SRI International

SYSTEM DESIGN DOCUMENT: NEXT-GENERATION INTRUSION DETECTION EXPERT SYSTEM (NIDES)

R. Jagannathan, Computer Science Laboratory
Teresa Lunt, Computer Science Laboratory
Debra Anderson, Computer Science Laboratory
Chris Dodd, Computer Science Laboratory
Fred Gilham, Computer Science Laboratory
Caveh Jalali, Computer Science Laboratory
Hal Javitz, Statistics Program
Peter Neumann, Computer Science Laboratory
Ann Tamaru, Computer Science Laboratory
Alfonso Valdes, Applied Electromagnetics and Optics Laboratory

SRI Project 3131
Contract N0039-92-C-0015

Prepared for:

Department of the Navy
Attn: SPAWAR 02-22D
Space and Naval Warefare Systems Command
Washington, DC 20363-5100

333 Ravenswood Avenue • Menlo Park, CA 94025-3493• (415) 326-6200 •FAX: (415) 326-5512 •Telex: 334486

Contents

List of Figures

March 9, 1993

1 Scope

1.1 System Overview

The purpose of NIDES (Next-generation Intrusion Detection System) is to detect intrusive and suspicious activities on computer systems in real time. Audit data, representing computer system activity of individual *subjects,* is collected by NIDES from one or more systems (known as *target hosts),* both statistical and rule-based analysis of the audit data is continuously performed, and the results are resolved and reported to a graphical user-interface (known as the *security officer user interface.)*

1.2 Document Overview

The purpose of this document is to provide a comprehensive description of the NIDES prototype software to enable a programmer to fully understand the structure and operation of the system.

The document is organized as follows. Section 2 presents an overview of the NIDES prototype. It includes an overview of the prototype consisting of a description of the prototype architecture, execution control, data flow, and resource requirements. It further includes the prototype design description in terms of core and infrastructural components and their embodiments as processes. Section 3 describes each of the core and infrastructural components as well as processes that embody them.

Section 4 describes the data files that are used by one or more components. Section 5 shows how the requirements allocated to components meet the requirements of the prototype itself. Section 6 highlights the essential differences between the NIDES prototype and the IDES prototype. Section 7 lists reports, papers, and manuals that are referred to in this document. Finally, Section 8 includes an alphabetical listing of all NIDES-specific terms along with their meanings as used in this document. The appendix consists of a description of the NIDES audit record format.

2 Design Overview

In this section, a top-level description of the NIDES prototype software is described.

2.1 Prototype Overview

The NIDES prototype monitors activities on multiple target hosts and reports any anomalous or suspicious activities as they occur to a security officer.

The NIDES prototype has two external interfaces:

1. Audit and accounting files on target hosts that are created by resident C2 auditing and Unix accounting daemons (system processes) under a predetermined directory on the file system on the target host.

2. An X/Motif-based security officer graphical user-interface display that allows the security officer to observe anomalous and suspicious activities and to manage the operation of the NIDES prototype. Anomalous and suspicious activities can also be reported using electronic mail.

2.1.1 Prototype Architecture

The underlying software design approach of the prototype is based on the spiral life-cycle method. In this method, a prototype is viewed as developing building blocks to be used by subsequent prototypes.

At the highest level of abstraction, the prototype is a dependency graph[1] of core components. Each core component depends on a set of infrastructural components. Each core component has well-defined interfaces, namely functions that hide the specific details of the X component implementation from other core components that depend on it. Similarly, each infrastructural component has well-defined interfaces that decouples the components use from its internals.

2.1.1.1 Core Components The core components of the NIDES prototype are as follows.

1. Audit-data generation component

2. Audit-data collection component

3. Statistical component

4. Rulebased component

5. Resolver component

[1] A directed edge in a dependency graph, from component A to component B means that component B depends on component A.

6. Security officer user interface component

The audit-data generation component generates NIDES-format audit records of activities of subjects (users) on a target system from C2 auditing and Unix accounting files. It is capable of being remotely started, stopped, and monitored.

The audit-data collection component is capable of gathering audit data generated by multiple target hosts as it is generated, provided the amount of audit data being generated is reasonable. This component guarantees that an audit record will be disposed only after it has been processed by the analysis components (statistical, rulebased, and resolver).

The statistical component detects masquerading users.

The rulebased component detects well-known types of intrusive or suspicious user behavior.

The resolver component analyzes the alerts issued by the statistical and rulebased components and reports only non-redundant alerts.

The security officer user interface component enables the following.

1. Real-time operation of NIDES, including displaying and reporting of alerts, selecting target hosts to be monitored, and reporting status of monitored target hosts.

2. Processing of previously recorded audit data using NIDES, including logging of alerts and managing of persistent store information used by NIDES.

The dependency graph of the core components is shown in Figure 1.

The security officer user interface component depends on the resolver component for obtaining alerts, on the audit-data collection component for obtaining the status of audit-data generation on various target systems, and on the audit-data generation component itself for its initiation and termination. The resolver component depends on the statistical and rulebased components for their respective analyses which, in turn, depend on the audit data collection component for audit-data records. The audit-data collection component obtains audit data from the various audit-data generation components.

2.1.1.2 Infrastructural Components The infrastructural components for realizing each core component of the NIDES prototype are as follows.

1. Process management component

2. Inter-process communication component

3. Persistent store component

4. Graphical user-interface component

The process management component manages the execution and interaction of core components. Each core component is encapsulated in a process. Interaction between core components encapsulated in processes is realized using inter-process communication.

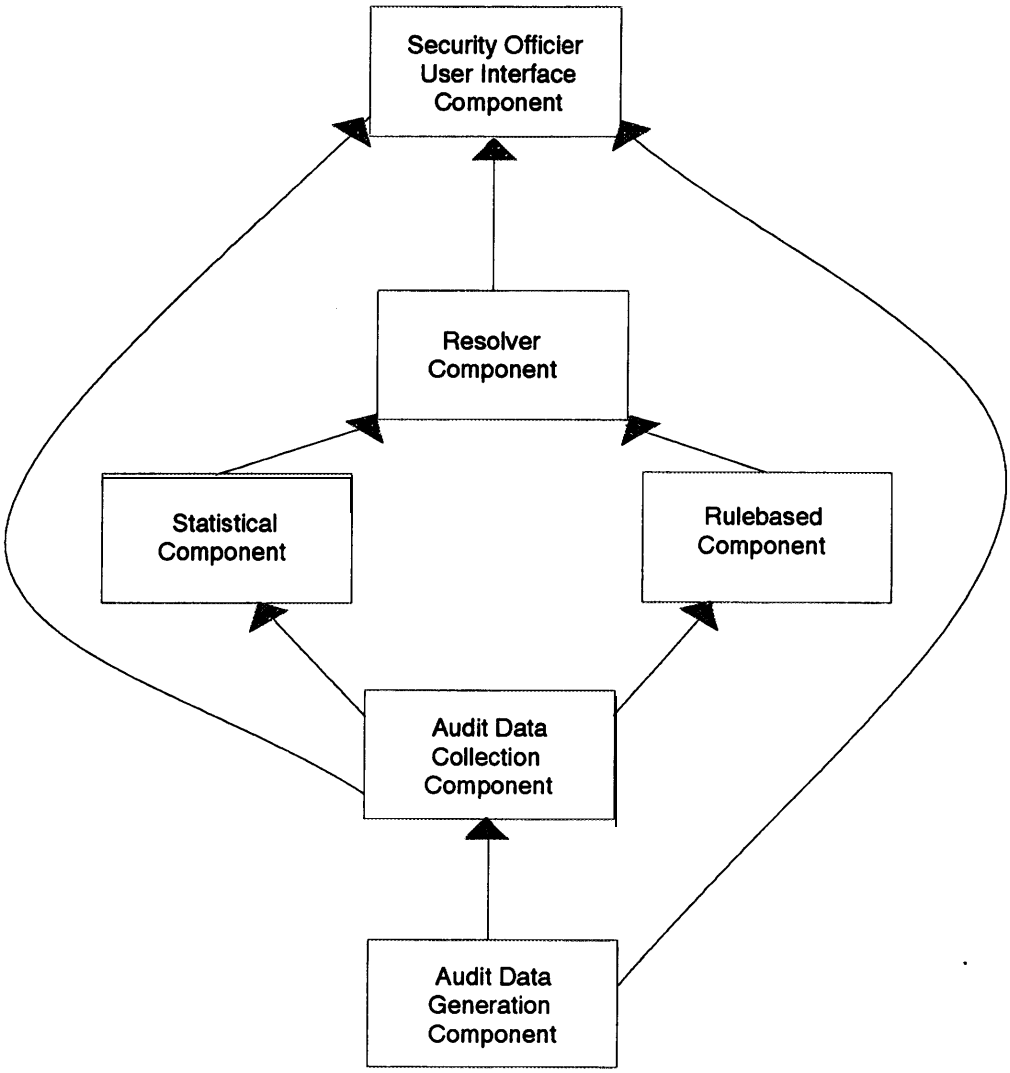

Figure 1: Core Component Dependencies

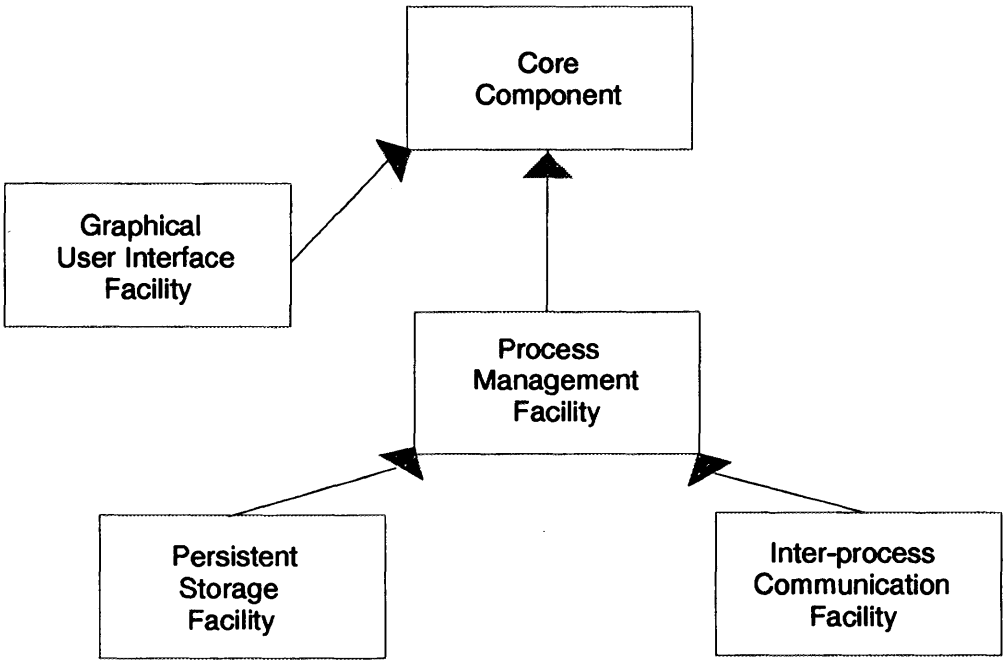

Figure 2: Infrastructural Component Dependencies of a Core Component

The model of process management used is the client/server model of distributed computing. Each process can be either a client or a server but not both. A client process is active it initiates interaction with a server process. A server process is reactive it responds to interaction initiated by one or more client processes.

Inter-process communication is implemented using the remote procedure call (RPC) mechanism. RPC uses an external data representation (XDR) of messages between processes to accommodate heterogeneity in the underlying hardware. Server processes export procedures that can be remotely invoked by clients. Both synchronous and asynchronous RPC are provided to be used as appropriate. Synchronous RPC means that the remote procedure call will be atomically executed, whereas asynchronous RPC means that the act of remote procedure invocation and the act of return from the procedure can be interleaved by other activities, including other remote procedure calls. This is strictly under server control and is transparent to clients.

The persistent store component provides a storage-independent way of storing and retrieving any internal data structure that is pertinent to the various core components. For the NIDES prototype, the persistent store facility is implemented using Suns Network File System (NFS).

The graphical user-interface facility, based on X/Motif, provides a location-independent window-based interface.

The dependency graph of the infrastructural components is shown in Figure 2. Each core component depends on the process management facility and, when pertinent, on the

graphical user-interface facility. The process management facility depends on the interprocess communication facility and the persistent store facility.

2.2 Prototype Design Description

In this subsection, we describe how the various components are integrated using the various infrastructural components.

The NIDES prototype is a collection of servers and clients, as shown in Figure 3. There are three servers: `SOUI` server, `Analysis` server, and `Arpool`. These are described below.

2.2.1 *SOUI* Server

The *SOUI* server is designed to be a server for both RPCs (issued by its clients) and X events (issued by the X display). It is important that all servers, and especially the *SOUI* server, not be indefinitely blocked. The strict client/server model allows blocking to be avoided. The *SOUI* server is supported by seven clients associated with it, which we generically refer to as *agents.* These are described below.

- Agent *agent_status* is responsible for obtaining the status of target systems being audited from *arpool.* The latest status is reported using the `put_status_of_targets` RPC to the *SOUI* server after being obtained from *arpool.*

- Agent *agent_alerts* is responsible for obtaining one or more alerts from the *analysis* server and providing them to the *SOUI* server for display and other purposes using the `put_alerts()` RPC.

