjects.

       Different intrusion-detection measures may be appropriate to different classes of subject. For example, for users whose computer usage is almost always during normal business hours, an appropriate measure might simply track whether activity is during normal hours or off hours. However, other users might frequently log in during the evenings as well, yet still have a distinctive pattern of use (e.g., logging in between 7:00 and 9:00 p.m. but rarely after 9:00 p.m. or between 5:00 p.m. and 9:00 p.m.); for such users, an intrusion-detection measure that tracks for each hour whether the user is likely to be logged in during that hour would be more appropriate. For still others for whom "normal" could be any time of day, a time-of-use intrusion-detection measure may not be meaningful at all. Our vision is for IDES to allow the security officer to activate or deactivate specific measures for particular subjects.

2. Different subjects can have different thresholds.

   Our vision is to allow the IDES security officer to adjust the false-positive rates individually for each subject. For example, the security officer might raise a user's false-positive rate (which simultaneously raises the true-positive rate) if he has reason to doubt a user's integrity or if the user's current assignment or security level requires closer scrutiny. The security administrator might lower a user's false-positive rate if he knows that a user's current assignment is changing or that the user has other legitimate reasons for changing his behavior.

3. Different subjects can have different profile update periods.

4. Different subjects can have different initial default profiles.

- Perform trend tests. We will develop and implement second-order intrusion-detection measures; that is, methods for conducting trend tests (to detect how fast a user's statistical profile changes). We will evaluate the effectiveness of using these trend tests to detect planned attacks in which a user gradually moves his or her behavior to a new profile, from which to safely mount an attack, or gradually increases the spread of the behavior considered normal.

- Enhance intrusion-detection rules (measures) to enable IDES to handle a wider variety of intrusions. To obtain the information necessary for this task, we will meet regularly with law enforcement agencies, UNIX experts, intrusion experts, and others to develop a realistic and useful rule base.

- Allow the capability of modifying intrusion-detection rules during execution time to enable IDES to adapt dynamically to a frequently changing target environment.

- Enhance IDES to be tolerant of missing or incomplete audit data or other failures.

- Upgrade existing Sun 3 equipment to Sun 4 equipment. This will allow us to upgrade obsolete equipment and to maintain at SRI the same version of IDES that is running at the FBI and elsewhere.

In addition, we plan to enhance the security officer interface to provide additional capabilities to the security officer.

The current user interface takes advantage of the windowing facility available on the Sun 3 workstation. This interface enables the security officer to receive reports, graphically view the amount of abnormal activity, and perform some predefined queries. A more powerful user interface would enable a security officer to tune the thresholds that control the alarm rate, activate and deactivate profiles, perform more complex queries, and view system-wide or user-specific information graphically, assisted by pop-up menus. Anomalies and summary reports will be presented graphically in ways that would enable the security officer to view a large amount of information easily.

9.7 IDES Experimentation

So as to gain an assessment of IDES's effectiveness at detecting intrusions in an operational setting, we will perform experiments using the installed IDES facility at FBI Headquarters in Washington D.C. Thus, we will continue to provide support to the FBI in the use of IDES and will continue to provide its personnel with regular updated versions of IDES. We will work closely with the FBI to design and enact a wide variety of experiments, such as staging intrusions, investigating the anomalies reported by IDES, and observing the effect of tuning various parameters in IDES.

9.8 Summary

SRI will develop the necessary technology to overcome several existing obstacles to widespread use of IDES. This includes improvements in the ease of use of IDES, reductions in its false-alarm rate, ensuring its resistance to attack, increasing its scalability, and extending it with respect to network vulnerabilities and attacks. We will design an architecture for the insertion of the Next Generation IDES into a large, complex network of computers.

Appendix A

PBEST - A Production-Based Expert System Tool

In this appendix[1], we present information on how to use PBEST, an expert-system tool created by Alan Whitehurst, formerly of SRI. We describe how to produce expert systems from rule specifications using the *pbcc* translator and how to create interfaces for the expert-system engines produced by PBEST. Besides this, we show how to produce and use interactive, window-based versions of expert-system engines. We describe the syntax of the PBEST expert-system specification language. Finally, we cover some efficiency considerations. We discuss implementation details primarily insofar as they bear on the above issues.

A.1 Introduction

The PBEST system consists of a rule translator, *pbcc,* a library of run-time routines, and a set of garbage-collection routines (these routines are copyright 1988 Hans-J. Boehm, Alan J. Demers; see the GC_README file included with the garbage-collector source). The rule translator accepts a set of rule and fact definitions and produces routines in the C language to assert facts and process rules. The run-time library contains the code that is constant for all expert systems, and includes routines to support interactive versions of expert-system engines. By setting the appropriate flags, the user can produce expert systems that run under Sunview or the IDES user interface library (the latter based on X-windows). (The Sunview version is not described in this document.) These environments allow the user to view the execution of the system, to single-step the execution, to set

[1]This appendix is a draft of a user's manual for the PBEST system. We would appreciate getting comments on both the manual and PBEST itself regarding bugs in PBEST, usefulness of PBEST and the manual, and improvements that would enhance either.

and remove breakpoints, to delete and assert facts, and to watch a trace of the effects of the rules as they fire.

To produce an expert system, the user creates a file or files containing rule and fact definitions, using any desired text editor. The user then runs this file through the *pbcc* translator, producing a file of C code. This code can be compiled normally, and linked with 1ibpb.a (the run-time routines), gc.o (the memory-allocating and garbage-collection routines), and the necessary window libraries, if the user wants a window version. The resulting program can be run as a stand-alone executable program. This process will usually be done under the control of *make* or a similar program.

A.2 Getting Started

In this section, we will give a working example of a trivial expert system and show how it is compiled and executed. Start by entering the following text into a text editor (donʽt worry about meaning right now; just concentrate on getting the text right):

```
‘
‘ Everything on the line after a backquote is a comment. You need
‘  not type comments in for this exercise if you don’t want
‘ to! You also do not need to worry about how things line
‘  up*  Just get the brackets and punctuation right.
‘

‘ Declare count ptypes with value field
ptype[count value:int]
‘ Declare signal ptypes with flag field
ptype[sig flag:int]

‘
‘ This rule fires if there are no count or signal facts.
‘  It asserts a count fact and signal fact with their fields
‘  set to 0. This example is set up to produce only one
‘  count fact and only one signal fact.
‘

rule[rO (#-10):
    [-count]
    [-sig]
  ==>
```

```
    [!|printf("Asserting count and sig facts.\n")]
    [+count|value=0]
    [+sig|flag=0]]


' This rule fires whenever the flag field of the signal
' fact is set to 1 and the count fact's value field is less
' than 100.
'

rule[rl:
    [+c:count|value<l00]
    [+s:sig|flag==1]
  ==>
    [!|printf ("Incrementing count, resetting sig flag.\n")]
    [/c|value+=l]                     ' Increment count value
    [/s|flag=0]]                      ' Reset signal flag

'
' This rule fires whenever the flag field of the signal
' fact is set to 0 and the count fact's value field is less
' than 100.


rule[r2:
    [+c:count|value<l00]
    [+s:sig|flag==0]
  ==>
    [!|printf("Removing old count fact, asserting new count fact,\
 setting sig flag.\n")]
    [+count|value=c.value+l]          ' Make a new count fact and
                                      ' set its value field to the old
                                      ' fact's value plus 1
    [-|c]                             ' Delete count fact
    [/s|flag=l]]                      ' Set signal flag

'
' This rule fires when the count fact's value gets to 100.
' It removes the count fact and prints a message.
' After this, no rules will be able to fire, so the engine
'  stops in the window version and exits the program in the
'  non-window version.
```

```
rule [r3 (#10):
    [+c:count|value>=l00]
  ==>
    [-|c]
    [!|printf ( Count >= 100.\n )]]
```

Save the file as 'tst.pbest'.

At this point, the user must translate, compile and link this file to produce an executable expert system. The user can use the *make* command to do this. We provide a sample *Makefile* at the end of this document; it allows one to automatically produce and maintain a particular expert system. Assuming things are set up as described in that section, the user can issue the command

```
make tst
```

at the shell prompt. The system should execute the commands necessary to produce the *tst* program. The user can then type

```
tst
```

at the shell prompt, and the expert system should execute.

To make the X-windows version, the user can type

```
make tstx
```

To run this program, the workstation must be running X-windows. Type

```
tstx
```

at the shell prompt.

Note that one could type

```
make
```

to produce the *tst* and *tstx* versions with one command.

A.3 Basic Syntax

To create an expert system, the user must define rules and ptypes. Ptypes are also known as pattern types. A complete specification of the syntax of PBEST can be found at the end of this document. For now, we will discuss example rule and ptype declarations.

The following is an example of a ptype declaration:

```
ptype [count value : int]
```

This declaration consists of the keyword *ptype,* an opening bracket, a name *count,* and a typed field, *value,* which is declared to be an integer. The declaration is terminated with a closing bracket.

The purpose of this declaration is to establish a pattern or template for facts. Each fact that exists in the expert system's knowledge base will be an instance of some ptype. Facts of ptype *count* will have one field, an integer called *value.* This declaration allows rules to refer to count facts, and to inspect and modify the value field of these facts. There will usually be many facts of a given ptype.

A ptype may have more than one field:

```
ptype [session userid : string,
               terminal : string,
               timeoutflag:int]
```

This ptype declaration establishes a pattern for facts with three fields: *userid* and *terminal,* both strings, and *timeoutflag,* an integer.

Rules can refer to facts that have fields matching particular things. For example, a rule could check for a session fact whose timeoutflag is 1 by including the following clause in its antecedent (we will look at the complete structure of rules a little later):

```
[+session | timeoutflag == 1]
```

The + sign after the opening bracket is used as a sort of "existential quantifier." That is, it allows a rule to check to see if any fact having certain characteristics exists. This clause, then, will match any session fact having a timeoutflag field with the value 1.

