A Common Intrusion Detection Framework*

Clifford Kahn	Phillip A. Porras	Stuart Staniford-Chen	Brian Tung
c.kahn@opengroup.org	porras@csl.sri.com	stanifor@cs.ucdavis.edu	brian@isi.edu
The Open Group	SRI	UC Davis	ISI

15 July 1998

Abstract

As intrusions and other attacks become more widespread and more sophisticated, it becomes beyond the scope of any one intrusion detection and response (ID&R) system to deal with them. The need thus arises for systems to cooperate with one another, to manage diverse attacks across networks and time. Heretofore, efforts toward "cooperation" have focused primarily on homogeneous components, with little if any attention toward standardization.

In this paper, we discuss the efforts of the Common Intrusion Detection Framework (CIDF) working group in designing a framework in which ID&R systems may cooperate with one another. We consider the issues involved in standardizing formats, protocols, and architectures to co-manage intrusion detection and response systems, and compare the strengths and weaknesses of previous approaches. We examine various ways that these systems and their components may be connected and related. We conclude with an overview of CIDF's current approach to providing a common intrusion specification language.

*The work presented in this paper is currently funded by a lot of nice people.

A Common Intrusion Detection Framework

15 July 1998

Abstract

As intrusions and other attacks become more widespread and more sophisticated, it becomes beyond the scope of any one intrusion detection and response (ID&R) system to deal with them. The need thus arises for systems to cooperate with one another, to manage diverse attacks across networks and time. Heretofore, efforts toward "cooperation" have focused primarily on homogeneous components, with little if any attention toward standardization.

In this paper, we discuss the efforts of the Common Intrusion Detection Framework (CIDF) working group in designing a framework in which ID&R systems may cooperate with one another. We consider the issues involved in standardizing formats, protocols, and architectures to co-manage intrusion detection and response systems, and compare the strengths and weaknesses of previous approaches. We examine various ways that these systems and their components may be connected and related. We conclude with an overview of CIDF's current approach to providing a common intrusion specification language.

1 Introduction

The intrusion detection research community has been pursuing techniques to detect and respond to malicious activity in computer systems for more than a decade. These efforts have led to a variety of techniques, which have been applied to numerous data sources and platforms (audit trails, application logs, network traffic, etc.). However, these projects have largely been pursued in isolation, with the collection methods, analysis results, and response directives rarely intended to be shared across intrusion detection systems. While many techniques may, in principle, lend themselves to applications beyond their project objectives (e.g., a signature analysis tool may well contain a fairly general inference engine), much of the effort within the project is directed at tuning the system for a very specific event stream and analysis objective. Furthermore, systems have primarily been closed, in the sense that their management interfaces have not been exportable outside their internal frameworks.

As a result, systems tend not to provide comprehensive detection coverage for malicious activity across a spectrum of systems. Incidents, however, often consist of a large series of widely distributed exploit steps. The incident may involve numerous systems, and it may attack networks, host operating systems, and applications. Thus, to effectively cover the potential threats, it is desirable to integrate multiple intrusion detection systems that provide complementary coverage. This paper therefore addresses the research questions involved in intrusion detection interoperability.

In the rest of this paper, we first define what we mean by two IDS systems interoperating (Section 2), and provide a range of scenarios that illustrate the degrees of possible interoperation. This is intended as a framework for the rest of our discussion. Next, in Section 3, we discuss the previous work that has been done in this area and show that while interesting things have been published, several unsolved problems remain.

Then, in Sections 4 and 5, we begin to discuss the efforts of the Common Intrusion Detection Framework (CIDF) working group to fill one of these gaps—the definition of a language to allow IDS systems to express and share intrusion relevant data. The CIDF working group is a consortium of researchers from a number of organizations who have been collaborating in an effort to create a specification adequate to allow their respective systems to interoperate. This paper presents one aspect of the CIDF specification—the Common

Intrusion Specification Language (CISL) which focuses on the common expression of events, analysis results, and response directives. Intrusion detection systems are logically decomposed into distinct tasks-oriented components, from which CIDF attempts to standardize a common encoding schema for expressing data that needs to be transmitted across component boundaries. We discuss the functional requirements of CISL in Section 4, and we examine the syntax and semantics of the language in Section 5.

Finally, the conclusion summarizes the status of the project and future directions.

2 Interoperating ID&R Components: Sample Applications and Scenarios

What does it mean for two ID&R systems to interoperate? Our definition is as follows

> Two intrusion detection and response systems are interoperating when they exchange data automatically, and as a result achieve some goal which neither could have achieved alone.

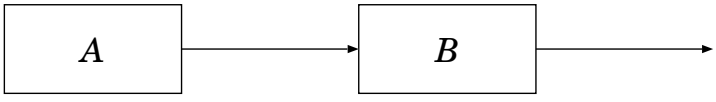
Various aspects of interoperability can be identified. For example, a necessary but not sufficient condition for full interoperability of two systems is *configuration interoperability*, which refers to the condition in which two systems can find each other and successfully send some data back and forth. This is not useful by itself, however; there must also be *syntactic interoperability*. This implies that the two systems both parse the syntax of the exchanged data correctly (eg. agree on data-types, byte ordering, etc). A stronger condition still is *intercomprehension*; this occurs when both systems agree on the meaning of the data as well as its syntax (to the degree necessitated by the algorithms each system uses).