- Agent *agent_server* is responsible for initiating and terminating the other servers, namely, the *arpool* and *analysis* servers. It does this by issuing the `get_control_server()` RPC to the *SOUI* server. Depending on the kind of request that is returned, the appropriate initiation or termination is accomplished. Errors in accomplishing the request are conveyed to *SOUI* server using the `put_server_error` RPC.

- Agent *agent_target* is responsible for initiating and terminating audit generation on target hosts. It does this by issuing the `get_control_target()` RPC to the *SOUI* server. Errors in accomplishing the request are conveyed to the *SOUI* server using the `put_target_error()` RPC. Audit generation activity on a target system is initiated and terminated by interacting with a target-resident daemon using RPC.

- Agent *agent_save* is responsible for saving audit data (as obtained from the *arpool* server through an RPC interface) using the `get_control_ar_storage()` RPC to start and stop saving audit data and using `ar_storage_error()` RPC to report errors.

- Agent *agent_email* is responsible for issuing email alerts, which it does by continually issuing the `email_alert()` RPC to the *SOUI* server.

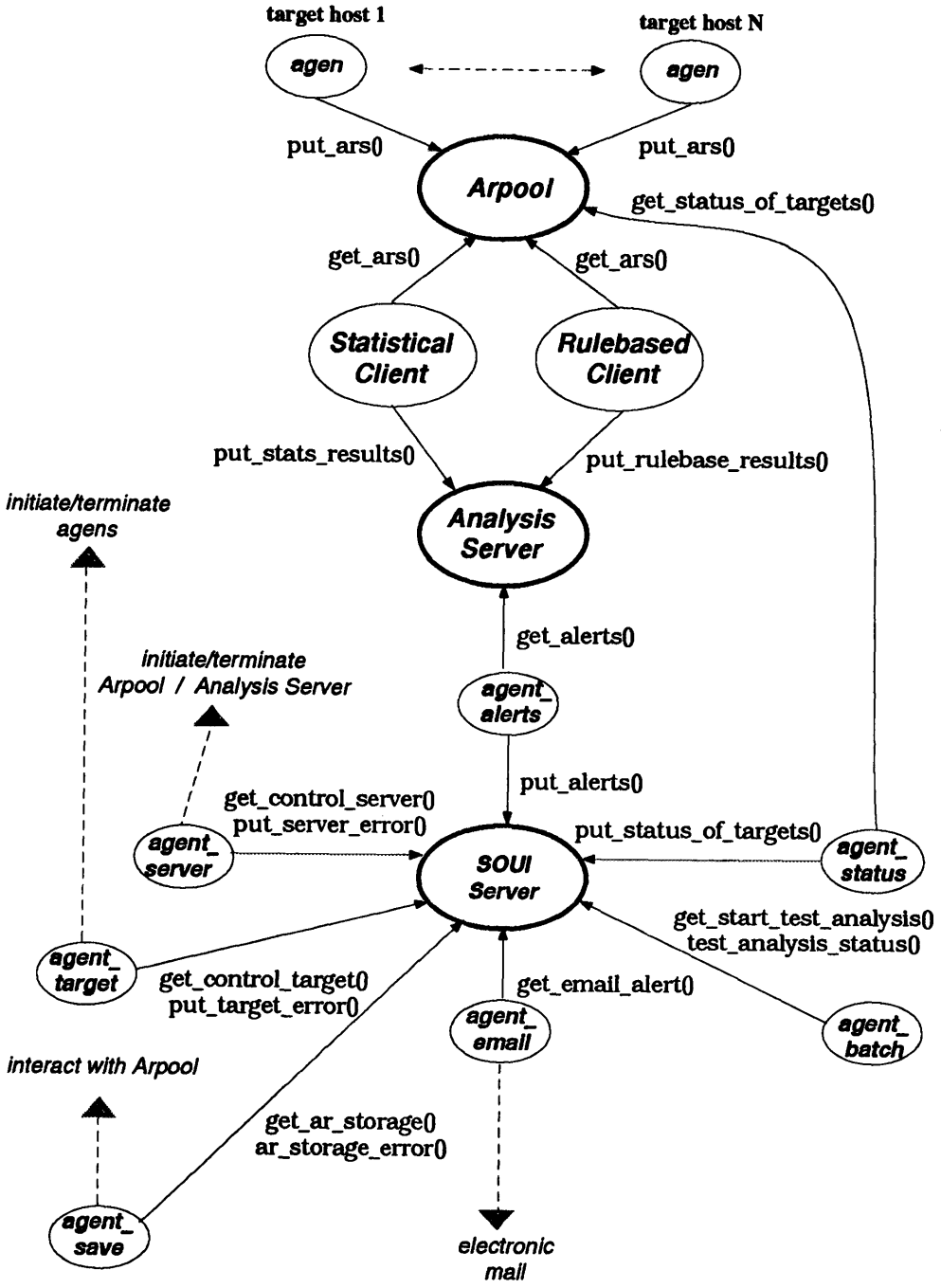

Figure 3: Client/Server Graph

- Agent *agent_batch* is responsible for applying the statistical and *rulebased* components to a file of audit data and recording the alerts in a logfile using the `get_start_test_analysis()` RPC. Status of the offline analysis is conveyed using the `test_analysis_status()` RPC.

2.2.2 *Analysis* Server

The *Analysis* server provides the `get_alerts()` RPC so that an agent can obtain alerts and provide them to the *SOUI* server. The *statistical* client issues `put_stats_results()` RPC to return the results of statistical analysis of one or more audit records. Similarly, the *rulebased* client issues the `put_rulebase_results()` RPC to return the results of rulebased analysis of possibly several audit records. Both *statistical* and *rulebased* clients obtain audit records using the `get_ars()` RPC from the *arpool* server. Also, *statistical* and *rulebased* clients make use of the persistent store facility to store statistical and rulebased information (see Section 2.1.1.2).

2.2.3 *Arpool* Server

The *arpool* server provides the `put_ars()` RPC for audit data generating clients to deposit audit records. It also provides RPC `get_ars()` to allow analysis clients to retrieve audit records. Note that each audit record is retained until all currently active clients of *arpool* have retrieved the record. *Arpool* also provides the status of target hosts using the `get_status_of_targets()` RPC invoked by a security-officer user-interface agent.

3 Detailed Design

The detailed design of the NIDES prototype is described by first considering the infrastructural components and then discussing each core component and its implementation using the client /server model.

3.1 Infrastructural Components

3.1.1 Process Management Component

The process management component is realized using the client/server model. A description of this model can be found in Suns Networking Programming Guide [6].

3.1.2 Inter-Process Communication Component

The Interprocess communication is based primarily on Remote Procedure Calls, similar to those defined by Sun with their `rpcgen`[2] tool. Our RPC tool `arpcgen` supports many more features than Suns tool, and accepts ANSI C, rather than Suns RPC Language.

The `arpcgen` tool takes as input ANSI C data type declarations and function prototypes (usually a `".h"` header file) and generates three C files: client-side RPC stubs, server-side RPC stubs, and XDR routines.

The generated code makes use of functions in the `arpc` library, which can be similarly grouped into client side, server side, and XDR routines.

The client-side routines are as follows:

`Status ipc_clnt_open(IPC *ipc, const char *name)`
: This routine opens a connection to server `name`. The resulting connection specifier is written into `*ipc`.

`Status ipc_clnt_close(IPC ipc)`
: This routine closes a connection opened with `ipc_clnt_open()`.

`IPC ipc_set_server(IPC server)`
: This routine specifies the server to be used in subsequent RPC calls. It returns the old server handle.

`Status ipc_onfail(IPC who, void (*what)(IPC, void *), void *arg)`
: This routine specifies an error handling function for the connection `who`. If an error occurs while talking to `who`, the function `*what` is called with `who` and `arg` as arguments.

The server-side routines are as follows:

[2] See the Sun Network Programming Guide [6], pp33-146

`void ipc_svc_dispatch(IPC from)`

This routine is produced by `arpcgen` in the server side stubs; it decodes an incoming RPC and calls the corresponding server function. Typically `ipc_src_dispatch` is passed to `ipc_svc_init`.

`Status ipc_svc_init(const char *name, void (*svc)(IPC))`

This routine initializes a server, registering it as `name` with the name server and using `svc` as the service dispatch routine (usually `ipc_svc_dispatch`).

`int ipc_svc_run(int poll)`

This routine handles incoming RPCs, using the previously specified dispatch routine (argument to `ipc_svc_init`). `If poll` is false (zero), it runs forever waiting for incoming RPCs calls and never returns. `If poll` is true (non-zero), it services at most one pending RPC from each client and returns without blocking. In this case, it returns `true` if there are more pending RPCs, or `false` if not.

`void ipc_svc_close(IPC who)`

This routine closes a connection. The client in question will see a failure on its IPC handle.

`void ipc_svc_shutdown()`

This routine closes all connections to all clients and shuts down the server so as to no longer accept any incoming RPCs. This should be called prior to exiting in order to do a clean shutdown of the RPC service.

`Status ipc_onfail(IPC who, void (*what)(IPC, void *), void *arg)`

This routine specifies an error handling function for the handle `who`. If an error occurs while talking to `who`, the function `what` is called with `who` and `arg` as arguments.

`typedef int (*authproc_t)(IPC)`

`authproc_t ipc_set_auth(authproc_t auth_fn)`

This routine sets up an authorization test for incoming RPCs. `Auth_fn` is called for each new client that requests a connection. The client is rejected if `auth_fn` returns false. `Ipc_set_auth` returns the old authorization function.

`void XrpcInit()`

`void XrpcAppInit(XtAppContext ctxt)`

These special functions allow an RPC server to coexist with an X toolkit application. After calls to `ipc_svc_init` and `XrpcInit`, a call to `XtMainLoop` causes the program to service incoming RPCs as well as Xt events. The latter form should be used if a non-default XtAppContext is used.

XDR routines

int rxdr_int(IPC ipc, void *p, void *ignore)

int rxdr_u_int(IPC ipc, void *p, void *ignore)

int rxdr_char(IPC ipc, void *p, void *ignore)

int rxdr_u_char(IPC ipc, void *p, void *ignore)

int rxdr_short(IPC ipc, void *p, void *ignore)

int rxdr_u_short(IPC ipc, void *p, void *ignore)

int rxdr_long(IPC ipc, void *p, void *ignore)

int rxdr_u_long(IPC ipc, void *p, void *ignore)

int rxdr_float(IPC ipc, void *p, void *ignore)

int rxdr_double(IPC ipc, void *p, void *ignore)

The above are the basic low level routines for translating from XDR format to machine format. Each of these takes an `ipc` channel, a pointer to the place to put the read object, and an extra pointer which is ignored.

int wxdr_int(IPC ipc, void *p, void *ignore)

int wxdr_u_int(IPC ipc, void *p, void *ignore)

int wxdr_char(IPC ipc, void *p, void *ignore)

int wxdr_u_char(IPC ipc, void *p, void *ignore)

int wxdr_short(IPC ipc, void *p, void *ignore)

int wxdr_u_short(IPC ipc, void *p, void *ignore)

int wxdr_long(IPC ipc, void *p, void *ignore)

int wxdr_u_long(IPC ipc, void *p, void *ignore)

int wxdr_float(IPC ipc, void *p, void *ignore)

int wxdr_double(IPC ipc, void *p, void *ignore)

The above are the low level routines for translating from machine format to XDR format. Each of these takes an `ipc` channel, a pointer to the machine object, and an extra pointer which is ignored.

The XDR routines are defined in this manner so that every XDR routine has the same signature. In this way, pointers to XDR routines can be passed around in a transparent manner regardless of what type of object the XDR routines handles, or even whether it is a read or a write routine.

int rxdr_opaque(IPC ipc, void *p, void *cnt)

int wxdr_opaque(IPC ipc, void *p, void *cnt)
These routines read and write `cnt` bytes of data pointed at by `p`.

int rxdr_void(IPC ipc, void *p, void *ignore)

int wxdr_void(IPC ipc, void *p, void *ignore)
These translate a type `void` object.

int rxdr_string(IPC ipc, void *p, void *ignore)

int wxdr_string(IPC ipc, void *p, void *ignore)

Translate a `char *` pointed at by `p` which is either a NULL pointer, or a pointer to a NULL terminated string of characters (a C string).

```
struct xdr_vector_info {
        u_int   size                            /* number of elements */
        u_int   elsize                          /* size of each element (bytes) */
        int     (*proc)(IPC, void *, void *)    /* XDR routine for elements */
        void    *extra                          /* third arg to proc */
        }
```

int rxdr_vector(IPC ipc, void *p, void *info)

int wxdr_vector(IPC ipc, void *p, void *info)

The above are routines to translate an array of objects of an arbitrary data type. The third argument points to a `struct xdr_vector_info,` which specifies the size of the array and information about the elements.

void *xdr_vector_info(struct xdr_vector_info *p, u_int size, u_int elsize, int (*proc)(IPC, void *, void*), void *extra)

This routine fills in a `struct xdr_vector_info.`

```
struct xdr_pointer_info {
        u_int size                              /* size of pointed to object */
        int     (*proc)(IPC, void *, void *)    /* XDR routine for pointed to object */
        void    *extra                          /* third arg to proc */
        }
```

int rxdr_pointer(IPC ipc, void *p, void *info)

int wxdr_pointer(IPC ipc, void *p, void *info)

void *xdr_pointer_info(struct xdr_pointer_info *p, u_int size, int (*proc)(IPC, void *, void*), void *extra)

Analogous to the `xdr_vector` routines above, these routines deal with pointers to a single object of some type.

