SRI International

Software Design Document: A002 • July 11, 1994
Version Description Document: A005 • July 11, 1994

Next Generation Intrusion Detection Expert System (NIDES)

Software Design, Product Specification, and Version Description Document

SRI Project 3131
Contract N00039-92-C-0015

Debra Anderson, Computer Science Laboratory
Thane Frivold, System Technology Division
Ann Tamaru, Computer Science Laboratory
Alphonso Valdes, Applied Electromagnetics and Optics Laboratory

Prepared for:

Department of the Navy
Space and Naval Warfare Systems Command
2451 Crystal Drive
Arlington, VA 22245-5200
Attn: SPAWAR, PD51E2, Mr. Robert Patton
SPAWAR, 02-22D
NRaD, Code 412
NRL, Code 5540
NSA, R23

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

Contents

1 Scope 1

2 Prototype Design 3
2.1 Architecture 3
2.1.1 Core Components 3
2.1.2 Infrastructural Components 4
2.2 Component Integration 6
2.2.1 *UI* Server 6
2.2.2 *Analysis* Server 9
2.2.3 *Arpool* Server 9

3 Detailed Design 11
3.1 Infrastructural Components 11
3.1.1 Process Management Component 11
3.1.2 Inter-Process Communication Component 11
3.1.2.1 Client-side Routines 11
3.1.2.2 Server-side Routines 12
3.1.2.3 XDR Routines 13
3.1.3 Persistent Storage Component 15
3.1.3.1 Persistent Storage Facility 15
3.1.3.2 Data Management Facility 19
3.1.4 Graphical User-Interface Component 23
3.2 Audit-Data Generation Component 24
3.2.1 Audit Data Gathering 24
3.2.2 Audit-Data Conversion 25
3.2.3 Data Structures 25
3.3 Audit-Data Collection Component 26
3.4 Statistical Component 28
3.4.1 Algorithm Summary 28
3.4.2 Data Structures 30
3.4.2.1 Profiles 30
3.4.2.2 Activity Data 34
3.4.2.3 Configuration Data 35
3.4.3 Functional Interfaces 37
3.5 Rulebased Component 40
3.5.1 Functionality 41
3.5.2 Data Structures 42
3.5.3 Functional Interfaces 44
3.6 Resolver Component 46
3.7 User Interface Component 47
3.8 Audit Generation Service 52

3.8.1 Data Structures 52
3.8.2 Functional Interfaces 52
3.9 Audit Collection Service 53
3.9.1 Data Structures 53
3.9.2 Functional Interfaces 53
3.10 Analysis Service 54
3.10.1 Statistical client 54
3.10.2 Rulebased client 55
3.11 Security Officer User Interface Service 55
3.11.1 Data Structures 56
3.11.2 Functional Interfaces 57
3.11.3 Agents. 58
3.11.4 Agent Interfaces 61

4 Data Files 63

5 Requirements Traceability 65

6 Differences between NIDES Beta and Alpha Prototypes 67

A NIDES Audit Record Format Description 69
A.1 Structure of the NIDES Audit Record 69
A.2 Mark Structure 74
A.3 Reading and Writing Audit Records 74

Glossary 75

References 79

List of Figures

1 Core Component Dependencies 5
2 Client/Server Graph 7

1 Scope

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 rulebased analysis of the audit data is continuously performed, and the results are resolved and reported to a graphical user interface.

This document, which provides a comprehensive description of the NIDES prototype software, is organized as follows. Section 2 presents an overview of the NIDES prototype, including a description of the prototype architecture, execution control, data flow, and resource requirements. It also discusses 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 and the 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 beta prototype and the NIDES alpha prototype. The Appendix contains a description of the NIDES audit record format. A glossary of terms and a list of references is also included.

2 Prototype Design

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

The prototype has two external interfaces:

1. Audit and accounting files on target hosts that are created by resident Sun OS C2/BSM[1] auditing and UNIX accounting daemons (system processes) under a predetermined directory on the file system on the target host.

2. An X/Motif-based 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 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[2] 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 component implementation from other core components that depend on it. Similarly, each infrastructural component has well-defined interfaces that decouple the component's use from its internals.

2.1.1 Core Components

The core components of the NIDES prototype, shown in the dependency graph of Figure 1, are:

- Audit-data generation component
- Audit-data collection component
- Statistical analysis component
- Rulebased analysis component
- Resolver component

---

[1] All product names mentioned in this document are the trademarks of their respective holders.

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

- Archiver component
- User interface component

The audit-data generation component generates NIDES format audit records of subject (user) activities on a target system from SunOS BSM, SunOS C2, and standard UNIX accounting audit data. 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 of only after it has been processed by each of the analysis components (statistical and rulebased).

The statistical component detects unusual activity, which may be indicative of masquerading users.

The rulebased component detects "well-known" types of intrusive, suspicious, or non-authorized user behavior.

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

The archiver component archives NIDES audit records into a structured set of archive files.

The user interface component enables the following.

1. Real-time operation of NIDES, including displaying and reporting of alerts, selecting target hosts to be monitored, reporting status of monitored target hosts, dynamic reconfiguration of statistical parameters, dynamic control of individual rules, and display of archived alerts and NIDES audit records
2. Processing of previously recorded NIDES audit data for after-the-fact analysis, experimentation, and configuration tailoring

The 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 host systems, and on the audit-data generation component for initiation and termination of audit-data generation. 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 archiver component depends on the audit-data collection component. The audit-data collection component obtains audit data from the various audit-data generation components.

2.1.2 Infrastructural Components

The infrastructural components for realizing each core component of the NIDES prototype are

- Process management component

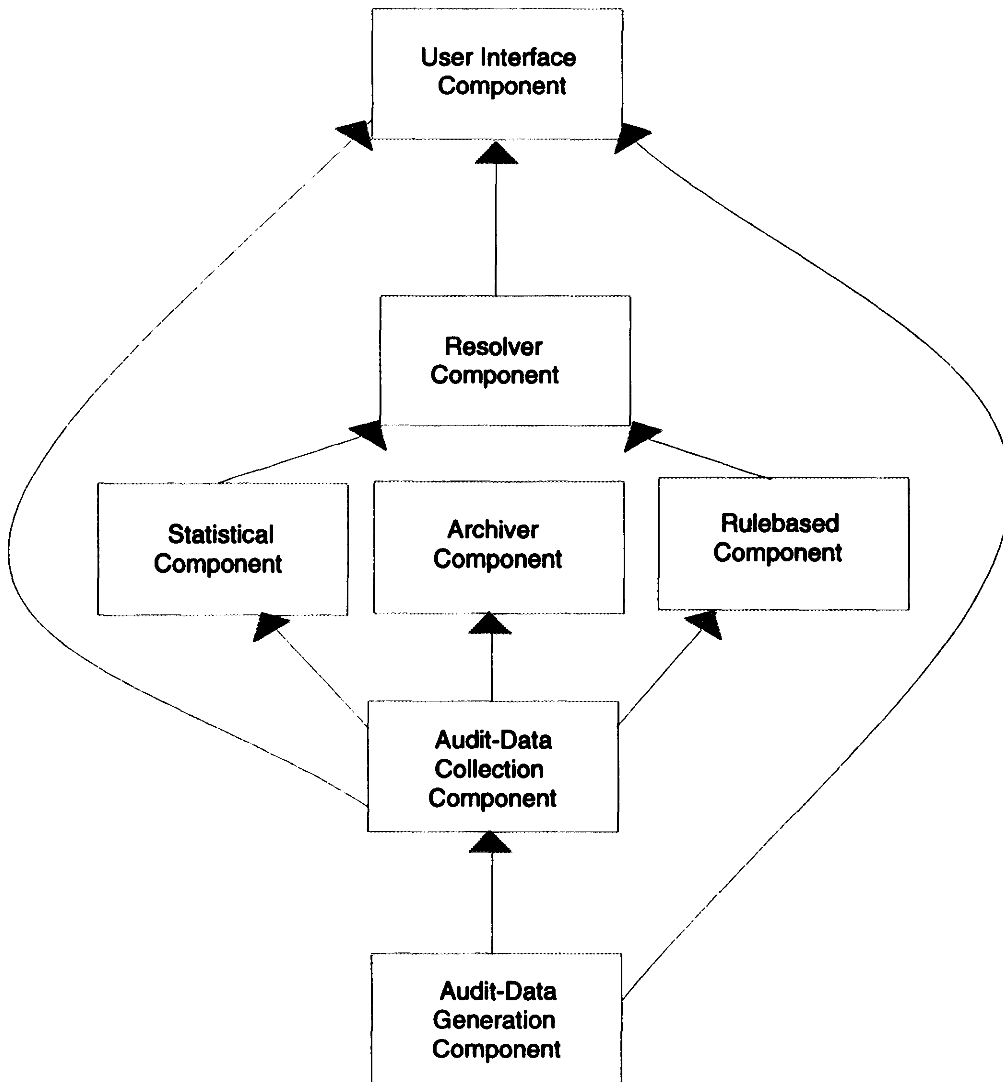

Figure 1: Core Component Dependencies

- Inter-process communication component
- Persistent storage component
- 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.

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 storage 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 storage facility is implemented using Sun's Network File System (NFS).

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

2.2 Component Integration

The core components are integrated by the infrastructural components.