We also note that a general specification that allows many ID&R systems to interoperate (at least pairwise), must do much more than is required of a special-purpose scheme for integrating two particular systems. The specification must contain enough detail that systems can intercomprehend each others data *even though they were not designed specifically to work together*, but only with another potentially unknown but specification compliant system. Out-of-specification agreements between system developers should not be needed.

We now discuss various ways that ID&R components may be interconnected to provide extra functionality or additional robustness and certainty. We are here considering *composition* of ID&R systems, and so will illustrate *compositional idioms*, useful ways of combining several systems to form a larger system. This will help guide us in defining a language that will provide intercomprehension, as opposed to simply interoperation. Without these idioms, we run the risk of defining language constructs that simply make the language heavier-weight, without aiding comprehension at all.

2.1 Analyzing

A gathers raw data and may analyze a relatively small amount of that data at a time. B takes a larger or longer view, analyzing or distilling several reports from A to produce a single report from B.



A may be a recorder of raw events, or a signature analyzer, or an anomaly-based detector. B in turn can be a signature analyzer, trying to determine whether several events reported by A are related in some

way. Or B may be an anomaly-based detector, trying to decide whether the weight of reports from A is sufficient to deserve attention, thus filtering out false alarms from A.

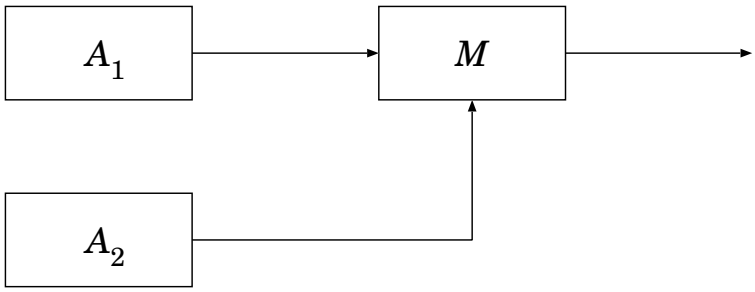
2.1.1 Example

For example, A might be Heberlein's Non-cooperative Service Recognition (NCSR) [9], and B might be an anomaly detector, such as NIDES [10]. A atempts to evaluate the behavior of network service connections against it empirical understanding of how client sessions on these network services typical behave. For example, A attempts to determine whether a connection to port 25 is *really* a mail transfer, or might be interactive, or whether the process running on port 80 looks like httpd, or more like sshd. As it encounters client connections that do not appear to match previous patterns of usage, it reports these connections to B.

B may then look at the organization-wide distribution of alerts from the NCSR boxes. When that distribution does something unusual—e.g., alerts show up where they normally do not, or in much greater numbers than usual—then it makes a report, which may go to a human monitoring the network.

2.2 Complementing

Two components A_1 and A_2 can complement each others' coverage. A third component, M, merges their output. Obviously, it is helpful if A_1 and A_2 produce output in the same notation.



A_1 and A_2 may detect different kinds of attacks. Perhaps A_1 detects attacks on the TCP/IP protocol, whereas A_2 detects attacks on certain applications.

Or A_1 and A_2 may detect the same kinds of attacks but on different machines. Indeed, A_1 and A_2 may be instances of the same detection program—or they may be different programs, just as there are many different web server programs.

This case includes extrapolation, or sequential trend analysis. If many instances of A start reporting a particular anomaly, M may predict more such anomalies. M may have to do extra work if the various instances of A do not report exactly the same set of events, but merely overlap in their reporting.

This case also includes the conversion of audit trails into a standard format to permit cross-host analysis.

2.2.1 Example

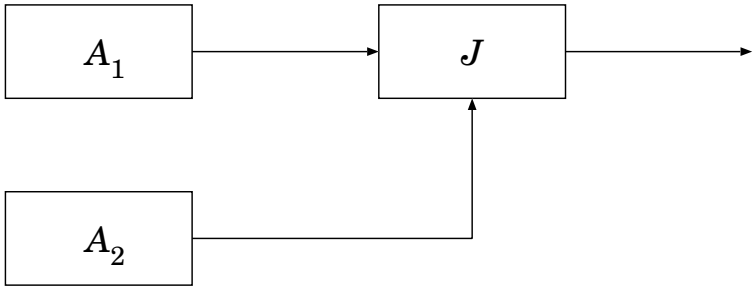
In one important scenario, each host runs software that continuously converts the host's audit log and other security log files into a common format.

One or more analyzers, running on dedicated hosts, analyze this data to quickly pinpoint attacks. Ideally, an administrator will be able to run several analyzers from different vendors, or switch from one analyzer to another, without having to modify or reconfigure most of the hosts.