The XDR routines generated by `arpcgen` are organized as routines named `rxdr_`*`type`* and `wxdr_`*`type`* for each data type defined in the input. The client stubs and server stubs make use of these XDR routines to copy arguments and results back and forth. In general, the programmer need not know about the details of the XDR routines and calling them, since this is handled by automatically generated code.

The `ipc_clnt_open` and `ipc_svc_init` both translate names to physical host and TCP port addresses by talking to the `ipc_nameserver`, which must be running on a well-known host and port. The mechanism used to specify the location of the nameserver is the environment variable `IPC_NAMESERVER`, which should be in the form *hostname:*`portnum`.

It is impossible to have more than one server in the same process. That is, the server-side stubs generated by two runs of `arpcgen` cannot be linked together to form a server that deals with the RPCs of both servers. A client may multiplex between multiple servers, but it is up to the programmer to make sure not to make an RPC to a server that doesnt support it.

3.1.3 Persistent Storage Component

The persistent store translates data objects to and from a machine-independent byte-stream format using the IPC component described earlier (see page 9). The byte-stream form of the data is stored in an NFS file system to permit transparent network access in the alpha implementation.

The persistent storage allows an arbitrarily hierarchical naming scheme, which is used to make independent *instances* in the alpha version of NIDES. Names are specified as *instance*/*name* to the persistent storage. An additional set of calls allows manipulation of entire instances.

Most of the data types used by the persistent store are defined in the other components. The persistent store has one data type, used to specify or return a list of names.

```
struct Name_list {
    struct Name_list   *next;
    string             name;
   };
typedef struct Name_list *Name_list;
```

The functional interfaces to the persistent store component are as follows.

1. `Status get_list_of_instance_names( Name_list **ilist )`
   This function reads the list of currently available NIDES analysis instances into `ilist`.

2. `Status copy_instance( string to_instance, string from_instance )`
   This function copies NIDES analysis instance `from_instance` into `to_instance`.

3. `Status create_instance( string instance )`
   This function creates NIDES analysis `instance` instance initialized appropriately.

4. `Status delete_instance( string instance )`
   This function deletes NIDES analysis instance `instance`.

5. `Status get_list_of_subjects(const string instance, Name_list **list)`
   This function reads the list of subjects who have profiles in `instance`.

6. `Status read_current_profile( const string profile_name, Curr_prof_struct *curr_profile )`
   This function reads current profile from persistent store object named `profile_name` into `curr_profile`.

7. `Status write_current_profile( const string profile_name, const Curr_prof_struct *curr_profile )`
   This function writes current profile `curr_profile` into persistent store object named `profile_name`.

8. `Status read_historical_profile( const string profile_name, Hist_prof_struct *hist_profile )`
   This function reads historical profile from persistent store object named `profile_name` into `hist_profile`.

9. `Status write_historical_profile( const string profile_name, const Hist_prof_struct *hist_profile )`
   This function writes historical profile `hist_profile` into persistent store object named `profile_name`.

10. `Status read_stats_config( const string stats_config_name, Statconfig_struct *config )`
    This function reads statistics configuration from persistent store object named `stats_config_name` into `config`.

11. `Status write_stats_config( const string stats_config_name, const Statconfig_struct *config )`
    This function writes statistics configuration `config` into persistent store object named `stats_config_name`.

12. `Status read_kb( const string kb_name, Rule_list *kb )`
    This function reads knowledge base in persistent store object called `kb_name` into `kb`.

13. `x Status write_kb( const string kb_name, const Rule_list *kb )`
    This function writes knowledge base `kb` into persistent object named `kb_name`.

3.1.4 Graphical User-Interface Component

Motif is an object oriented set of X-compatible programming tools which support creation of a large set of user interface objects called *widgets.* A widget can be a text display area, a menu button, a label, or any object that is displayed to the user and possibly manipulated. Another class of widgets, called *managers,* control the organization of the display and the arrangements and actions of the widgets that comprise the user interface. The Motif software libraries provide numerous functions to create and manipulate widgets. In addition, many Motif functions provide the capability to create high-level objects that are comprised of many widgets with a single function. The Motif model is based upon *X-windows,* and also utilizes the `Xt` libraries. The NIDES security-officer user-interface service is built using both the Motif libraries, and the `Xt` libraries.

For more information about the Motif library, refer to the Open Software Foundations publications on Motif [2].

3.2 Audit-Data Generation Component

The audit-data generation component consists of two modules: audit record gathering and audit record conversion. Both of these modules are encapsulated in the *agen* utility which runs as a client of *arpool* on each target host. These modules and utilities are described below.

3.2.1 Audit Data Gathering

The audit data gathering module is a set of functions which read data from system log files and return these data in the NIDES audit record format. The functions are instantiated for each distinct type of audit data.

The following defines the interface of each of these functions.

- `open( void )`

  This function opens the default system log file for a certain type of audit data. A value of -1 is returned on error, and 0 on success.

- `seek_eof( void )`

  This function seeks to the end of the current log file. This function should be called after `open` to discard or skip over stale audit data. A value of -1 is returned on error, and 0 on success.

- `get( ia_node **rp )`

  This function reads the next audit record and converts it to the NIDES audit record format (using the appropriate conversion function – see Section 3.2.2). It is possible for multiple NIDES audit records to have been generated, so this function returns a list of NIDES-formatted audit records. A value of -1 is returned on error, otherwise the length of the list is returned. A value of 0 indicates an empty list. The list is returned in the first result-parameter.

  Note that this routine implements a *polling* model of checking for the availability of new audit data. Also, this function must be able to deal with logging facilities that span multiple files. For example, UNIX accounting can arbitrarily stop updating one file and begin writing into a new file. To deal with this, the module must occasionally poll for the existence of a new accounting file.

3.2.2 Audit-Data Conversion

Currently, Unix C2 and Unix accounting audit data are supported. The conversion functions are described below.

- int c2_to_ialist(const audit_record_t *au, ia_node **rp)
  This routine converts an UNIX C2 audit record into a list of NIDES audit records, returned in the second parameter. This function returns 0 on success and -1 on error.

- int pacct_to_ia(const pacct_rec *, ia_audit_rec *)
  This routine converts a UNIX accounting record into a NIDES audit record, returned in the second parameter. This function returns 0 on success and -1 on error.

3.2.3 Data Structures

The data structures used by the functions defined above are described next. First, we describe the Unix C2 audit record; then, we describe the Unix accounting record; and finally we describe a structure for maintaining list of NIDES-formatted audit records.

```
/* from /usr/include/sys/audit.h */

struct audit_record {
short au_record_size;     /* size of audit record */
short au_record_type;     /* its type */
unsigned int au_event;    /* the event */
time_t au_time;           /* the time */
uid_t au_uid;             /* real uid */
uid_t au_auid;            /* audit uid */
uid_t au_euid;            /* effective uid */
gid_t au_gid;             /* real group id */
```

```
short au_pid;             /* process id */
int  au_errno;            /* error code */
int au_return;            /* a return value */
blabel_t  au_label;       /* audit label */
short  au_param_count;    /* # of parameters */
};
typedef  struct  audit_record  audit_record_t;

/* from /usr/include/sys/acct.h  */

typedef  struct {
        char             ac_flag;         /* Accounting flag */
        char             ac_stat;         /* Exit status */
        unsigned short   ac_uid;          /* Accounting user ID */
        unsigned short   ac_gid;          /* Accounting group ID */
        short            ac_tty;          /* control typewriter */
        long             ac_btime;        /* (time_t) Beginning time */
        unsigned short   ac_utime;        /* (comp_t) Accounting user time */
        unsigned short   ac_stime;        /* (comp_t) Accounting system time */
        unsigned short   ac_etime;        /* (comp_t) Accounting elapsed time */
        unsigned short   ac_mem;          /* (comp_t) average memory usage */
        unsigned short   ac_io;           /* (comp_t) number of chars transferred */
        unsigned short   ac_rw;           /* (comp_t) number of blocks read or written */
        char             ac_comm[8];      /* Accounting command name */
}  pacct_rec;


typedef  struct  ia_node  ia_node;
struct  ia_node {
        ia_node          *next;
        ia_audit_rec     *ia;             /* IDES-formatted audit record
                                             see appendix for details */
} ;
```

3.3 Audit-Data Collection Component

The audit data collection component is designed to be a building block for a server capable of scheduling or managing multiple RPC requests. The audit record collection component is responsible for managing the flow of audit records such that the analysis components of NIDES see a consistent view of all audit records. Furthermore, audit records from many target hosts are multiplexed into a single stream of audit records.

All analysis components of NIDES must see a consistent view of the audit data. In order to achieve this, audit records from multiple target hosts must be centrally collected and assigned unique identifiers. Each analysis component of NIDES can then request audit data from this centralized location. Since all analysis components fetch audit data from the same server, it is easy to ensure that all analysis components receive the same audit data.

The central repository of audit records in the audit data collection component is referred to as the *pool* of audit records. The pool is simply a first-in first-out queue of audit records with multiple producers and multiple consumers. Producers are entities which add audit records into the pool, while consumers fetch these audit records. The audit data collection component keeps track of the number of consumers in order to correctly determine when an audit record can be discarded; that is, an audit record is discarded only when every consumer has received it.

In order to bound the memory requirements of the audit record pool, the audit data collection component enforces a flow-control mechanism using high-water and low-water marks. If the number of audit records in the pool exceeds a predetermined high-water mark, then a no more flag is returned to the producer that the producer is expected to honor by not adding any more audit records until further notice. When audit records are consumed and the number of audit records in the pool falls below the low-water mark, then new audit records are once again accepted and stored in the pool. Producers may install a call-back function to be notified of this condition. It is expected that a low-water mark of 256 audit records and a high-water mark of 768 will function adequately, although these parameters can be changed.

Audit records in the pool are managed using a reference count scheme. An audit record is kept in the pool until every client has requested that record. Thus, it is possible for consumers to request audit records at different rates. Again, to bound the memory requirements of the pool, the flow-control mechanism prevents one consumer from getting too far ahead of other consumers. In other words, since all but the fastest consumer have not consumed the audit records in the pool, the records must be kept in the pool until the slower consumers have received the records. Since there is a bound on the number of audit records in the pool, the faster consumer will reach a point at which no new audit records are available in the pool and the faster consumers are blocked by the flow-control scheme.

If there are no consumers, then audit records are accepted until the high-water mark is reached. When the high-water mark is reached, further audit records are accepted, but a no more flag is returned. In addition, a call-back function may be specified to be called when more audit records can be accepted. The main intent is to block the producers from generating and transmitting audit data when there is no room in the pool.

When the first consumer attaches to the pool, the pool is flushed such that this consumer can fetch only audit records records produced after the consumer attached.

When additional consumers attach, each will fetch audit records beginning with the oldest audit record stored in the pool at that time.

When a particular audit record has been read by all consumers, it is deleted. If this deletion causes the pool size to drop below the low-water mark, the call-back function for

each producer is called to signal that they may resume the generation and transmission of audit data.

When consumers request audit records, they receive audit records in the order that they were received by the audit data collection component.

The functions for the audit data collection component are as follows:

- `int put_records(int nrec, ia_audit_rec **iav)`
  This function appends an array of NIDES audit records to the pool. The memory allocated for the audit records is inherited by this function; that is, the caller must not reference these audit records when this function returns. However, the `iav` vector should be freed by the caller. This function returns TRUE if the pool is full, else FALSE.

- `int get_records(client_info *c, ia_audit_rec **avec, long *count)`
  This function returns a copy of the next `*count` audit records. The value of `*count` should be greater than 0. The value of `*count` is modified to reflect the number of audit records actually returned if this function returns TRUE. If the returned records are not referenced by any other consumers, then they are deleted from the pool. `Avec` is a result parameter for returning audit records. The caller is responsible for freeing the audit records and the vector `avec`. The parameter `c` is used to identify the consumer making this request.

  This function returns FALSE if no audit records were available, and TRUE if one or more audit records were returned.

- `client_info *get_consumer_slot(IPC clnt_handle)`
  This function creates and returns a consumer context to be used by the `get_record` function. `Clnt_handle` is an opaque data type that must uniquely tag each consumer. Once a context is created, audit records are kept in the `pool` for this context until they are fetched by this context.

- `void zap_consumer_slot(client_info *c)`
  This function destroys the specified consumer context and performs any necessary garbage collection in the pool.

  Data structure `ia_audit_rec` is defined in the appendix. Data structures `client_info` and `clnt_handle` have not been described because their description is quite detailed.

3.4 Statistical Component

This section is organized as follows. We first summarize the statistical algorithms and relevant data elements. Then we list the major data structures that map to the statistical data elements referenced in the algorithm. Finally, we provide details on how the interface functions are implemented, and, where appropriate, indicate how the functions are mapped to the algorithms.

3.4.1 Algorithm Summary

The basic statistical approach in NIDES is to compare a users short-term behavior to the users historical or long-term behavior. In comparing short-term behavior to long-term behavior, the statistical component is concerned with both long-term behaviors that do not appear in short-term behavior as well as short-term behaviors that are not typical of long-term behavior. Whenever short-term behavior is sufficiently unlike long-term behavior, a warning flag is raised. This statistical approach requires no *a priori* knowledge about what type of behavior would result in compromised security.

The number of audit records or number of days that constitute short-term and long-term behavior can be set through the specification of a half life. For example, if the security officer wants short-term behavior to reflect on the order of 200 audit records, a half life of approximately 100 audit records should be specified. This will assure that the 200th audit record has only one-quarter the influence of the most recent audit record, the 400th audit record has only one-sixteenth of the influence, etc. Similarly, a reasonable half life for a long-term profile might be 30 days.