The NIDES prototype is a collection of servers and clients, as shown in Figure 2. There are three servers: *UI, Analysis,* and *Arpool.*

2.2.1 *UI* Server

The *UI* 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 *UI* server, not be indefinitely blocked. The strict client/server model allows blocking to be avoided. The *UI* server is supported by seven clients, which we generically refer to as *agents.*

- Agent *agent_status* is responsible for obtaining the status and audit record counts for active target systems from *arpool.* Target status is obtained from the `Arpool`

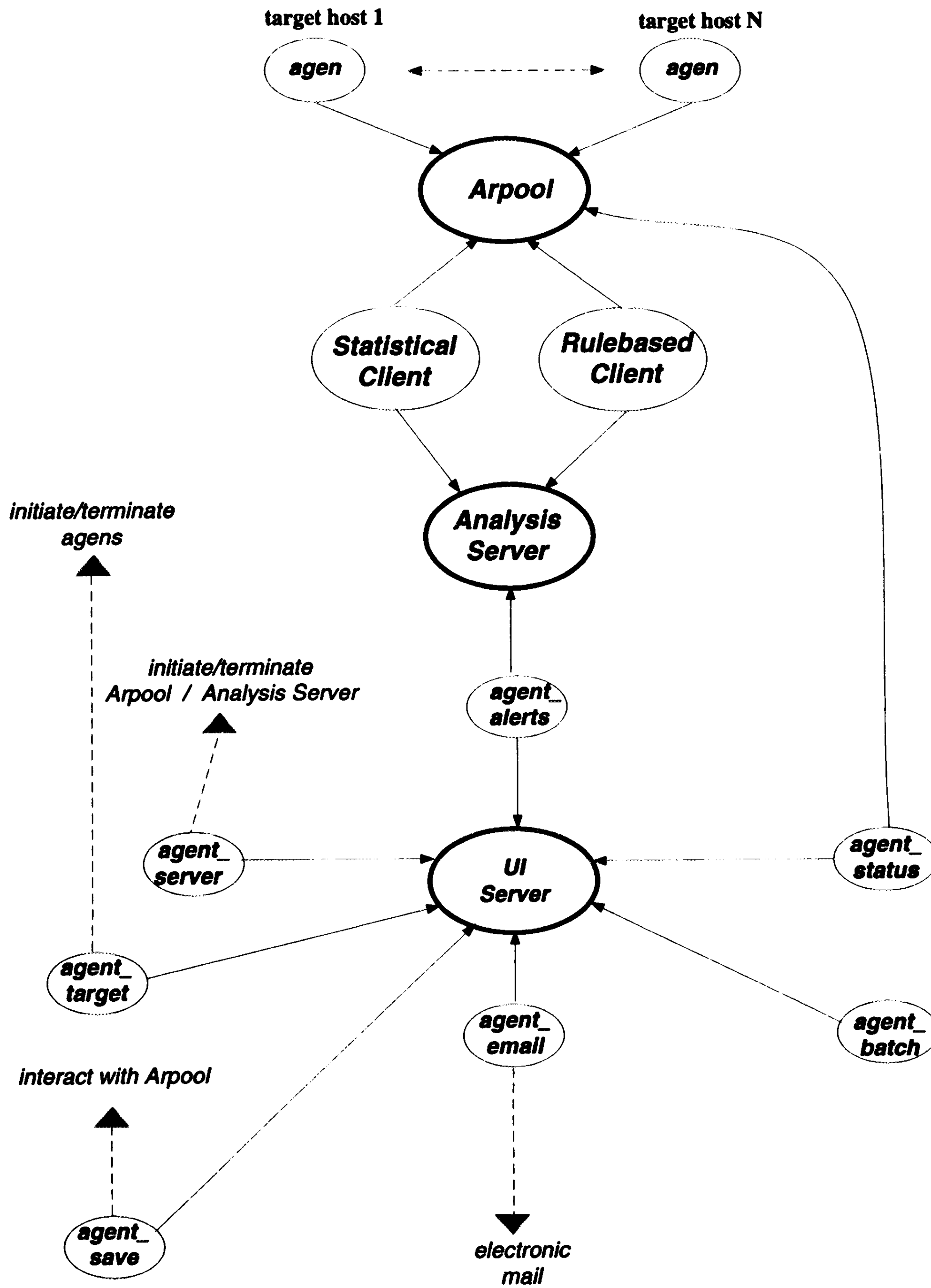

Figure 2: Client/Server Graph

server with the `arpool_get_status()` RPC and is reported to the *UI* server with the `put_status` RPC.

- In the alpha version of NIDES, agent *agent_alerts* was responsible only for transferring alerts from the *analysis* server to the *UI* server. In the beta version of NIDES, the role of *agent-alerts* has expanded to support two-way communication between the *analysis* server and the *UI* server.

  Information sent from the *UI* server to the *analysis* server includes alert filtering, real-time analysis reconfiguration, and manual profile update requests. Information sent from the *analysis* server to the *UI* server includes alert reports, alert statistics for individual target hosts, and reconfiguration status.

  Alert filtering is obtained from the *UI* server with the `get_alert_filter_list()` RPC and is reported to the *analysis* server with the `put_alert_fiIter_list()` RPC. Real-time analysis reconfiguration is obtained from the *UI* server with the `get_realtime_reconfig()` RPC and is reported to the *analysis* server with the `put_reconfig()` RPC. Manual profile update requests are obtained from the *UI* server with the `get_stats_client_update_list()` RPC and are reported to the *analysis* server with the `put_stats_update_list()` RPC. Alert reports are obtained from the *analysis* server with the `get_alerts()` RPC and are reported to the *U1* server with the `put_alert()` RPC. Alert statistics for individual target hosts are obtained from the *analysis* server with the `get_alert_stats()` RPC and are reported to the *UI* server with the `put_alert_stats()` RPC. Reconfiguration status is obtained from the *analysis* server with the `get_reconfig_status ()` RPC and is reported to the *UI* server with the `put_client_reconfig_status()` RPC.

- Agent *agent_server* is responsible for starting and stopping the analysis processes — namely, the *arpool* and *analysis* servers. Start and stop requests are obtained from the *UI* server with the `get_control_server()` RPC. Status and error information concerning a request is reported to the *UI* server with the `put_server_error` RPC.

- Agent *agent_target* is responsible for processing requests to start and stop the audit generation process on target hosts — namely, *agen.* Start and stop requests are obtained from the *UI* server with the get_control_target ( ) RPC. Status and error information concerning a request is reported to the *UI* server with the put_target_error RPC. The actual starting and stopping of audit generation activity on a target system is performed by a target hosts server — namely, *agend.*

- Agent *agent_save* is responsible for starting and stopping the audit record archive processes — namely, *archiver.* Start and stop requests are obtained from the *UI* server with the `get_ar_control_storage()` RPC. Status and error information concerning a request is reported to the *UI* server with the `ar_storage_error()` RPC.

- Agent *agent_email* is responsible for issuing e-mail alerts. Alert messages and a list of recipients are obtained from the *UI* server with the `email_alert ()` RPC. There is no mechanism to report errors.

- Agent *agent_batch* is responsible for initiating NIDES tests with archived audit data. Test parameters are obtained from the *U1* server with the `get_start_test_analysis()` RPC. Status and error information concerning a request is reported to the *UI* server with the `test_analysis_status()` RPC.

2.2.2 *Analysis* Server

The *analysis* server provides RPC interfaces for both *agent_alerts* and the analysis components — namely, the *statistical* client and the *rulebased* client. The agent interface includes the `get_alerts()`, `put_alert_filter_list()`, `put_reconfig()`, `put_stats_update_list()`, `get_alert_stats()`, and `get_reconfig_status()` RPCs described above.

The *statistical* client reports the results of statistical analysis with the `put_stats_results()` RPC to the *analysis* server. Similarly, the *rulebased* client reports the results of rulebased analysis with the `put_rulebase_results()` RPC to the *analysis* server. In addition, the *statistical* and *rulebased* clients make use of the persistent storage facility to store statistical and rulebased information (see Section 2.1.2).

2.2.3 *Arpool* Server

The *arpool* server provides RPC interfaces for *agent_status,* both *statistical* and *rulebased* clients, and the audit data generating clients — namely, *agen.*

The agent interface includes the `arpool_get_status()` RPC described above.

The *statistical* and *rulebased* clients obtain NIDES audit records from the *arpool* server with the `arpool_get_ar_vec()` RPC. The *arpool* server retains each audit record until all currently active clients have retrieved the record.

The *agen* processes running on multiple target systems report NIDES audit records to the *arpool* server with the `arpool_put_ar_vec()` RPC.

3 Detailed Design

Our description of the NIDES prototype first considers the infrastructural components, and then defines 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 Sun's Networking Programming Guide [7].

3.1.2 Inter-Process Communication Component

The Inter-process communication is based primarily on Remote Procedure Calls, similar to those defined by Sun with the `rpcgen`[3] tool. Our RPC tool `arpcgen` supports many more features than Sun's tool, and accepts ANSI C, rather than Sun's 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.

3.1.2.1 Client-side Routines The client-side routines are as follows:

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

`Status ipc_clnt_close(IPC ipc)`
: Closes a connection opened with `ipc_clnt_open()`.

`IPC ipc_set_server(IPC server)`
: 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)`
: 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.

[3] See the Sun Network Programming Guide [7], pp 33-146.

3.1.2.2 Server-side Routines

The server-side routines are as follows:

`void ipc_svc_dispatch(IPC from)`
: 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))`
: 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)`
: 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)`
: Closes a connection. The client in question will see a failure on its IPC handle.

`void ipc_svc_shutdown()`
: 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)`
: 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)`
: 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()` and

`void XrpcAppInit(XtAppContext ctxt)`
: Special functions that 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.

3.1.2.3 XDR Routines The XDR routines are defined in a manner in which 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 handle, or even whether it is a read or a write routine.

The following basic low-level routines translate from XDR format to machine format. Each routine takes an `ipc` channel, a pointer to the place to put the read object, and an extra pointer that is ignored.

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 following low-level routines translate from machine format to XDR format. Each routine takes an `ipc` channel, a pointer to the machine object, and an extra pointer that 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 following routines 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).

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

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

The following 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)

The following routines translate a type `void` object.

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

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

The following routines 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.

```
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 following routine fills in a `struct xdr_vector_info`.

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

The following routines, which are analogous to the `xdr_vector` routines described earlier, deal with pointers to a single object of some type.

