ELSEVIER

Computer Networks 34 (2000) 547–570

COMPUTER NETWORKS

www.elsevier.com/locate/comnet

Intrusion detection using autonomous agents

Eugene H. Spafford, Diego Zamboni *

Center for Education and Research in Information Assurance and Security, 1315 Recitation Building, Purdue University, West Lafayette, IN 47907-1315, USA

Abstract

AAFID is a distributed intrusion detection architecture and system, developed in CERIAS at Purdue University. AAFID was the first architecture that proposed the use of autonomous agents for doing intrusion detection. With its prototype implementation, it constitutes a useful framework for the research and testing of intrusion detection algorithms and mechanisms. We describe the AAFID architecture and the existing prototype, as well as some design and implementation experiences and future research issues. © 2000 Elsevier Science B.V. All rights reserved.

Keywords: Intrusion detection; Software agents; Distributed systems; Security; Perl

1. Introduction

The intrusion detection field has grown considerably in the last few years, and a large number of intrusion detection systems have been developed to address different needs. Intrusion detection is clearly necessary with the growing number of computer systems being connected to networks. We describe an architecture for intrusion detection and system monitoring based on autonomous agents that serves as a research framework for intrusion detection techniques and algorithms.

We start by defining some common terms and the motivation for using autonomous agents in an intrusion detection system.

1.1. Intrusion detection

Intrusion detection is defined as "the problem of identifying individuals who are using a computer system without authorization (i.e., 'crackers') and those who have legitimate access to the system but are abusing their privileges (i.e., the 'insider threat')" [25]. We add to this definition the identification of *attempts* to use a computer system without authorization or to abuse existing privileges. Our working definition matches the one given by Heady et al. [15], where an intrusion is defined as "any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource", disregarding the success or failure of those actions.

The dictionary definition of the word *intrusion* [24] does not include the concept of an insider abusing his or her privileges to perform unauthorized actions, or attempting to do so. A more accurate phrase to use is *intrusion and insider abuse detection*. In this document we use the term intrusion to mean both intrusion and insider abuse.

An intrusion detection system is a computer system (possibly a combination of software and hardware) that attempts to perform intrusion detection, as defined above. Most intrusion detection systems try to perform their task in real time [25],

* Corresponding author.
E-mail addresses: spaf@cerias.purdue.edu (E.H. Spafford), zamboni@cerias.purdue.edu (D. Zamboni).

PII: S1389-1286(00)00136-5

but there are also intrusion detection systems that do not operate in real time, either because of the nature of the analysis they perform (e.g., [20]) or because they are geared for forensic analysis [13,32].

Intrusion detection systems are usually classified as host-based or network-based [25]. Host-based systems base their decisions on information obtained from a single host (usually audit trails), while network-based systems obtain data by monitoring the traffic in the network to which the hosts are connected.

The definition of an intrusion detection system does not include preventing the intrusion from occurring, only detecting it and reporting it to an operator. There are some intrusion detection systems (for example, [6]) that try to react when they detect an unauthorized action occurring. This reaction usually includes trying to contain or stop the damage, for example, by terminating a network connection.

1.2. Desirable characteristics of an intrusion detection system

Crosbie and Spafford [10] defined the following characteristics as desirable for an intrusion detection system:

- It must *run continually* with minimal human supervision.
- It must be *fault tolerant* by being able to recover from system crashes, either accidental or caused by malicious activity. Upon startup, the intrusion detection system must be able to recover its previous state and resume its operation unaffected.
- It must *resist subversion*. The intrusion detection system must be able to monitor itself and detect if it has been modified by an attacker.
- It must impose a *minimal overhead* on the systems where it runs, to avoid interfering with the systems normal operation.
- It must be *configurable* to accurately implement the security policies of the systems that are being monitored.
- It must be *adaptable* to changes in system and user behavior over time. For example, new applications being installed, users changing from one activity to another or new resources being available can cause changes in system use patterns.

As the number of systems to be monitored increases and the chances of attacks increase we also consider the following characteristics as desirable:

- It must be *scalable* to monitor a large number of hosts while providing results in a timely and accurate manner.
- It must provide *graceful degradation of service*. If some components of the intrusion detection system stop working for any reason, the rest of them should be affected as little as possible.
- It must allow *dynamic reconfiguration*, allowing the administrator to make changes in its configuration without the need to restart the whole intrusion detection system.

1.3. Distributed and centralized intrusion detection systems

Intrusion detection systems are also usually classified by the way their components are distributed

- A *centralized intrusion detection system* is one where the analysis of the data is performed in a fixed number of locations, independent of how many hosts are being monitored. We do not consider the location of the data collection components, only the location of the analysis components. Some intrusion detection systems that we classify as centralized are: IDES [11,12,22,23], IDIOT [7,21], NADIR [17] and NSM [16].
- A *distributed intrusion detection system* is one where the analysis of the data is performed in a number of locations proportional to the number of hosts that are being monitored. Again, we only consider the locations and number of the data analysis components, not the data collection components. Some intrusion detection systems that we classify as distributed are: DIDS [29,30], GrIDS [4,31], EMERALD [26] and AAFID [1].

1.3.1. How centralized and distributed intrusion detection systems compare

Table 1 comments on the advantages and disadvantages of centralized and distributed intrusion

detection systems with respect to the desirable characteristics described in Section 1.2.

We can see that centralized intrusion detection systems have some advantages over distributed intrusion detection systems, but these advantages are not insurmountable through technical means. Centralized intrusion detection systems, however, have some fundamental limitations, such as their lack of scalability and the difficulty to provide graceful degradation of service. The intrusion detection field has been shifting in the last few years towards designing and building distributed intrusion detection systems (e.g., [1,2,18,26,31]). In this paper, we discuss one approach to building such a system using autonomous agents.

1.4. Autonomous agents

A software agent can be defined as [3]:

> ...a software entity which functions continuously and autonomously in a particular environment...able to carry out activities in a flexible and intelligent manner that is responsive to changes in the environment...Ideally, an agent that functions continuously...would be able to learn from its experience. In addition, we expect an agent that inhabits an environment with other agents and processes to be able to communicate and cooperate with them, and perhaps move from place to place in doing so.