The following paragraphs in this section discuss some of the specifics of the algorithms implemented in the statistical component. For a more rigorous description of the algorithms, the reader should read our earlier reports [4, 5].

Aspects of subject behavior are represented as measures (for example, file access, CPU usage, hour of use). We refer to a subjects *profile* as the set of measure values associated with short-term and long-term behavior. We have classified the NIDES measures into four groups: activity intensity, audit record distribution, categorical and ordinal. These different classifications serve different purposes. The activity intensity measures determine whether the volume of activity generated is normal. The audit record distribution measure determines whether for recently-observed activity (say, the last few hundred audit records generated), the types of actions being generated are normal. The categorical and ordinal measures determine whether within a type of activity (say, accessing a file), the behavior over the recent past that affects that action is normal.

We use a vector called `Q` that quantifies each measure, and this quantification is recorded into a frequency distribution. By observing the values of `Q` over many audit records, and by selecting appropriate intervals for categorizing `Q` values, we build a historical distribution for `Q`. 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. Generally speaking, small values of `Q` are indicative of a recent past that is similar to historical behavior, while large values of `Q` represent dissimilar behavior.

We use another vector called `S`, that is a transformation of `Q` such that `S` is small whenever `Q` is small, and large whenever `Q` is large; this transformation can be viewed as a rescaling of the magnitude of `Q`. The transformation of `Q` to `S` requires knowledge of the historical distribution of `Q`. It is actually a simple mapping of the percentiles of the distribution of `Q` onto the percentiles of a half-normal distribution (which we call `TPROB`).

Finally, we have the T^2 statistic, which is a summary judgment of the abnormality of many measures, and is, in fact, the sum of the squares of the individual measures in `S`. For each audit record generated by a subject, the single test statistic value T^2 is computed that summarizes the degree of abnormality of the subjects behavior in the near past. 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). We keep an historical distribution of the T^2 statistic, and we use this distribution as a basis for determining whether or not a score value is anomalous enough to warrant an alert. We have currently selected as a default the 99.9th percentile of the historical distribution of T^2 score values as the default critical level of concern for the security officer (this percentile value may be changed at any time).

It is important to realize that the complex process just described is based upon many days of audit data processing. Like any other process that relies on probability curves, the more data contributing to the distribution, the more stable and accurate the information becomes. The training of these distribution tables is a key factor in the effectiveness of the statistical component, and although we have not provided a formal (algorithmic) explanation of how such training is accomplished in this document, the training concepts are implemented in the component. Some of these will be described later in the section in the context of functional interfaces.

3.4.2 Data Structures

The following constants are used throughout the statistical component data structures and functions.

```
#define MAXMEASURES 46           /* number of measures */
#define MAXMEASURES7 MAXMEASURES*7 /* 7x number of measures */
#define NUMBINS        32        /* number of bins for Q */
#define  HFSQMAXMEASURES  ((MAXMEASURES*MAXMEASURES+MAXMEASURES)/2)
                                 /* for one-half of a symmetrical matrix */
#define MAXSUMRAREPROB 0.01
                                 /* maximum sum of rare category probabilities */
#define  CATRAREPROB    0.01     /* "rare" probability value */
#define MINPROB        1.0/4096.0
                                 /* minimum probability for categories */
```

The statistical component utilizes three types of data structures:

- profiles
- activity
- configuration

We describe each of these structure types in the remainder of this section.

3.4.2.1 Profiles There are three main data structures that comprise a subjects profile. Each of these is described below.

`Curr_prof_struct` represents the short-term profile, and is generally updated on a per-audit-record basis. Each subject must have its own current profile containing the following information:

- `subjid`. This is an integer value that uniquely identifies the subject of this profile.
- `subjname`. This is the character string representation (also unique) that identifies the subject.
- `prevtstamp`. This is the long integer value that contains the timestamp of the last audit record processed for this subject. This number represents the number of seconds elapsed since January 1, 1970 (ref: `ctime()` in any Unix manual).
- `thresh_red`. This value is the percentile level at which we consider a score threshold to be critical. For example, if `thresh_red` value is 0.1, then any score value exceeding this percentile for this subject will result in an alert status.
- `score_red`. This value is the score that corresponds to the `thresh_red` percentage value.
- `thresh_yellow`. This value is the percentile level at which we consider a score threshold to be in a warning level. For example, if `thresh_yellow` value is 1.0, then any score value exceeding this percentile for this subject would be in a warning status.
- `score_yellow`. This value is the score that corresponds to the `thresh_yellow` percentile value.
- `actvd_measures`. This field is an array of integers that represent the active measures for this subject. Each element in the array represents one measure, and it is set to 1 if active, 0 otherwise.
- `q`. This field is an array of type doubles of size `MAXMEASURES`. It represents the `Q` values for a profile. The values for this array are recomputed for each measure observed in an audit record.
- `qcount`. This field is a (`MAXMEASURES` by `NUMBINS`) matrix of integers that keeps track of the daily count of `Q` values falling into the appropriate bins. This matrix is reset to 0 after each profile update.
- `s`. This field is an array of type double of size `MAXMEASURES` that represents the `S` values for a profile. This array is reset to 0 after each profile update. This array is recomputed for every measure observed in each audit record.
- `dailycnt`. This field is an array of integers of size `MAXMEASURES` representing the number of times each measure was observed during the day. It is reset to 0 at profile update time.

- `cats`. This field is an array of pointers to a list of categories for each measure. Each list is sorted in ascending order of category probability. See description of `Catnode` for more detail (page 24).
- `scorehistn`. This field contains the value of the aged count of scores that have been produced for this subject.
- `nextcatid`. This field is an array of size of `MAXMEASURES`. It keeps track of the next category ID number available for assignment for a particular measure (each category has a unique ID). It is incremented after a new category for the given measure has been assigned a value.
- `t2cnt`. This field is an array of size of `MAXMEASURES7`. It keeps track of the daily score counts, and is reset to 0 at profile update time.
- `dailysum`, `dailysum` and `utilvec1`. These fields are unused in the current implementation of the statistical component. However, they remain in this structure for future incorporation of additional profile data.
- `hashed_cats`. This field is a hashed table for all the categories defined for this subject. This table is primarily used for quick access. It can be considered volatile data (i.e., it does not need to be stored in permanent storage).

`Hist_prof_struct` represents the long-term profile, and is generally updated once a day. As with the current profile structure, each subject must have its own historical profile; hence, for each `Curr_prof_struct`, there should always be a corresponding `Hist_prof_struct`. The historical profile structure contains the following information.

- `subjid`. This is an integer value that uniquely identifies the subject of this profile.
- `subjname`. This is the character string representation (also unique) that identifies the subject.
- `lastupdate`. This is the long integer value that represents the time of the last historical profile update for this subject. This number represents the number of seconds elapsed since January 1, 1970 (ref: `ctime()` in any Unix manual).
- `nupdates`. This is the integer value that represents how many updates this profile has gone through. It is incremented by one each time this profile is updated.
- `qp`. This field is a (`MAXMEASURES` by `NUMBINS`) matrix of type double integer that represents the historical distribution of `Q` values within each bin (interval). This matrix is recomputed at profile update time using the daily counts accumulated in `qcount` from the current profile.
- `tp`. This field is a (`MAXMEASURES` by `NUMBINS`) matrix of type double integer that represents the tail probabilities of the `qp` historical distribution. See Section 3.4.1 for a more detailed explanation. This matrix is recomputed at profile update time using the recomputed `qp` values.

- `rareprob`. This field is an array of size `MAXMEASURES`. It represents the sum of all the categories for a particular measure with rare probabilities (as defined by `CATRAREPROB`). It is recomputed at profile update time, and has a cap value `(MAXSUMRAREPROB)`.
- `maxrareprob`. This field is an array of size `MAXMEASURES`. It represents the maximum category probabilities computed to be less than `CATRAREPROB` for each measure. It is used in conjunction with `rareprob` to keep track of new and/or rare categories observed for each measure. It is recomputed at profile update time, and must never be greater than `CATRAREPROB`.
- `ncats`. This field is an array of size `MAXMEASURES` representing the aged number of categories for each measure. This value is used to smooth the normalization of during score computation. It is recalculated at profile update time, incorporating the most recent count of categories for each measure.
- `histn`. This field is an array of size `MAXMEASURES` representing the historically-aged effective-`N` for each measure (i.e., the aged number of times each measure was observed). It is recomputed and aged at profile update time.
- `t2dist`. This field is an array of size `MAXMEASURES7`. It represents the historical T^2 score distribution, and is used to determine new score threshold values. It is recomputed at profile update time. The first 200 slots represent 0.1 score points, and the remainder of the array slots represent whole score point.
- `utilvec2`. This field is an array of size `MAXMEASURES` representing the historically-aged number of active measures. It is recomputed at profile update time.
- `halfinv, histmean, histcorr, qbin`. These fields are unused in the current implementation of the statistical component. However, they remain in this structure for future incorporation of additional profile data.

`Catnode` represents a category for a particular measure. It contains both current and historical information, but is generally stored as part of the current profile. The fields for this data structure are defined as follows:

- `catid`. This field is the integer code of this category. It is unique within a measure only.
- `cmid`. This field specifies the measure that this `catnode` belongs to. Together with the category id `catid`, these two numbers are unique throughout all the categories for all measures for a subject.
- `catname`. This field contains the character string identification of this category.
- `catprob`. This field contains the historically-aged probability for this category within this measure. It is updated at profile update time.
- `catcount`. This field keeps track of how many times this category was accessed since the last profile update. It is reset to zero at profile update time.

- `agecnt, agecntsq`. These fields represent the aged count and square of the aged count for each category (i.e., how many times the category was observed). Together with the `catcount` field, these two fields make up the short-term profile for the subject for this category. It is updated whenever it is observed in an audit record.
- `catflags`. This field is a vector of bit flags that indicate any peculiarities for the category (such as a first-time seen category).
- `catnext`. This field is a pointer to another `Catnode` data structure. Categories within a measure are represented as linked lists.

3.4.2.2 Activity Data Before the statistical component can compute any scores, audit data must be converted into a representation that can be used for processing. To support this, the following two data structures are used.

`Measure`. Measures are defined in the statistical component configuration file. Except where indicated, all of the fields defined for this data structure may be reconfigured by the security officer.

- `mid`. This field is an integer value that represents the measure id. It is unique and should not be modified.
- `mname`. This field is a mnemonic character string representation of the measure. It should not be modified.
- `mdesc`. This field is used for a more verbose description of the measure.
- `mtype`. This field indicates the type of the measure. The types are continuous (ordinal), categorical, and binary continuous [5]. (Note that intensity measures are actually continuous measures, and the audit record distribution measure is categorical, and thus do not have a different type associated with them.)
- `mflags`. This field indicates whether or not the measure is activated (1 if active, 0 otherwise).
- `mqmax`. This field represents the maximum `Q` value that can be computed for this measure. This value is used to properly scale the intervals of `Q` for an even distribution (i.e., bell-shaped curve).
- `mscalar`. This field represents a scalar value only used for continuous measures, and is used to evenly distribute continuous measure values across 32 bins.
- `mweight`. This field represents a weighting factor for the measure. It is currently unused.

`Activity`. Each audit record is transformed into a series of activity units. Activity units are represented in a vector of size `MAXMEASURES`, thus providing a one-to-one mapping of observation units to measures.

- `mid`. This field identifies the measure to which this activity is maps.

  `m_val`. This field is a structure of different data types. Observation of an activity can be represented in several ways, depending on the type of measure. For a continuous measure, the activity is a `float` (or `double`) value. For a categorical measure, the activity is a character string (name of a file, terminal, host, etc.). For binary continuous measures, the activity is set to 1. If the activity is not observed in that particular audit record, the `m_val` values would be 0, `null`, and 0 respectively for each measure type.

3.4.2.3 Configuration Data The statistical component has a variety of configurable parameters; all of these are stored in the following data structure. There is only one set of configuration data per instance of the statistical component.

`Statconfig_struct`. This data structure contains all the configurable parameters in the statistical component. Only a trained security officer should be allowed to modify these parameters, particularly since changing some of these requires the profiles to be retrained before the statistical scores becoming meaningful again.

- `arec_hlife`. This field is the audit record half-life that is the basis for short-term profile aging. It is represented in units of audit records.
- `prof_hlife`. This field is the profile half-life that is the basis for long-term profile aging. It is represented in units of days.
- `arec_gamma`. This field is the aging factor applied to each count in the short-term profile (computed from `arec_hlife`) .
- `prof_gamma`. This field is the aging factor applied to each count in the long-term profile (computed from `prof_hlife`) .
- `corr_cutoff`. This field is the correlation cutoff value for the correlation matrix. It is currently unused.
- `min_effn`. This field is the minimum effective-`N` value for all measures. This is essentially the number of (aged) audit records that should get processed by the statistical component in order to begin building a reasonable historical profile.
- `traindays`. This is the number of days that are required for profile training.
- `measures`. This is the table of measures for the statistical component. It serves as the default configuration for a new subject (it is possible for subjects to have different measures activated, although this can potentially become quite complicated).
- `nmeas_active`. This field represents the number of activated measures.
- `statmode`. This value is a bit field that indicates which modes the statistical component should be running. One example of a mode is whether or not the

updater should be invoked by the main statistical component (as opposed to being independently started from an external process). This field is generally used for development and experimentation purposes only.