A rule may also check to see that there is no fact of a given type using the - syntax. The following example checks to see that there is no session fact with the user id of THISUSER:

```
[-session | userid == "THISUSER"]
```

```
  ==>
    [/c|value+=l]         ' Increment count fact's value field
    etc.
```

Note that modifications are implemented as if the original fact had been negated and a new fact with the modified fields had been added. Anyone using the interactive windowing system will notice that when a fact gets modified, it gets a new number. This allows the system to give priority in binding to modified facts over facts that were created more recently but are "inactive."

Here is an example of a complete rule declaration that demonstrates some of these techniques:

```
    rule[SimuLogon(#l;*):
         [+tr:transaction]
         [+se:session|userid == tr.userid]
         [?|se.terminal != tr.terminal]
       ==>
         [!|printf( SimuLogon:  user %s at terminals %s, %s\n ,
                  tr.userid, tr.terminal, se.terminal)]
         [-|tr]
         [-|se]]
```

This rule detects a user logging in on a terminal while already logged in somewhere else. It works by checking for some transaction fact (these facts must be instances of a *transaction* ptype that the user declared), and checking to see if there is any session fact with the same *userid* field. If such a session fact exists, it compares the *terminal* fields of the transaction and session facts to see if they are different. If they are different, the rule fires. This rule assumes that some other rule will look for login actions and create session facts for each login.

The syntax of a rule is completely determined by the delimiters (brackets and ==>). The above rule is formatted in a readable fashion, but such formatting is optional.

A rule declaration, then, begins with the keyword rule. It then has a name section followed by a colon (:). The name section of the above rule consists of the name *(SimuLogon)* followed by a set of options in parentheses. We will give a description of all possible options later. The #l option gives the rule a ranking of 1. This means that if several rules can fire, this rule will fire before rules with a lower ranking. Thus, if this rule and a rule with a ranking of 0 can both fire, this one will be selected to fire. The asterisk (*) option means the rule is repeatable. This means that the rule can be fired repeatedly without some other rules firing in the meantime. By default, rules are not repeatable, since once a rule's antecedents are satisfied, they will continue to be satisfied forever, and

```
  ==>
    [/c|value+=1]          ' Increment count fact’s value field
    etc.
```

Note that modifications are implemented as if the original fact had been negated and a new fact with the modified fields had been added. Anyone using the interactive windowing system will notice that when a fact gets modified, it gets a new number. This allows the system to give priority in binding to modified facts over facts that were created more recently but are “inactive.”

Here is an example of a complete rule declaration that demonstrates some of these techniques:

```
    rule[SimuLogon(#l;*):
         [+tr : transaction]
         [+se:session|userid == tr.userid]
         [?|se.terminal != tr.terminal]
       ==>
         [!|printf ( SimuLogon : user %s at terminals %s, %s\n",
                   tr.userid, tr.terminal, se.terminal)]
         [-|tr]
         C-|se]]
```

This rule detects a user logging in on a terminal while already logged in somewhere else. It works by checking for some *transaction* fact (these facts must be instances of a *transaction* ptype that the user declared), and checking to see if there is any *session* fact with the same *userid* field. If such a session fact exists, it compares the *terminal* fields of the transaction and session facts to see if they are different. If they are different, the rule fires. This rule assumes that some other rule will look for login actions and create session facts for each login.

The syntax of a rule is completely determined by the delimiters (brackets and ==>). The above rule is formatted in a readable fashion, but such formatting is optional.

A rule declaration, then, begins with the keyword *rule.* It then has a name section followed by a colon (:). The name section of the above rule consists of the name *(SimuLogon)* followed by a set of options in parentheses. We will give a description of all possible options later. The #l option gives the rule a ranking of 1. This means that if several rules can fire, this rule will fire before rules with a lower ranking. Thus, if this rule and a rule with a ranking of 0 can both fire, this one will be selected to fire. The asterisk (*) option means the rule is repeatable. This means that the rule can be fired repeatedly without some other rules firing in the meantime. By default, rules are not repeatable, since once a rule's antecedents are satisfied, they will continue to be satisfied forever, and the rule would fire again and again. As we will see below, this rule deletes the facts that satisfy it and so prevents such a loop.

The body of a rule consists of a sequence of antecedent clauses, the delimiter ==>, and a sequence of consequent clauses. Each antecedent clause consists of some test. The first antecedent clause

```
[+tr : transaction]
```

checks for a transaction fact. It gives it the alias of *tr*. The clause

```
[+se : session|userid == tr.userid]
```

checks for a session fact with the same value in the userid field as the tr fact already found. If it finds such a fact, it gives it the alias of se. The third clause

```
[?|se.terminal != tr.terminal]
```

checks to see if the terminal field in the session fact is different from the one in the transaction fact.

If all the antecedent clauses of a rule are satisfied, the rule “fires”; that is, its consequent clauses get executed. This rule has three consequent clauses. The first clause

```
[! |printf ("SimuLogon: user %s at terminals %s, %s\n",
  tr.userid, tr.terminal, se.terminal)]
```

is a call to a C function. The ! | syntax indicates such a call. This call can be an arbitrary C function. *pbcc* recognizes most of the built-in C functions; the user must declare user-written ones (we will describe how to do this later). Such calls can reference the fields in the facts as C structures. This is done using the fact alias followed by a clot (.) and the field name.

The clauses

```
[-|tr]
```

and

```
[-|se]
```

remove the *tr* and *se* facts from the knowledge base. This is indicated by the - | syntax followed by the fact alias. There are three reasons to remove facts. First, doing so prevents the rule from firing over and over with the same facts satisfying it. Second, removing facts from the knowledge base when they are no longer needed makes the system run more quickly. This is because when a new fact is asserted into the knowledge base, any rule with an antecedent clause that the new fact matches must compare the new fact to all other facts matching the other antecedent clauses of the rule to see if some set of facts matches the whole antecedent. In the above example, a new transaction fact will cause the rule to check all session facts to see if any of them have the same userid field. Thus, removing unneeded session facts can shorten this check.

Note that if a fact can match an antecedent clause in more than one rule, no rule should remove it unless all the rules that may need it have used it. Checking this can be done with marks, which we will describe later.

A third reason to remove facts once they are not needed is to conserve memory. Obviously, if the system is to run for long periods of time, things must be removed from the knowledge base at the same rate as they are added to it, or else the knowledge base will ultimately overflow the memory of the system.

A.4 More Syntax

This section describes the syntax of PBEST in more detail. It includes descriptions of the mechanism for declaring external functions and for marking facts.

The PBEST system allows the user to call arbitrary C language functions within rules. However, it only knows about certain built-in C language functions. If you want to use your own function or a function that *pbcc* does not recognize, or if you want to use an auxiliary variable, you must declare your intention to *pbcc.* You do this using the `xtype` syntax. For example, say you have written a function to get data from the outside world. This function, as a side effect, asserts facts into the knowledge base, and returns an integer code indicating whether it was able to get some new data. You would include the following statement in your code:

```
  xtype [get_data:int]
```

pbcc would then recognize get_data as a valid function call.

Similarly, if you wanted an integer variable named returncode and a constant named END_OF_FILE, you could include the statements

```
  xtype[returncode:int]
  xtype [END_oF_FILE :int]
```

in your code. You could then write the following rule:

```
rule [get-data (#-99; *) :
     [?|'returncode  !=  'END_OF_FILE]
  ==>
      [! | 'returncode =  get_data()]]
```

This rule will check to see that the `returncode` variable is not set to `END_OF_FILE,` and if not, it will set its value to the result of calling the `get_data()` function. Note that the `returncode` variable should be initialized to a value not equal to `END_OF_FILE` (the user can write a statement to do this in the auxiliary C code section).

Notice that in the antecedent clause, both `returncode` and `END_OF_FILE` are preceded by quote marks. This syntax indicates to *pbcc* that these identifiers have been declared by *xtype.* The quote marks are optional in the consequent.

It is possible to have a function call in an antecedent clause. For example, one could write

```
 xtype[GOOD_DATA:int]
 .
 .
 .
rule[test_data (#-99;*):
      [?|get_data() == 'GOOD_DATA]
 ==>
  etc.
```

The code for user-writ ten C language functions or variable declarations can be included in the body of the rule file if the programmer desires. The syntax for this is simply

```
  {
    ...C language statements.. .
  }
```

Such code can be included at any point in the file outside of the body of a ptype, xtype, or rule declaration.