This scenario exploits the fact that although the forms of audit logs and other logs vary from one operating system to another, the content is similar. The means of recognizing an attack have much in common from one operating system to another. So, whereas the data-gathering logic must be operating-system specific, the analysis logic can be much more generic.

2.3 Reinforcing

Two components A_1 and A_2 may *reinforce* each others' findings. This is important because intrusion detectors tend to have many false positives. However, two detectors that use different analytical methods normally have false positives at different times. J judges their output, typically accepting it only when A_1 and A_2 agree.



For example, A_1 may do statistical anomaly detection. It may report an anomaly when a statistic such as the number of connections reaches an unusual value. A_2 may do a signature-based analysis, reporting an attack when it observes a known attack pattern.

2.3.1 Example

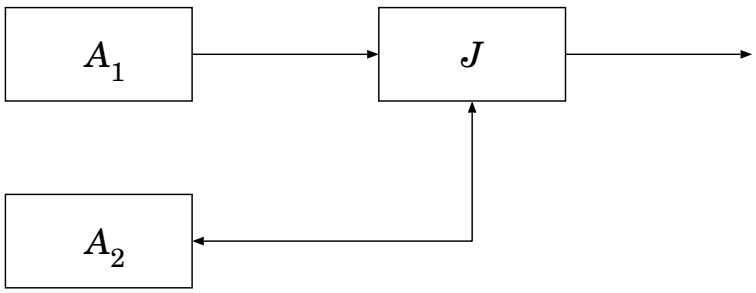
A_1 is a detector that looks at network traffic and generates reports about suspicious usage. (NSM [8], NID [11], and ASIM [5] are examples of such detectors.) Unfortunately, A is inaccurate and often produces reports on uninteresting connections. A_2 is a host-based detector that monitors accounting and audit data and tries to report whenever someone gains root privileges on a host (either legitimately or not). When J finds a time and host coincidence between these two detectors, then it creates a report for an administrator or for another component.

2.4 Verifying

After A_1 reports an attack, J asks A_2 whether it sees signs of the same attack. If A_2 reports that it also sees the attack, then J considers the report verified and endorses it.

2.4.1 Example

There are frameworks in which many hosts cooperate to diagnose and respond to attacks. One such framework is the trace-and-suppress system (such as IDIP [3]). A trace-and-suppress system traces an



attack toward its source. The attack must involve a series of packets, not just one or two. The trace-and-suppress approach has been successful against flooding attacks, password-guessing attacks, and the like.

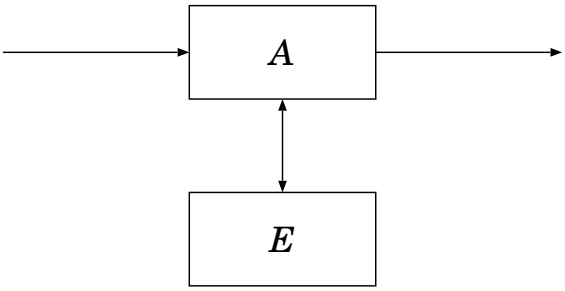
The network must be partitioned into enclaves separated by boundary controllers. One or more detectors reside with each boundary controller, and they monitor the traffic crossing that boundary (in one or both directions). Other detectors may be separate from boundary controllers, and may be either host-based or network-based.

A detector A_1 detects an attack and reports it to a boundary controller J. The boundary controller then asks its resident detectors A_2 whether they also see the attack. If so, the boundary controller takes steps to suppress the harmful traffic, and also reports the attack to the upstream boundary controllers. These controllers repeat the process, treating the first J as their A_1. This continues until the attack is isolated to a single enclave. (The Internet at large may be considered a single enclave.)

It is important to have a standard so that new analyzers and responders designed to handle new attacks can easily plug into an installed framework. It is important as well that different analyzers in different places be able to determine whether they are seeing the same event. Analyzers written by different vendors must be able to understand each others' reports, at least partially.

2.5 Adjusting Monitoring

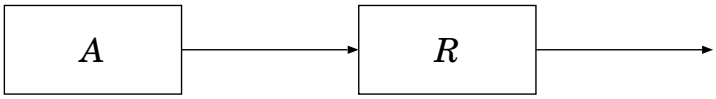
A adjusts monitoring depending on what warnings it receives. A sends instructions to E about what to monitor. In turn, A receives information from E, analyzes it, and passes it on.



The E at the bottom can actually do analysis, which follows from the principle of composition.

2.6 Responding

Suppose an analyzer determines that a particular process is attacking its host, or that a particular network connection is being used to launch attacks. What does the analyzer do? Alerting a human is fine, but



human response is slower than machine response, especially in the middle of the morning. Just as buildings often have sprinklers that respond to a fire while the fire fighters are on their way, so intrusion detection systems often need some measure of automatic response. But the analyzer cannot, in most cases, respond directly. It must send a prescription to some other component.