- `command_classes`. This field contains lists of special commands or hosts that have been assigned to a particular group (compilers, editors, local hosts, etc.). It is a hash table that contains all of these commands and hosts.
- `thresholds`. This field contains the threshold levels used to determine when a score value should be reported to the security officer. These represent the percentiles where the score distribution should be considered in alert status. Currently, there are two levels specified: red for critical, yellow for warning. The default settings for these values are 99.9 and 99 respectively.

There are several data structures that are stored in lists or tables that require fast access (such as categories and command lists). A generic data structure is available to support a hashing scheme for various types of structures. Utility functions are available to create, examine, and manipulate these hash tables.

3.4.3 Functional Interfaces

1. `Status make_activity_vector( const ia_audit_rec *audit_rec, const Hashnode *commd_classes[], const Hashnode *subj_commd_list[], long *prev_timestamp, Activity *activity_vec[] )`
   This function creates the activity vector that represents what was observed by the given audit record. It extracts the necessary information from the NIDES audit record and puts them into the activity vector. In some cases, some data conversion is necessary (for example, the timestamp value in the audit record is represented in Unix long integer form, and the hour and day must be deciphered from this value).

   To obtain the subject name for this audit record, this function looks at the audit user name first. If this is not available, then it will use the regular user name. It assumes that at least one of these fields is not null.

   The interarrival time (used for the activity intensity measures and the interarrival measure) is computed from the timestamp of the audit record and the timestamp of the last audit record processed for this subject.

   If the audit record indicates that a command was invoked, this function will first check to see if this command is any one of the special commands defined for this target system (mailer, editor, compiler, etc.); these commands are provided in the argument *commd_classes*. If a command has previously not been seen for this subject, then it is added to the subjects command lists (both the general commands and special commands). Some action types have predefined command names associated with them, and so the command (program) names are assigned to these predefined names.

If a measure is a binary continuous type, then the activity vector for this measure will contain either a 0 or 1 to indicate whether or not this measure was observed (1 means observed). If a measure is continuous, the activity vector is assigned the appropriate numerical value; if this measure is not observed in the audit record, then the value is set to 0. If a measure is categorical, the activity vector location is assigned the character string representation of the category ID; if the measure is not observed, the corresponding location in the activity vector for this measure will be set to null.

Some of the network-related measures require knowledge of whether a specified host name is local or remote. Local means that the host is on the same local area network; all other remote hosts are considered remote. The list of local hosts is defined in the *commd_classes* argument.

2. `Status compute_score( const Activity *activity_vec[], const int intarrtime, const Statconfig_struct *config_params, const Hist_prof_struct *hist_profile, Curr_prof_struct *curr_profile, double *score)`

This function is the heart of the statistical anomaly detection analysis. It implements the algorithms that compare the subjects short-term profile against its historical long-term profile, to produce a score value that represents the degree to which the short-term profile differs from the long-term profile.

This computation is performed for each audit record. This function takes an activity vector that represents the measures observed in a subjects audit record and updates the short-term profile for this subject. The short-term profile is calculated according to the type of measure (ordinal, categorical, audit record distribution or activity intensity). In the case of ordinal and categorical measures, this is done by finding the observed category for the measure and determining the probability of this category in the recent past (defined by the audit record half-life value). The audit record distribution measure is calculated similarly, where the categories are the types of measures themselves (and hence each of the categories for this measure are updated for those measures that have been observed). The short-term profile for the activity intensity measures are based on the amount of time that has elapsed since the last audit record for this subject (hence the interarrival parameter is needed).

Special consideration is given to never-seen-before categories and categories with a very low probability. For new categories, we have a separate category that keeps track of appearances of never-seen-before categories, and if this category reaches a level that differs greatly from the long-term profile for this category, the `Q` (and hence `S`) value for this measure will be significantly high. Rare categories are treated similarly.

Once the short-term profile is computed, a `Q` value is produced for each observed measure that indicates some comparison of the short-term to the historical profile. These `Q` values are mapped to corresponding `S` values, and this vector of comparison values is squared and summed to produce a T^2 score that basically measures abnor-

mality. The distribution table for T^2 is represented in tenths of score points for the first 200, and whole numbers thereafter, so some conversion techniques are used to accommodate this.

In addition to the short-term versus long-term comparison, this function also records cumulative activity of a subject for this period (a period in this case is defined as the time between profile updates). This is done by merely independently counting the number of observations for the categories, `Q` values (which bins they fall into), and the resultant T^2 scores. These counters are later incorporated into the historical profile at the next profile update *(see update_profile()* on page 29).

3. `Status update_profile(const Statconfig_struct *config, Curr_prof_struct *curr_profile, Hist_prof_struct *hist_profile )`

This function implements the algorithms that create the long-term profile. Each time this function is called, the long-term profiles are updated with the recent set of activity. The cumulative activity for a subject since the last profile update is incorporated into the previous long-term profile and aged according to the profile half life defined in the statistical configuration, and a new historical profile is built for this subject. All cumulative activity counters are reset to zero to begin the next period.

There are three levels of profile updating. At the lowest level, the probability values for individual measure categories are adjusted according to the frequency of observation during the day. If a measures category falls below a minimum probability value `(MINPROB)`, it gets dropped from the category list, in order to avoid an unbounded growing list of categories for any measure. Finally, the cached hash table that contains the individual measure categories is rebuilt (since some categories may have been dropped because they fell below the minimum probability). The next level of profile update occurs at the `Q` distribution level. Again, the observed values for `Q` (defined in the *qcount* field of the current profile) are folded into the historical distribution and aged appropriately, and hence a new `Q` distribution is computed. Finally, the T^2 score distribution table is updated in a likewise manner. The new 99.9 th percentile is recalculated and is subsequently used to determine whether a score should be reported as anomalous to the security officer. Again (as noted in *compute_score()* page 30), the T^2 distribution table is represented in tenths of score points for the first 200, and whole numbers thereafter, so conversion techniques are used to deal with this.

Based on the training stage of the profile, only certain portions of the profile are updated. The training period is broken up into three stages. This is done by dividing the total number of training days specified in the configuration by three (rounded off to the nearest whole number), and hence each trimester is the calculated number of days. During the first stage, only the categorical probabilities are updated. In the second trimester of training, the long-term profiling for the `Q` distribution takes place. Finally, in the last trimester, T^2 scores are profiled. Once the training period is complete, all parts of the long-term profile are updated each time this function is called, and only then can the T^2 scores be considered valid.

4. `Status make_def_profile( Curr_prof_struct *curr_prof, Hist_prof_struct *hist_prof, const Measure *measures[] )`
   This function creates a default profile. The data structures for the current and historical profiles must be allocated prior to making this call.

   A default frequency distribution table is created for both the `Q` and the tails of the `Q` distribution `(TPROB)`. We do this to ensure that the sum of the probabilities in each row add up to 1.0. We evenly distribute the probability among the first 10 bins for each measure, and zero out the rest. For the T^2 distribution table, 1.0 is filled in for the first score slot.

   Certain measures will have predefined categories from the start. For example, ordinal (counting) measures will automatically have 32 categories, as we map the observed values for these measures logarithmically into bin values between 0 and 31. By default, the audit record distribution measure will have all the initially activated measures as its categories. Default probabilities for these predefined categories are evenly distributed (1/32 for ordinal measures, 1/`active-measures` for the audit record distribution measure). As for categorical measures, some measures, such as the hours-of-use and days-of-use measures, have a finite set of categories (24 hours of the day and 7 days in the week), so the categories are predefined for these measures. In addition, all the categorical measures have a new category associated with them, and hence this category is preallocated in this function (default is `MINPROB`). Finally, the hash table for all measure categories is created for the above predefined categories.

   Score thresholds are set to default values: 99^{th} percentile for warning status, and the 99.9^{th} percentile for critical status.

5. `Status reconfig_stats( Statconfig_struct *config, Curr_prof_struct *curr_prof, Hist_prof_struct *hist_prof )`
   This function takes the given statistical configuration and applies it to the subjects profile. It checks to make sure that all the configuration data is valid. The aging factors are computed from the given half-life values. The number of active measures is recomputed (some may have been reactivated or turned off) and applied to the profile. If the score threshold cutoffs have been changed, the corresponding score values are recomputed.

6. `Status check_anomaly( const double score, const Curr_prof_struct *cprof, const Hist_prof_struct *hprof, int measures[], int training_days, Stats_analysis *anomaly_rec )`
   This function determines whether the score obtained from *compute_score* is anomalous enough to be reported to the security officer. A *Stat_analysis* data structure is passed in to be filled with relevant information.

   If the score is above the critical threshold zone (defined in the subjects profile), then this function sets the alert status to critical. If the profile is still in training mode, all scores are reported as safe.

In addition to the above, this function determines the top five measures that contributed to the score. This is done by examining all the `S` values and selecting the highest five values. These data, along with the alert status, are returned in the *anomaly_rec* argument.

3.5 Rulebased Component

The rulebased component uses a rulebase generated by the PBEST tool (see Appendix of [5]). This tool takes a rulebase specification and translates it into a series of functions for each rule (these functions implement a modified version of the Rete algorithm [1]). It also generates functions for asserting all the different types of facts into the knowledge base. Besides the code generated by the PBEST tool, there is a support library that includes code for the rulebase engine and other support code that remains constant for all rulebases. None of these functions are visible to the NIDES programmer; all external interaction with the rulebased component is conducted using the interface functions described in the section on functional interfaces, below.

3.5.1 Functionality

This section describes the hidden functionality of the rulebase. All the functions described here are normally invisible to the NIDES programmer.

The rulebased component does its inference using a modified Rete algorithm. The main difference between the PBEST implementation and the Rete algorithm as described by Forgy is that PBEST does not represent the knowledge base state in the form of the networks Forgy discusses; instead it uses lists and functions to serve the purpose of the network nodes.

Each rule consists of a set of functions. These functions are called the `ante1, ante2, binding` and `concl` functions. An `ante1` function is called with a message invoking one of several actions: *assert, negate,* or *select binding*. When a fact is negated, the `ante1` functions are called with the fact and a negate action message. These functions remove the fact from the list of facts they know about. The act of asserting a fact into the knowledge base consists of calling all the `ante1` functions with the fact and an assert action message. The `ante1` function checks to see if it is interested in the kind of fact being asserted whether the fact is of the type it looks for and whether the data in the fact meets the tests the rule was given for that kind of fact. That is, the `ante1` function checks antecedent clauses of the form

```
[+ev:event^BLOG|action == ia#BAD_LOGIN]
```

and it does these tests at fact-assert time. Finally, when the component wants the rules to determine whether they can fire, it calls the `ante1` functions with a select binding action message.

The select binding message causes the `ante1` functions to call their `ante2` functions. The `ante2` functions check to see if their rule is active. If their rule is inactive, they return

without doing any of their tests. If the rule is active, the `ante2` functions perform the inter-fact tests and consistency checks to determine whether a rule can fire, as well as any other arbitrary tests that were specified in the rules antecedent. They test clauses of the form

```
[+bp:bad_password|userid == ev.real_userid]
```

where `ev.real_userid` is a field from a fact that was already tested by the `ante1` function, from a clause such as

```
[+ev:event|action == ia#BAD_LOGIN]
```

They also test clauses such as

```
[?|kb_check_local_host(ev.rhost, do.domain_name) == bool#FALSE]
```

that implement some arbitrary test. The `ante2` functions also check fact marks, which are dynamic and may change after a fact has been asserted.

If the `ante2` function decides that its rule can fire, it calls its rules binding function with the facts that allow it to fire. The binding function returns a binding consisting of the rule and a list of the facts allowing the rule to fire. The `ante2` function stores this in the binding slot for the rule. If the rule cant fire, it stores a null binding in the binding slot.

The next step in most production systems is called *conflict resolution.* Conflict resolution means selecting one rule to fire when many have indicated that they can fire. In the PBEST system, conflict resolution effectively occurs at compile time, when rules are ordered by two specific criteria: rank (also known as priority), and order of occurrence. After all the rules have selected their bindings, the first rule that has a valid binding is the rule that will fire.

Once the rulebase engine has selected a rule to fire, it invokes the `concl` function of that rule with the facts that allowed the rule to fire as arguments. The `concl` function implements the rule actions.

3.5.2 Data Structures

The data type definitions for this component are as follows.

```
struct factlist{
  /* struct fact is dynamically defined when rulebase specification is
  translated */
  struct fact *fact;
  struct factlist *next;
  struct factlist *prev;
};
```

The `struct factlist` data structure is used to store lists of the facts that are bound by the rules.

```
struct factheader{
  struct factlist *fl;
  struct factheader *next;
  struct factheader *prev;
};
```

The `struct factheader` data structure stores lists of factlists.

```
struct bind{
  struct rulelist *rule;
  struct factlist *facts;
};
```

The `struct bind` data structure is used to store a rule together with the collection of facts that make the rule firable.

```
struct bindlist{
  struct bind *binding;
  struct bindlist *next;
  struct bindlist *prev;
};
```

The `struct bindlist` data structure stores lists of firable bindings.

```
struct rulefields {
  void (*ante1)();
  void (*concl)();
  char *name;
};
```

The `struct rulefields` data structure stores pointers to the functions that implement the rule and the name of the rule.

```
struct rulelist{
  struct rulefields r;
  struct factheader *fh;        /* Facts this rule has bound to. */
  struct bindlist *bestbinding;/* Best binding with which to fire. */
  char *name;                   /* Rule name. */
  int repeat;                   /* Rule repeatability. */
  int rank;                     /* Rule priority. */
  int active;                   /* Can the rule currently fire? */
  long ante_secs;               /* Cumulative seconds spent executing antecedent. */
  long ante_usecs;              /* Cumulative microseconds spent executing antecedent. */
  long  conc_secs;              /* Cumulative seconds spent executing conclusion. */
```

```
  long conc_usecs;              /* Cumulative microseconds spent executing conclusion. */
  long rule_firings;            /* Number of times consequent was executed. */
  char *text;                   /* Rule text. */
  char *sourcefile;             /* Full path name of rule source. */
  struct rulelist *next;
  struct rulelist *prev;
};
```

The `struct rulelist` data structure contains all the data needed to implement a rule, together with a pointer to a list of the facts, if any, to which the rule is bound.

```
typedef enum {
  ADD_RULE,
  DELETE_RULE,
  MODIFY_RULE,
} ia_rb_action;
```

The `ia_rb_action` enumeration contains the possible configuration actions.

```
struct config_action {
  string rule_name;
  ia_rb_action action_code;    /* see enumerated types listed above */
};
```

The `struct config_action` data structure is used to pass the configuration action messages to the rulebased component.

```
struct rule_info {
string rule_path;
string rule_text;
int active;
};
```

The `struct rule_info` data structure is used to return information about a rule when a `get_rule_info()` request is made.

```
struct fact_info {
int count;
string fact_rep[count];
};
```

The `struct fact_info` data structure is used to return the human-readable representations of the facts in the knowledge base in response to a `get_fact_info()` request.