Table 1
Comparison between centralized and distributed intrusion detection systems with respect to the desirable characteristics described in Section 1.2

Characteristic	Centralized	Distributed
Run continually	A relatively small number of components need to be kept running.	Harder because a larger number of components need to be kept running.
Fault tolerant	The state of the intrusion detection system is centrally stored, making it easier to recover it after a crash.	The state of the intrusion detection system is distributed, making it more difficult to store in a consistent and recoverable manner.
Resist subversion	A smaller number of components need to be monitored. However, these components are larger and more complex, making them more difficult to monitor.	A larger number of components need to be monitored. However, because of the larger number, components can cross-check each other. The components are also usually smaller and less complex.
Minimal overhead	Impose little or no overhead on the systems, except for the ones where the analysis components run, where a large load is imposed. Those hosts may need to be dedicated to the analysis task.	Impose little overhead on the systems because the components running on them are smaller. However, the extra load is imposed on most of the systems being monitored.
Configurable	Easier to configure globally, because of the smaller number of components. It may be difficult to tune for specific characteristics of the different hosts being monitored.	Each component may be localized to the set of hosts it monitors, and may be easier to tune to its specific tasks or characteristics.
Adaptable	By having all the information in fewer locations, it is easier to detect changes in global behavior. Local behavior is more difficult to analyze.	Data are distributed, which may make it more difficult to adjust to global changes in behavior. Local changes are easier to detect.
Scalable	The size of the intrusion detection system is limited by its fixed number of components. As the number of monitored hosts grows, the analysis components will need more computing and storage resources to keep up with the load.	A distributed intrusion detection system can scale to a larger number of hosts by adding components as needed. Scalability may be limited by the need to communicate between the components, and by the existence of central coordination components.
Graceful degradation of service	If one of the analysis components stops working, most likely the whole intrusion detection system stops working. Each component is a single point of failure.	If one analysis component stops working, part of the network may stop being monitored, but the rest of the intrusion detection system can continue working.
Dynamic reconfiguration	A small number of components analyze all the data. Reconfiguring them likely requires the intrusion detection system to be restarted.	Individual components may be reconfigured and restarted without affecting the rest of the intrusion detection system.

For our purposes, we define an *autonomous agent* (henceforth *agent*) as a software agent that performs a certain security monitoring function at a host.

We term the agents as *autonomous* because they are independently-running entities (i.e., their execution is scheduled only by the operating system, and not by another process). Agents may or may not need data produced by other agents to perform their work. Additionally, agents may receive high-level control commands – such as indications to start or stop execution, or to change some operating parameters – from other entities. Neither of these characteristics detriments our definition of agent autonomy.

1.4.1. Using autonomous agents to build a better distributed intrusion detection system

Because agents are independently-running entities, they can be added, removed and reconfigured without altering other components and without having to restart the intrusion detection system. Agents can be tested on their own before introducing them into a more complex environment. An agent may also be part of a group that performs different simple functions but that can exchange information to derive more complex results than any one of them may be able to obtain on their own.

We analyze the performance of an intrusion detection system built using autonomous agents with respect to the desirable characteristics listed in Section 1.2:

Continuous running: If an intrusion detection system is constituted of a number of autonomous agents, some of them may be taken off-line for maintenance or for other reasons while others keep running, therefore providing intrusion detection functionality on a continuous manner.

Fault tolerance: The storage and recovery of global state is still problematic, as described in Section 1.3 for distributed intrusion detection systems. Autonomous agents, however, would be able to keep local state and recover it upon startup.

Resist subversion: Self-monitoring in autonomous agents is a difficult problem, but it is also the subject of current research (e.g., [14]). One possibility is to have agents do cross-verification, with each agent being checked periodically by several others.

If an agent collects network information related to the host where it is running, we reduce the possibility of insertion and evasion attacks [27], which are also a form of subversion, and to which network-based intrusion detection systems are usually subject.

Minimal overhead: Well-designed agents can impose a minimal load on the system where they are running. Furthermore, agents can be enabled and disabled dynamically, making it possible to use resources only for the tasks needed at each moment.

Configurable: An agent can be configured (or even implemented) specifically for the needs of the host where it will run. Autonomous agents therefore provide the possibility for fine-grained configuration abilities.

Adaptable: Each agent has local knowledge that makes it able to adapt to changes in local behavior. By having a higher-level view of the system state, it may be possible to adapt to changes in global behavior as well. One way of adapting is to automatically add or remove agents to monitor things that are deemed interesting at a certain point in time.

Scalable: Agents are deployed to the hosts that need to be monitored. Therefore, an intrusion detection system built using agents can grow simply by deploying new agents as needed. The bottleneck may be the communication mechanisms between the agents and the central coordination components, if they exist, as well as the processing capabilities of those components. Both of these problems can be solved, as has been shown by proposed schemes to build intrusion detection systems without the need of a central controlling entity [18,34] and that minimize communication needed between components [18].

Even when using traditional communication schemes, the intrusion detection system can be made scalable by organizing the agents in a hierarchical structure. This idea was proposed by Crosbie and Spafford [9] and was also used by Staniford-Chen et al. [31].

Graceful degradation of service: If one agent stops working for any reason, one or two things may happen

- If the agent produces results on its own, only its results will be lost. All other agents will continue to work normally.
- If the data produced by the agent was needed by other agents, that group of agents may be impeded from working properly. Even in this case, the data dependencies between agents are known in advance, so the consequences of failure can be predicted.

In any case, the damage is restricted to at most a set of agents. Thus, if the agents are properly organized in mutually independent sets, the degradation of the service provided by the intrusion detection system will be gradual and proportional to the number of agents that stop functioning.

Dynamic reconfiguration: The ability to start and stop agents independently of each other creates the possibility of reconfiguring the intrusion detection system without having to restart it. If we need to start collecting a new type of data or monitoring for a new kind of attacks, the appropriate agents can be started without disturbing the ones that are already running. Similarly, agents that are no longer needed can be stopped, and agents that need to be reconfigured can be sent the appropriate commands without having to restart the whole intrusion detection system.

Additional benefits: Using agents as data collection and analysis entities also provides the following desirable features:

- Because an agent can be programmed arbitrarily, it can obtain its data from an audit trail, by probing the system where it is running, by capturing packets from a network, or from any other suitable source. Thus, an intrusion detection system built using agents can cross the traditional boundaries between host-based and network-based intrusion detection systems.
- Because agents can be stopped and started without disturbing the rest of the intrusion detection system, agents can be upgraded as increased functionality is required from them. As long as their external interface remains compatible, other components need not even know that the agent has been upgraded.
- If agents are implemented as separate processes on a host, each agent can be implemented in the programming language that is best suited for the task that it has to perform.

2. The AAFID architecture

We propose an architecture called autonomous agents for intrusion detection (AAFID) for building intrusion detection systems that use agents as their lowest-level element for data collection and analysis.

2.1. History of the AAFID architecture

What was to become the AAFID architecture was first proposed by Crosbie and Spafford in 1994 [8]. In this paper, the authors proposed (for the first time in the published literature) the idea of using autonomous agents for performing intrusion detection, and suggested that the agents could be evolved automatically using genetic programming so that the intrusion detection system would automatically adjust and evolve according to user behavior.

The idea of using genetically programmed agents was never fully implemented and tested. However, the idea of using agents for intrusion detection kept evolving, and between 1995 and 1996 the AAFID architecture was developed in the COAST Laboratory. The initial architecture had the hierarchical structure that remains to date, included monitors, transceivers and agents, and was used to implement the first prototype of the system.

From 1997 to date the AAFID architecture evolved with the addition of filters and the separation of the user interface from the monitor. The new architecture has been used for the implementation of the latest prototype.