Another important feature of PBEST is the ability to mark and unmark facts, and to test for these marks. Since marks can have names, an expert system can mark a fact with different marks, and check for these marks by name. One use of this feature is to make sure that all the rules that can possibly use a fact have had the chance to do so. To do this, it helps to organize rules in groups, making sure that at least one rule in a group will always fire, and that each rule in the group will mark the fact with the same mark if it does fire. Here is an example of how this would work:

```
'*******************************************************
' Rule to check for unsuccessful logins for a single user
'*******************************************************

'
' Predicate for counting login errors for a single user
' (lieucount stands for login_item_error_user_count)


ptype[lieucount
        user_id:integer,
        day:integer,
        second:integer,
        maximum:integer,
        current:integer]


' Rule for first unsuccessful login for this user
'

rule[UNSLOG1:
        [+lie:login_item_error^UNSLOG]
        [-lieucount|user_id==lie.user_id]
==>
        [!|printf("UNSLOG1: Found bad login for user %d\n",
                lie.user_id)]
        [+lieucount|user_id=lie.user_id,
                    day=lie.days,
                    second=lie.seconds,
                    maximum=4,
                    current=1]
        [$|lie:UNSLOG]]


' Rule for counting unsuccessful logins after the first,
' up to lieuc.maximum
'

rule[UNSLOG2:
        [+lie:login_item_error^UNSLOG]
        [+lieuc:lieucount|user_id==lie.user_id]
        [?|lieuc.current <= lieuc.maximum - 1]
==>
        [/lieuc|current+=l]
        [!|printf("UNSLOG2: Found bad login for user %d, number %d\n",
               lie.user_id, lieuc.current)]
        [$|lie:UNSLOG]]



' Now we have seen the maximum number of bad logins for this user
'

rule[UNSLOG:
        [+lie:login_item_error^UNSLOG]
        [+lieuc:lieucount|user_id==lie.user_id]
        [?|lieuc.current == lieuc.maximum]
==>
        [!|printf("UNSLOG: Found %d bad logins for user %d\n",
               lieuc.maximum, lie.user_id)]
        [$|lie:UNSLOG]
        [-|lieuc]]


'*******************************************************
' Rule to check for unsuccessful logins for the host system
'*******************************************************

'
' Predicate for counting bad logins for a host
' (liehcount stands for login_item_error_host_count)
'

ptype[liehcount
        day:integer,
        second:integer,
        maximum:integer,
        current:integer]


' Test rule for bad logins
```

```
‘
‘ Rule for first unsuccessful login for this host


rule[UNSLGX1:
        [+lie:login_item_error^UNSLGX]
        [-liehcount]
==>
        [!|printf("UNSLGX1: Bad login for host\n"]
        [+liehcount|day=lie.days,
                    second=lie.seconds,
                    maximum=10,
                    current=11
        [$|lie:UNSLGX]]


‘ Rule for counting unsuccessful logins after the first,
‘ up to liehc.maximum


rule[UNSLGX2:
        [+lie:login_item_error^UNSLGX]
        [+liehc:liehcount|urrent <= liehc.maximum - 1]
==>
        [/liehc|current+=l]
        [!|printf("UNSLGX2: Found %d bad logins for host\n",
                liehc.current)]
        [$|lie:UNSLGX]]


‘ Now we have seen the maximum number of bad logins for this host


rule[UNSLGX:
        [+lie:login_item_error^UNSLGX]
        [+liehc:liehcount|current == liehc.maximum]
==>
        [!|printf("UNSLGX: Found %d bad logins for host\n",
                liehc.maximum)]
        [$|lie:UNSLGX]
        [-|liehc]]
```

```
'
' Remove login item errors seen by rule groups UNSLOG and UNSLGX
'

rule[UNSLGGC:
        [+lie:login_item_error$UNSLOG]
        [+lie:login_item_error$UNSLGX]
==>
        [!|printf("Removing bad login fact with timestamp %s %s\n",
                lie.days, lie.seconds)]
        [-|lie]]
```

This example incorporates two groups of rules, one for single users and one for host systems. The rules relating to single users are named UNSLOG1, UNSLOG2, and UNSLOG, while the ones relating to hosts are UNSLGX1, UNSLGX2, and UNSLGX. Each rule in both groups looks for login item error facts. The single user rules all mark facts with UNSLOG, while the host rules all use UNSLGX. Each rule also checks to see that the fact it is looking at has not already been marked by its rule group. The example includes a rule that will remove facts that have been "seen" by both rule groups.

The syntax for all this is as follows. If a rule wants to mark a fact, it includes a clause like :

```
[$|lie:UNSLGX]
```

in its consequent. This clause is taken from the consequent of the fact named UNSLGX. The $| followed by the fact alias and the mark name indicates the marking of the fact. If a rule wants to check for a fact marked with UNSLGX, it would include the clause

```
[+lie:login_item_error$UNSLGX]
```

in its antecedent. This clause is taken from the antecedent of the UNSLGGC rule.

If a rule wants to check for a fact that has not been marked with a certain mark, it should include a clause of the form

```
[+lie:login_item_error^UNSLGX]
```

in its antecedent. Similarly, if it wants to remove the mark from a fact, it should include a clause of the form

```
[^|lie:UNSLGX]
```

in its consequent.

The UNSLGCC rule gives an example of the technique for removing facts that multiple rules may want to use. This rule tries to find a fact that has been marked by both rule groups. If it does find such a fact, it removes it.

One problem with the marking scheme is that we have not yet provided a robust way to check for a single fact's being marked by multiple marks. Looking at the UNSLGCC rule, we notice that there are two antecedent clauses, each checking for a fact with a particular mark. There is no guarantee that they will both select the same fact, though in actuality they will because they will each scan the list of login item error facts in the same way. However, this depends on several things, including the fact that the rule UNSLGCC follows all the rules that may use the facts it removes, so they will fire first (it would probably be better to give the UNSLGCC rule a lower priority than these other rules). So this way of checking a fact for more than one mark does work, but it is not really clean or robust. We will probably fix the syntax for checking marks at a later time.

A.5 Communicating with the Outside World

The trivial expert system defined above has no need to obtain information from the outside; it creates its own facts and processes them. Even this system, however, reports a result to the user, and thus communicates with the outside world. In general, an expert system can both get information from the outside world and produce information. It does this by using its ability to invoke arbitrary C language function calls.

For facts to be added to the knowledge base, they must be *asserted.* From a C language point of view, this happens when the function *assert_< ptypename >* gets called with the proper arguments. That is, for each ptype the *pbcc* translator creates a function that allows facts of that type to be asserted. To use these functions the user can write rules like the following:

```
'   This rule reads data into the knowledge base using the
'   get_fact_record() routine. It has a very low
'   priority so it doesn't add new facts until the old ones
'   have been processed.


rule[get_fact_record_data (#-99;*):
     [?|'retval != 'END_OF_FILE]
```

```
    [+c : count]
 ==>
    [/c|value += 1]
    [!| retval = get_fact_record()]]
```

The heart of this rule is the call to the C function *get_fact_record().* Besides this, the rule checks for end of file and keeps a count of the number of records read. The get_fact_record() function will read a record from somewhere (a file), move it into a series of variables, then call the *assert_transaction()* function with these variables for arguments.

Here is example code showing how the get_fact_record() function might look:

```
int get_fact_record()
{

  int i, reader() ;

  /*
    Read a record from the file into the buffer strings.
  */
  i = reader(infp,(struct iovec *)iov1, 26);
  if (i < 1)
    if (i == -1) {
      fprintf (stderr,  Problem with data file, error %d\n , errno) ;
      return(NO_DATA) ;
    } else {
      fprintf(stderr,  Reached end of file. \n ) ;
      return (END_OF_FILE) ;
    }

  assert-transaction (
                          dbid,
                          cpuid,
                          repdate,
                          reptime,
                          recdate,
                          rectime,
                          day,
                          week,
                          recordtype,
                          jobname,
                          userarea,
                          usertype,
                         userid,
                         altuserid,
                         terminal,
                         logon,
                         program,
                         idesfile,
                         command,
                         response,
                         duration,
                         enqueue,
                       );
  return(GOOD_DATA);
}
```

For clarity, lots of detail is left out of this example. The get_fact_record() routine, however, simply calls reader() to read a record into the proper buffer strings, then calls assert_transaction() to assert a transaction fact with this data into the knowledge base. Note that the reader() function must be written by the user, but the assert_transaction() function will be produced by the *pbcc* translator when it sees the following ptype declaration in the rule file:

```
  ptype [transaction
    dbid:string,
    cpuid:string,
    repdate:string,
    reptime:string,
    recdate:string,
    rectime:string,
    day:string,
    week:string,
    recordtype:string,
    jobname:string,
    userarea:string,
    usertype:string,
    userid:string,
    altuserid:string,
    terminal:string,
    logon:string,
    program:string,
    idesfile:string,
    command:string,
    response:string,
    duration:string,
    enqueue:string
]
```

The steps in making an input interface can be summarized as follows.

- Declare ptypes for the kinds of facts you want to assert into the knowledge base.
- Write a C language function that calls the proper assert_< *ptypenane* >() function.
- Write a rule that includes in one of its consequents or antecedents a call to this C language function.

A.6 Other Programming Considerations

By default, *pbcc* creates a stand-alone C program. If the user invokes *pbcc* with the -e flag, the program that results will not contain a *main0* function. This allows the user to "embed" the expert system in a larger program or to deal with special initialization requirements. Any time the expert system is used in an embedded fashion, the user-written main() function must call the initialization routines *gc_init()* and *pb_init()* (in that order). The gc_init() routine initializes the garbage-collecting memory allocator, while pb_init() initializes the expert system itself. The program must call these functions before calling any other expert-system related functions.

If the program needs to transfer control to the expert system at some point, it should then call the *engine()* function. This function will return when there are no more rules that can fire. There is no provision for finer-grained control of the expert system at this level.

If the program intends to use the X-Windows version of the expert system, it should not call engine(); instead, it should call the function *pb_xmonitor().* This function uses *argc* and *argv,* the complete call should be

```
pb_xmonitor(argc, argv);
```

This allows the X server to get access to X specific command line arguments.

Note that the expert system uses its own memory-allocation system that is not compatible with *malloc()* and its related functions. A program that wants to allocate memory that may be passed to expert system functions should not use malloc() or calloc(); instead, it should request memory with the functions *gc_malloc()* and gc_malloc_atomic(). The gc_malloc() function allocates memory that may contain pointers, while gc_malloc_atomic() can be used to allocate memory for things like strings that do not contain pointers. These functions zero the memory they allocate.

One of the things the gc_init() function does is allocate a default heap from which to allocate memory. If the programmer knows that the program will use lots of memory and create lots of garbage, it may speed things up somewhat to force expansion of the heap. (Note that an expert system that removes facts will create lots of garbage.) The function *expand_hp()* does this. By expanding the heap, the program can run longer before it needs to collect garbage. Note that if the memory allocator is unable to reclaim enough memory by garbage collection to satisfy a memory request, it will automatically expand the heap.

Since these functions use garbage collection to reclaim memory when necessary, the program does not need to explicitly free the memory it obtains from them. However, functions to explicitly free memory are provided.

For a complete description of the garbage collector, see the GC_README file included with the source code.

A.7 The PBEST Interactive Window System

This section describes the interactive window system that *pbcc* produces when it is invoked with the -x flag. Figure 1 shows the display produced by running the *tstx* program described in Section A.2. Note that this actually consists of two windows, the main window and the menu window, arranged to overlap.