3.5.3 Functional Interfaces

The interface definitions for the rulebased component are as follows.

1. `Status init_kb( struct rulelist **kb )`
   Produce the state of the initial knowledge base into `kb`. As a result of this call, the pointer `kb` will point to the head of a properly initialized rulebase. This function also calls internal initialization functions needed to set up the state of the knowledge base. It currently returns 0 (meaning no error) whenever it returns.

2. `Status config_kb( struct rulelist **kb, const struct config_action *action )`
   Configure a knowledge base in `kb` using configuration action in `action`. This function allows run-time configuration of the rulebased component. The current configuration capabilities consist of adding, modifying or removing rules from the knowledge base while the system is operating. This function depends on the system link editor providing dynamic linking and loading functionality and thus is not portable. (This function is not called in the current release.)

   The function `config_kb()` returns -1 for memory or other system failures; -2 for invalid rule name, -3 if it is passed a config action that it doesnt understand, and 0 for success.

3. `Status deduce_kb( const ia_audit_rec *ar, const struct kb_state *kb, Rulebase_analysis *result )`
   This function analyzes the audit record in `ar` using the knowledge base `kb` and records its analysis in `result`. It works by asserting the current audit record into the component's knowledge base using the automatically-generated assert function for audit record facts. It then calls the rulebase engine. The rulebase engine performs all possible analysis on the audit record that was asserted, and then removes the audit record from the knowledge base.

   This function always returns a 0 (meaning no error) result.

4. `Status get_rule_info_kb( const struct kb_state *kb, const string rule_name, struct rule_info *info )`
   This function gets information about the rule name contained in `rule_name` from `kb` and records it in `info`. It scans the knowledge base for the given rule name, and if it finds the rule with that name it returns information about the rule. This information consists of the name of the source file from where the rule came, the text of the rule, and whether the rule is active. (Rule activity is a very transient state, and thus the last piece of information may not be very useful.)

   The function returns 0 if it finds the rule and -1 if it cant find it.

5. `get_fact_info_kb( const struct kb_state *kb, struct fact_info *info )`
   This function gets information about the facts in the knowledge base from `kb` and

records this information in `info` using a human-readable representation. There is no global list of the knowledge bases facts; instead, each rule maintains a list of the facts it is interested in (this is called binding to a fact). Thus, this function must scan the list of rules and extract the facts each rule has bound to. Since more than one rule can bind to a given fact, the function sorts the list of facts and discards duplicates. It then creates the printed representations of the facts and sorts these representations by fact ID number. The function is thus fairly expensive, since it requires two sorts: a step to discard duplicates, and a step to create the printed representations.

Besides the printed representations, the function puts a count of the number of facts into the `info` structure.

This function returns -1 for failures such as running out of memory and 0 for success.

3.6 Resolver Component

The resolver component analyzes the alerts issued by the statistical and rulebase components and reports only non-redundant alerts.

The data structures of the resolver component are as follows.

```
typedef enum {
        SAFE,              /* Everything is okay */
        CRITICAL,          /* Critical Alert */
} ia_result_code;

typedef struct audit_record_info {
    ia_timeval timestamp; /* time audit record was generated */
    ia_seqno rseq;        /* sequence number of audit record */
    string host;          /* name of target host generating the
                             audit record */
    string subject;       /* name of subject (user) generating the
                             audit record */
} audit_record_info;

typedef struct Rulebase_analysis {
     ia_result_code result_code;  /* see enumerated types */
     string significance;         /* significance of rule firing */
     string rule_name;            /* name of rule that fired and generated
                                     this record */
} Rulebase_analysis;

typedef struct Stats_result {
    Stats_analysis analysis;      /* statistical analysis */
```

```
    audit_record_info ar_info;     /* audit record information */
} Stats_result;

typedef struct Rulebase_result {
    Rulebase_analysis analysis;    /* rulebase analysis */
    audit_record_info ar_info;     /* audit record information */
} Rulebase_result;

typedef struct Alert {
     string message ;              /* string of "\n" terminated lines */
} Alert;
```

The resolver has the following functional interface.

- `Status resolve( const Stats_result *stats_result, const Rulebase_result *rulebase_result, Alert *resolver_rec )`
  This function resolves the analysis from the statistical component `(stats_result)` and the rulebase component `(rulebase_result)` and returns it in `resolver_rec`. Every intrusion flagged by the rulebased component becomes an alert. Every anomaly flagged by the statistical component becomes an alert if one or more of the following is true.

  1. The previous audit record from this subject was not anomalous.
  2. The previous audit record from this subject had a different top measure.
  3. The score for this audit record is at least 1.5 times as high as the score in the previous *reported* alert for this subject.

  Function `resolve` returns the number of alerts, which is always 0 or 1 in the current implementation. There are no error conditions possible with the current implementation.

3.7 Security Officer User Interface Component

The Security Officer User Interface component provides the security officer with alert and status information and enables the security officer to manage the operation of the NIDES prototype.

The component consists of the following function interfaces.

- `put_alert`
- `email_alert`
- `control_target`
- `target_error`

- control_server
- server_error
- put_status
- control_ar_storage
- ar_storage_error
- put_seq_no
- start_test_analysis
- test_analysis_status
- get_list_of_instance_names
- create_instance
- copy_instance
- delete_instance

Each of the functions listed above are either invoked on behalf of the security officer or on behalf of some other component of the NIDES prototype.

We consider the functions in detail below.

- put_alert(struct Alert_message *)
  When the put_alert() function is invoked, the alert denoted by the argument is presented to the security officer. The presentation mode is given by the global variable alert_mechanism, which can have one of these four values:

  1. EMAIL_ALERT
  2. POPUP_ALERT
  3. EMAIL_POPUP_ALERTS
  4. NO_ALERT_MECHANISM

  The variable alert_mechanism is initialized to NO_ALERT_MECHANISM, thus disabling presentation of alerts to the security officer by default. The variables value can be modified by the security-officer at any time.

- email_alert(string, string)
  This function when invoked posts an email message denoted by the first parameter to a list of recipients denoted by the second parameter. This function is invoked as a result of function put_alert() being invoked (see page 38) and EMAIL_ALERT or EMAIL_POPUP_ALERTS being set in the variable alert_mechanism.

- `control_target( string, int )`
  This function, when invoked, initiates or terminates the audit generation activity on a target host. The first parameter specifies the name of the target host, and the second parameter specifies whether audit generation needs to start or cease.

- `target_error( string, int )`
  When the `target_error()` function is invoked, the name of target host given by the first parameter is displayed to the security officer along with the error message, which is denoted by the second parameter. The error message is one of the following types.

  1. `TARGET_NOT_STARTED`
  2. `ERROR_ON_START_TARGET`
  3. `TARGET_NOT_STOPPED`
  4. `ERROR_ON_STOP_TARGET`

  This function is invoked only after function `control_target()` (see page 39) has been previously invoked.

- `control_server( string, int )`
  This function, when invoked, initiates or terminates a server associated with NIDES on a specific host. The name of the server is given by the first parameter, and whether to start or stop the server is given by the second parameter.

- `server_error( string, int )`
  When the `server_error()` function is invoked, the name of server given by the first parameter is displayed to the security officer along with the error or status message, which is denoted by the second parameter. The message is one of the following types.

  1. `ERROR_ON_START_ARPOOL`
  2. `ERROR_ON_START_ANALYSIS`
  3. `ERROR_ON_STOP_ARPOOL`
  4. `ERROR_ON_STOP_ANALYSIS`
  5. `STOP_ARPOOL_DONE`
  6. `STOP_ANALYSIS_DONE`

  This function is invoked only after function `control_server()` (see page 39) has been previously invoked.

- `put_status( struct host_list * )`
  When function `put_status()` is called, the status `(UP/DOWN)` of each of the target hosts is displayed to the security officer. If the list is already being displayed, it is updated to reflect any changes in status.

- `control_ar_storage( string, int, string )`
  This function, when invoked, causes audit data from the audit data collection component to be written into a file. The first parameter indicates the host where collection is to take place. The second parameter indicates whether collection needs to start or stop. The third parameter indicates the name of the file where audit data is to be recorded.

- ar_storage_error(string, string, int)
  When function `ar_storage_error()` is invoked, an error message is displayed to the security officer to the effect that audit data archival has failed. The first parameter is a character string representing the host where the audit data archival process was executing, the second parameter is a character string representing the name of the file where the archive data was written, and the third parameter is an integer error code which is currently not used. This function is invoked only after function `control_ar_storage()` has been previously invoked (see page 40).

- put_seqno(long, long, long)
  When function `put_seqno()` is invoked, the status of NIDES processing is displayed to the security officer. The three parameters are the sequence number of the last audit record received by the audit collection component, the sequence number of the last audit record processed by the resolver component, and the total number of alerts recorded since startup. The display consists of the latest cumulative alert count and elapsed time since startup.

- `start_test_analysis( string, string, string )`
  This function, when invoked, uses the profile instance denoted by the first parameter to process audit data in NIDES format from the file denoted by the second parameter. The function writes the alerts to the file denoted by the third parameter.

- test_analysis_status(string, string, int)
  When function `test_analysis_status()` is invoked, the status of the test analysis batch run is displayed to the security officer. The first parameter is a character string representing the host where the test analysis is executing, the second parameter is a character string representing the name of the output file where the results of test analysis are written, and and the third parameter is an integer status code. Based upon the status code, the user will be notified of the status of the test analysis run as follows.

  1. `BATCH_DONE`
  2. `BATCH_ERROR`

  This function is invoked only if function `start_test_analysis()` (see page 40) has been previously invoked.

- `string *get_list_of_instance_names( )`
  This function, when invoked, returns the list of known profile instances.

- `create_instance( string )`
  This function, when invoked, creates an empty profile instance denoted by its parameter.

- `copy_instance( string, string )`
  This function, when invoked, copies the profile instance denoted by the first parameter to a new instance denoted by the second parameter.

- `delete_instance( string )`
  This function, when invoked, deletes the profile instance denoted by its parameter.

3.8 Audit Generation Service

The audit generation service consists of two processes: *agend* which is the server and *agen* which is the active agent process of the server. The purpose of *agen* is to gather audit data on the resident system, convert them to the NIDES audit record format on the fly, and then forward these audit records to the audit collection service.

Agen can operate in two modes, which can be selected with the use of command line arguments. In the first mode, *agen* runs in a fault-tolerant mode. That is, it tries to recover from communication errors that occur between it and the audit collection service by retrying the failed operation. In the second mode, *agen* terminates when a communication error is detected.

Agen is typically invoked by *agend,* which is responsible for starting and stopping *agen.*

The following specifies the interface of *agend.* First we present the data structures used by *agend* and then we describe the functional interfaces.

3.8.1 Data Structures

The data structures used by the audit generation service are as follows.

```
const DEFAULT_AGEND_PORT = 3131;    /* TCP/IP port on which
                                       agend waits for incoming
                                       requests.
                                    */

typedef enum agend_rval {
        AGEND_ERR = -1,
        AGEND_OK = 0,
        AGEND_RUNNING,
} agend_rval;
```

```
agend_rval  agend_start_agen(string  agen_arg);
agend_rval  agend_stop_agen(void);
```

3.8.2 Functional Interfaces

The functions exported as remote procedure calls (RPC) for *agend* are described below.

- `agend_start_agen( string )`
  This function starts an instance of *agen* on the target machine. *Agen* interprets the parameter as the address of audit collection service to connect to. It is typically a string of the form `hostname:port`. The function returns `AGEND_ERR` if an unforeseen error occurred, or `AGEND_OK` if *agen* was invoked successfully, and `AGEND_RUNNING` if a previous instance of *agen* is still active. Note that only one instance of *agen* is allowed to be running at any time.
- `agend_stop_agen()` This function terminates a previous invocation of *agen.* It returns `AGEND_ERR` if an unforeseen error occurs or no *agen* is active, and `AGEND_OK` if *agen* has been terminated successfully.