2.6.1 Example

To stop a flooding attack, an analyzer might send a prescription to a router to discard certain packets rather than routing them. This is important to standardize so that different routers, firewalls, etc. can accept the same prescriptions.

3 Relation to Previous Work

We now discuss previous work particularly relevant to ID&R interoperation. The primary technical issues raised in this discussion are

- the dissection of intrusion detection systems into distinct task-oriented components
- the development of a canonical notation and semantic model for expressing intrusion-detection-relevant information across these logical component boundaries
- strategies for managing the interoperation of components based on this common language

3.1 Canonical Notations

Canonical notations for expressing intrusion detection relevant data has been pursued for several years. Such a notation would be particularly interesting to the intrusion detection community in that, if widely adopted, it could greatly reduce the cost of integrating new intrusion detection tools. Some example work in this area includes

Standard Audit Trail Format: Bishop proposed a standard audit trail format in 1995 [1]. Each record represents a single event, and consists of a variable number of name-value pairs that define attributes of the event represented by the record. The format defines syntactic separators for the fields and records, and appropriate escaping mechanisms for when separators must be used within fields. However, the format does not carry precise typing information for field values. Data is in ASCII and how, dates, etc, get represented in ASCII is not specified precisely. This would be required for the format to be helpful in achieving syntactic interoperability. The format is completely flexible and extensible, in the sense that any attribute name can be used in an event. The meaning of attribute names is not defined by the paper—this is explicitly out of scope though a few comments are made about likely common attributes. Defining the meaning of the fields would be an essential prerequisite for semantic interoperability.

Intrusion Detection Input Requirements Price performed a helpful survey in 1997 [13], which analyzed the audit data needs of five different intrusion detection systems built by the research community (NIDES, DIDS, NADIR, IDIOT, and STAT/USTAT). Price compared the input needs of these systems to operating system audit data provided by a selection of commercial operating systems (based on the system documentation), noting the various mismatches. Price's study suggests that all the operating system trails, and the custom formats used by the IDS systems, all are organized basically into (Subject, Action, Object) triplets. Additionally, records provide time and sequence number information. Price provides very long lists of all the various attributes which it is desirable for an IDS to be able to deduce from any particular audit record.

X/Open Distributed Audit System (XDAS) The Open Group has published a preliminary standard for XDAS [6], intended to become part of the X/Open suite of standards. XDAS provides formats and APIs for auditing of events which are relevant at a global distributed system level. The record format itself conforms to the (Subject-Action-Object) triplet view of audit records, but, interestingly, adds an Observer role that contains a range of information (names, numerical ids, ownership attributes, etc). XDAS provides an audit record format which is of interest here. Syntactically, an audit record consist of all specified fields, in order, in ISO LATIN-1 characters. Fields are of variable length and are colon separated. There is no flexibility as to which fields may appear—all must. So this format is space expensive, but the semantics of all the fields is fairly well defined.

X/OPEN Event Managment Systems (XEMS) The Open Group also published a specification for Event Management Systems (XEMS) [7]. Whereas XDAS is targeted at storing audit-trails for after-the-fact analysis of security-relevant events, XEMS is targetted for real time notification of *any* system events (not just those of security relevance). XEMS provides a set of APIs for event producers and consumers to connect to the service. Additionally, there are API calls for configuration and management of the service itself. This format is similar in form to Bishop's, but also allows for binary data. Thus each provides type information with it, in addition to its name and value. Also, XEMS has mechanisms to define and manage event-schemas. Schemas are particular types of events which can be created at run time. The schema basically defines a list of attributes which must be defined in events of this type. XEMS also manages event filters, which consist of logical expressions on the attributes in an event (in disjunctive normal form). These filter out events which do not match the expression. XEMS is clearly complete enough to allow syntactic interoperability, but cannot support semantic interoperability unless systems have out-of-specification agreements on the meaning of fields in records.

3.2 Cooperative Security Monitoring

Several efforts have argued the potential benefits in sharing analysis reports among intrusion detection modules, but focus on configuration interoperability, rather than on data semantics. Some example work includes

- Frinke, et.al., proposed a language for specifying and modeling interactions between peer intrusion detection monitors, and attempted to enumerate the principles and requirements for interoperation among peer intrusion detection components [4]. A prototype implementing some of these concepts, HummingBird, has also been developed.
- The EMERALD project proposed a building-block approach where monitors are deployed as services throughout a network [12]. These monitors may operate independently, and can also operate

cooperatively in a recursive hierarchy that correlates analysis results at higher layers of abstraction (i.e., from the analysis of individual components and services, to domain-wide analyses, and then to enterprise-layer analyses.

- Purdue University proposed the development of an Autonomous Agent intrusion detection framework [2]. Agents may be spread throughout a system to operate independently. In addition, highly suspicious activity may cause an agent to `RAISE` the alert (i.e., broadcast the anomaly to other agents).