```
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)
```

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 about 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 be sure not to make an RPC to a server that does not support it.

3.1.3 Persistent Storage Component

The persistent storage component of the NIDES beta release is divided into two subcomponents, a persistent storage subcomponent which is based on the NIDES alpha release persistent storage component, and a new data management facility subcomponent (DMF).

3.1.3.1 Persistent Storage Facility

The persistent storage facility allows an arbitrary hierarchical naming scheme, which is used to make independent *instances* in the beta version. The persistent storage facility defines five data types, used to store configuration and reconfiguration information.

```
typedef struct instance_config {
char *comment ;
long timestamp, last_arec_timestamp;
```

```
int result_filter;
} instance_config;


typedef struct Stats_config {
    struct Name_list    *no_updates;
    unsigned long update_offset;
    struct profile_config *pconfig;
    int update_method,cache_size;
} Stats_config;

typedef struct Rulebase_config {
long nactions;
struct config_action actions[nactions];
} Rulebase_config;


typedef struct anal_config {
struct instance_config config;
struct Stats_config stconfig;
struct Rulebase_config rbconfig;
} anal_config;


typedef struct anal_reconfig {
struct Stats_reconfig *stats_reconfig;
struct Rulebase_reconfig *rulebase_reconfig;
int result_filter;
} anal_reconfig;
```

The persistent storage facility provides the following library functions that allow manipulation of persistent storage data.

`Status get_list_of_instance_names( Name_list **ilist )`
: Reads the list of currently available NIDES instances into `ilist`.

`Status copy_instance( string to_instance, string from_instance )`
: Copies the NIDES instance `from_instance` into `to_instance`.

`Status create_instance( string instance )`
: Creates the NIDES instance `instance` initialized appropriately.

`Status delete_instance( string instance )`
Deletes the NIDES instance `instance`.

`Status delete_stats_profile( string instance ,string profile)`
Deletes the NIDES profile for subject profile contained in instance `instance`.

`Status get_list_of_subjects(const string instance, Name_list **list)`
Reads the list of subjects who have profiles in `instance` into the list `list`.

`Status read_current_profile(string instance, const string profile_name, Curr_prof_struct *curr_profile )`
Reads the current profile from instance `instance` for the subject named `profile_name` into `curr_profile`.

`Status write_current_profile( string instance, const string profile_name, const Curr_prof_struct *curr_profile )`
Writes the current profile `curr_profile` for subject named `profile_name` for the instance `instance`.

`Status read_historical_profile(string instance, const string profile_name, Hist_prof_struct *hist_profile )`
Reads the historical profile for subject `profile_name` into `hist_profile` from instance `instance`.

`Status write_historical_profile(string instance, const string profile_name, const Hist_prof_struct *hist_profile )`
Writes the historical profile `hist_profile` for subject named `profile_name` into instance `instance`.

`Status read_stats_config( const string instance, Statconfig_struct *config )`
Reads the statistics configuration for instance `instance` into `config`.

`Status write_stats_config( const string instance, const Statconfig_struct *config )`
Writes the statistics configuration `config` into instance `instance`.

`Status delete_stats_config( const string instance, const Statconfig_struct *config )`
Deletes the statistics configuration `config` from instance `instance`.

`Status read_rulebase_config( const string instance, struct Rulebase_config *config )`
Reads the rulebase configuration for instance `instance` into `config`.

`Status write_rulebase_config( const string instance, struct Rulebase_config *config )`
Writes the rulebase configuration `config` into instance `instance`.

`Status read_kb( const string instance, struct rulebase *kb )`
: Reads knowledge base from instance `instance` into `kb`.

`Status write_kb( const string instance, struct rulebase *kb )`
: Writes the knowledge base `kb` into instance `instance`.

`Status read_instance_config( const string instance, struct instance_config *i )`
: Reads the instance configuration into `i` from instance `instance`.

`Status write_instance_config( const string instance, struct instance_config *i )`
: Writes the instance configuration `i` into instance `instance`.

`Status read_anal_config( const string instance, struct anal_config *i )`
: Reads the analysis configuration into `i` from instance `instance`.

`Status write_anal_config( const string instance, struct anal_config *i)`
: Writes the analysis configuration `i` into instance `instance`.

`Status read_instance_reconfig( const string instance, struct anal_reconfig *i )`
: Reads the instance reconfiguration into `i` from instance `instance`.

`Status write_instance_reconfig( const string instance, struct anal_reconfig *i )`
: Writes the instance reconfiguration `i` into instance `instance`.

`Status delete_instance_reconfig( const string instance`
: Deletes the instance reconfiguration for instance `instance`.

`Status read_recipient_list ( struct Name_list *list )`
: Reads the recipients list and places it in list.

`Status read_targethost_list ( struct Name_list *list )`
: Reads the target host list and places it in `list`.

`Status read_priv_user_list ( struct Name_list *list )`
: Reads the privileged user list and places it in `list`.

`Status read_alert_filter_list ( struct Nameint_list *list )`
: Reads the alert filter list and places it in `list`.

`Status read_mandatory_rulelist ( struct Name_list *list )`
: Reads the mandatory rule list and places it in `list`.

`Status write_recipient_list( struct Name_list *list )`
: Writes the recipients list `list`.

`Status write_targethost_list ( struct Name_list *list )`
: Writes the target, host list `list`.

`Status write_priv_user_list( struct Name_list *list )`
Writes the privileged user list `list`.

`Status write_alert_filter_list( struct Nameint_list *list )`
Writes the alert filter list `list` .

`Status write_mandatory_rulelist( struct Name_list *list)`
Writes the mandatory rule list `list`.

`Status delete_alert_filter_list()`
Deletes the alert filter list.

`Status privileged_user( )`
Determines whether the user running `nides` is privileged.

3.1.3.2 Data Management Facility

The DMF is maintained as a library of routines separate from the other persistent storage routines; however, the DMF is an integral part of the overall persistent storage component. The main features that distinguish the DMF functionality from the persistent storage facility routines are:

- The data stored by the DMF is presumed to be voluminous, and thus must be compressed in order to save disk space.
- The data stored by the DMF must be easily accessible once stored, and thus must be indexed according to major search criteria.

The DMF does not define any new data types for storage purposes; however, it does define the following new data types used for reading and writing existing data structures.

```
typedef enum
{
    DMF_ERROR        = -3, /* indicates various errors               */
    DMF_NOT_FOUND    = -2, /* indicates certain items were not found */
    DMF_NO_MORE_DATA = -1, /* indicates no more data available       */
    DMF_OK           =  0  /* indicates successful execution         */
} dmf_return_codes;

typedef enum
{
    DMF_RULEBASE   = 1,
    DMF_STATISTICS = 2,
    DMF_BOTH       = 3
```

```
} component;

struct Interval
{
    time_t start;
    time_t end;
};