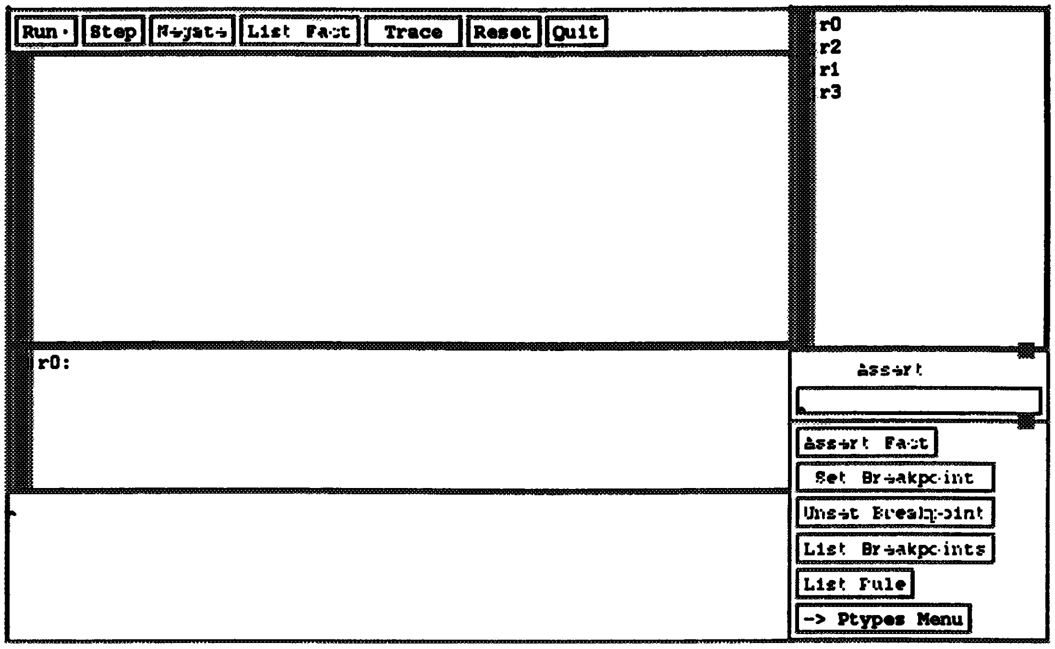
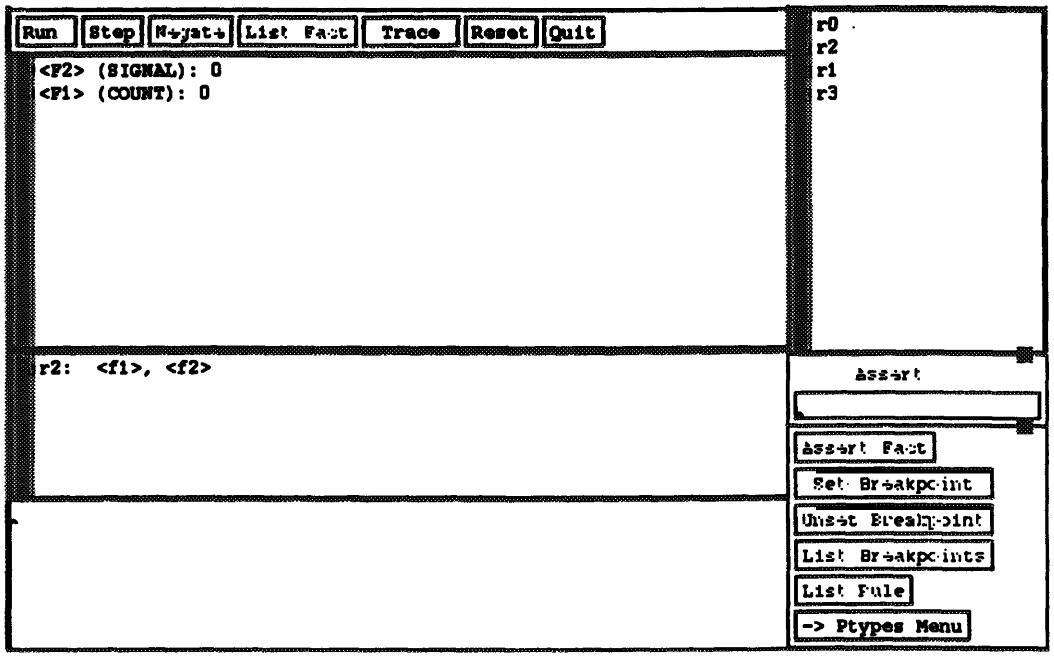


Figure A.l: The PBEST Interactive Window

The main window has four areas. From top to bottom they are: the main buttons panel; the facts list, which is empty in Figure A.l; the firable rules list, which contains r0; and the message window, which is empty. Notice that the *Negate* and *List Fact* buttons are ghosted. Since the user has not selected a fact (as there are not yet any to select), it is impossible to negate or list a fact.

The menu window, on the right, has three areas: the menu list, a user-input area marked "Assert", and the menu buttons panel. The menu window is currently listing rules. There are four rules listed, r0 through r4. All of the menu buttons relating to rules are ghosted since the user has not selected any rule. The *Assert Fact* button is also ghosted, since ptypes are not being displayed.



```
<F2> (SIGNAL): 0
<F1> (COUNT): 0
```

```
r0
r2
r1
r3
```

```
r2:   <f1>, <f2>
```

Figure A.2: The Result of Clicking on the *Step* Button

Figure A.2 shows the result of clicking on the *Step* button. Since r0 was the only rule that could fire, the engine executed the conclusion of r0. The clauses in the conclusion of r0 asserted two facts into the fact base. These are shown in the facts list as <F1> and <F2>. They are identified with their types and the values of their fields. Note that facts get displayed in inverse chronological order, so the most recent facts will be visible by default. Since the facts list is a scrollable window, the user can view any fact, but usually the most recent ones are of interest.

Besides the facts, the firable rules list has changed. It now contains the line

```
r2:   <f1>, <f2>
```

This line says that r2 can fire, and the facts that make it firable are facts <F1> and <F2>. (Note that it is possible for many combinations of facts to make a rule firable. In such cases, the more recent facts will be selected to activate the rule.)

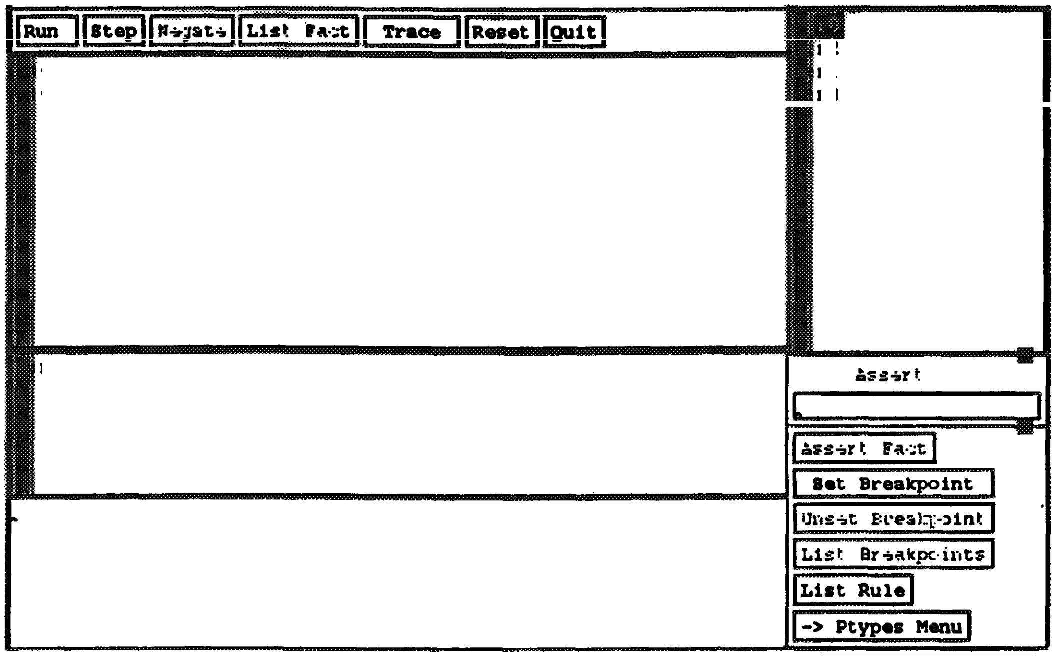


Figure A.3: The Result of Selecting a Rule from the Rules List

Figure A.3 shows the result of selecting a rule from the rules list in the menu window. In this instance, the user has clicked on r0. Two of the buttons in the menu buttons panel have become active.