- White, Fisch, and Pooch proposed the development of a cooperating security management system for peer-based intrusion detection, which attempts to facilitate the cooperative operation of independent network intrusion monitoring tools [15]. Their project involved the integration of a network intrusion detection monitoring tool with a local security manager component, collectively referred to as Cooperative Security Managers (CSMs). When combined, the security manager component performs intrusion response handling, and they suggest the sharing of intrusion information across CSM boundaries (though details are not provided).

4 Language Requirements

Thus far, we have argued that for ID&R systems to interoperate and cooperate, we should outline ways that various systems and their components might interact. From such an outline, we can then construct a data format or language using which those interactions can in fact take place.

It should be evident that not all languages are suitable. When a language lacks a flexible grammar and a suitably rich vocabulary, we will soon encounter situations and events which are simply inexpressible in that language. Conversely, it is possible for a language to be *too* flexible and *too* rich, providing the observer of an event with such a bewildering array of ways to describe the event that it would be hopeless to select the one any other system would understand.

We shall soon see that a language appropriate for describing intrusions and prescribing responses will be subject to many such mutually opposite constraints. In this section, we discuss those constraints in some detail. We will follow this in Section 5 with a description of our current design and approach to satisfying these requirements.

4.1 Impact of the CIDF Architecture

Under the CIDF model, components in general may receive an input stream, use this input to drive their internal analytical processing, and pass the results on to other components. In other words, the output of one component may be the input of another.

Notice that the nature of these inputs and outputs may be quite different at various places and times in the system. Components that are in some sense close to the data sources are more likely to produce raw event descriptions; components that are further downstream may consume these event descriptions and produce analyses of them; and yet other components may consume these analyses and produce recommendations for responses.

In spite of these widely varying types of information being exchanged, the fact that all of these components are interconnected and related makes it beneficial for us to find a common way to express them. This would allow intrusion detection components that conduct other activities such as state monitoring to participate

as well. As an additional rationale, consider that responses may involve further reconnaisance or specialized event gathering, for instance, so that there is potentially an extensive feedback loop within the intrusion detection system at large.

However, these commonalities should not obscure the fact that they are still different in important ways. Without going into the specific nature of events and their analyses, we note that event records simply represent observations made by a component on a system. Many of these records may involve ordinary computation or routine maintenance. As a result, these records are likely to be produced in high volume. Also, since many of these activities can be witnessed by anyone logged into the system, the security requirements for the event records may be relatively low.

On the other hand, analysis results and countermeasure recommendations are strongly correlated with suspected attacks. They are therefore comparatively scarce. And because they may affect an intrusion system's reasoning about an attack, or its decision-making process on how to react, they will probably have stronger security requirements.

These differences will affect what transport mechanisms are used to move them around; messages that are very common will likely use transports that are optimized for performance, whereas messages that are scarcer will use transports that provide reliability and assurance. The language used by the intrusion detection components should not place restrictions on the transport and security mechanism(s) used, and vice versa.

4.2 Language Goals and Requirements

In light of the foregoing discussion, any language used to exchange information about attacks should have the following qualities:

1. Expressive. Components should be able to express a wide range of intrusion- and misuse-related phenomena and prescriptions. It should be broad enough to enable one to manage a system of intrusion detection and response components easily and scalably. To be more specific, this language should be able to express at least the following:

   - causal relationships among events
   - the roles of objects in events, such as initiator
   - properties of objects, such as name
   - relationships of objects to other objects, such as owner
   - response prescriptions calling for halting particular processes, sessions, or other activities
   - contingent response prescriptions

   However, this same language should also be...

2. Unique in expression. Components should not be able to express a given sentiment in a nearly infinite number of ways; instead, there should be one or a small number of "natural" expressions. If a sender and a receiver can agree on the *objects* of interest, but not on the *way* they will express information about those objects, then they should still be able to understand each other. If they cannot, then the language is too arbitrary.

3. Precise. The meaning of an utterance in the language should be well-defined. Two receivers reading the same message must not draw mutually contradictory conclusions from it. But this language should also be...

4. Layered. Specific senses should be expressed as cases of general senses, so that different receivers with different requirements can discern as much as they need from a message. There should therefore be a mechanism in the language by which specific concepts are defined in terms of more general ones.

5. Self-defining. Consumers that receive a report should be able to interpret messages to the degree that they need to, without recourse to out-of-band negotiation. It should be self-evident from a message how each datum within it should be interpreted. But the language should also be...

6. Efficient. Messages should consume as little of system resources as possible. Especially, it should be possible to omit contextual information in most of a sequence of similar messages.

7. Extensible. If a group of producers and consumers decide on additional information they wish to be able to express, they can define a way of doing so with a minimum of trouble and not lose any compatibility with other CIDF components that do not know their extension. These extensions may be arbitrarily complex. But the language should also be...

8. Simple. Producers should be able to encode information quickly; consumers should be able to extract the information they need without having to do excessive processing. Components with a need to be compact and inexpensive should be able to send and receive simple messages without "understanding" the language as a whole, simply by filling in blanks and/or extracting from them the required information.

9. Portable. The language should support a variety of platforms and transport mechanisms—support meaning that the environment should not fundamentally limit what information is exchanged. However, it should also be...