2.2. Overview

A simple example of an intrusion detection system that adheres to the AAFID architecture is shown in Fig. 1(a). This figure shows the four components of the architecture: agents, filters,

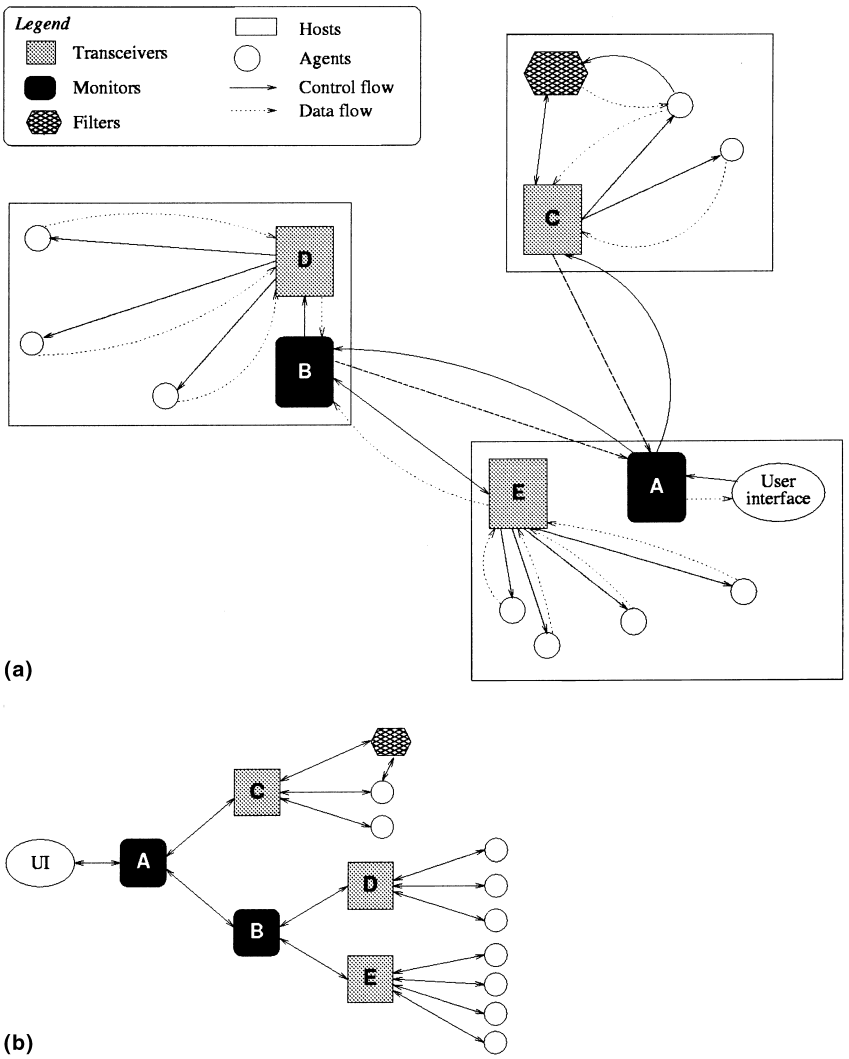

Fig. 1. Physical and logical representations of a sample intrusion detection system that follows the AAFID architecture (called an AAFID system). (a) Physical layout of the components in a sample AAFID system, showing agents, filters, transceivers and monitors, as well as the communication and control channels between them. (b) Logical organization of the same AAFID showing the communication hierarchy of the components. The bidirectional arrows represent both the control and data flow between the entities. Notice that the logical organization is independent of the physical location of the entities in the hosts.

transceivers and monitors. We refer to each one of these components as AAFID entities or simply *entities*, and to the whole intrusion detection system constituted by them as an *AAFID system*.

An AAFID system can be distributed over any number of hosts in a network. Each host can contain any number of *agents* that monitor for interesting events occurring in the host. Agents may use *filters* to obtain data in a system-independent manner. All the agents in a host report their findings to a single *transceiver*. Transceivers are per-host entities that oversee the operation of all the agents running in their host. They have the ability to start, stop and send configuration

commands to agents. They may also perform data reduction on the data received from the agents. The transceivers report their results to one or more *monitors*. Each monitor oversees the operation of several transceivers. Monitors have access to network-wide data, therefore they are able to perform higher-level correlation and detect intrusions that involve several hosts. Monitors can be organized in a hierarchical fashion such that a monitor may in turn report to a higher-level monitor. Also, a transceiver may report to more than one monitor to provide redundancy and resistance to the failure of one of the monitors. Ultimately, a monitor is responsible for providing information and getting control commands from a user interface. Fig. 1(b) shows the logical organization corresponding to the physical distribution depicted in Fig. 1(a).

We now describe each component in greater detail.

2.3. Components of the architecture

2.3.1. Agents

An agent is an independently-running entity that monitors certain aspects of a host, and reports to the appropriate transceiver. For example, an agent could be looking for a large number of *telnet* connections to a protected host, and consider their occurrence as suspicious. The agent would generate a report that is sent to the appropriate transceiver. The agent does not have the authority to directly generate an alarm. Usually, a transceiver or a monitor will generate an alarm for the user based on information received from agents. By combining the reports from different agents, transceivers build a picture of the status of their host, and monitors build a picture of the status of the network they are monitoring.

Agents do not communicate directly with each other in the AAFID architecture. Instead, they send all their messages to the transceiver. The transceiver decides what to do with the information based on agent configuration information.

The architecture does not specify any requirements or limitations for the functionality of an agent. It may be a simple program that monitors a specific event (for example, counting the number of telnet connections within the last 5 min, which is an existing agent in the current AAFID implementation), or a complex software system (for example, an instance of IDIOT [7] looking for a set of local intrusion patterns). As long as the agent produces its output in the appropriate format and sends it to the transceiver, it can be part of the AAFID system.

Agents may perform any functions they need. Some possibilities (which have not been used by any existing AAFID agents) are:

- Agents may evolve over time using genetic programming techniques, as suggested in [9].
- Agents may employ techniques to retain state between sessions, allowing them to detect long-term attacks or changes in behavior. Currently, the architecture does not specify any mechanisms for maintaining persistent state.
- Agents could migrate from host to host by combining the AAFID architecture with some existing mobile-agent architecture.

Agents can be written in any programming language. Some functionality (e.g., reporting, communication and synchronization mechanisms) is common to all the agents, and can be provided through shared libraries or similar mechanisms. Thus, a framework implementation, like the one described in Section 3, can provide most of the tools and mechanisms necessary to make writing new agents a relatively simple task.

2.3.2. Filters

Filters are intended to be both a data selection and a data abstraction layer for agents.

In the original AAFID architecture, each agent was responsible for obtaining the data it needed. When the first prototype was implemented, this approach showed the following problems:

- On a single system, there may be more than one agent that needs data from the same data source. This is common in Unix with multi-function log files (such as `/var/adm/messages`). Having each agent read the data on its own meant duplicating the work of reading the file, parsing it and discarding unnecessary records.
- There may be agents that can provide a useful function under different versions of Unix, or

even under different architectures (such as Windows NT). However, the data needed by the agent may be located in different places in each system and may be stored in different formats. This meant having to write a different agent for each system, that knows where to find the data and how to read it.