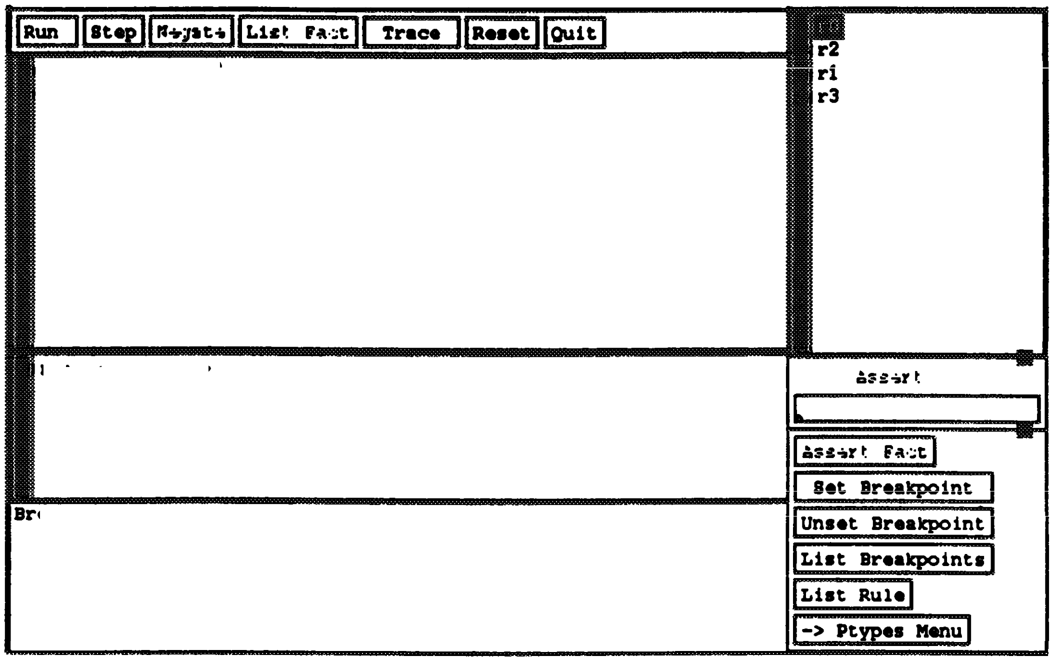
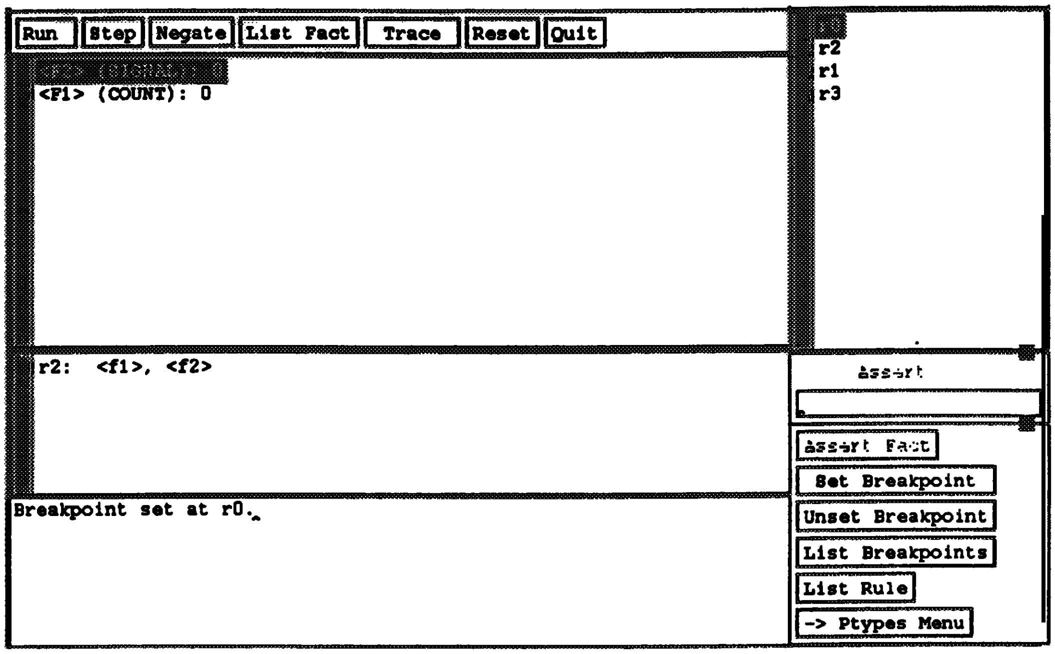


Figure A.4: The Effect of Clicking on the *Set Breakpoint* Button

Figure A.4 shows the effect of clicking on the *Set Breakpoint* button in the menu buttons panel. The message window says that a breakpoint has been set at r1, and the *Unset Breakpoint* and *List Breakpoints* buttons in the menu buttons panel have become active.



```
<F1> (COUNT): 0
```

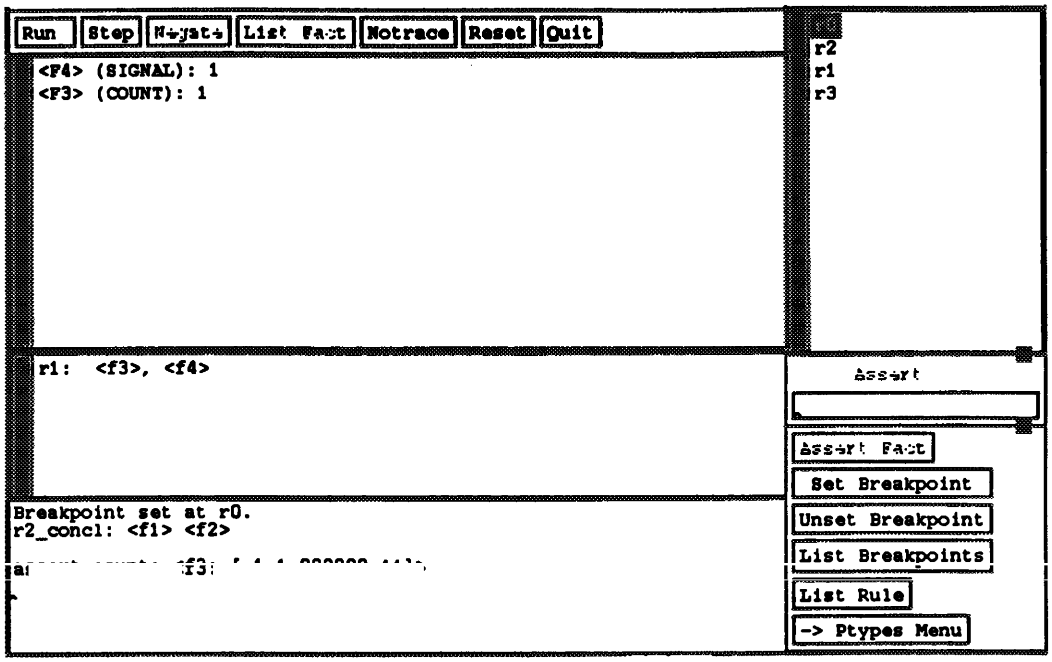
```
r2
r1
r3
```

```
r2:  <f1>, <f2>
```

```
Breakpoint set at r0.
```

Figure A.5: The Effect of Selecting a Fact in the Facts List

Figure A.5 shows the effect of selecting a fact in the facts list. The *Negate* and *List Fact* buttons in the main buttons panel have become active.



```
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

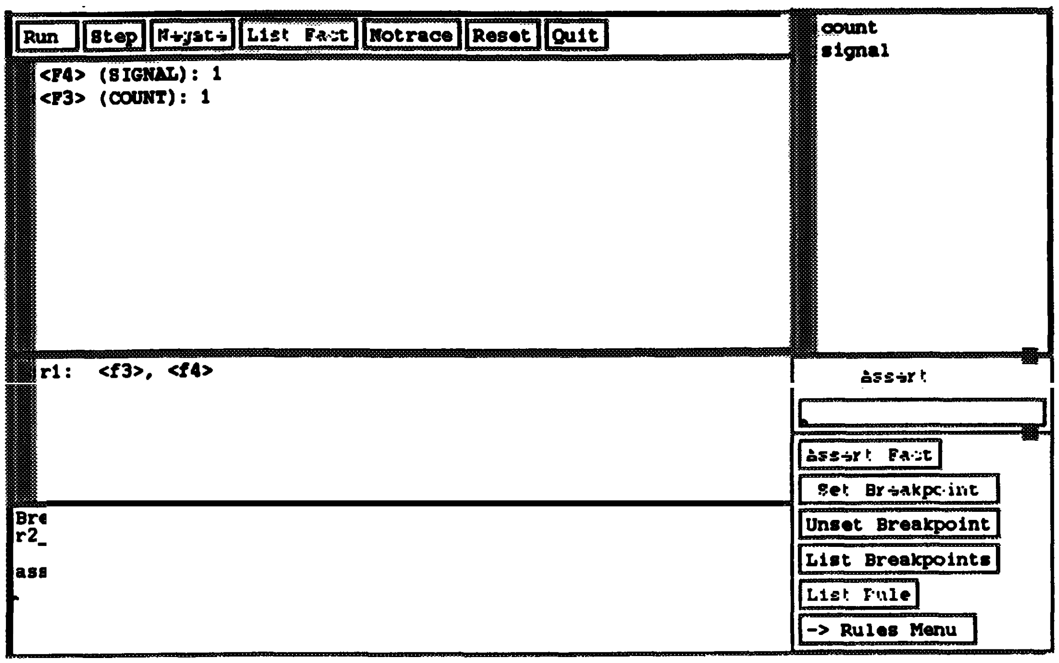
```
r2
r1
r3
```

```
r1:  <f3>, <f4>
```

```
Breakpoint set at r0.
r2_concl: <f1> <f2>

a[illegible]f3[illegible]
```

Figure A.6: The Result of Clicking on the *Trace* Button

Figure A.6 results from the user's clicking on the *Truce* button in the main buttons panel and then clicking on the *Step* button. Notice that the *Trace* button has changed into a *Notrace* button. Besides this, each time a rule fires, the engine displays messages in the message window, indicating some of its effects. In this case, r2 has deleted <Fl>, asserted a new count fact, <F3>, and modified <F2>. Note that when a fact gets modified, the engine does this by deleting the old fact and asserting a new fact with the fields of this fact set to their new values. Thus, <F2> has become <F4>. This means that the engine will look at newly modified facts before looking at older facts when it tries to find groups of facts to fire a rule.



```
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

```
r1:  <f3>, <f4>
```

```
Bre
r2_

ass
```

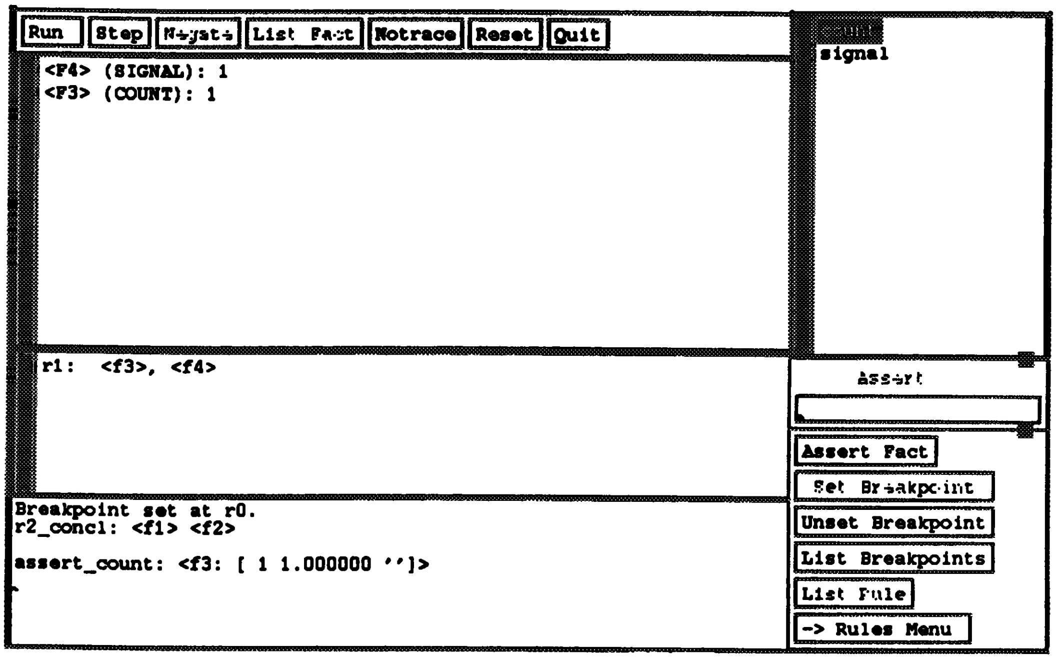
Figure A.7: The Result of Clicking on the -> *Pypes Menu* Button

Figure A.7 shows the result of clicking on the -> *Ptypes Menu* button. This button lets you switch the menu display between ptypes and rules. Once you are displaying ptypes, you can select a ptype and assert a new fact of this ptype into the knowledge base. This can be useful for making sure a rule will fire in the presence of a fact with certain characteristics.