3.9 Audit Collection Service

The audit collection service consists of a server *(arpool)* which is responsible for collection of audit records from all target hosts and distribution of audit records to all active clients of *arpool.*

3.9.1 Data Structures

The data structures used by the audit collection service are as follows.

```
struct arpool_vec {
        struct  ia_audit_rec        *rec[nrec];
};

struct arpool_status {
        long                    lowater,
                                hiwater;
        long                    npool;
        long                    max_rseq_hi;
        unsigned long           max_rseq_lo;
        struct arpool_producer {
                string          hostname;
```

```
        } producer[nproducers];
        struct arpool_consumer {
                string          hostname;
                long            rseq_hi;
                unsigned long   rseq_lo;
        } consumer[nconsumers];
};
```

3.9.2 Functional Interfaces

The functions exported as remote procedure calls are as follows.

- `int put_ar_vec(struct arpool_vec *)`
  This function deposits a vector of audit records in the pool of audit records maintained by *arpool* and returns the value 0 upon completion.

- `struct arpool_vec *get_ar_vec(void)`
  This function retrieves a vector of audit records stored in *arpool* as its return result.

- `struct arpool_status *arpool_get_status(void)`
  This function obtains status information maintained in *arpool* regarding *agen* processes on remote target hosts that are depositing audit records, and on local client processes that are retrieving audit records. It also provides information about the usage of the audit record pool.

3.10 Analysis Service

The analysis service consists of a server *(analysis* server) and two client processes: the *statistical* client and the *rulebased* client. The server itself embodies the resolver component (see Figure 3).

The *analysis* service defines the following functions, which can be invoked as remote procedure calls (RPC).

- `void put_stats_results(int, Stats_result * )`

- `void put_rulebase_results(int, Rulebase_result * )`

- `AlertVec *get_alerts()`

- `AnalInfo getAnalInfo()`

The *analysis* server receives a stream of `Stats_results` from the *statistical* client and a stream of `Rulebase_results` from the *rulebased* client. It matches corresponding pairs of `Stats_results` and `Rulebase_results,` and invokes the resolver on each pair (using the function `resolve` — see page 37), queuing the resulting alerts, if any. It provides those alerts

to the security officer user interface when function `get_alerts()` is invoked. It provides the sequence number of the most recently processed audit record, as well as the total number of alerts reported so far, when function `getAnalInfo()` is invoked.

The *analysis* server assumes that there is precisely one statistical client, one rulebased client, and one agent reading alerts on behalf of the security officer user interface. It queues alerts indefinitely, until the agent for the security officer user interface reads them, so if the user interface is running behind, the analysis server may require arbitrary amounts of memory to store these alerts. In practice, this is unlikely to occur.

3.10.1 Statistical client

The *statistical* client invokes two functions using RPC, one defined in *arpool* and one defined in the *analysis* server (see Figure 3). The functions invoked using RPC are as follows.

- From *arpool*
  ```
  struct arpool_vec *get_ar_vec()
  ```
- From *analysis* server
  ```
  void put_stats_results(int count, Stats_result *results)
  ```

The *statistical* client gets audit records from *arpool* with `get_ar_vec`. For each audit record, the *statistical* client extracts the subject name and reads the profile for that subject from the persistent storage. It then runs the audit record through the statistics with `make_activity_vector` and `check_anomaly`. It sends the results to the *analysis* server after each block of audit records with `put_stats_results`.

The *statistical* client also maintains an in-memory cache of profiles, so as to reduce access time through the secondary medium. All dirty in-cache profiles are flushed every midnight. Profiles are eliminated from the cache at the same time if they havent been accessed in the preceding 24 hours.

Every midnight, the *statistical* client updates the historical profile for every subject in the cache.

The *statistical* client depends on the persistent storage facility to store all profiles between invocations.

The *statistical* client flushes dirty profiles to persistent storage in the event of any fatal error. It detects failures in *arpool* or the *analysis* server. It detects `TERMINATE` signals and treats them as an error condition. It verifies the consistency of every profile read from persistent storage. In some cases it may be able to continue even if the profile is corrupt, by regenerating it from scratch.

The *statistical* client keeps all the profiles used in a day in its in-memory cache, so it requires a large amount of virtual memory – around 200K for each profile.

3.10.2 Rulebased client

The *rulebased* client invokes two functions using RPC, one defined in *arpool* and one defined in the *analysis* server (see Figure 3). The functions invoked using RPC are as follows.

- From *arpool*
  `struct arpool_vec *get_ar_vec( )`
- From *analysis* server
  `void put_rulebase_results(int count, Rulebase_result *results)`

The *rulebased* client gets audit records from *arpool* with `get_ar_vec( ).` It passes each audit record through the rulebase with `deduce_kb( ),` and sends the results on to the *analysis* server with `put_rulebase_results( ).`

3.11 Security Officer User Interface Service

The security officer user interface service is responsible for presenting information received from the other services to the security officer, and to allow the security officer to manage the operation of the prototype itself.

The agent interface is part of the security officer user interface service. It is responsible for managing the agents, and all communications with the agents. It exports a number of remote procedure calls that are used by the agents to exchange information with the security officer user interface *(SOUI)* server, as well as non-RPC calls that are used by the rest of *SOUI* server to delegate jobs to the agents.

3.11.1 Data Structures

The *SOUI* server uses the following data structures.

```
struct email_msg {
    string      to;          /* recipient list */
    string      msg;         /* text of the message to send */
    };

struct control_cmd {
    string      host;
    int         action;      /* action code */
    string      a1,a2,a3;    /* extra arguments used by some actions */
    };

/* Action, result, and error codes */
enum {  START_ARPOOL=101,
        STOP_ARPOOL=102,
        START_ANALYSIS=103,
        STOP_ANALYSIS=104,
        START_TARGET=105,
        STOP_TARGET=106,
        BATCH_START=107,        START_BATCH=107,
```

```
BATCH_STOP=108,          STOP_BATCH=108,
START_COLLECTION=1O9,
STOP_COLLECTION=11O,

ERROR=0,
RUNNING=201,
DONE=202,
UP=203,
DOWN=204,

ERROR_ON_START_ARPOOL=1,
ERROR_ON_STOP_ARPOOL=2,
ERROR_ON_START_ANALYSIS=3,
ERROR_ON_STOP_ANALYSIS=4,
ERROR_ON_START_TARGET=5,
ERROR_ON_STOP_TARGET=6,

STOP_ARPOOL_DONE=205,
STOP_ANALYSIS_DONE=206,

TARGET_UP=203,
TARGET_DOWN=204,
BATCH_ERROR=0,
BATCH_RUNNING=201,
BATCH_DONE=202,
/*   0- 99 error codes */
/* l00-199 action codes */
/* 200-299 nonerror result/information codes */
};
```

3.11.2 Functional Interfaces

The following functions are used by the *SOUI* server to start and stop its agents.

1. `Status start_agents()`
   This function forks and executes each of the seven agents, remembering their process-ids for later use by `kill_agents()`.

2. `void kill_agents()`
   This function kills all the agents started by `start_agents()`.

The *SOUI* server exports the following functions as remote procedure calls that can be invoked by agents.

- `void put_alert(Alert *)`
- `email_msg *get_email()`
- `control_cmd *get_control_target()`
- `void target_error(string host, int errcode)`
- `control_cmd *get_control_server()`
- `void server_error(string host, int errcode)`
- `control_cmd *get_start_test_analysis()`
- `void test_analysis_status(string host, int errcode, string output)`
- `control_cmd *get_control_ar_storage()`
- `void ar_storage_error(string host, int errcode, string filename)`
- `void put_status(struct hostlist *status)`
- `void put_seqno(long arpool_seqno, long anal_seqno, long anal_alertno)`

3.11.3 Agents

The *SOUI* server has seven agent processes, each of which has its own task to perform (see Figure 3). These agents are described below.

1. *Agent_alerts*
   This agent is responsible for getting alerts from the *analysis* server and reporting them to the *SOUI* server. The RPCs invoked are as follows.

   - From *analysis* server
     `AlertVec *get_alerts()`
   - From *SOUI* server
     `void put_alert(Alert *)`

   This agent is a simple loop, getting a block of alerts with `get_alerts()`, and reporting them one at a time with `put_alert()`

   The agent determines if the *analysis* server has failed or is not running and waits for it to be started.

2. *Agent_batch*
   This agent is responsible for starting and stopping test runs for the test facility. It also reports errors or completion back to the *SOUI* server. The RPCs invoked are as follows.

- From *SOUI* server

```
control_cmd *get_start_test_analysis()
void test_analysis_status(string host, int errcode, string output)
```

This agent gets commands from the *SOUI* server to start and stop test analyses, and reports any unusual occurrences relating to the analyses. It is implemented as a simple loop that get commands with `get_start_test_analysis()` and carries out the commands by forking and spawning child processes and sending signals to them. It also catches signals from these children and reports errors and completion status back to the *SOUI* server with `test_analysis_status()`.

3. *Agent_email*
This agent is responsible for sending *email* on behalf of the *SOUI* server. The RPC invoked is as follows.

- From *SOUI* server

```
email_msg *get_email()
```

This agent is a simple loop calling `get_email()` and invoking `sendmail` to send *email* messages.

4. *Agent_save*
This agent is responsible for starting and stopping the archiving of audit data to files. The RPCs invoked are as follows.

- From *SOUI* server

```
control_cmd *get_control_ar_storage()
void ar_storage_error(string host, int errcode, string filename)
```

This agent is a simple loop much like *agent_batch*. It gets commands to stop and start archiving with `get_control_ar_storage()`, and carries out those commands by forking and spawning child processes and sending signals to them. It catches signals and reports back status with `ar_storage_error()`.

5. *Agent_server*
This agent is responsible for starting and stopping the *analysis* server and *arpool,* and reporting on their status to the *SOUI* server. The RPCs invoked are as follows.

- From *SOUI* server

```
control_cmd *get_control_server()
void server_error(string host, int errcode)
```

This agent is a simple loop much like *agent_batch*. It gets commands to stop and start archiving with `get_control_server()`, and carries out those commands by forking and spawning child processes and sending signals to them. It catches signals and reports back status with `server_error()`.

6. *Agent_status*
   This agent is responsible for polling *arpool* and the *analysis* server to determine the current status of the NIDES system. The RPCs invoked are as follows.

   - From *arpool*
     ```
     arpool_status *arpool_get_status()
     ```
   - From *analysis* server
     ```
     AnalInfo getAnalInfo()
     ```
   - From *SOUI* server
     ```
     void put_status(struct hostlist *status)
     void put_seqno(long arpool_seqno, long anal_seqno, long anal_alertno)
     ```

   This agent is a simple loop. It gets information about currently running targets and the latest audit record sequence number from *arpool* with `arpool_get_status()`. It gets the last audit record sequence number and the last alert number from the *analysis* server with `getAnalInfo()`. If the targets have changed since the last poll, it reports the status of the targets that have changed with `put_status()`. In any case, it reports the sequence numbers and alert count with `put_seqno()` to the *SOUI* server. It then sleeps for 10 seconds before repeating.

   The agent detects failure on the part of *arpool* or the *analysis* server, and waits for that component to be restarted.

7. *Agent_target*
   This agent starts *agen* on target hosts by communicating with the *agend* daemon on those target hosts. The RPCs invoked are as follows.

   - From *SOUI* server
     ```
     control_cmd  *get_control_target()
     void target_error(string host, int errcode)
     ```
   - From *agend*
     ```
     agend_rval  agend_start_agen()
     agend_rval  agend_stop_agen()
     ```

   This agent is a simple loop that gets commands from the *SOUI* server with `get_control_target()` and makes calls to `agendstart_agen()` or `agend_stop_agen()` on the requested target host as appropriate. If the RPC to *agend* fails, it reports this with `target_error()`.

   Error handling in this agent is minimal. It detects RPC failures and reports errors back to the *SOUI* server if it cannot invoke RPC on *agend* on the target host, or if *agend* reports an error. If *agen* fails after it has been successfully started by *agend,* no error is reported by *agend.*

3.11.4 Agent Interfaces

Associated with each agent, there is an agent interface. The agent interface maintains several queue-pairs for sending commands to agents. Each queue-pair has two queues a queue of messages to be sent to an agent, and a queue of agents waiting for messages. Only one of these will be non-empty at any time. For each queue-pair, there are two functions; a local function and an associated function that is exported for remote invocation by the agent.

The local function takes a message as an argument and either dequeues an agent and sends the message to it, or, if the agent queue is empty, it queues the message. The RPC function takes no arguments and dequeues and returns a message to the agent, or, if the message queue is empty, it queues the agent. This arrangement allows multiple instances of the same agent to efficiently handle many messages. The current version only ever starts one instance of each agent.

There five of these queue-pairs for five of the agents. The other two agents report information to the *SOUI* server and consequently do not need a queue-pair.

1. email queue, used by *agent_email,*
   accessed by `email_alert()` and `get_email()`

2. server queue, used by *agent_server,*
   accessed by `control_server()` and `get_control_server()`

3. target queue, used by *agent_target,*
   accessed by `control_target()` and `get_control_target()`

4. batch queue, used by *agent_batch(),*
   accessed by `start_test_analysis()` and `get_start_test_analysis()`

5. ar_storage queue, used by em agent_save,
   accessed by `control_ar_storage()` and `get_control_ar_storage()`

For all of these agents except *agent_email,* there is also an error and status reporting RPC function that is passed on to the *SOUI* server. These are `server_error()`, `target_error()`, `test_analysis_status()`, and `ar_storage_error()` defined in Section 3.7.

There are three information-reporting RPC functions (defined in Section 3.7), `put_alert()`, `put_status()`, and `put_seqno()`, that are passed on to the *SOUI* server for it to display.

4 Prototype Data Files

The following files/directories are required for the NIDES prototype to operate successfully.