Both of these problems are solved through the introduction of filters. Filters provide a subscription-based service to agents, and have two functions.

Data selection: There exists only one filter per data source, and multiple agents can subscribe to it. When an agent subscribes to a filter, it specifies which records it needs (using some criteria like regular expressions), and the filter only sends to the agent records that match the given criteria. This eliminates duplicate work in reading and filtering data.

Data abstraction layer: Filters implement all the architecture- and system-dependent mechanisms for obtaining the data that agents need. Therefore, the same agent can run under different architectures simply by connecting to the appropriate filter. This makes it easier to reuse code and to run AAFID under different operating systems.

2.3.3. Transceivers

Transceivers are the external communications interface of each host. They have two roles: control and data processing. For a host to be monitored by an AAFID system, there must be a transceiver running on that host.

In its control role, a transceiver performs the following functions:

- Keeps track and controls execution of agents in its host. The instructions to start and stop agents can come from configuration information, from a monitor, or as a response to specific events (for example, a report from one agent may trigger the activation of other agents to perform a more detailed monitoring of the host).
- Responds to commands issued by its monitor by providing the appropriate information or performing the requested actions.

In its data processing role, a transceiver has the following duties:

- Receives reports generated by the agents running in its host.
- Does appropriate processing on the information received from agents.
- Distributes the information received from the agents, or the results of processing it, either to other agents or to a monitor, as appropriate.

2.3.4. Monitors

Monitors are the highest-level entities in the AAFID architecture. They have control and data processing roles that are similar to those of the transceivers. The main difference is that monitors can control entities that are running in several different hosts whereas transceivers only control local agents.

In their data processing role, monitors receive information from all the transceivers they control, and can do higher-level correlations and detect events that involve several different hosts. Monitors have the capability to detect events that may be unnoticed by the transceivers.

In their control role, monitors can receive instructions from other monitors and they can control transceivers and other monitors. Monitors have the ability to communicate with a user interface and provide the access point for the whole AAFID system. Monitors implement an interface that includes mechanisms for accessing the information that the monitor has, for providing commands to the monitor, or to send commands to lower-level entities such as transceivers and agents.

If two monitors control the same transceiver, mechanisms have to be employed to ensure consistency of information and behavior. The AAFID architecture does not currently specify the mechanisms for achieving this consistency.

2.3.5. User interfaces

The most complex and feature-full intrusion detection system can be useless if it does not have good mechanisms for users to interact with it.

The AAFID architecture clearly separates the user interface from the data collection and processing elements. A user interface has to interact with a monitor to request information and to provide instructions.

This separation allows different user interface implementations to be used with an AAFID system. For example, a graphical user interface (GUI) could be used to provide interactive access to the intrusion detection system, while a command-line based interface could be used in scripts to automate some maintenance and reporting functions.

2.4. Communication mechanisms

The transmission of messages between entities is a central part of the functionality of an AAFID system. Although the AAFID architecture does not specify which communication mechanisms have to be used, we consider the following to be some important points about the communication mechanisms used in an AAFID system:

- Appropriate mechanisms should be used for different communication needs. In particular, communication within a host may be established by different means than communication across the network.
- The communication mechanisms should be efficient and reliable in the sense that they should (a) not add significantly to the communications load imposed by regular host activities, and (b) provide reasonable expectations of messages getting to their destination quickly and without alterations.
- The communication mechanisms should be secure in the sense that they should (a) be resistant to attempts of rendering it unusable by flooding or overloading, and (b) provide some kind of authentication and confidentiality mechanism.

The topics of secure communications, secure distributed computation and security in autonomous agents have been already studied [14,19], and possibly some previous work can be used in AAFID implementations to obtain communication channels that provide the necessary characteristics.

3. The AAFID implementation

An implementation of the AAFID architecture has been an important part of the project because it enables us to test ideas on real situations. In this section we describe the objectives, design and implementation decisions that have been made in the AAFID implementations.

3.1. History of implementations

The first AAFID prototype was implemented between 1995 and 1996, based on the first specification of the AAFID architecture. This prototype was implemented by a combination of programs written in C, Bourne shell, AWK and Perl. Its main objective was to "put something together" to test the initial feasibility of the architecture, and to get some feedback on design and implementation decisions. This first implementation was only tested internally in the COAST Laboratory.

The second implementation, which has evolved into the current version of the AAFID prototype, was started in late 1997, and has evolved to incorporate the more recent changes in the architecture, such as filters. This implementation is referred to as the $AAFID_2$ prototype. In September 1998, $AAFID_2$ was released for the first time to the public. The first release included the basic system, a few agents, and had been tested under Solaris.

In September 1999, a second public release was made. The main change in the new release was the introduction of a new event-processing mechanism (see Section 3.4.5). The new release also included improved development support, and was tested under Linux as well as Solaris.

The latest version of $AAFID_2$ is described in the following sections.

3.2. Objectives of the current prototype

The $AAFID_2$ prototype was designed with a set of specific objectives in mind.

Road-test the architecture. The AAFID architecture seems adequate for an intrusion detection system from a design point of view but we are interested in getting feedback from its use in real situations, in real networks, and facing real attacks and problems. The main objective in the development of $AAFID_2$ was to be able to use it and ship it to people and organizations that can put it to

use, evaluate both the architecture and the implementation, and provide feedback that enables us to identify weaknesses and possibilities for improvement.

Usability. One of our high priorities was to make it as easy to use as possible. The different modules are as independent as possible, and each one of them has a defined interface that allows users to run and interact with it easily.

Configurability and extensibility. Another priority was to make it easy to configure the behavior of the components of $AAFID_2$, even at run time. The modules themselves are easy to configure, and the overall structure of the system (including the layout of modules in the monitored systems, and their interconnections) is easy to specify and set up.

Minimum installation requirements. Many components of $AAFID_2$ are likely to change frequently during the development and testing phases. For this reason, we tried to make it possible to provide a minimum set of programs that have to be preinstalled in the monitored hosts, and have all the other modules distributed automatically when they are needed.

No focus on performance. Although we want $AAFID_2$ to be as efficient as possible, our focus during its development was not in performance, but on identifying and evaluating the characteristics that we want in an AAFID system. Once these features are set, a more efficient implementation can be done based on them.

No focus on agent security. The use of autonomous agents poses security concerns, and ensuring the integrity and authenticity of an agent is a difficult problem. We decided not to address this problem in the implementation of the $AAFID_2$ prototype, because it is a subject of current research [14], and it does not affect the detection capabilities of the AAFID architecture (although it affects its applicability as a production system).

Provide an infrastructure for development. $AAFID_2$ provides a simple intrusion detection system, but more importantly it provides the infrastructure for the implementation of more complex systems by allowing the implementer to use all the underlying facilities (e.g., communications and synchronization) to build complex features without having to worry about the architectural mechanisms.

3.3. Design notes

In this section, we describe some of the design decisions that we took for the development of the $AAFID_2$ prototype.