To assert a fact, perform the following steps.

- Make the menu display ptypes.
- Select the desired ptype.
- Click on the *Assert Fact* button.
- Type the fact fields into the Assert box in response to the prompts that will show up above the typing area.

The following sequence of figures illustrates this process.



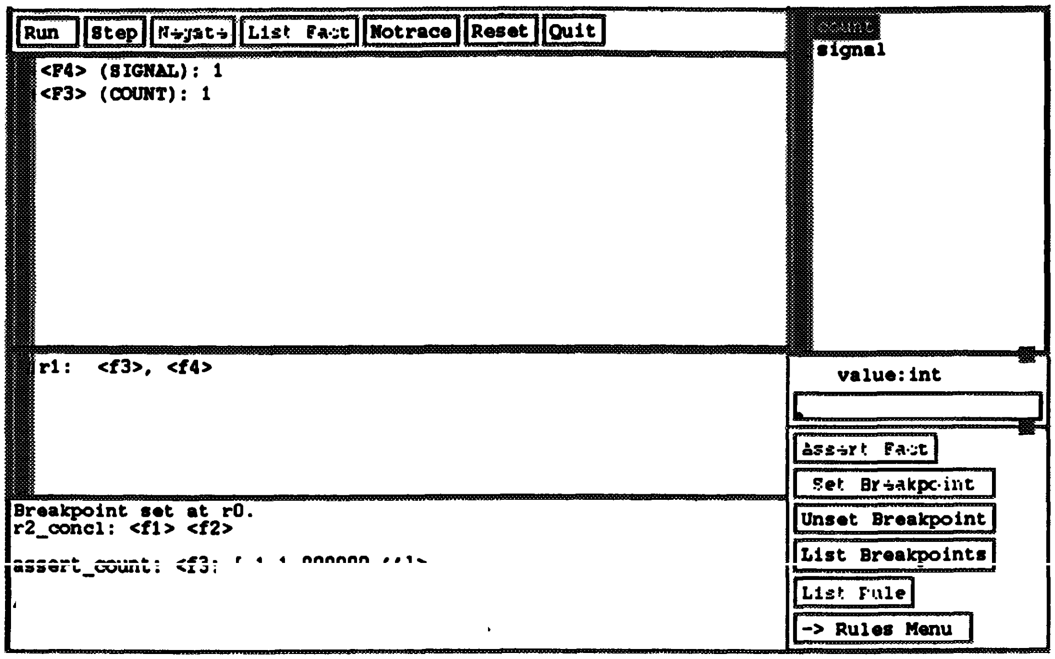
```
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

```
r1:  <f3>, <f4>
```

```
Breakpoint set at r0.
r2_concl: <f1> <f2>

assert_count: <f3: [ 1 1.000000 '']>
```

Figure A.8: Selecting the *count* Ptype

In Figure A.8, the user has selected the *count* ptype from the list of ptypes in the menu window. The *Assert Fact* button has become active.



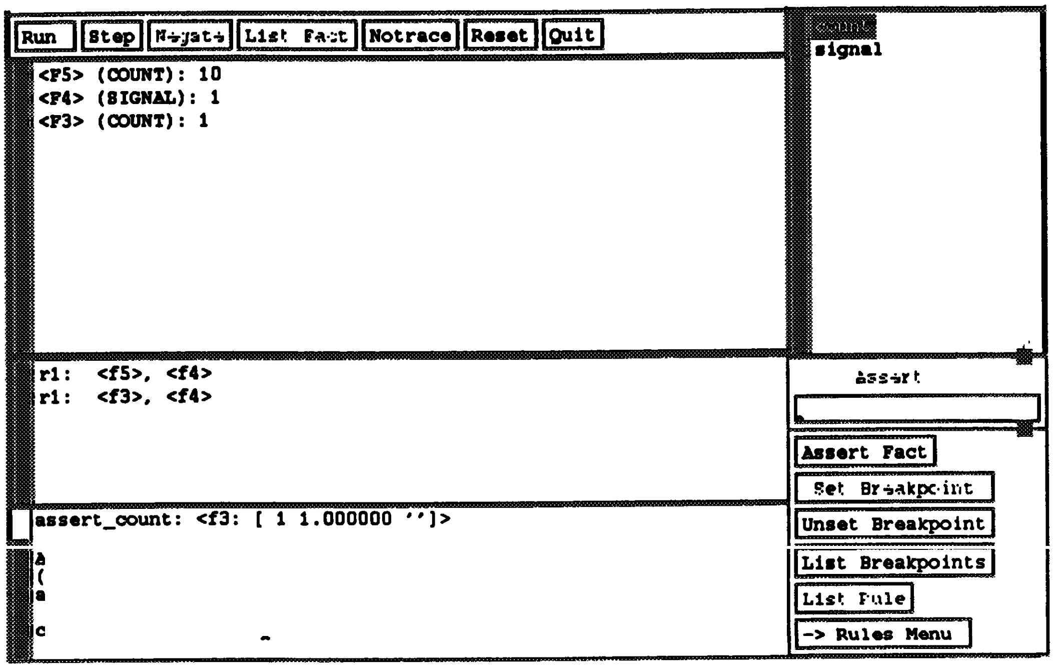
```
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

```
r1:  <f3>, <f4>
```

```
Breakpoint set at r0.
r2_concl: <f1> <f2>

assert_count: <f3: [illegible]
```

Figure A.9: The Effect of Clicking on the *Assert Fact* Button

The user has clicked on the *Assert Fact* button in Figure A.9. It becomes inactive (you have to finish asserting a fact before you can assert a new one). A message appears in the message window, indicating that the user is asserting a count fact, and explaining how to abort the assertion process. A prompt, value:int, has appeared in the entry area of the menu window.



```
<F5> (COUNT): 10
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

```
r1:  <f5>, <f4>
r1:  <f3>, <f4>
```

```
assert_count: <f3: [ 1 1.000000 '']>
```

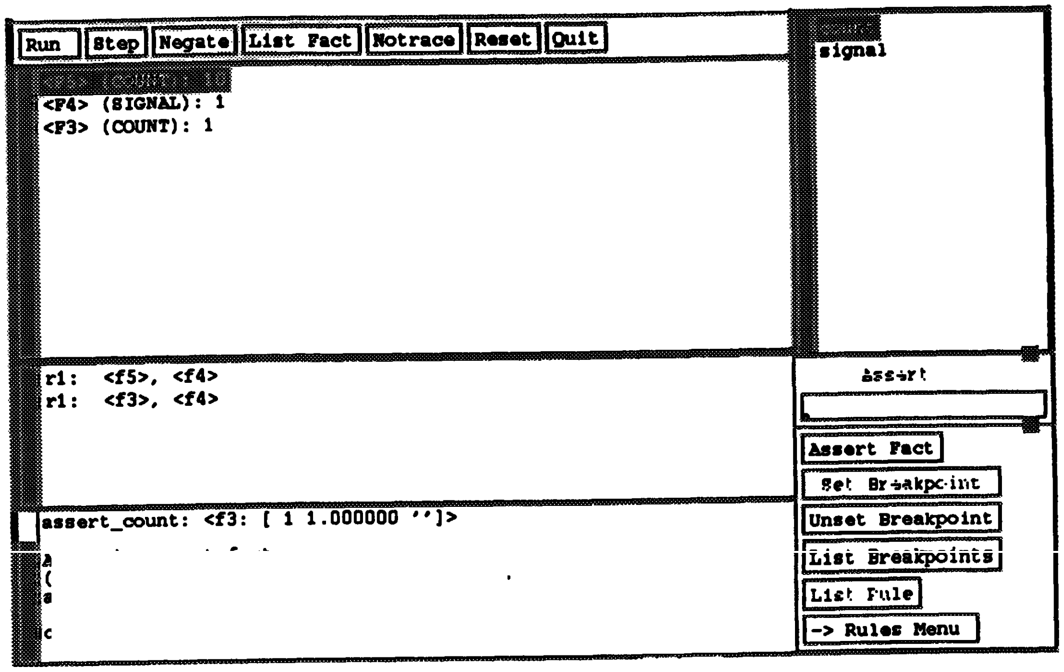
Figure A.10: Fact is Asserted

In Figure A.10, the user typed in the number 10 and pressed <return>.

Since this is the only field for that fact, the fact gets asserted. A message to this effect shows up in the message window, and the fact itself shows up in the facts list window.

Note that if the user had typed in some invalid input, for example, lo instead of 10, this input would not be accepted. Instead, the system would beep when the <return> key was pressed.

Facts can also be removed from the knowledge base with the *Negate* button in the main buttons panel.



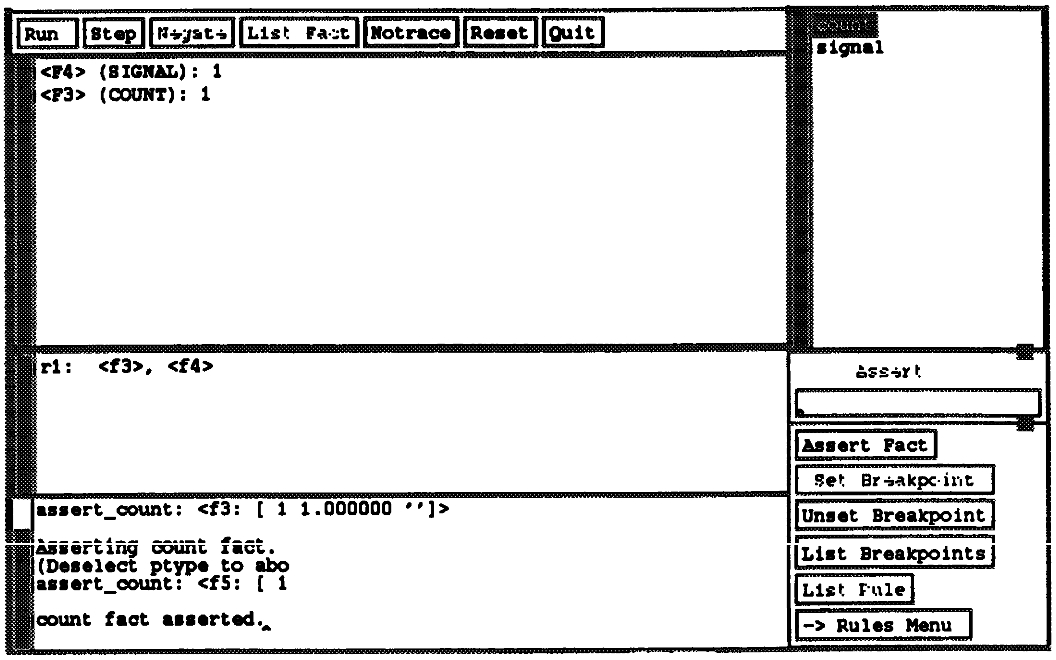
```
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

```
r1:  <f5>, <f4>
r1:  <f3>, <f4>
```

```
assert_count: <f3: [ 1 1.000000 '']>
```