10. Easy to implement. This is the most practical requirement. If the language is too difficult to implement, then it will simply not be used.

4.3 The Proposed Approach

In this section, we will describe, in general terms, our proposed approach. We will begin with a general language construct, called S-expressions [14]. S-expressions are simply recursive groupings of tags and data; typically, the grouping is done with parentheses, as in Lisp. Here is an simple S-expression:

```
(HostName 'ten.ada.net')
```

This S-expression simply groups two terms, `HostName` and `'ten.ada.net'`. It does not, on its own, provide any semantic interpretation of this pairing. That is left up to the language definition. The advantage of S-expressions is that they provide an explicit association between terms, without limiting what those terms and their groupings might express.

To achieve self-definition, we will add a feature to the S-expressions. Each S-expression will begin with a tag, such as `HostName`, that gives some semantic "clue" to the interpretation of the rest of the S-expression. For this reason, these tags will be called Semantic IDentifiers, or SIDs for short. In the example above, we are instructed to interpret the string `'ten.ada.net'` as a hostname.

As we shall see, these tags can not only give descriptive information such as a hostname, but they can also tag actions taken by an entity, or specify whether an entity acted or was acted upon, and so forth. For example, we may surround the above expression with other SIDs that will indicate whether the host with that name was the location of a deleted file, whether it was the source of ping packets, and so forth.

As a consequence, S-expressions, when used with SIDs, allow a wide variety of sentiments to be expressed. Like English, however, they suffer from ambiguity in the sense that a single sentiment may be capable of many different expressions.

To achieve uniqueness in expression, we must formulate rules and guidelines for constructing S-expressions, so that only a subset of all possible S-expressions will be produced. This will reduce the likelihood that two components will not be able to understand each other even when they are interested in the same kind of data.

Another drawback of S-expressions and SIDs is that they do not on their own provide a compact representation. In order to make them easy to remember, a SID may have a name that is quite long. For instance, our language contains a SID called `IntentionallyHelpedCause`; this SID is 24 letters long, and left as ASCII, would take a corresponding long space in a communication stream between two components. Therefore, we will define an encoding that is both compact and simple to parse.

5 S-Expressions

In this section, we will describe how SIDs and S-expressions are used to express information about intrusions and anomalies and their related paraphernalia.

5.1 An Introductory Example

We introduce our language by means of an example. The reader is not expected to understand the organization of the example; nevertheless, part of the aim of the language is to make comprehension by non-educated readers at least plausible.

```
(InSequence
    (Login
        (Context
            (Time '14:57:36 24 Feb 1998')
        )
        (Initiator
            (HostName 'big.evil.com')
        )
        (Account
            (UserName 'joe')
            (RealName 'Joe Cool')
            (HostName 'ten.ada.net')
            (ReferAs 0x12345678)
        )
    )
    (Delete
        (Context
```

```
            (HostName 'ten.ada.net')
            (Time '14:58:12 24 Feb 1998')
        )
        (Initiator
            (ReferTo 0x12345678)
        )
        (Source
            (FileName (ExtendedBy UnixFullFileName) '/etc/passwd')
        )
    )
    (Login
        (CriticalContext
            (ReturnCode (ExtendedBy CIDFReturnCode) Failed)
            (Comment '/etc/passwd missing')
        )
        (Context
            (Time '15:02:48 24 Feb 1998')
        )
        (Initiator
            (HostName 'small.world.com')
        )
        (Account
            (UserName 'mworth')
            (RealName 'Mary Worth')
            (HostName 'ten.ada.net')
        )
    )
)
```

A rough English translation would of this S-expression would be

> Three actions took place in sequence, at the times indicated. First, someone logged into the account named `'joe'` (real name `'Joe Cool'`) at `'ten.ada.net'`, from a host named `'big.evil.com'`. Then, about a half-minute later, this same person deleted the file `'/etc/passwd'` from `'ten.ada.net'`. Finally, some five minutes later, a user attempted but failed to log in to the account `'mworth'` at `'ten.ada.net'`. The attempted login was initiated by a user at `'small.world.com'`.

Notice that SIDs take arguments, and that some SIDs take simple data as arguments, while other SIDs take other S-expressions as arguments. Furthermore, some SIDs denote actions, while others describe time and location, and still others describe or identify objects participating in the action.

5.2 Types of SIDs

At the heart of a sentence are *verb SIDs*. Normally, we think of verbs as denoting some action (which may sound somewhat event-centric), but they may also denote a recommendation, for instance, or description of state. An example of a verb SID is `Delete`.

A sentence which has exactly one verb is called a *simple* sentence. Sentences may be joined using conjunctions; the result is called a *compound* sentence. Sentences may also contain other sentences; they are called *complex* sentences.

Now, a verb on its own cannot tell much. If all we know is the verb `Delete`, we can appreciate that something was (presumably) deleted, but we don't know who deleted what, at what time, in what place. So a verb SID takes as argument a sequence of S-expressions that tell us these various aspects. Those S-expressions are headed by *role SIDs*. A role SID governs information about the "players" associated with the verb. The role denotes what part an entity plays in a sentence. An example of a role SID is `Initiator`.