typedef struct _Handle *handle;
typedef struct _ListHandle *list_handle;
typedef struct Interval *interval;
```

The DMF provides the following library functions that allow reading and writing of NIDES specific data structures.

`create_audit_database(const string dbname, const string root)`
: Takes a database name string and creates a database for storing NIDES audit records. If the value of `root` is not `NULL,` the value will be used as the "root" path for the database; otherwise, the appropriate path will be determined from the `IDES_ROOT` environment variable. This routine returns `DMF_OK` for success, or DMF error codes for failure.

`create_result_database(const string dbname, const string root)`
: Takes a database name string and creates a database for storing NIDES result records. If the value of `root` is not `NULL,` the value will be used as the "root" path for the database; otherwise, the appropriate path will be determined from the `IDES_ROOT` environment variable. This routine returns `DMF_OK` for success, or DMF error codes for failure.

`create_alert_database(const string dbname, const string root)`
: Takes a database name string and creates a database for storing NIDES alert records. If the value of `root` is not `NULL,` the value will be used as the "root" path for the database; otherwise, the appropriate path will be determined from the `IDES_ROOT` environment variable. This routine returns `DMF_OK` for success, or DMF error codes for failure.

`handle new_audit_handle(const string dbname, const string root)`
: Creates a `handle` that is used in subsequent DMF routines for both reading and writing of `ia_audit_rec` data structures.

`handle new_result_handle(const string dbname, const string root)`
: Creates a `handle` that is used in subsequent DMF routines for both reading and writing of `AnalResult` data structures.

`handle new_alert_handle(const string dbname, const string root)`
Creates a `handle` that is used in subsequent DMF routines for both reading and writing of `AnalResult` data structures.

`void close_handle(handle h)`
Closes and frees a handle to any type of DMF database. If the handle was used for writing data, all data is flushed and all open files are closed.

`list_handle new_audit_list_handle(const string dbname, const string root)`
Creates a `list_handle` that is used in subsequent DMF routines for both reading and writing of `ia_audit_rec` data structures.

`list_handle new_result_list_handle(const string dbname, const string root)`
Creates a `list_handle` that is used in subsequent DMF routines for both reading and writing of `AnalResult` data structures.

`list_handle new_alert_list_handle(const string dbname, const string root)`
Creates a `list_handle` that is used in subsequent DMF routines for both reading and writing of `AnalResult` data structures.

`void close_list_handle(list_handle lh)`
Closes and frees a list_handle to any type of DMF database. If the list_handle was used for writing data, all data is flushed and all open files are closed.

`int write_audit_rec(ia_audit_rec *ar, handle *h)`
Writes the provided audit record data to the DMF database identified by the provided handle. The appropriate compression and indexing is performed automatically.

`int write_result_rec(struct AnalResult *rr, handle *h)`
Writes the provided result record data to the DMF database identified by the provided handle. The appropriate compression and indexing is performed automatically.

`int write_alert_rec(struct AnalResult *rr, handle *h)`
Writes the provided alert record data to the DMF database identified by the provided handle. The appropriate compression and indexing is performed automatically.

`int select_audit_recs(const string user, const interval time_int, const int countflag, int *count, handle h)`
Indicates the criteria for selecting audit records for a single user. The user and time interval must be specified. The number of records found can be obtained by setting the `countflag` variable; the total number of records will be written to the integer indicated by the `count` variable. The provided handle will be modified to reflect the selection criteria; the handle should not be modified until the desired records have been acquired (using `get_audit_rec()`).

`int select_result_recs(const string user, const interval time_int, const int countflag, int *count, handle h)`
Indicates the criteria for selecting result records for a single user. The user and time interval must be specified. The number of records found can be obtained by setting the `countflag` variable; the total number of records will be written to the integer indicated by the `count` variable. The provided handle will be modified to reflect the selection criteria; the handle should not be modified until the desired records have been acquired (using `get_result_rec()`).

`int select_result_recs(const string user, const interval time_int, const component comp, const int countflag, int *count, handle h)`
Indicates the criteria for selecting result records for a single user. The user, component, and time interval must be specified. The number of records found can be obtained by setting the `countflag` variable; the total number of records will be written to the integer indicated by the `count` variable. The provided handle will be modified to reflect the selection criteria; the handle should not be modified until the desired records have been acquired (using `get _alert_rec ( )`).

`int select_user_audit_recs(const int userc, string *userv, const interval time_int, const ia_audit_action action, const int countflag, int *count, list_handle lh)`
Indicates the criteria for selecting audit records for a list of users. The user list, audit action type, and time interval must be specified. The number of records found can be obtained by setting the `countflag` variable; the total number of records will be written to the integer indicated by the `count` variable. The provided list handle will be modified to reflect, the selection criteria; the list handle should not be modified until the desired records have been acquired (using `get_list_audit_rec()`).

`int select_user_result_recs(const int userc, string *userv, const interval time_int, const int countflag, int *count, list_handle lh)`
Indicates the criteria for selecting result records for a list of users. The user list and time interval must be specified. The number of records found can be obtained by setting the `countflag` variable; the total number of records will be written to the integer indicated by the `count` variable. The provided list handle will be modified to reflect the selection criteria; the list handle should not be modified until the desired records have been acquired (using `get_list_result_rec()`).

`int select_user_alert_recs (const int userc, string *userv, const interval time_int, const component comp, const int countflag, int *count, list_handle lh)`
Indicates the criteria for selecting result records for a list of users. The user list, component, and time interval must be specified. The number of records found can be obtained by setting the `countflag` variable; the total number of records will be written to the integer indicated by the `count` variable. The provided list handle will be

modified to reflect the selection criteria; the list handle should not be modified until the desired records have been acquired (using `get_list_alert_rec()`).

`int get_audit_rec(ia_audit_rec *ar, handle *hp)`
Acquires individual audit records, in chronological order, from a handle previously configured by a call to `select_audit_recs()`. The return value will be `DMF_OK` if the record was successfully acquired, `DMF_NO_MORE_DATA` if there are no more records meeting the selection criteria, or `DMF_ERROR` if an internal error was encountered.

`int get_result_rec(struct AnalResult *rr, handle *hp)`
Acquires individual result records, in chronological order, from a handle previously configured by a call to `select_result_recs()`. The return value will be `DMF_OK` if the record was successfully acquired, `DMF_NO_MORE_DATA` if there are no more records meeting the selection criteria, or `DMF_ERROR` if an internal error was encountered.

`int get_alert_rec(struct AnalResult *rr, handle *hp)`
Acquires individual alert records, in chronological order, from a handle previously configured by a call to `select_alert_recs()`. The return value will be `DMF_OK` if the record was successfully acquired, `DMF_NO_MORE_DATA` if there are no more records meeting the selection criteria, or `DMF_ERROR` if an internal error was encountered.

`int get_list_audit_rec(ia_audit_rec **ar, list_handle lh)`
Acquires individual audit records, in chronological order, from a list handle previously configured by a call to `select_user_audit_recs ()`. The return value will be `DMF_OK` if the record was successfully acquired, `DMF_NO_MORE_DATA` if there are no more records meeting the selection criteria, or `DMF_ERROR` if an internal error was encountered.

`int get_list_result_rec(AnalResult **rr, list_handle lh)`
Acquires individual alert records, in chronological order, from a list handle previously configured by a call to `select_user_alert_recs()`. The return value will be `DMF_OK` if the record was successfully acquired, `DMF_NO_MORE_DATA` if there are no more records meeting the selection criteria, or `DMF_ERROR` if an internal error was encountered.

`int get_list_alert_rec(AnalResult **rr, list_handle lh)`
Acquires individual alert records, in chronological order, from a list handle previously configured by a call to `select_user_alert_recs()`. The return value will be `DMF_OK` if the record was successfully acquired, `DMF_NO_MORE_DATA` if there are no more records meeting the selection criteria, or `DMF_ERROR` if an internal error was encountered.

3.1.4 Graphical User-Interface Component

Motif is an object-oriented set, of X-compatible programming tools that 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,* controls 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 [3].

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.

3.2.1 Audit Data Gathering

The audit data gathering module is a set of functions that 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 )` 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 )` 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 )` 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. This function must be able to deal with logging facilities that span multiple files. For example, C2 and BSM accounting can arbitrarily stop updating one file and begin writing into a new file. To deal with this, the module examines the audit data stream for file continuation records and polls for the existence of new accounting files.

3.2.2 Audit-Data Conversion

Currently, SunOS BSM, SunOS C2, and standard UNIX accounting audit data are sup_ ported. The conversion functions are as follows.

`int BSMProcessRecord(int fd)`
Converts a SunOS BSM audit record read from the file associated with `fd` into a list of NIDES audit records. This function returns 1 if any NIDES audit records were generated, 0 if no NIDES audit records were generated, and -1 on error.

`ia_node *BSMCurrentNodes()`
Returns the list of NIDES audit records generated by the last call to the function `BSMProcessRecord()`. This function returns a pointer to the list of available records, or a NULL pointer if no records are available.

`int c2_to_ialist(const audit_record_t *au, ia_node **rp)`
Converts a SunOS C2 audit record into a list of NIDES audit records, returned in the second parameter. This function returns 0 on success, or -1 on error.

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

3.2.3 Data Structures

The data structures used by the functions defined above are described below. The SunOS C2 audit record is defined, followed by the standard UNIX accounting record format. A structure for maintaining a list of NIDES-formatted audit records is defined last.

While both SunOS C2 and standard UNIX accounting audit data have fixed audit record formats, the SunOS BSM audit data format permits a large degree of variability by representing all audit data as a collection of attribute/value pairs where each attribute has a differing format. For that reason, unlike the SunOS C2 and standard UNIX conversion, the SunOS BSM audit data conversion generates NIDES audit records directly without going through a conversion to an intermediate format. The following data structures define the SunOS C2 and standard UNIX accounting audit data record formats.

```
/* 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/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;            /* NIDES-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 are 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 that 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 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 count, ia_audit_rec **avec)`

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 `avec` vector should be freed by the caller. This function returns TRUE if the pool is full, and otherwise returns FALSE.

`int get_records(client_info *client, ia_audit_rec **avec, long *count)`

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 `client` is an opaque data type used to identify the consumer making a given request.

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

3.4 Statistical Component

The NIDES statistical analysis component maintains historical statistical profiles for each user and raises an alarm when observed activity departs from established patterns of use for an individual. The historical profiles are updated regularly, and older data "aged" out with each profile update, so that NIDES adaptively learns what to expect from each user. This type of analysis is intended to detect intruders masquerading as legitimate users. Statistical analysis may also detect intruders who exploit previously unknown vulnerabilities and who cannot be detected by any other means. The statistical analysis is also expected to be particularly useful in detecting activity that may be indictative of fraud and abuse.

3.4.1 Algorithm Summary

The basic statistical approach in NIDES is to compare a user's short-term behavior to the user's 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 and 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 specifics of the algorithms implemented in the statistical component are discussed here; a more rigorous description of the algorithms can be found in [5].

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, and so forth. Similarly, a reasonable half-life for a long-term profile might be 30 days.

Aspects of subject behavior are represented as measures (for example, file access, CPU usage, hour of use). We refer to a subject's *profile* as the set of measure values associated with short-term and long-term behavior. We have classified the NIDES measures into five groups: activity intensity, audit record distribution, categorical, continuous, and binary. 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 continuous measures determine whether within a type of activity (say, accessing a file), the behavior over the recent past that affects that action is normal. The binary measures note if a particular type of activity occurred.

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 use 32 intervals for **Q** for each measure, with logarithmic 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 a simple mapping of the tail probabilities (which we call **TPROB**) of the distribution of **Q** onto that of a half-normal distribution.

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 subject's 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 a 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 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).

The statistical analysis is based upon many days of audit data processing. Like any other process that relies on empirical probability distributions, 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 (see [5]), the training concepts are implemented in the component. Some of these are described in Section 3.4.3.

3.4.2 Data Structures

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

```
#define MAXMEASURES 49            /* number of measures */
#define MAXMEASURES7 MAXMEASURES*7 /* 7x number of measures */
#define NUMBINS       32          /* number of bins for Q */
#define MINPROB       1.0/4096.0  /* minimum probability for categories */
#define CATRAREPROB   0.01        /* default rare category total probability */
```

The statistical component utilizes three types of data structures:

- Profiles
- Activity
- Configuration

3.4.2.1 Profiles

A subject's profile comprises three main data structures.

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` — An integer value that uniquely identifies the subject of this profile.
- `subjname` — The character string representation (also unique) that identifies the subject.