Figure A.11: Selecting a Fact to Negate

The user has selected a fact to negate in Figure A.11. The *Negate* and *List Fact* buttons have become active.



```
<F4> (SIGNAL): 1
<F3> (COUNT): 1
```

```
r1:  <f3>, <f4>
```

```
assert_count: <f3: [ 1 1.000000 '']>

Asserting count fact.
(Deselect ptype to abo
assert_count: <f5: [ 1

count fact asserted.
```

Figure A.12: The Effect of Clicking on the *Negate* Button

In Figure A.12, the user has clicked on the *Negate* button. The fact gets removed from the knowledge base. The facts list is updated to reflect this, and the *Negate* and *List Fact* buttons return to their inactive state.

We can summarize the functions of the buttons as follows:

Run The Run/Stop toggle. Clicking on it puts the expert-system engine in free-running mode if it is stopped, or stops it if it is running. The button label changes appropriately.

Step Drives the engine through one rule-firing cycle. It can also be used to stop the system if it is running.

Negate Removes the selected fact from the knowledge base.

List Fact Lists the selected fact. It is useful mostly when the fact has many fields that cannot all fit in the fact list window.

Trace Toggles tracing of the execution of the engine. When tracing is enabled, the engine prints messages indicating what it is doing in the message window. The label in the button will change to reflect what clicking on the button will do.

Reset Deletes all facts and breakpoints.

Quit Terminates the execution of the program and returns control to the shell.

The buttons in the menu buttons panel work as follows.

Set Breakpoint Associates a breakpoint with the selected rule; when the running engine tries to fire a rule with a breakpoint, it will stop instead. Clicking on *Run* or Step after the engine stops at a breakpoint causes the engine to execute the conclusion of the rule with the breakpoint and then continue as appropriate. Note that breakpoints are not removed when they are hit. If the engine tries to fire the rule again later, the breakpoint will again cause it to stop.

Unset Breakpoint Removes the breakpoint from the selected rule, if the rule has one. If the rule doesn't have a breakpoint, the program displays a message in the message window indicating that the selected rule had no breakpoint.

List Breakpoints Causes the program to print a list of rules with breakpoints in the message window.

List Rule Causes the program to print some information about the selected rule in the message window.

-> Ptypes Menu / -> Rules Menu Toggles the menu window between displaying ptypes and rules.

Assert Fact Allows the user to assert an instance of a selected ptype as a fact into the knowledge base.

A.8 Certainties and Justifications

The PBEST system currently supports three kinds of certainty factors: standard (MYCIN-like), Bayesian, and fuzzy. These can be specified by a command line argument to *pbcc.* See Section A.11 on invoking *pbcc* for how to do this. We do not presently anticipate using certainty factors in the IDES project. Should this change, we will incorporate a more complete description of certainty factors into this document.

PBEST also has a justification mechanism; however, this is not well supported. We currently consider it to be superfluous in light of PBEST's ability to execute arbitrary C language functions. Should our opinion on this matter change, we will modify both the implementation and the documentation to reflect whatever role we anticipate for justifications.

A.9 A Sample Makefile

The following is a *Makefile* that automates the production of expert systems from specification files. To produce both a window (based on the X Window System) and nonwindow version from the same set of rules, just give the command

```
make
```

at the shell prompt. To create only the X-Windows version, give the command

```
make tstx
```

at the shell prompt. To create the nonwindow version, give the command

```
make tst
```

at the shell prompt.

This file uses certain macro abbreviations that specify the locations of the libraries and other files needed to make the executable programs. The user should make sure that these are set correctly.

```
# Directory structure macros -- should only need to change
# the value of the ROOT macro
ROOT=/home/caesar

IDES=$(ROOT)/ides
IDESLIB=$(IDES)/lib

# Where the PBEST translator is
PBCC= $(IDES)/bin/pbcc

# Show version number.
PBCCFLAGS= -v

# The garbage collector and runtime libraries
PBCCLIBS=$(IDESLIB>/gc.o  $(IDESLIB)/libpb.a

# The window libraries
XLIBS= $(IDESLIB)/libiui.a -lXaw -lXmu -1Xt -1X11

# C compiler stuff -- need the system v compatibility
# version
CC= /usr/5bin/cc

# Include symbols, search the ides and
# global include directories
CFLAGS= -g -I$(IDES)/include -I/usr/global/include

#
.SUFFIXES: .pbest
#

############################################################
# Everything above here can be used as a skeleton Makefile.
# The only thing that should need to change is ROOT, which
# should be set when the system is installed.
#
# Things below should be used more as an example. The main
# thing is to make sure each language processor knows how to
# find the libraries it is looking for.
############################################################

SRC=         tst.pbest
```

```
OBJ=          tst.o
XOBJ=         tstx.o

#
# NOTE:  You must use TABS (^I) between the target and the
#        dependencies, and to start off the command lines
#        below each target-dependency specification.
#        Failure to do this will cause the make program to
#        refuse to do what you want and to produce a series
#        of uninformative error messages
# this space made by tabs
#      ||||||
#      VVVVVV
all:         tst tstx

tst:         $(OBJ)
             $(cc) -o $@ $(OBJ) $(PBCCLIBS)

tstx:        $ (XOBJ)
             $(CC) -o $@ $(XOBJ) $(PBCCLIBS) $(XLIBS)

tst.o:       tst.c
             $(CC) $(CFLAGS) -c tst.c

tst.c:       tst.pbest
             $(PBCC) $(PBCCFLAGS) -o $@ tst.pbest

tstx.o:      tstx.c
             $(CC) $(CFLAGS) -c tstx.c

tstx.c:      tst.pbest
             $(PBCC) $(PBCCFLAGS) -x -o $@ tst.pbest

end:
```

A.10 PBEST Syntax Diagrams

This section includes syntax diagrams for the PBEST expert-system specification language. The diagrams use fairly standard conventions. For example, <object a> ::=

<object b> means that the object on the left has the syntax shown on the right. "Literals" or keywords are in bold face. This includes brackets, bars, plus signs, and so on that are written in boldface. The same symbols in lightface have a different meaning, as follows.

- | (nonbold vertical bar) indicates alternatives; that is, we would say

```
<statement >      ::= <ptypedef>
                  |    <ruledef>
                  |    <ccode>
                  |    <xrefdef>
                  |    <comment >
```

  to indicate that a statement can be a ptypedef, a ruledef, an instance of ccode, an xrefdef, or a comment.

- * (nonbold superscript asterisk) indicates that an item can occur as many times as desired, but need not appear. For example,

```
<ante>            ::=  [<clause>]*
```

  says that an antecedent consists of zero or more clauses.

- + (nonbold superscript plus) indicates that an item can occur as many times as desired, but must appear at least once. For example,

```
<cons>            ::=  [<action>]+
```

  says that a consequent must consist of at least one action clause (though it may have as many more as desired).

- [ ] (nonbold brackets) surrounding an item indicate grouping. In the examples above, the brackets indicate that the asterisk and plus apply to whatever is inside the brackets. If the brackets surround an item, but are not followed by an asterisk or plus, they indicate that the item may occur zero times or once. For example

```
<marktest> ::= $[<name>]
             |    ^[<name>]
```

  indicates that a mark test has an optional name after the dollar sign or up caret.