Like verb SIDs, role SIDs take as argument a sequence of S-expressions. These S-expressions are atom clauses (see below), and identify or describe the entity playing the indicated role. An S-expression headed by a role SID is called a *role clause*.

The purpose of role SIDs is to provide some context for the atom SIDs, which actually carry the load of describing entities. We made the observation that describing an entity can be separated from identifying what role that entity plays within a sentence. Instead of having a single atom SID `InitiatorHostName`, for instance, we have a role SID `Initiator` and an atom SID `HostName`.

The benefits of this division include clarity of presentation as well as a reduction in the number of SIDs. If we imagine that there are m different roles to play, and n different ways to describe each role, then separating role SIDs from atom SIDs reduces the number of SIDs needed to express these combinations from mn to $m + n$.

There is also a class of role SIDs that describe the location and time of an action, as well as its outcome (failure, success, or something in between). These are called *modifier* role SIDs.

The SIDs inside role clauses are *atom SIDs*. Atom SIDs are the workhorses of S-expressions. Verb and role SIDs are placeholders that organize the sentence, but the atom SIDs are required to instantiate those placeholders. Within an `Initiator` role SID, for instance, one will typically find atom SIDs that describe and identify that `Initiator`. Examples of atom SIDs are `Username` and `IPv4Address`. An S-expression headed by an atom SID is called an *atom clause*, or sometimes a *SID-data pair*.

By far the great majority of SIDs are atom SIDs. There are atom SIDs that describe users, those that describe processes, those that describe machines, those that describe times, those that describe statistical data, and so forth. In addition, there are also special-purpose atom SIDs to express notions for Kerberos principals, for instance, or for objects in the Andrew File System. The list of atom SIDs is potentially quite long. Part of the ongoing work in the CIDF effort is to generate a list of atom SIDs to cover the major areas that intrusion detection and response systems would find of interest.

Conjunction SIDs, as their name implies, join sentences together. The most common case is simply `And`. For example,

```
(And <Sentence1> <Sentence2>)
```

means only that `Sentence1` and `Sentence2` both hold. It says nothing about the order in which they occurred, or about any causal relation between the two. There are other conjunction SIDs which *do* express some of these relationships, however. An example is `InSequence`, which occurred in the example above. `InSequence` joins two or more sentences together, and implies that they occurred in forward chronological order (earliest to latest).

Referent SIDs allow one to link two or more parts of the same sentence together (or possibly, two or more different sentences). There are only two referent SIDs, `ReferAs` and `ReferTo`. They are syntactically atom SIDs—they take a single long integer as their data type—but they differ from other atom SIDs in that the integer is semantically opaque. Roughly speaking, a `ReferAs` "catches" a reference made by a `ReferTo`.

In the example above, the clause headed by the role SID `Account` contains the referent SID `ReferAs`. By this means, the object playing the `Account` role here is given a "name," the value `0x12345678`. This allows later reference to this object by name, rather than by describing it anew.

This is done in the clause headed by the role SID `Initiator` under the verb SID `Delete`. Instead of being described explicitly here, the initiator is denoted by making a reference, using the referent SID `ReferTo`, to the object described in the `Account` clause.

The referent SIDs can also be used in a similar way to refer to previous sentences, rather than just role clauses within sentences.

It is important to note that referent SIDs *identify* an object, rather than simply describe it. In other words, the use of the `ReferTo` SID indicates that the component generating this message believes the referred to object to be the same as the previously described object, not just one matching that description.

The scope rule applying to referent SIDs is as follows: The value of a referent clause is the verb or role within which it is found (roughly speaking), provided that that verb or role is in the same thread. A thread is defined as follows:

- Any two parts of an S-expression (which may contain other S-expressions) is in the same thread. For instance, the entire example in Section 4.1 is all in the same thread.
- If sentences are being transported with a GIDO header, then then a thread is further extended to cover all GIDOs with the same originator ID and thread ID.

5.3 SID Extensions

It is not expected that any component will understand all SIDs. A component concerned with Unix notions will often not be worried about X.500-related SIDs. Nevertheless, many X.500-related SIDs have their complements in the Unix world, and the Unix component will want to capture this information, even if it isn't cognizant of the exact use of this information in the X.500 world. For instance, a user's real name is a user's real name, although in Unix it might be the name in `/etc/passwd` associated with the user's account, and in X.500 it may be a Common Name. If these two concepts were expressed with two completely distinct SIDs, then we would lose much of the benefits of data sharing.

SID extensions are designed to address this. SID extensions allow one to specify information in a relatively generic fashion, and then give more specialized receivers extra information about a SID that specifies more precisely how it is to be used. For instance, an X.500 Common Name would be expressed as follows:

```
(RealName (ExtendedBy X500CommonName) 'Joe Cool')
```

In this example, `RealName` is the base SID, and `X500CommonName` is the SID extension. Most components would be able to understand the `RealName` SID, and would be able to capture the fact that the a user with the real name `'Joe Cool'` is in question here. Additionally, any component who understands X.500 could implement the `X500CommonName` extension, allowing it to determine that the real name is registered as a Common Name, along with any implications of that fact.