- `prevtstamp` — A long integer value representing the timestamp of the last audit record processed for this subject. This number represents the number of seconds elapsed since January 1, 1970 (see `ctime()` in any UNIX manual).
- `score_red` — The score that corresponds to the `thresh_red` false positive percentage value (which is found in the `profile_config` struct).
- `score_yellow` — The score that corresponds to the `thresh_yellow` false positive percentage value (which is found in the `profile_config` struct).
- `actvd_measures` — An array of integers that represent the active measures for the subject. Each element in the array represents one measure, and it is set to the logical OR of the measure's active/inactive and trained/untrained status.
- `q` — An array of type float 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.
- `nmeas_active` — Maintains the count of active measures (irrespective of training status). A measure is considered active when it is configured to contribute to statistical score calculation (anomaly dectection).
- `qcount` — A matrix of integers (`MAXMEASURES` by `NUMBINS`) 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.
- `cats` — An array of pointers to a list of categories for each measure. Each list is sorted in ascending order of category probability. See the description of `Catnode` for more detail (page 33).
- `s` — An array of type float of size `MAXMEASURES` that represents the **S** values for a profile. This array's elements are recomputed for every measure observed in each audit record.
- `dailycnt` — 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.
- `scorehistn` — The value of the aged count of scores that have been produced for this subject.
- `hashed_cats` — A hashed table for all the categories defined for this subject. This table is primarily used for quick access.
- `nextcatid` - 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` — An array of size of `MAXMEASURES7`. It keeps `track` of the daily score counts, and is reset to 0 at profile update time.

- neffn — An array of size MAXMEASURES that tracks the daily aged Effective-N (the counts in dailycnt do not reflect short-term aging). See the glossary and [l] for a description of Effective-N.

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 — An integer value that uniquely identifies the subject of this profile.
- subjname — The character string representation (also unique) that identifies the subject.
- lastupdate — The long integer value representing the time of the last historical profile update for this subject. This number represents the number of seconds elapsed since January 1, 1970 (see ctime() in any UNIX manual).
- nupdates — An integer value that tells the number of profile updates. It is incremented by 1 each time the profile is updated.
- qprob — A matrix (MAXMEASURES by NUMBINS) of type float representing the historical distribution of Q values within each bin (interval). This matrix is recomputed at profile update time using the daily counts accumulated in the qcount stored in the current profile.
- tprob — A matrix (MAXMEASURES by NUMBINS) of type float representing the tail probabilities of the qprob historical distribution. See Section 3.4.1 for a more detailed explanation. This matrix is recomputed at profile update time using the recomputed qprob values.
- rareprob — An array of size MAXMEASURES representing the sum of all the category probabilities for the categories classified as RARE. The sum of the probabilities does not exceed the user-configurable quantity maxsumrare (see page 35). It is computed at update time, and has a cap value of maxsumrare.
- maxrareprob — An array of size MAXMEASURES representing the maximum observed category probability of all categories classified into the RARE group. 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 maxsumrare .
- ncats — An array of size MAXMEASURES representing the aged number of categories for each measure. This value is used to "smooth" the normalization of **Q** during score computation. It is recalculated at profile update time, incorporating the most recent count of categories for each measure.
- histn — 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` — An array of size `MAXMEASURES7` representing the historical `T2` score distribution used to determine new score threshold values. It is recomputed at profile update time. The first 200 slots represent 0.1 score point, and the remainder of the array slots represent whole score points.
- `measactv_histn` — A smoothed estimate of the number of active measures as they vary from update to update. This is required to properly adjust the `T2` statistics as the number of measures contributing to the scoring changes. It is recomputed at profile update time.
- `upd_to_train` — An integer array of size `MAXMEASURES` that tracks the number of updates required in the current training phase.
- `tphase` — An integer array of size `MAXMEASURES` that reflects the training phase the measure is in, which is some combination of the predefined constants `TCATS` for category training, **TQ** for **Q** training, or **TT2** for **T2** distribution training.
- `effn_since_train` — A float array of size `MAXMEASURES` that tracks the Effective-N since the current training phase started. The phase is exited when both the updates to train go down to zero and the Effective-N since training began exceeds one third of the configurable constant `mineffn` (see the definition of struct `measure` on 34).

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

- `catid` – The integer code of this category. It is unique within a measure only.
- `cmid` — Specifies the measure that this `catnode` belongs to.

  The category id `catid` and `cmid` pair are unique throughout all the categories for all measures for a subject.
- `catname` – The character string identification of this category.
- `catprob` — The historically aged probability for this category within this measure. It is updated at profile update time.
- `catcount` — Tracks how many times this category was observed since the last profile update. It is reset to zero at profile update time.
- `agecnt` — The aged count for each category (i.e., how many times the category was observed, weighted by short-term aging). It is updated whenever it is observed in an audit record.
- `prevobscnt` — Tracks the count of the audit records (since the last update) when this category was last observed. Needed for the recursive Q calculation.
- `catflags` — A vector of bit flags that indicate any peculiarities for the category (such as a first-time seen category or whether it is in the RARE group).

- catnext — 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 — An integer value representing the measure id that is unique and should not be modified.
- mname — A mnemonic character string representation of the measure that should not be modified.
- mdesc — Stores a verbose description of the measure.
- mtype — Indicates the type of the measure, continuous, categorical, or binary. (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 — Indicates if the measure is activated (1 if active, 0 otherwise).
- mqmax — Contains a configurable upper limit on the Q value and is used to properly scale the intervals of Q for an "even" distribution (i.e., one that uses an adequate number of the preallocated Q bins).
- mscalar — A scalar value only used for continuous measures to evenly distribute the measure values across 32 bins.
- mweight — A weighting factor for the measure currently unused.
- mineffn — The minimum effective-n (number of observations weighted by the short-term aging factor) before a measure is allowed to contribute to scoring, regardless of the number of updates observed for the measure.
- short_hlife — The (user-configurable) short-term half-life in number of audit records for the measure.
- short_gamma — The short-term aging factor computed from the above short-term half-life used to age the observations.

Activity. Each audit record is transformed into a vector 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 — Identifies the measure to which this activity is mapped.

- `m_val` — The observed activity value for the measure that is a structure containing different data types. The 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 (e.g., name of a file, terminal, host). For binary measures, the activity is set to 1 if observed. If the measure activity is not observed in the 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 global 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 becomes meaningful again.

- `prof_hlife` — The long-term profile half-life that is the basis for long-term profile aging, represented in units of profile updates.
- `arec_gamma` — The aging factor applied to each count in the short-term profile, currently overridden by the `short_gamma` parameter in the `measure` struct.
- `prof_gamma` — The aging factor applied to historical counts in the long-term profile at update time, computed using `prof_hlife`.
- `corr_cutoff` — The correlation cutoff value for the correlation matrix, currently unused.
- `traindays` — The configurable minimum number of profile updates that are required for profile training.
- `maxsumrare` — The configurable maximum sum of probabilities for categories in the RARE group. At update time, all categories whose cumulative sum is less than or equal this value are grouped into the RARE category.
- `measures` — The table of measures for the statistical component, serving as the default configuration for a new subject.
- `nmeas_active` — Represents the number of activated measures.
- `statmode` — A bit field that indicates which modes the statistical component should be running. An example mode determines if 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 only for development and experimentation purposes.
- `command_classes` — Contains lists of special commands or hosts that have been assigned to a particular group (e.g., compilers, editors, local hosts). It is a hash table that contains all commands and hosts.

- `thresholds` — Contains the threshold levels used to determine when a score value should be reported to the security officer. These levels represent the percentiles where the score 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, and are user-configurable. These correspond to settings of 0.1 percent and 1 percent for the structure items `thresh_red` and `thresh_yellow`, respectively.
- `numtempdirs` — Tracks the number of temporary directories, that is, directories whose contents are considered "temporary files" and thus do not generate categories in the historical profile.
- `tempdirs` — A class item list containing directory prefixes of temporary directories (configurable through the tmp_dirs class list).

Stats_reconfig. This data structure contains reconfigured changes to the configuration structure. 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 becomes meaningful again.

- `type_chg` — An integer denoting the type of configuration changes requested.
- `prof_hlife, traindays, maxsumrare, thresh_yellow, thresh_red` — These fields potentially contain (depending on `type_chg`) new values for any subset of the corresponding fields in the `Statconfig_struct`.
- `cache_size` — Contains a new value for the profile cache size (in number of profiles cached during real-time operation) if a change to the cache size has been made.
- `update_method` — Contains a new specification of the update method (none, by system clock, or by audit record timestamp), if a change to the update method configuration has been made.
- `no_updates` — A list of subjects who should *not* have their profiles updated during the daily profile update.
- `update_offset` — The number of seconds past midnight that the daily profile update should occur.
- `meas_chg` — Contains a field marking the types of changes requested in `chg_mark`, followed by new values for any combination of `mflags, short_hlife, mineffn, qmax` or `scalar`.
- `chg_list` — Contains changes to class list items.

Several data structures, stored in lists or tables, 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

`Status make_activity_vector( const ia_audit_rec *audit_rec, const Hashnode *commd_classes[], const Hashnode *subj_commd_list[], int numtempdirs, Nameint_list *tempdirs[], long *prev_timestamp, Activity *activity_vec[] )`

This function creates the activity vector that represents the measures observed in the audit record under analysis. It extracts relevant information from the NIDES audit record and puts it into the activity vector. In some cases, some data conversion is performed (e.g., 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 inter-arrival time (used for the activity intensity measures and the inter-arrival measure) is computed from the timestamp of the audit record and the timestamp of the last audit record processed for the 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 (e.g., mailer, editor, compiler); 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 subject's 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 of the binary type, then the activity vector for this measure will contain either a 0 or 1 to indicate whether 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 is 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.

`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, float *score)` This function is the heart of the statistical anomaly detection analysis. It implements the algorithms that compare the subject's short-term profile against its 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 subject's audit record and updates the short-term profile for this subject. The short-term profile is calculated according to the type of measure (continuous, categorical, audit record distribution or activity intensity). In the case of continuous and categorical measures, this is done by finding the observed category for a 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 is updated for those measures that have been observed). The short-term profiles for the activity intensity measures are based on the amount of time that has elapsed since the last audit record for a subject (hence the inter-arrival parameter is needed).