3.3.1. Communication model

The logical organization of the AAFID architecture (see Fig. 1(b)) maps directly to a hierarchical model of communication. This model is shown in Fig. 2, and based on it we make the following terminology definitions:

Definition 1 (*Upstream*, *downstream*, *above and below*). From a given level in the AAFID communication model as depicted in Fig. 2, we identify levels that are closer to the root of the hierarchy as being *upstream*, and levels that are closer to the leaves of the tree as *downstream*. One entity is *above* another one if it can be reached by following communication paths upstream, and it is *below* if it can be reached by following communication paths downstream. For example, in Fig. 2, entity A is above entities B, C, D and E; entities D and E are below entities A and B, but not below entity C, because there is no upstream-only or downstream-only communication path from C to D and E.

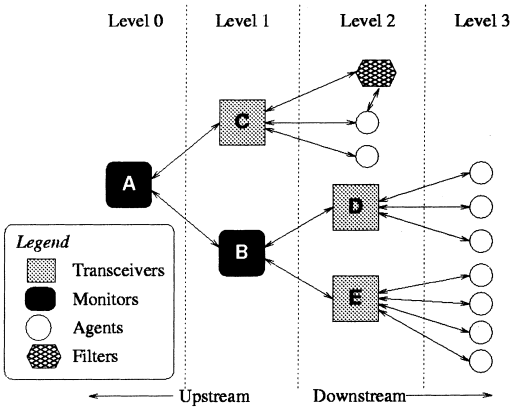

Fig. 2. Model of communication in $AAFID_2$. The arrows represent the communication channels (both for data flow and control) between the entities.

Definition 2 (*Super-entities and sub-entities*). With respect to a specific entity, *sub-entities* are those that are below it, and *super-entities* are those that are above it. For example, in Fig. 2, entity A is a super-entity of all the others and entities D and E are sub-entities of both A and B.

Definition 3 (*Parent and child entities*). For any given entity, a super-entity that is in the level immediately above it is called a *parent entity*. Similarly, a sub-entity that is in the level immediately below it is called a *child entity*. For example, in Fig. 2, entity A is the parent of both C and B, which are its children.

Definition 4 (*Up and down*). When an entity sends a message to an entity that is above it, we say that it sends the message *up*. If it sends a message to an entity that is below it, we say that the message is sent *down*.

Definition 5 (*Controller entity*). A *controller entity* is one that can exert control over a set of other entities. Usually the controller of an entity is its parent entity.

In this model, we can identify layers of entities, with the following properties:

- There is a single node at the top of the hierarchy (the root).
- An entity can only have direct communication with entities immediately above and below its own. The only exception to this rule is the communication between agents and filters, which occurs between entities in the same layer.
- An entity can only have one parent, but it can have multiple children.
- Agents and filters are in the leaves of the tree, and thus can only send messages up. Agents and filters can also send messages between themselves, but only when they are siblings (both are children of the same transceiver).
- The entity in the root of the tree is always a monitor. However, monitors can also appear in intermediate levels.
- The only entity that may be above the root monitor is a user interface, or some other program that is used to control the whole intrusion detection system.

3.3.2. Functionality needed for each entity

One of the first phases in the development of $AAFID_2$ was the identification of the functionality expected from each type of entity, as well as their communication requirements.

3.3.2.1. All entity types. All entities in $AAFID_2$ must be independently-running programs. It must be possible for a user to start an entity from the command line and interact with it by giving messages and commands from the keyboard, but it must also be possible for the entity to be started by its controller entity.

All entities have a unique identifier and a description. The identifier must provide a unique way of referencing an entity. The description must be a brief human-readable description of the entity and its functionality.

All entities must react adequately to a set of messages that allow basic operations such as stopping the entity and querying it for information. Additionally, each entity may define its own commands for implementing specific functionality.

All entities must be able to exchange messages with their controller entity. Any messages received from the controller entity must be processed as soon as possible. The controller entity may be in the same host (for example, a transceiver is the controller entity for its agents), or in a different host (for example, a monitor is the controller entity for the transceivers). Ideally, the implementer of an entity should not have to worry about whether the communication channel is local or over the network.

We also decided that $AAFID_2$ should include facilities in the entity infrastructure for debugging and generation of log messages.

3.3.2.2. Agents. Agents collect information from the system where they run and monitor for specific situations that may indicate a security problem. The general behavior of an $AAFID_2$ agent is described by Algorithm 1. This structure allows agents to react to different types of events, including timing, file and signal events.

Algorithm 1. Generic behavior of an agent. Lines in italics represent sections that have to be provided by the author of the agent.

```
{Instantiation of the agent}
{Instantiation-time initialization}
Set event handlers
{Execution of the agent}
Run-time initialization
Set more event handlers
Contact necessary filters
Enter Event loop
```

In the initial release of $AAFID_2$, agents had a much stricter structure based on a polling mechanism. This structure is shown in Algorithm 2. The polling structure is sufficient for many agents, which periodically monitor for some activity or event, and report their findings. However, it also has severe limitations. The general structure is able to implement the simplified structure.

Of the tasks in the algorithms, only the lines shown in italics are specific to each agent. All the other functions are provided by the infrastructure to allow the creation of new agents in a minimum amount of time.

Algorithm 2. Polling structure for an agent. Lines in italics represent sections that have to be provided by the author of the agent.

```
{Instantiation of the agent}
Instantiation-time initialization
{Execution of the agent}
Run-time initialization
loop
   Perform checks
   if [abnormal condition detected] then
      Generate STATUS_UPDATE message
      with new status information
   if [STOP message was received] then
      Cleanup and exit
   Process other inputs
   Sleep for a certain amount of time (inter-
   check period)
```

3.3.2.3. Filters. Filters are the only entities in the AAFID architecture that can communicate with another entity at their same level in the hierarchy (see Fig. 1(b)). Filters are started by transceivers, but receive commands from and provide data to agents directly. Therefore, a filter needs to be able to receive control data from two channels (not only one, as all other entities), and must be able to provide data through a "side channel" that is different from the normal communication link to its controlling entity.

Filters are data sources. Therefore, they must have facilities for accessing files and other sources of data on the system where they run. On occasion, a filter may need to run with increased privileges to be able to access system data.

3.3.2.4. Transceivers. Transceivers are in charge of controlling all the agents running in a host. Therefore, they do not need remote communication capabilities downstream, but they need to be able to communicate with a large number of local entities. Agents may provide both data (in the form of status updates) and commands (for example, to request an additional module needed for execution), and the transceiver may need to send commands to the agents (for example, to set a parameter value). Transceivers also need to be able to respond to commands from monitors, as well as provide them with status updates. Finally, if a transceiver receives a message from an agent that it is not able to interpret, the message should be forwarded to the transceiver's parent entity for further processing.

3.3.2.5. Monitors. Monitors are the most complex entities with respect to communication capabilities. A monitor must have all the functionality of a transceiver, because it also needs to be able to control local entities. Additionally it must be able to start and control remote entities. In particular, it must be able to start new transceivers or monitors in remote hosts and communicate with them. The monitor only communicates with monitors and transceivers and not with all the agents that may be running in a host. This abstraction corresponds with the AAFID architecture and communication model, and it helps scalability because it reduces the number of entities that a monitor has to track.