An extended SID always takes the same *type* as the unextended (base) SID. Furthermore, the interpretation of the SID extension always *entails* the interpretation of the base SID, so that there is no danger of misinterpreting the whole clause by ignoring the SID extension.

5.4 Defining New SIDs

In order to satisfy our efficiency requirement, we permit new SIDs to be defined which are in essence macros built out of existing SIDs. For example, one can define a `DeleteFile` SID as follows:

```
(def DeleteFile ($username $hostname $filename $time)
    (Delete
        (Context
            (HostName $hostname)
            (Time $time)
        )
        (Initiator
            (UserName $username)
        )
        (Source
            (FileName $filename)
        )
    )
)
```

Then, later, we can write

```
(DeleteFile 'joe' 'ten.ada.net' '/etc/passwd' '14:58:12 24 Feb 1998')
```

and that will be expanded by the receiver to

```
(Delete
    (Context
        (HostName 'ten.ada.net')
        (Time '14:58:12 24 Feb 1998')
    )
    (Initiator
        (UserName 'joe')
    )
    (Source
        (FileName '/etc/passwd')
    )

)
```

The general form of a def construct is

```
(def <NewSID> (<DummyArgumentList>)
    <SIDExpansion>
)
```

When used, `NewSID` must take on as many arguments as there are in `DummyArgumentList`, and each argument must be of the type implied by the `SIDExpansion`. The value of this use is `SIDExpansion` with each dummy argument replaced by the actual argument given.

The primary motivation behind the `def` structure is to allow many identical messages with the same data format to be represented compactly, with only the data filling the majority of the message, since only one SID needs to be recorded per message, rather than one SID per data item.

6 Conclusion

In this paper, we considered the issues involved in designing a framework in which intrusion detection and response (ID&R) systems may be connected as components and work together to provide additional functionality as a larger, integrated system. We discussed various ways that such systems are connected today, and showed how they can be abstracted so as to provide a common interface. We examined the requirements of any language designed to allow two or more ID&R components to communicate with one another about attacks and other events and system conditions, and we described one approach to designing a language to satisfy those requirements.

This work is far from complete and is an ongoing effort. In the future, we will study how components may find each other, given only their spheres of interest. We will also examine practical issues involved in integrating this work into existing systems.

References

[1] M. Bishop. A standard audit trail format. In *Proceedings of the 18th National Information Systems Security Conference, Baltimore*, pages 136–145, 1995.

[2] M. Crosbie and E.H. Spafford. Active defense of a computer system using autonomous agents. Technical report, Department of Computer Sciences, CSD-TR-95-008, Purdue University, West Lafayette IN, 1995.

[3] Boeing Defense and Space Group. Protocol definition. intruder detection and isolation protocol concept. *Boeing Document Number D658-10732-1*, 199X.

[4] D. Frinke, T. Johnson, J. Marconi, and D. Polla. Towards a distributed architecture for cooperative intrusion detection. *Pending*, 199X.

[5] GAO. Information security: Computer attacks at department of defense pose increasing risks. *GAO Report Number AIMD-96-84*, 1996.

[6] The Open Group. *Distributed Audit Service Preliminary Specification (Company Review Version.* The Open Group, Apex Plaza, Forbury Road, Reading, Berkshire, RG1 1AX, UK, 1997.

[7] The Open Group. *Systems Management: Event Management Service Preliminary Specification.* The Open Group, Apex Plaza, Forbury Road, Reading, Berkshire, RG1 1AX, UK, 1997.

[8] L. T. Heberlein, K. N. Levitt, and B. Mukherjee. A method to detect intrusive activity in a networked environment. In *Proceedings of the 14th National Computer Security Conference*, pages 362–371, Washington, D.C., 1991.

[9] Todd Heberlein. *Non-cooperative Service Recognition.* http://www.hokie.bs1.prc.com/ia/bbheberl/tsld020.htm, 1997.

[10] H.S. Javitz and A. Valdes. The sri ides statistical anomaly detetor. In *IEEE Symposium on Research in Security and Privacy*, 1991.

[11] Lawrence Livermore National Laboratory. *Network Intrusion Detector.* http://ciac.llnl.gov/cstc/nid/, 1998.

[12] P.A. Porras and P.G. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *National Information Systems Security Conference*, pages 353–365, Baltimore, MD, October 1997.

[13] K. Price. Host based misuse detection and conventional operating systems' audit data collection. Master's thesis, Purdue University, 1997.

[14] Ronald Rivest. *S-expressions.* Internet Draft draft-rivest-sexp-00.txt, 1997.

[15] G.B. White, E.A. Fisch, and U.W. Pooch. Cooperating security managers: A peer-based intrusion detection system. pages 20–23, February 1996.