Special consideration is given to never-seen-before categories and categories with a very low probability. For never-seen-before categories, we have a separate category; if appearances of never-seen-before categories reach a level that differs greatly from the long-term profile for the never-seen-before category, the **Q** (and hence **S**) value for the measure will be significantly high. RARE categories are scored as a group, using the cumulative observed sum of RARE probabilities.

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 long-term 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 abnormality. 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 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 long-term profile at the next profile update (see *update_profile()*).

`Status update_profile(const Statconfig_struct *config, Curr_prof_struct *curr_profile, Hist_prof_struct *hist_profile, long event_time )` 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 long-term profile is built for the 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 observa-

tion 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 a new Q distribution is computed. Finally, the T^2 score distribution table is updated in a similar 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()* — see page 37), the T^2 distribution table is represented in tenths of score points for the first 200, and whole numbers thereafter.

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 (rounding up to the next 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 also 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.

`Status make_def_profile( char *subjname, Curr_prof_struct *curr_prof, Histprof_struct *hist_prof, profile_config *config )` 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, continuous (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 (l/32 for continuous measures, `l/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 cat-

egory is preallocated in this function (with default probability of `MINPROB`). Finally, the hash table for all measure categories is created for the above predefined categories.

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

`Status evaluate_stat_reconfig( Stats_reconfig *recfg, profile_config *config)` This function evaluates the statistical reconfiguration for errors and inconsistencies. If it returns without error, reconfiguration items are applied. Some items can be immediately applied; others are deferred to the next update.

`Status apply_immediate_stat_reconfig( Stats_reconfig *recfg, profile_config *config)` This function takes the given statistical reconfiguration and applies those changes that can be done immediately (such as turning measures ON or OFF) to the subject's profile.

`Status apply_deferred_stat_reconfig( Stats_reconfig *recfg, profile_config *config)` This function applies reconfiguration changes that are done at profile update time, (e.g., measure and **Q** scaling parameters and new half-life values). 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.

`Status check_anomaly( float 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 value (defined in the subject's 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 an SRI developed rulebase tool (see [l]). 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 [2]). It also generates functions for asserting all the different types of facts into the knowledge base. Besides the code generated by the rulebase 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 Section 3.5.3.

3.5.1 Functionality

The rulebased component does its inference using a modified Rete algorithm. The main difference between the SRI rulebase implementation and the Rete algorithm as described by Forgy [2], is that SRI's tool 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, all of which are normally invisible to the NIDES programmer. These functions are called the `ante1`, `ante2`, `binding`, and `concl`. 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 these tests are performed 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 rule's 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 rule's 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 can't 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 SRI rulebase 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 (*antel)();
  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.

`Status init_kb( struct rulebase **kb )`

Initializes a knowledge base. 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.

`Status config_kb( struct rulelist **kb, const struct config_action *action )`

Configures 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.

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 doesn't understand, and 0 for success.

`Status deduce_kb( const ia_audit_rec *ar, const struct rulebase *rb, Rulebase_analysis *result )`

Analyzes the audit record in `ar` using the knowledge base `rb` 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.

`Status get_rule_info_kb( const struct rulelist *kb, const string rule_name, struct rule_info *info )`

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 a 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.

The function returns 0 if it finds the rule and -1 if it cannot find it.

`get_fact_info_kb( const struct rulelist *kb, struct fact_info *info )`

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 base's 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 printing the 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.

`get_sorted_rulelist_kb()`
Returns the list of rules in the rulebase.

3.6 Resolver Component

The resolver component analyzes the alerts issued by the statistical and rulebased components and reports nonredundant "critical" results. The data structures of the resolver component are as follows.

```
typedef enum {
    SAFE,               /* everything is okay */
    WARNING,            /* activity to consider investigating */
    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;      /* target host generating the audit record */
    string       subject;   /* subject (user) generating the audit record */
} audit_record_info;

typedef struct Stats_analysis {
    stats_result_code   result_code;      /* see enumerated types */
    int                 top5meas[5];      /* top 5 measures */
    ftype               topSval[5];       /* top 5 "S" values */
    ftype               score_threshold;  /* prevailing "red" threshold */
    ftype               score;            /* score for this record */
} Stats_analysis;

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

typedef struct Stats_result {
    Stats_analysis     anal; /* statistical analysis */
    audit_record_info ar;    /* audit record information */
} Stats_result;

typedef struct Rulebase_result {
```

```
    Rulebase_analysis anal;   /* rulebased analysis */
    audit_record_info ar;     /* audit record information */
} Rulebase_result;

typedef struct AnalResult {
    struct audit_record_info  ar;              /* key audit record fields */
    struct Stats_analysis     st_res;          /* statistical analysis */
    struct Rulebase_analysis  rb_res;          /* rulebased analysis */
    string                    alert_message;   /* text of generated alert */
} AnalResult;
```

The resolver has the following functional interfaces.

`int resolve(const Stats_result *stats, const Rulebase_result *exsys, AnalResult *result, int *reportit)`

Resolves the analysis from the statistical component, `stats`, and the rulebased component, `exsys`, and returns it in `result`. The flag `reportit` will indicate whether or not an alert should be sent to the *UI* server.

Every result tagged as `CRITICAL` by the rulebased component becomes an alert. Every result tagged as `CRITICAL` 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.

The function `resolve()` returns the number of alerts generated, which is always either 0 or 1. The flag `reportit` will be set only if an alert is generated, but, because of alert filtering for individual subjects, the flag `reportit` may not be set for all alerts.

`int result_type(AnalResult *result)`

Determines the overall result level (`SAFE`, `WARNING`, or `CRITICAL`) of any given analysis result record. Both the statistical component and rulebased component results are consulted.

3.7 User Interface Component

The 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`
- `start_test_analysis`
- `test_analysis_status`
- `put_alert_stats`
- `put_test_progress`
- `put_client_reconfig_status`
- `Name_list *get_stats_client_update`
- `Name_list *get_exsys_client_update`
- `struct Stats_reconfig *get_stats_client_reconfig`
- `struct Rulebase_reconfig *get_exsys_client_reconfig`
- `int get_result_filter`
- `Nameint_list *get_alert_filter_list`

Each of the functions listed above is invoked either on behalf of the security officer or on behalf of some other component of the NIDES prototype. The functions perform as follows.

`put_alert( struct AnalResult *result )`

Presents the AnalResult `result` to the security officer. The presentation mode is given by the global variable `alert_mechanism`, which can have one of these four values.

- `EMAIL_ALERT`
- `POPUP_ALERT`

- EMAIL_POPUP_ALERTS
- 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 variable's value can be modified by the security officer at any time.

`email_alert( string message, string header)`
Posts the email message `message` to a list of recipients contained in `header`. This function is invoked as a result of function `put_alert()` being invoked (see page 48) and `EMAIL_ALERT` or `EMAIL_POPUP_ALERTS` being set in the variable `alert_mechanism`.

`control_target( string hostname, int action)`
This function, when invoked, initiates or terminates audit generation activity based on the `action` flag on the target host specified by `hostname`.

`target_error( string hostname, int error_code)`
Displays the name of the target host given by `hostname` to the security officer along with the error message, which is determined by `error_code`. The error code is one of the following types.

- TARGET_NOT_STARTED
- ERROR_ON_START_TARGET
- TARGET_NOT_STOPPED
- ERROR_ON_STOP_TARGET

This function is invoked only after the function `control_target()` (see page 49) has been previously invoked.

`control_server( string hostname, int action)`
Initiates or terminates a NIDES server based on the `action` value on a specific host, `hostname`.

`server_error( string server_name, int error_code)`
Displays the name of server `server_name` to the security officer along with the error or status message, denoted by `error_code`. Possible error codes are

- ERROR_ON_START_ARPOOL
- ERROR_ON_START_ANALYSIS
- ERROR_ON_STOP_ARPOOL
- ERROR_ON_STOP_ANALYSIS
- STOP_ARPOOL_DONE

- STOP_ANALYSIS_DONE

This function is invoked only after the function control_server() (see page 49) has been previously invoked.

put_status(struct host_list *hosts)
Displays the status (UP or DOWN) of each target host, as listed in hosts, to the security officer. If the list is already being displayed, it is updated to reflect any changes in status.

control_ar_storage(string host, int action, string archive_name)
Causes audit data from the audit data collection component to be written into an audit data archive by the archiver process. The host parameter indicates the host where collection is to take place. The action parameter indicates whether collection needs to start or stop. The archive_name parameter indicates the name of the archive where audit data is to be recorded.

ar_storage_error(string host, string archive, int code)
Displays an error message to the security officer to the effect that audit data archival has failed. The host parameter represents the host where the audit data archival process was executing, the archive parameter represents the name of the file where the archive data was written, and the code parameter is an error code that is currently not used. This function is invoked only after the function control_ar_storage() has been previously invoked (see page 50).

start_test_analysis(string host, int code, string instance, string adset, string testname)
Invokes an execution of the batch_analysis process that uses the instance instance to process audit data in NIDES format from the audit data set adset on the host host with a testname of testname. The batch_analysis process writes the results to an archive named testname. The code parameter is not used by the NIDES user interface.

test_analysis_status(string host, int code, string testname)
Displays the status of a batch_analysis process to the security officer. The host parameter represents the host where the batch_analysis is executing, the testname parameter is the name of the test and code is the status code. Based upon the status code, the user will be notified of the status of the batch_analysis process as follows.

- BATCH_DONE
- BATCH_ERROR

This function is invoked only if the function start_test_analysis() (see page 50) has been previously invoked.

`put_alert_stats( struct host_list *hosts)`

Updates the target host and system status displays with the alert record counts processed by each configured target host. The information is contained in the `hosts` parameter.