A monitor must also be able to listen for connections from remote entities. This is because a

transceiver or monitor may be started on its own in a host and need to be controlled by an already existing monitor. The new entity must be able to contact the monitor and register with it by sending a CONNECT message.

Once a remote entity is started or a connection request has been processed, the monitor must be able to communicate with it in the same way as with local entities.

Finally, monitors have the responsibility of acting as repositories for all the information that their sub-entities may need. For example, a transceiver may be started with few local resources, and when it needs a module whose code is not locally present, it will request it of its monitor. The monitor must be able to locate the necessary code and provide it to the transceiver. If the monitor itself does not have the requested code, it must forward the request to its own controller entity. If the code eventually arrives, the original request from the transceiver must be immediately fulfilled.

3.3.3. Object model

Based on the requirements and design decisions described in Section 3.3.2 we designed the class hierarchy shown in Fig. 3. These classes correspond almost directly with the functionality described in Section 3.3.2. In the Perl implementation, all the class names are prefixed with AAFID:: (for example, the full name of Entity is AAFID::Entity), but for the sake of space and clarity we only make reference to them by their distinctive name in the rest of this paper.

Subclasses inherit all the base functionality from their subclass, and can extend it. For example, a Monitor behaves essentially as a ControllerEntity, but it adds functionality and modifies some of the features of the base class.

3.3.4. Entity status

Each entity in $AAFID_2$ is able to keep a status value, which represents whether it has detected a problem, and the seriousness of the problem. Each entity must be able to report its status upon request. The status of the whole intrusion detection system is computed from the status of all the individual entities.

The status is kept using entity parameters. All entities must keep its status as a numeric indicator in a parameter called Status, and a textual description of it in a parameter called Message.

The value of Status must be a number between 0 and 10 inclusive, where 0 means "all normal" and 10 means "extremely alarmed", and the values in between represent different degrees of importance of the problem detected. No formal definitions have been given for the values of the status indicator, and in $AAFID_2$ it is up to each entity to assign them.

3.3.5. Messages and commands

Messages in the communication model flow through the edges of the graph shown in Fig. 2. At the architectural level, we do not worry about the semantic contents of the messages, so we defined a message format with the following fields for $AAFID_2$ messages:

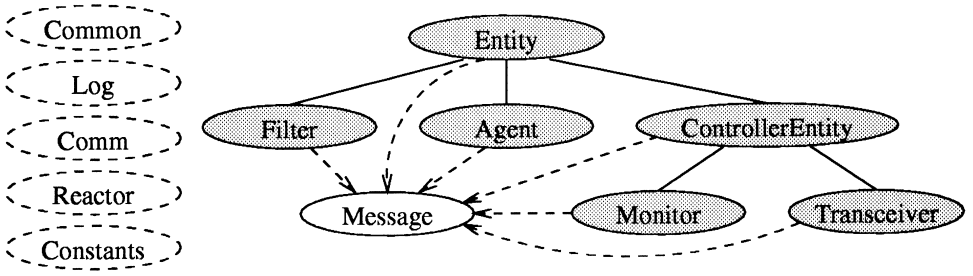

Fig. 3. Class hierarchy in $AAFID_2$. Solid lines represent inheritance relationships between the object classes. Dashed lines represent "uses" relationships, meaning that a class uses another one internally. Grey ovals represent the classes in the main class hierarchy, that implement the primary entity types. Dashed ovals represent auxiliary classes that are not instantiated as objects, but that contain methods that are used by all the other classes.

- Message type.
- Message subtype.
- Source identifier.
- Destination identifier.
- Time stamp.
- Data.

This generic structure can be used by entities to represent any type of information. The kind of message is denoted by the type field. This field also implicitly determines the format of the data field (the receiving entity must know the message type and must know how to interpret the data field accordingly). The subtype field can be used to provide additional information about the contents of the message, as well as to indicate the specific semantic interpretation that should be given to the data field.

The source and destination identifiers must contain information that uniquely identifies the sending and receiving entities. The destination field may be empty if the message is sent over a one-to-one communication channel, where there is no ambiguity about who the receiver is.

The time stamp contains a unique representation of the time when the message was generated. $AAFID_2$ uses the number of seconds elapsed since the Unix epoch (1 January 1970), which is a commonly available value in programming languages under Unix.

Table 2 describes the base message types that we have defined for $AAFID_2$. The content of the data field is specified for each one.

The COMMAND message type provides entities with the capability of defining their own functionality, which can be accessed through special commands by parent entities or users. We have also defined a basic set of commands to which any entity in $AAFID_2$ must respond. This standard set of commands is shown in Table 3.

3.4. Implementation notes

We now describe some of the implementation decisions and issues we faced while developing $AAFID_2$.

3.4.1. Selection of a programming language

$AAFID_2$ is implemented in Perl 5 [28,33]. The choice of this programming language was influenced by the following factors:

- *Ease of prototyping*. Perl is a good language for quickly prototyping and testing ideas. This was one of our objectives for the $AAFID_2$ prototype, so Perl was a good choice.
- *Portability*. Perl can be used in most versions of Unix, as well as on Windows 95 and NT. This makes it extremely attractive to program a distributed application that we eventually want to run in a large number of systems.
- *Extreme flexibility*. Being an interpreted language, Perl provides an unprecedented flexibility in what a program can do. For example, we can add code to a module at run time, even code provided by the user. This allows the modification of parameters, internal variables, and even subroutines without the need to restart the programs. In our prototype, it also makes it easier to distribute code over the network, because the code can be shipped as source and evaluated by the recipient.

$AAFID_2$ was implemented using the object-oriented features of Perl 5. This makes it possible to directly implement the object model described in Section 3.3.3.

During the development and testing of $AAFID_2$ we came to also realize some of the disadvantages of using Perl for the implementation:

- *Big resource usage*. The Perl interpreter is a large program, and it has to be in memory whenever a Perl program is executed. This results in large memory and CPU usage footprints.
- *Excessive flexibility*. The flexibility that we praised as an advantage can also be a problem. For example, the lack of strict type checking and the looseness with which Perl code can be written can lead to programs that are difficult to maintain when their size grows. We found that we had to be particularly careful in documenting and structuring the code to avoid problems. Additionally, the object model in Perl has significant differences with those of other object-oriented programming languages, which can lead to confusions and awkwardness in the programs.