Several abbreviations are used. Some may be confusing:

- relop - relational (comparison) operator
- assop - assignment operator
- funcall - C language function call
- rcf - rule certainty factor

- exstat - externally defined statement

```
<pbprog>     ::=  [<statement>]+

<statement > ::=  <ptypedef>
               |   <ruledef>
               |   <ccode>
               |   <xrefdef>
               |   <comment>

<ptypedef>   ::=  ptype [ <name> <fields> ]

<fields>     ::=  <field>[,<field>]*

<field>      ::=   <name>
               |   <name>:<typename>

<typename>   ::=  integer
               |   float
               |   list
               |   string
               |   symbol

<ruledef>    ::=   rule [ <name> [(<opts> )] : <ante> => <cons> ]

<opts>       ::=  <opt>[;<opt>]*
```

```
<opt >            |   <rank>
                  |   <repeat>
                  |   <certainty>
                  |   <explanation>


<rank>           ::=   # <integer>


<repeat >        ::=   *


<certainty>      ::=   ~<real>


<explanation>    ::=   <string>


<ante>           ::=   [<clause>]*


<clause>         ::=   [+<pname>[<marktest>][<restrictions>]]
                  |    [-<name>[<marktest>][<restrictions>]]
                  |    [?<restrictions>]


<pname>          ::=   <name> : <name>
                  |    <name>


<marktest>       ::=   $[<name>]
                  |    ^[<name>]


<restrictions>   ::= | <restricts>


<restricts>      ::= <restrict>[,<restrict>]*


<restrict >      ::= <expr> <relop> <expr>
                  |   <expr>
```

```
<expr>          ::= ( <expr> )
                |   <expr> + <expr>
                |   <expr> - <expr>
                |   <expr> * <expr>
                |   <expr> / <expr>
                |   <expr> % <expr>
                |   <expr> >> <expr>
                |   <expr> << <expr>
                |   <expr> & <expr>
                |   <expr> ^ <expr>
                |   <expr> | <expr>
                |   <expr> && <expr>
                |   <expr> || <expr>
                |   <pvalue>


<relop>         ::= ==
                |   !=
                |   >
                |   <
                |   >=
                |   <=


<pvalue>        ::= <name>
                |   ' <name>
                |   <integer>
                |   <real>
                |   <string>
                |   <pfield>
                |   <list >
                |   <funcall>
```

```
<assop>         ::=  =
                 |   +=
                 |   -=
                 |   *=
                 |   /=
                 |   %=
                 |   >>=
                 |   <<=
                 |   &=
                 |   ^=
                 |   |=
                 |   %=

<pfield>        ::= <name> . <name>

<list>          ::= [ [<pvalue>)*]

<funcall>       ::= <name> ( [<arglist>] )

<arglist>       ::= [<pvalue>],+

<cons>          ::= [<action>]+

<action>        ::= [<negate>]
                 |   [ <mark>]
                 |   [<unmark>]
                 |   [ <assert>]
                 |   [<modify>]
                 |   [ <execute> ]

<negate>        ::=  -| [<name>]+,

<mark>          ::=  $| <name> [ : <name> ]

<unmark>        ::=  ^| <name> [ : <name> ]
```

```
<assert >       ::=  +  <name>  [<arestricts>][<rcf>]

<modify>        ::=  /  <name>  [<arestricts>][<rcf>]

<arestricts>    ::=  |  <arestrict>[,<arestrict>]*

<arestrict>     ::=  <expr>  <assop>  <expr>

                |     <expr>
<execute>       ::=       !|[<exstat>]T

<exstat>        ::=  <xassign>
                |     <funcall>

<xassign>       ::=   [']<name>  <assop>  <expr>

<rcf>           ::=   ~  <real>

<ccode>         ::=  {  <string>  }

<xrefdef>       ::=  xtype  [  <name>  :  <typename>  ]

<name>          ::= [A-Za-z]+[A-Za-z0-9_]I'

<string>        ::= "  [<any  character>]*  "

<integer>       ::=   [0-9]+

<real>          ::= [0-9]*. [o-g]*
```

```
<comment>        ::=    ' [<any character>]* <newline>
```

A.11 Invoking *pbcc*

NAME

pbcc - translate a PBEST source program into C code.

SYNOPSIS

pbcc [-c[s|f|b]] [-d|-x] [-e] [-j] [-vI [-o <file>] [<file>]

DESCRIPTION

pbcc is a program for translating expert-system rule bases written in the language of PBEST into C language source code for compilation. The following is a description of the command line arguments that the user can give when invoking *pbcc.*

-c[s|f|b] Turns on certainty factors. `-cs` specifies standard certainty factors that implement the Mycin strategy, `-cf` specifies fuzzy certainty factors, and `-cb` specifies Bayesian certainty factors.

-d Turns on the Sunview-based debugging options. This will create a source module that, when compiled and linked with the `pb` library, will produce a window-based monitor for the expert system. See also the `-x` option below. Note that either `-d` or `-x` can be specified, but not both.

-e Suppresses the generation of a `main()` function. In this case, the user must provide a `main()` function that includes calls to the initialization routines `gc_init()` and `pb_init()`, and a call to one of `engine()`, `pb_monitor()` or `pb_xmonitor()`, depending on which version of the expert system is being generated.

-j Turns on the generation of fact justifications.

-0 <file> Places output into the file named <file> in the current working directory. If this option is not specified, output is sent to standard output.

-v Causes the version number and release date of the current version of `pbcc` to be printed.

-x Turns on the X-windows-based debugging options. This will create a source module which, when compiled and linked with the `pb` library, will produce a window-based monitor into the expert system. Note that either -d or -x can be specified, but not both.

Bibliography

[l] T. F. Lunt, R. Jagannathan, R. Lee, S. Listgarten, D. L. Edwards, P. G. Neumann, H. S. Javitz, and A. Valdes. *Development and Application of IDES: A Real-Time Intrusion-Detection Expert System.* Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1988.

[2] T. F. Lunt, R. Jagannathan, R. Lee, A. Whitehurst, and S. Listgarten. Knowledge-based intrusion detection. In *Proceedings of the 1989 AI Systems in Government Conference,* March 1989.

[3] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection system. In *Proceedings of the 1988 IEEE Symposium on Security and Privacy,* April 1988.

[4] T. F. Lunt. Real-time intrusion detection. In *Proceedings of COMPCON Spring 89,* March 1989.

[5] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, and C. Jalali. A progress report. In *Proceedings of the Sixth Annual Computer Security Applications Conference,* December 1990.

[6] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, C. Jalali, H.S. Javitz, A. Valdes, and P.G. Neumann. *A Real- Time Intrusion-Detection Expert System.* Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1990.

[7] T. F. Lunt. Using statistics to track intruders. In *Proceedings of the Joint Statistical Meetings of the American Statistical Association,* August 1990.

[8] T. F. Lunt. IDES: An intelligent system for detecting intruders. In *Proceedings of the Symposium: Computer Security, Threat and Countermeasures,* Rome, Italy, November 1990.

[9] H.S. Javitz and A. Valdes. The SRI statistical anomaly detector. In *Proceedings of the 14th National Computer Security Conference,* October 1991.

[10] T.D. Garvey and T.F. Lunt. Model-based intrusion detection. In *Proceedings of the 14th National Computer Security Conference,* October 1991.

[11] J. P. Anderson. *Computer Security Threat Monitoring and Surveillance.* Technical report, James P. Anderson Company, Fort Washington, Pennsylvania, April 1980.

[12] H. S. Javitz, A. Valdes, D. E. Denning, and P. G. Neumann. *Analytical Techniques Development for a Statistical Intrusion Detection System (SIDS) Based on Accounting Records.* Technical report, SRI International, Menlo Park, California, July 1986. Not available for distribution.

[13] T. F. Lunt, J. van Horne, and L. Halme. *Automated Analysis of Computer System Audit Trails.* In *Proceedings of the Ninth DOE Computer Security Group Conference,* May 1986.

[14] T. F. Lunt, J. van Horne, and L. Halme. *Analysis of Computer System Audit Trails--Initial Data Analysis.* Technical Report TR-85009, Sytek, Mountain View, California, September 1985.

[15] T. F. Lunt, J. van Horne, and L. Halme. *Analysis of Computer System Audit Trails-Intrusion Characterization.* Technical Report TR-85012, Sytek, Mountain View, California, October 1985.

[16] T. F. Lunt, J. van Horne, and L. Halme. *Analysis of Computer System Audit Trails-Feature Identification and Selection.* Technical Report TR-85018, Sytek, Mountain View, California, December 1985.

[17] T. F. Lunt, J. van Horne, and L. Halme. *Analysis of Computer System Audit Trails-Design and Program Classifier.* Technical Report TR-86005, Sytek, Mountain View, California, March 1986.

[18] J. van Horne and L. Halme. *Analysis of Computer System Audit Trails-Training and Experimentation with Classifier.* Technical Report TR-85006, Sytek, Mountain View, California, March 1986.

[19] J. van Horne and L. Halme. *Analysis of Computer System Audit Trails-Final Report.* Technical Report TR-85007, Sytek, Mountain View, California, May 1986.

[20] T. F. Lunt. Automated audit trail analysis and intrusion detection: A survey. In *Proceedings of the 1lth National Computer Security Conference,* October 1988.

[21] D. E. Dennning and P. G. Neumann. *Requirements and Model for IDES-A Real-Time Intrusion Detection System.* Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1985.

[22] D. E. Denning. An intrusion-detection model. *IEEE Transactions on Software Engineering,* 13(2), February 1987.

[23] National Computer Security Center. *Department of Defense Trusted Computer System Evaluation Criteria.* DOD 5200.28-STD, Department of Defense, December 1985.

[24] R. Jagannathan and A.A. Faustini. GLU: A system for scalable and resilient large-grain parallel processing. In *Proceedings of 24th Hawaii International Conference on System* Sciences, Kauai, Hawaii, January 1991.

[25] R. Jagannathan and A.A. Faustini. *The GLU Programming Language.* Technical Report SRI-CSL-90-11, Computer Science Laboratory, SRI International, Menlo Park, California, November 1990.

[26] R. Jagannathan. Transparent multiprocessing in the presence of fail-stop faults. In *Proceedings of the Third Workshop on Large-Grain Parallelism,* Pittsburgh, Pennsylvania, October 1989.

[27] R. Jagannathan. *A Descriptive and Prescriptive Model for Dataflow Semantics.* Technical Report CSL-88-5, Computer Science Laboratory, SRI International, Menlo Park, California, May 1988.

[28] John D. Lowrance, Thomas D. Garvey, and Thomas M. Strat. A framework for evidential-reasoning systems. In *Proceedings of the National Conference on Artificial Intelligence,* pages 896-903,445 Burgess Drive, Menlo Park, California, August 1986. American Association for Artificial Intelligence.