`put_test_progress( string instance, string testname, string adset, int records, int alerts, int starttime)`

Reports the progress of active NIDES test runs approximately every 10 seconds. The numbers of audit records (`records`) processed and alerts (`alerts`) generated are reported for each test.

`put_client_reconfig_status ( struct updateStatus *status)`

Acknowledges the receipt and application of deferred reconfiguration data by the analysis component. When this function is called, it signals the user interface to clear out any pending reconfiguration data for the real-time instance.

`Name_list *get_stats_client_update()`

Returns the list of subjects whose profiles are to be updated. This function is called when the security officer wishes to manually update one or more specific profiles.

`struct Stats_reconfig *get_stats_client_reconfig()`

Returns reconfiguration data for the real-time instance statistical analysis, such as measures to be turned ON or OFF and statistical parameters. Some reconfiguration elements must be deferred until the next profile update period, and hence the user interface assumes that the reconfiguration is "pending" until acknowledgment is received via a call to `put_client_reconfig_status`.

`struct Rulebase_reconfig *get_exsys_client_reconfig()`

Returns reconfiguration data for the real-time instance rulebased analysis (i.e., turning rules ON or OFF). All rulebase reconfigurations are done immediately.

`int get_result_filter()`

Returns any new reconfiguration of the result archive filter for the real-time instance. Reconfiguration of this data will cause the resolver to either increase or decrease the amount of result data archived, depending on the level requested. The available levels of filtering are

1. Critical Only
2. Warnings and Above (default filter value)
3. All Results

`Nameint_list *get_alert_filter_list()`

Returns a list of subjects whose alert reporting status is to be modified. By default, all alerts are reported to the user interface. This configuration can be changed so that

only statistical or rulebased alerts (or none at all) are to be reported for a particular subject. All alerts will still be archived regardless of alert reporting filter configuration.

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 it to the NIDES audit record format on the fly, and 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 discussion specifies the data structures and functional interfaces for *agend.*

3.8.1 Data Structures

The following data structures are used by the audit generation service:

```
const DEFAULT_AGEND_PORT = 7777;    /* 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 following functions are exported as remote procedure calls (RPCs) for *agend:*

`agend_start_agen( string agen_arg)`

Starts an instance of *agen* on the target machine. *Agen* interprets the parameter `(agen_arg)` as the address of the 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. Only one instance of *agen* is allowed to be running at any time.

`agend_stop_agen()`
: Terminates a previous invocation of *agen*. This function 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)* that 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 following data structures are used by the audit collection service:

```
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 following functions are exported as remote procedure calls:

`int put_ar_vec(struct arpool_vec *audit_record_vector)`
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)`
Retrieves a vector of audit records stored in *arpool* as its return result.

`struct arpool_status *arpool_get_status(void)`
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. This function 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 2).

The *analysis* service defines the following functions, which can be invoked as RPCs.

- `void put_stats_results(int count, Stats_result *vector)`
- `void put_rulebase_results(int count, Rulebase_result *vector)`
- `AlertResultVec *get_alerts()`

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 47), queuing the resulting alerts, if any. It provides those alerts to the user interface when function `get_alerts()` 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 user interface. It queues alerts indefinitely, until the agent for the 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 2). The functions invoked using RPC are

- From *arpool*
  `struct arpool_vec *arpool_get_ar_vec()`

- From *analysis* server
  `void put_stats_results(int count, Stats_result *vector)`

The *statistical* client gets audit records from *arpool* with `arpool_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. All in-cache profiles that need to be cleared are flushed every midnight. Profiles are eliminated from the cache at the same time if they have not been accessed in the preceding 24 hours.

Every midnight, the *statistical* client updates the historical profiles for all subjects who have been active since the last update occurred.

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

The *statistical* client flushes profiles that need to be checkpointed 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.

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 2). The functions invoked using RPC are as follows

- From *arpool*
  `struct arpool_vec *arpool_get_ar_vec()`

- From *analysis* server
  `void put_rulebase_results(int count, Rulebase_result *vector)`

The *rulebased* client gets audit records from *arpool* with `arpool_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 user interface service is responsible for presenting information received from the other services to the security officer, and for allowing the security officer to manage the operation of the prototype itself.

The agent interface is part of the 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 user interface *(UI)* server, as well as non-RPC calls that are used by the *UI* server to delegate jobs to the agents.

3.11.1 Data Structures

The *UI* 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=109,
        STOP_COLLECTION=110,

        ERROR=0,
        RUNNING=201,
        DONE=202,

        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 */
/* 100-199 action codes */
/* 200-299 nonerror result/information codes */
};
```

3.11.2 Functional Interfaces

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

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

`void kill_agents()`
: Kills all the agents started by `start_agents()`.

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

- `void put_alert(AnalResult *Alertinfo)`
- `void put_alert_stats(struct host_list *Alert_stats)`
- `void put_client_reconfig_status(struct updateStatus *status)`
- `Nameint_list *get_alert_filter_list()`
- `anal_reconfig *get_realtime_reconfig()`
- `Name_list *get_stats_client_target()`
- `Name_list *get_exsys_client_target()`
- `email_msg *get_email()`
- `control_cmd *get_control_target()`
- `void target_error(string host, int code)`
- `control_cmd *get_control_server()`

- `void server_error(string server, int code)`
- `control_cmd *get_start_test_analysis()`
- `void test_analysis_status(string host, int status, string testname)`
- `void put_test_progress(string instance, string testname, string adset, int records, int alerts, int start_time)`
- `control_cmd *get_control_ar_storage()`
- `void ar_storage_error(string host, int code, string archive)`
- `void put_status(struct hostlist *status)`

3.11.3 Agents

The *UI* server has seven agent processes, each performing specific tasks (see Figure 2).

1. *Agent_alerts*
   This agent is responsible for getting all results and alerts from the *analysis* server and reporting them to the *UI* server. It also passes back and forth analysis reconfiguration data and acknowledgments between the user interface and the analysis component. The RPCs invoked are

   - From *analysis* server
     ```
     AnalResultVec  *get_alerts()
     struct host_list *get_alert_stats()
     void put_alert_filter_list(Nameint_list *)
     void put_stats_update_list(Name_list *)
     void put_exsys_update_list(Name_list *)
     Nameint_list  *get_reconfig_status()
     ```

   - From *UI* server
     ```
     void put_alert(AnalResult *)
     void put_alert_stats(struct host_list *)
     Nameint_list *get_alert_filter_list()
     Name_list  get_stats_client_update()
     Name_list get_exsys_client_update()
     void put_client_reconfig_status(updateStatus *)
     ```

   This agent is a loop that performs a number of functions. It gets a list of hosts and an alert count, for each. The alert counts include the total number of alerts generated since the target host was up, and a count of alerts over the past hour. *Agent_alerts*

then gets a block of analysis results with `get_alerts()`, and reports them one at a time with `put_alert()`. It next determines if there are any alert filter configuration requests from the user interface and, if so, passes them on to the *analysis* server.
In addition, *agent_alerts* manages real-time analysis reconfiguration between the user interface and the analysis server. If there are any reconfiguration requests waiting at the user interface, it, fetches the reconfiguration data and sends it *to* the *analysis* server for application. Once the *analysis* server has applied the reconfiguration changes, *agent_alerts* passes back an acknowledgment to the user interface to inform the security officer of the new reconfiguration. This agent also handles manual profile update requests.
Also, *agent_alerts* gets alert statistics about currently active target hosts from the *analysis* server with the `get_alert_stats()` call. This information is reported to the *UI* server with the `put_alert_stats()` call. Polling is done approximately every 10 seconds.

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 test errors and completion to the *UI* server. The RPCs invoked are as follows.

- From *UI* server

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

This agent gets commands from the *UI* server to start test analyses, and reports any unusual occurrences relating to the analyses. It is implemented as a simple loop that gets 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 *UI* server with `test_analysis_status()`. This agent does *not* return the periodic status of test runs that are shown in the user interface. This information is sent directly from the *batch_analysis*.

3. *Agent_email*

This agent is responsible for sending e-mail on behalf of the *UI* server. The RPC invoked is as follows.

- From *UI* 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 *UI* server
     ```
     control_cmd *get_control_ar_storage()
     void ar_storage_error(string host, int errcode, string archive)
     ```

   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, the statistical component, the rulebased component, and *arpool,* and reporting on their status to the *UI* server. The RPCs invoked are as follows.

   - From *UI* 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 analysis and arpool processes with the `get_control_server()` call, 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* to determine the current status of the NIDES system. The RPCs invoked are as follows.

   - From *arpool*
     ```
     arpool_status *arpool_get_status()
     ```

   - From *UI* server
     ```
     void put_status(struct hostlist *status)
     ```

   This agent is a simple loop. It gets information about currently running target hosts and the latest audit record sequence number from *arpool* with `arpool_get_status()`. If the target hosts'status have changed since the last, poll, or if the total number of audit records (including the hourly count) from that host has increased since the last count, this information is reported to the UI server with `put_status()`. Polling is done approximately every 10 seconds.

The agent detects failure on the part of *arpool,* 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 *UI* 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 *UI* server with `get_control_target()` and makes calls to `agend_start_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 *UI* 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* an error cannot be reported in the absence of an RPC request. For this reason, *agent_target* polls the *agend* servers of the running target hosts to verify that the *agen* process is still running. Polling is done approximately every 10 seconds.

3.11.4 Agent Interfaces

An agent interface is associated with each agent. 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 nonempty 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 starts only one instance of each agent.

There are five of these queue-pairs for five of the agents. The other two agents report information to the *UI* 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 *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 *UI* 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 *UI* server for it to display.

4 Data Files

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

- `/etc/security/audit`
  Directory `/etc/security/audit` contains SunOS C2 or BSM 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 user IDs into user names. The file should exist on each target host.

- `storage`
  Directory `storage` is read from and written into by the persistent storage infrastructural component. It uses subdirectories `adsets`, `instances`, and `dmf`. Directory `storage` itself is under `<IDES_ROOT>`, which is a site-dependent environment variable.

- `mandatory_rules`
  File `mandatory_rules` is read by the UI component. It contains a list of rules that may not be disabled. It is stored under `<IDES_ROOT>/etc` where `<IDES_ROOT>` is a site-dependent environment variable.

- `priv_users`
  File `priv_users` is read by the UI component. It contains a list of users with privileged access to NIDES functionalities. It is stored under `<IDES_ROOT>/etc` where `<IDES_ROOT>` 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.

- `stats_config`
  File `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/instance/stats_config`.

5 Requirements Traceability

The requirements for the NIDES prototype and the extent to which they have been satisfied [4], are as follows.

- **Acceptable detection performance:**
  Minimal false positives and maximal true positives.
  - 1% to 5% false positives for the statistical component.
  - reporting of twenty-five known intrusion types using the rulebased component.

- **Real-Time Operation:**
  Anomaly detection 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:**
  Maintains 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 Beta and Alpha Prototypes

The essential differences between the NIDES beta prototype and the NIDES alpha prototype are as follows.

1. The NIDES beta prototype supports real-time analysis component reconfiguration, as well as experiment reconfiguration. Reconfiguration of the rulebase (addition and deletion of rules) is supported and statistical analysis parameters, measures, and training values can be reconfigured. For a complete description of reconfiguration capabilities see [1].
2. The NIDES beta prototype supports audit data and result data archival and retrieval using a set of search parameters.
3. The NIDES beta prototype supports filtering of alert and result data.
4. The NIDES beta prototype supports system performance tuning configuration.
5. The NIDES beta prototype includes an expanded rulebase over what was provided with the alpha version.
6. The NIDES beta prototype supports privileged and non-privileged user functions.

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 11). 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 appendix.

Contents of an NIDES Audit Record

The first nine data items of the NIDES audit record always exist, and the remaining data items are optional. A data item exists if it is "check-marked" in the `mark` data item that always exists (see page ***74).***

The NIDES audit record has the following data items.

`version`
: Audit record structure version number. This should be 4 — that is the fourth version of the audit record structure.

`rseq`
: A monotonically increasing sequence 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`
: Corresponds to the time stamp when this record was received by NIDES *(arpool* server).

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

`atime`
: 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`
Name of target host.

`audit_src`
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`
Activity that resulted in the generation of an audit record.

`mark`
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`
Corresponds to the actual user name. It is the user's authenticated (actual) ID rather than the user's current ID (see `uname`). For example, on UNIX, this should not change with superuser enables `(su)`.

`auname_label`
Security label associated with `auname`.

`uname`
User's current ID. It might not correspond to the user's actual ID (see `auname`).

`uname_label`
Security label associated with `uname`.

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

`ttyname`
Name of the terminal associated with the action.

`cmd`
Name of the command associated with the action.

`arglist`
List of command arguments associated with the action.

`syscall`
Number of the system call or the operation code associated with the action.

`errno`
Error code from the action.

`rval`
: Return value from this action.

`res_utime`
: User CPU time for this action.

`res_stime`
: System CPU time for this action.

`res_rtime`
: Elapsed real time for this action.

`res_mem`
: Amount of memory consumed in executing the action.

`res_io`
: Amount of terminal I/O performed in executing the action.

`res_rw`
: Amount of disk I/O performed in executing the action.

`ouname`
: Alternate user name, as in the argument of superuser enable `(su)`.

`ouname_label`
: Security label associated with `ouname`.

`remoteuname`
: Remote user name for actions involving remotely initiated activity.

`remoteuname_label`
: Security label associated with `remoteuname`.

`remotehost`
: Remote hostname for actions involving remotely initiated activity.

`path0`
: File name associated with the action.

`path0_type`
: File type of `path0`.

`path0_label`
: Security label associated with `path0`.

`path1`
: Another file name associated with the action.

`path1_type`
File type of `path1`.

`path1_label`
Security label associated with `path0`.

Data Structures

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

`ia_seqno`
NIDES sequence numbers are represented 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 1.

`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`
NIDES security label.

`ia_ftype`
Specifies the type of file, 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`
Identifies the source of the audit data.

`ia_audit_action`
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 — that is, 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 were 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, and otherwise returns 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 *)
```