Table 2
Standard message types defined in $AAFID_2$

NOTYPE	
Subtypes	N/A
Description	Indicates that the message has not been initialized. This value can also be used in the message subtype field if that field has no specific value, or its value is irrelevant
Data field	N/A
CONNECT	
Subtypes	CHILD, PARENT, FILTER, GUI
Description	Sent by an entity when it starts. It can be sent upstream to a parent entity, downstream to children entities, and by filters and user interfaces. The subtype field indicates who is sending the message
Data field	Information about the entity. In $AAFID_2$ it contains the entity's identifier and description
DISCONNECT	
Subtypes	CHILD, PARENT, FILTER GUI
Description	Signals termination of the entity that sends the message. The purpose of this message is to allow the receiving entity to perform actions to maintain consistent internal information
Data field	Same as in CONNECT
STATUS_UPDATE	
Subtypes	Irrelevant
Description	The message contains a status update that the sender wants the recipient to have
Data field	Current status and descriptive message in the form of two subfields called Status and Message that contain the current numeric status and a descriptive message of the situation. The format of these fields is the Perl syntax for hash specification. For example `Status =>0`, `Message =>"No problems"`
COMMAND	
Subtypes	Command name or RESULT
Description	Specifies a command that the receiving entity must execute. The subtype field contains the name of the command to execute. If a command produces a result, it will be sent back to the entity that requested the command in a message of type COMMAND and subtype RESULT. Each entity can define its own commands in addition to the standard set. See Table 3 for the list of standard commands defined in $AAFID_2$
Data field	Named parameters to the command, in Perl hash-specification format (`Key => Value`, separated by commas). In a RESULT message, it contains the result produced by the command (which should also be in the form of named parameters) plus a special parameter called Command that contains the name of the command that produced the result
STOP	
Subtypes	Irrelevant
Description	Stops the execution of the entity, performing any necessary cleanup actions. This should usually involve at least sending a DISCONNECT message to any parent and children entities with which the entity has communication
Data field	Irrelevant

- *Security problems*. Some of Perl's features, like the ability to modify the code at run time, can also lead to security problems. However, the current implementation of $AAFID_2$ is intended as a proof of concept and not as a production system, so this was a secondary concern for our purposes.

We believe that the most significant advantages of using Perl are the ease of prototyping and the capability to modify the code at run time.

3.4.2. Entities and parameters

In Perl 5, objects are represented by *blessed references* [33]. A reference can point to any kind of Perl variable, including scalars, arrays, strings or hashes. A common technique in Perl Object-Oriented programming is to represent objects with a reference to an anonymous hash [5]. Hashes in Perl are arrays where the indices can be any value, not only numbers. In particular, the indices of a hash can be strings, and can be used

Table 3
Standard commands defined in $AAFID_2$[a]

STOP	
Parameters	None
Description	Has the same effect as a STOP message
Returns	N/A
EVAL	
Parameters	`Code =>` "*Perl code to execute*"
Description	Allows the execution of arbitrary code in the context of the entity that receives the message. This command is mostly for experimental purposes, for two reasons: it is easy to implement in Perl, but may be impossible to achieve in other languages; and it opens the possibility for security problems. The Code parameter contains the code to execute
Returns	If the code executes without problems, does not produce a return value. If an error occurs, the result contains an ErrorMessage parameter that contains the description of the error
SET_PARAMS	
Parameters	`Parameter =>` *Value* pairs, separated by commas
Description	Allows the specification of values for internal entity parameters, as described in Section 3.4.2
Returns	If the list of parameters and values contains an error, the command returns a description of the error. Otherwise it produces no return value
GET_PARAMS	
Parameters	`Params =>` "*param1*, *param2*,..."
Description	Allows the retrieval of internal entity parameters. The Params parameter must be a string containing a comma-separated list of parameter names
Returns	A list of parameter name–value pairs containing the requested parameters and their values
DUMP_YOURSELF	
Parameters	None
Description	Instructs the entity to produce an internal representation of its current state
Returns	A string representation of the entity (which is actually the current values of all its parameters), contained in the Me parameter. This representation can be used to examine the internal state of the entity, or possibly to initialize another entity to the same state

[a] These commands are specified as subtypes of a message of type COMMAND (see Table 2).

to store named values. An anonymous hash is a hash that is not assigned to a variable, but that can be accessed through a reference stored in a variable. In our case, the reference to the anonymous hash is stored as the representation of the object.

Using an anonymous hash reference to represent an object has the advantage that the hash can be used as the object's internal name space. This makes up for the lack of data inheritance in Perl's object model. Within its own hash reference (the hash reference represents the object, so we normally say "within itself") the object can store any kind of values by name, as if they were instance variables.

In $AAFID_2$ each entity object is represented by an anonymous hash that contains the parameters for that specific entity. The parameter names are used as indices, and the value of each parameter is stored in the corresponding element of the hash. Because a single hash in Perl can store different types of elements simultaneously, each parameter can be of a different type without causing any problems.

An example may make this mechanism clearer. An agent may have the internal representation shown in Fig. 4.

The notation `Key => Value` is used in Perl to specify the elements of a hash. Thus, we may deduct from the information in the figure that the agent has a check period (CheckPeriod) of 10 seconds and its current status (Status) is zero, among other things.

Because each class instance has its own hash, each entity can keep its own state separate from others.

```
{
  Description => 'Check .rhosts files',
  CheckPeriod => 10,
  Status => '0',
  Message => 'Read::Write:',
  MyUsers => {'jdoe'},
  EntityID => 'server1:CheckRhosts:1.06:0',
  InstanceNumber => '0',
  ClassCount => \\1
}
```

Fig. 4. Sample internal representation of an agent.

The Entity class defines a number of methods that can be used to set and query entity parameters.

By convention, none of the parameters used internally by the base classes in $AAFID_2$ starts with the prefix "My", so authors of agents and other entities can use paramaters starting with that prefix (for example, MyUsers and MyKeys) assured that they will not have conflicts with internal parameters.

3.4.3. Messages

3.4.3.1. Format. Messages in $AAFID_2$ are represented by objects of the class *Message*. The main purpose of this class is to store the fields as defined in Section 3.3.5, but also to allow its conversion to and from a format suitable for sending over the network. Inside an entity the messages are stored as an object, but before sending them to another entity, they are converted to a single line of text with the following format:

```
TYPE SUBTYPE FROM TO TIME DATA
```

An entity that receives a message parses it and converts it back into a Message object. Internally, all the fields are represented as strings. The DATA portion of the string representation may contain spaces, depending on the contents of the DATA field.

The utilization of strings to represent messages was decided to make it simpler for human users to provide messages by hand and to monitor the flow of messages between entities. Future implementations may decide to use a more efficient format, particularly for sending messages over a network.

3.4.3.2. Reacting to messages. The behavior of the different message types and commands is implemented through regular Perl subroutines following certain conventions, and thus is easy for someone implementing a new entity type to add functionality.

To add support for a new message type (for example NEWTYPE) a subroutine called `message_NEWTYPE` (where the type name is in uppercase) has to be implemented. When a message of the new type is received, the subroutine will be automatically invoked as an instance method of the entity, which means that a reference to the entity will be its first argument. The second argument will be a Message object containing the message that triggered the call. The subroutine is free to do any processing that it needs. If no return value is necessary, the function should return the `undef` value. If a value is returned, it should be another Message object, and it will be transmitted as-is to the entity that sent the original message.