- `/etc/security/audit`
  Directory `/etc/security/audit` contains C2 audit files that are read by *agen*. These files are created by the system audit daemon on the various target hosts.

- `/var/adm/pacct`
  Directory `/var/adm/pacct` contains accounting files that are read by *agen*. These files are created by the system accounting daemons on the various target hosts.

- `/etc/passwd`
  File `/etc/passwd` is used by *agen* to resolve numeric userids into user names. The file should exist on each target host.

- `storage`
  Directory `storage` is read from and written into by the persistent store infrastructural component. In particular, it uses subdirectories `kb`, `profile`, and `stats_config`. Directory `storage` itself is under `<IDES_ROOT>` which is a site-dependent environment variable.

- `rb_config`
  File `rb_config` is read by the rulebased component. It contains site-specific information that gets stored in the knowledge base of the rulebased component. It is stored under `<IDES_ROOT>/etc` where `<IDES_ROOT>` is a site-dependent environment variable.

- `IDES_stats_config`
  File `IDES_stats_config` is read by the statistical component. It contains statistics customization data. It is created at system initialization. It is stored under `<IDES_ROOT>/storage/stats_config`.

5 Requirements Traceability

The requirements for the NIDES prototype are given along with the extent to which they have been satisfied [3].

- **Acceptable detection performance:** *Minimal false positives and maximal true positives.*
  - 1% to 5% false positives for the statistical component.
  - reporting of eight known intrusion types using the rulebased component.
- **Real-Time Operation:**
  Anomaly detection almost within minutes of occurrence.

  Processing typically completed within 15 seconds of audit-data reception; it could take longer depending on volume of audit data.
- **Portability:**
  Straightforward migration to different hardware and different operating systems.

  All NIDES-specific software is in ANSI C and all infrastructural facilities are established or *de facto* standards.
- **Usability:**
  Simple, flexible and comprehensive user interface for the security officer.

  X-based graphical user interface with both online and detailed written documentation.
- **Open:**
  Ability to enhance existing capabilities, incorporate new capabilities and extend target environment.

  Architecture facilitates addition and enhancement of core components as well as expansion of target environment.
- **Scalability:**
  Maintain level of performance for increasing rates of audit data generation in the target environment.

  The prototype is capable of processing audit data arriving at the rate of 50 audit records per second without degrading real-time performance.

6 Differences between NIDES and IDES Prototypes

The essential differences between the NIDES prototype and the IDES prototype [5] are as follows.

1. The NIDES prototype is designed to be a coherent, integrated system using components with well-defined interfaces. The IDES prototype was less integrated and used ad-hoc interfaces to its components.

2. The NIDES prototype makes a distinction between core and infrastructural components as well as between a component and its embodiment as a process. The IDES prototype made no such distinctions.

3. The NIDES prototype is highly portable and extendable, whereas the IDES prototype, due to its dependencies on operating system idiosyncracies, was less so.

4. The NIDES prototype has a comprehensive security-officer user interface that allows observance of anomalous and suspicious activity as well as management of the prototype system and the target hosts that report to it. It also has a well integrated test facility for conducting experiments. The IDES prototypes user interface, while useful, was disjointed.

5. New statistical algorithms are used by the statistical component of the NIDES prototype. The algorithms used in NIDES are described in [5]. The algorithms used in IDES are described in [4].

6. NIDES includes an augmented rulebase over what was provided with IDES.

7 Referenced Documents

[1] Charles L. Forgy. *Rete: A Fast Algorithm for the Many Pattern/Many Object Pattern Match Problem.* Technical report, Carnegie-Mellon University, Pittsburgh, Pennsylvania, 1982.

[2] D. Heller. *Motif Programming Manual.* OReilly and Associates, 632 Petaluma Avenue, Sebastopol, California 95472, OSF/Motif Version 1.1 edition, September 1991.

[3] R. Jagannathan, T.F. Lunt, F.M. Gilham, A.F. Tamaru, C.F. Jalali, P.G. Neumann, D.A. Anderson, T.D. Garvey, and J.D. Lowrance. *Requirements Specification: Next-Generation Intrusion Detection Expert System (NIDES).* SRI Project 3131 Deliverable, September 1992. Contract Number N0039-92-C-0015.

[4] T.F. Lunt, A.F. Tamaru, F.M. Gilham, R. Jagannathan, C.F. Jalali, H.S. Javitz, A. Valdes, and P.G. Neumann. *A Real-Time Intrusion-Detection Expert System, Interim Report.* Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1990.

[5] T.F. Lunt, A.F. Tamaru, F.M. Gilham, R. Jagannathan, C.F. Jalali, H.S. Javitz, A. Valdes, P.G. Neumann, and T.D. Garvey. *A Real-Time Intrusion-Detection Expert System, Final Report.* Technical report, Computer Science Laboratory, SRI International, Menlo Park, California, 1992.

[6] *Network Programming Guide.* Sun Microsystems, Inc., Mountain View, California, Revision A of 27 March, 1990 edition. Part Number 800-3850-10.

8 Notes

Acronym or Term	Meaning
agen	audit generation client process
agend	audit generation daemon
agent	client process
arpool	audit collection server process
client	active process
IDES	Intrusion Detection Expert System
Motif	An X library
NIDES	Next-Generation Intrusion Detection Expert System
RPC	Remote Procedure Call
server	reactive process
SOUI	Security Officer User Interface
subject	entity on target host being audited
target host	computer system that is being monitored by NIDES
X	*de facto* graphical user interface standard
XDR	External Data Representation

APPENDIX

A NIDES Audit Record Format Description

The following describes the standard NIDES audit record format.

A.1 Structure of the NIDES Audit Record

The NIDES audit record can be declared by including `audit_rec.h`. This file includes `audit_rec_xdr.h` which is generated from a specification in `audit_rec_xdr.ax` using `arpcgen` described earlier (see page 9). The `arpcgen` utility generates routines to read and write the data structures associated with the NIDES audit record in a hardware-independent manner. These routines are described later in this section.

A.1.1 Contents of an NIDES Audit Record

The first nine data items of the NIDES audit record always exist. Rest of the data items of the NIDES audit record are optional; a data item exists if it is check-marked in the mark data item which always exists (see page 66).

The following describes the data items of the NIDES audit record.

`version`
: Audit record structure version number. This should be 4, i.e., the fourth version of the audit record structure.

`rseq`
: This is a sequence number which is monotonically increasing number that uniquely identifies an audit record for NIDES. When an audit record is first generated, `rseq` is the same as `tseq`; however, every time several sources of audit data are merged, this value is resequenced to preserve its properties.

`recvtime`
: This corresponds to the time stamp when this record was received by NIDES *(arpool* server).

`tseq`
: This is the target-host sequence number. It is a monotonically increasing number that uniquely identifies an audit record on a particular target-host.

`atime`
: This corresponds to the time stamp at which the audit record was generated on the target host. Note that the time stamp is determined by the clock on the target host and may, in some cases, exceed the time stamp indicated by `recvtime`.

`hostname`
This is the name of target host.

`audit_src`
This identifies the auditing subsystem that created the audit record. For example, it could be created from C2 auditing, or from accounting data, or from an application.

`action`
This describes the activity that resulted in an audit record to be generated.

`mark`
This is an array of bits whose length is the number of non-mandatory data items of this structure. For every optional data item in this structure that exists, the corresponding bit is set. For those optional data items that do not exist, the corresponding bits are not set.

`auname`
This corresponds to the actual user name. It is the users authenticated (actual) ID rather than the users current ID (see `uname`). For example, on Unix, this should not change with superuser enables (`su`).

`auname_label`
This is the security label associated with `auname`.

`uname`
This is the users current ID and might not correspond to the users actual ID (see `auname`).

`uname_label`
This is the security label associated with `uname`.

`pid`
This is Process ID on the target host that performed the action (as specified by `action`).

`ttyname`
This is the name of the terminal associated with the action.

`cmd`
This is the name of the command associated with the action.

`arglist`
This is the list of command arguments associated with the action.

`syscall`
This is the number of the system call or the operation code associated with the action.

`errno`
This is the error code from the action.

`rval`
This is the return value from this action.

`res_utime`
This is the user-CPU time for this action.

`res_stime`
This is the system-CPU time for this action.

`res_rtime`
This is the elapsed real time for this action.

`res_mem`
This is the amount of memory consumed in executing the action.

`res_io`
This is the amount of terminal I/O performed in executing the action.

`res_rw`
This is the amount of disk IO performed in executing the action.

`ouname`
This is the alternate user name, as in the argument of superuser enable (`su`).

`ouname_label`
This is the security label associated with `ouname`.

`remoteuname`
This is the remote user name for actions involving remotely-initiated activity.

`remoteuname_label`
This is the security label associated with `remoteuname`.

`remotehost`
This is the remote hostname for actions involving remotely initiated activity.

`path0`
This is a file name associated with the action.

`path0_type`
This is a file type of `path0`.

`path0_label`
This is the security label associated with `path0`.

`path1`
This is another file name associated with the action.

`path1_type`
This is a file type of `path1`.

`path1_label`
This is the security label associated with path0.

A.1.2 Data Structures

These data structures have been defined for use with the NIDES audit record.

`ia_seqno`
NIDES sequence numbers are represented in this way as a pair of 32 bit numbers yielding a 64 bit sequence number.

The following operators have been provided for this type.

void `IA_SEQNO_INC(ia_seqno *)` Increment the sequence number by one.

`int IA_SEQNO_EQL(ia_seqno *sn1, ia_seqno *sn2)` Compare the two sequence numbers for equality.

`ia_timeval`
NIDES time stamps represent seconds plus nanoseconds since 1970 GMT the seconds portion of this structure is compatible with a UNIX *time_t*.

`ia_label`
This is the NIDES security label.

`ia_ftype`
This specifies type of file as one of the following.

`IA_FTYPE_VOID:` an error condition.

`IA_FTYPE_REG:` a regular file.

`IA_FTYPE_TMP:` a temporary or scratch file.

`IA_FTYPE_PRIV:` a privileged file such as the UNIX password file.

`ia_audit_src`
This identifies source of the audit data.

`ia_audit_action` This represents the audited action or event using a set of predefined, system independent events.

`IA_VOID:` represents an undefined action that should be treated as an error.

`IA_DISCON:` target host lost contact with NIDES host (or vice versa).

`IA_ACCESS:` a catch-all file reference i.e., a file was referenced for a purpose other than defined by other actions.

`IA_OPEN:` a file was opened.

`IA_WRITE:` a file was written.

`IA_READ:` a file was read.

`IA_DELETE:` a file was deleted.

`IA_CREATE:` a file was created.

`IA_RMDIR:` a directory was deleted.

`IA_CHMOD:` the permissions, access control list, or dates of a file was changed.

`IA_EXEC:` a program was executed (initiated).

`IA_CHOWN:` ownership of a file was changed.

`IA_LINK:` a symbolic or hard link was made from one file to another where `path0` field denotes the original file and `path1` denotes the new file name.

`IA_CHDIR:` a user changed his working directory.

`IA_RENAME:` a file was renamed where `path0` denotes the original file name and `path1` denotes the new file name.

`IA_MKDIR:` a directory was created.

`IA_MOUNT:` a file system was mounted (imported).

`IA_UNMOUNT:` a file system was unmounted.

`IA_LOGIN:` a user has logged in.

`IA_BAD_LOGIN:` a login attempt has failed.

`IA_SU:` a user changed user IDs.

`IA_BAD_SU:` a user ID change failed.

`IA_RESOURCE:` no action occurred and only resource info is provided; this should probably be subsumed in `IA_UNCAT`.

`IA_EXIT:` a process terminated.

`IA_LOGOUT:` a user logged out.

`IA_UNCAT:` a catch-all for actions that do not fit into any other action.

`IA_RSH:` a successful remote shell (action) has occurred.

`IA_BAD_RSH:` a `IA_RSH` attempt has failed.

`IA_PASSWD:` a user has changed his password.

`IA_RMOUNT:` a file system has been mounted remotely (exported).

`IA_BAD_RMOUNT:` an `IA_RMOUNT` has failed.

`IA_PASSWD_AUTH:` a username/password tuple has been verified and matched.

`IA_BAD_PASSWD_AUTH:` a username/password tuple has been verified and mismatched.

A.2 Mark Structure

The *mark* field in the NIDES audit record is used to specify which fields in the NIDES audit record are valid for each audit record. The following macros are declared in *audit_rec.h* and may be used to access the mark structure.

`IA_MARK_SET(ia_audit_rec *, ia_mark_e)`
: Set the mark associated with `field_id`. Field *field_id* is any one of the constants defined in *<IDES_ROOT>/include/audit_rec.h* of the form `IA_M_*`.

`IA_MARK_CLR(ia_audit_rec *, ia_mark_id)`
: Clear the mark associated with `ia_mark_id`.

`IA_MARK_ISSET(const ia_audit_rec *, field_id)`
: Test the mark associated with `field_id`. Returns 1 if the mark is set, else 0.

`IA_MARK_ZERO(ia_audit_rec *)`
: Clear all marks in this audit record.

A.3 Reading and Writing Audit Records

The following functions are generated by *arpcgen* to read and write an NIDES audit record:

```
int rxdr_ia_audit_rec(XDR *, ia_audit_rec *, void *)
int wxdr_ia_audit_rec(XDR *, const ia_audit_rec *, void *)
```

For general purpose I/O on a UNIX file descriptor, the above XDR structure must be initialized and destroyed using these provided functions.

```
int xdr_fdinit(XDR *, int fd)
void xdr_fdend(XDR *)
```