Glossary

Accounting Audit Data The standard UNIX accounting system. Designed primarily for keeping track of resource utilization (e.g., connection time, CPU usage) for billing purposes. The accounting records generated are of minimal utility when other forms of audit data are available (e.g., C2 or BSM).

Agen Audit data generation client process. A single `agen` process runs on each of the actively monitored target hosts, translating all the supported, native audit data into canonical NIDES audit records, and providing them to the `arpool` process. The UNIX version of the agen process currently supports three native audit record formats: SunOS BSM version 1, SunOS C2, and standard UNIX accounting.

Agend Audit data generation daemon process. A single `agend` process runs on each of the actively monitored target hosts. `Agend` accepts and acts upon requests to start or stop an `agen` process on a NIDES target host.

Agent A NIDES client process that facilitates communication between NIDES server processes.

Arpool One of the core NIDES processes. The `arpool` process accepts canonical NIDES audit records from the `agen` process on all the actively monitored target hosts and presents the audit records as a single data stream to the analysis components of NIDES.

Archiver One of the core NIDES processes. The `archiver` process accepts canonical NIDES audit records from the `arpool` process and stores them on disk, in a compressed format, to facilitate future reference when investigating activity that generated alerts.

BSM The most recent auditing system developed for SunOS. The BSM (Basic Security Module) generates audit records derived from low-level UNIX activity (e.g., reading/writing/assessing/deleting a file, changing directory, running a program).

C2 An older, now obsolete, auditing system developed for SunOS. C2 generates audit records derived from low-level UNIX activity (e.g., reading/writing/assessing/deleting a file, changing directory, running a program). Its name is derived from a specific security rating described in the ‘Orange Book” (see [6]). It should not be confused with the generic computer security rating of C2.

Client An *active* NIDES process. A client process initiates communications/requests with NIDES server processes.

IDES_ROOT The NIDES environment variable that determines the directory where the NIDES software resides. This variable must be set prior to running any NIDES software.

Instance An analysis configuration, and the set of profiles associated with that configuration.

Minimum effective n The minimum count of records in the long-term profile that must be accumulated before the scoring mechanism is considered reliable. It is measure-specific.

Native Audit Record An audit record specific to a given auditing system. Native audit records are converted by the `agen` process into a canonical NIDES audit record format for analysis and storage. Once the audit data are converted, NIDES no longer makes use of a native audit record. The UNIX version of the agen process currently supports three native audit record formats: Sun OS BSM version 1, Sun OS C2, and standard UNIX accounting.

NIDES Next-Generation Intrusion Detection Expert System.

NIDES Audit Record A canonical audit record format capable of representing all supported native audit record information. NIDES audit records are used for analysis and storage. Once the audit data are converted, NIDES no longer makes use of a native audit record.

Persistent Storage NIDES maintains databases of many types under its normal operation. These databases include an audit record archive, analysis result archive, instances (user profiles and analysis configuration data) and miscellaneous configuration files (e.g., privileged user lists). All of these databases and files are part of the NIDES persistent storage facility. The persistent storage facility provides a set of library functions to all NIDES components, allowing them to read and write data to the various databases and configuration files.

Remote Procedure Call (RPC) An action in which a process calls a procedure that is executed by another process. The NIDES architecture is composed of many processes that communicate via RPCs. For example, when the NIDES analysis components (statistical and rulebased) need an audit record to analyze, both components make an RPC to the `arpool` process to ask for the next audit record; the `arpool` process makes an RPC in the form of a response providing an audit record to the analysis processes.

Resolver The NIDES analysis process that receives results from the statistical and rulebased analysis components and determines if an alarm should be reported.

Result A result is generated for every audit record processed by the NIDES analysis components. Results are categorized into three levels: safe, warning, and critical. The level of a result is assigned by the resolver component based on the levels assigned by the statistical and rulebased analysis components. An NIDES alert is reported when the resolver determines that a critical-level result should be assigned alert status.

Sequence Number Numbers assigned by the NIDES `agen` and `arpool` processes to the audit records processed by NIDES. Two sequence numbers are assigned to each audit record. The `agen` process assigns a target host sequence number that is unique for the duration of the current `agen` process execution on the target host. This number is referred to as the *target sequence number*. The `arpool` process assigns a sequence number to all audit records it receives; this number is unique across all NIDES target hosts and monotonically increases for the duration of the current `arpool` process. This number, referred to as the *audit record sequence number,* is used to identify the audit record when alerts are reported by NIDES. When `arpool` is first started it begins with en a sequence number of 0.

Server A *passive* NIDES process that responds to communications/requests from NIDES client processes.

Subject The entity for which NIDES maintains profiles and performs anomaly detection. In the NIDES paradigm, the subject (e.g., a user of the system) initiates actions (e.g., file copy) that act on objects (e.g., files).

Target Host A host computer that is monitored (or can be monitored) by NIDES.

Test A batch run of NIDES with archived data, typically done to examine the impact of parameter changes or establish detection rates

X A *de facto* graphical user interface standard.

XDR External Data Representation.

References

[1] Debra Anderson, Thane Frivold, Ann Tamaru, and Alfonso Valdes. *NIDES User Manual/Computer System Operators Manual — Beta Release. Report, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, June 1994.*

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

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

[4] *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. SPA WAR Contract Number N0039-92-C-0015.*

[5] *Harold S. Javitz and Alfonso Valdes. The NIDES Statistical Component Description and Justification. Annual report, SRI International, 333 Ravenswood Avenue, Menlo Park, CA 94025, March 1994.*

[6] *NCSC. Department of Defense Trusted Computer System Evaluation Criteria (TCSEC). Report, National Computer Security Center, December 1985.*

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