We have provided a simpler interface for implementing new commands, rather than new message types. For this reason we consider that most extensions to the basic infrastructure functionality should be done through new commands and not new message types. For each command there must be a subroutine called `command_CMD`, where CMD is the command name, in uppercase. When the command is received, the subroutine will be called with three arguments: a reference to the entity itself, the Message object that triggered the call, and a hash containing all the parameter name–value pairs passed in the data field of the message. Although the whole message is available, most commands should only need to examine the hash to extract the values of their parameters and act accordingly. To produce a return value, the command subroutine should return a hash reference containing name–value pairs that will be sent to the entity that requested the command in a message of type COMMAND and subtype RESULT. If there is no return value, the subroutine must return `undef`.

3.4.4. Communication mechanisms

The choice and implementation of the communication mechanisms in $AAFID_2$ was done

with two main objectives in mind: transparency and system independence. To the extent that it is possible, entities do not have to know whether they communicate over a network or within the local host, even when completely different mechanisms may be used. We have put the portions of code that deal with the implementation differences in specific places so that it is easy to locate and modify. Most of the mechanism-specific or platform-specific code is in the Comm and Reactor classes.

$AAFID_2$ uses TCP connections for communication over the network, and Unix pipes for communication within the same host. These mechanisms were chosen because they are readily available in any Unix system, and both provide reliable end-to-end communication. The downside is that none of them provide encryption or authentication capabilities.

3.4.5. Event model

All entities in $AAFID_2$ operate around a single event loop that processes different types of events. Entities operate by defining the appropriate event handlers and waiting for the corresponding events to occur. The event mechanism is implemented by the Reactor class.

In the latest implementation of $AAFID_2$, an entity can set handlers for the following types of events:

File handles. In Unix systems most objects can be accessed through file handles. For $AAFID_2$, the most relevant objects are sockets and pipes. An entity can set a handler for a file handle, and the event loop will cause it to block until data are available. By blocking, the entity uses little resources while it is waiting.

Files. A common application for an agent or a filter is to read data from a file. A file is also accessed through a file handle. However, because of the way regular files are implemented in Unix, it is not possible for a process to block on file handle that represents a regular file. Therefore, the event engine processes regular files differently, by polling periodically from them instead of blocking.

Time. The event engine can also react to time events. Both one-shot and repeating events can be scheduled by an entity, and the event mechanism causes the entity to block until the next time event occurs.

Signals. Finally, the event mechanism can cause an entity to react to Unix signals. The entity can define the appropriate handlers, and they are called automatically when the corresponding signal occurs.

An entity can define handlers for as many events as it wants, including different types of events. The event engine, whenever possible, causes the entity to block until the next event occurs, to reduce resource usage.

3.4.6. Entity loading and execution

The ability to invoke a new local entity and control it is implemented in the ControllerEntity class, and thus is inherited by monitors and transceivers (see Fig. 3). The Monitor class extends this capability by allowing the invocation of remote entities. Additionally each entity must be able to run as a stand-alone program. We now describe the mechanisms used for each of these types of entity execution modes.

3.4.6.1. Entity execution. Every entity must have a method called `run` that will be called as the entry point for the execution of the entity. When the `run` method returns, the execution of the entity is considered finished.

For instantiation-time initialization, every entity may have a method called `Init`, which will be called at the time the object is created. For run-time initialization, an entity may define a method called `runtimeInit`. The difference between the two types of initialization can be seen in Fig. 5. Instantiation-time initialization occurs when the new entity is created inside its corresponding transceiver, whereas run-time initialization occurs once the entity starts executing in its own process. Therefore, any initialization that affects the state of the process (for example, opening files, or creating other child processes) has to be done in the run-time initialization to avoid affecting the transceiver.

For cleanup activities when its execution terminates, an entity may define a method called `Cleanup`.

The Entity class provides the infrastructure for a working entity, including placeholder methods

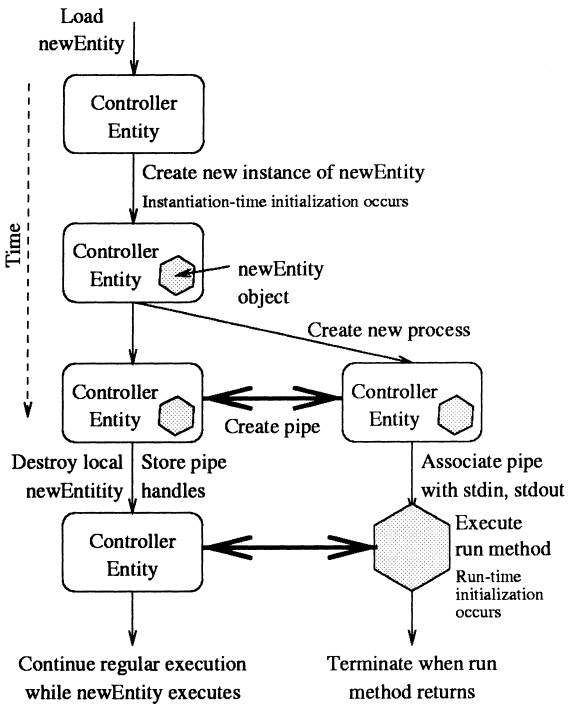

Fig. 5. Steps for loading and executing a local entity, called newEntity in this example. Time is represented along the vertical axis, and each vertical layer represents changes in the state of the entities as time passes.

for `run`, `Init`, `runtimeInit` and `Cleanup`, as well as the code for automatically doing setup activities such as generating an entity identifier, sending a CONNECT message up upon startup and a DISCONNECT message upon termination. These activities are done in the `new` method (constructor) defined by Entity, and should be inherited by all other entities. For this reason, subclasses of Entity must not override the constructor, but provide their functionality through `Init` and `runtimeInit`.

ControllerEntity and its subclasses recognize a command called START, which receives a parameter called Class that contains a specification of the entity that needs to be started, in one of the following forms:

- "Classname" requests the execution of a local entity of the given class.
- "Host:Classname" requests the execution of an entity of the given class in the host specified. Recognized only by Monitor.

3.4.6.2. Stand-alone entity execution. To allow an entity to be loaded as a stand-alone program, there must be a mechanism for automatically creating an instance of the class and invoking its `run` method when the entity is executed. However, this code must not be executed when the entity is loaded from another one.

In $AAFID_2$, the mechanism used to solve this problem is a subroutine called `_EndOfEntity`, implemented by the Entity class, which must be invoked in the last line in the source file of an entity. This subroutine checks whether the file is being loaded by itself or as a module from another program. In the first case, it creates an instance of the entity, calls its `run` method, and exits when execution terminates. In the second case, nothing is done and a value of 1 is returned, which is the customary way in Perl of signaling that a module file was loaded correctly.

When an entity is loaded stand-alone, its standard input and output are not redirected and the user is able to type messages and see the results in the terminal.

3.4.6.3. Local execution of an entity from another program. When an entity needs to be executed from another program (for example, a transceiver loading an agent), the following steps are performed:

1. Load the entity class using Perl's `use` statement.
2. Create a new instance of the entity using its `new` method.
3. Create a new process, where the new entity will be executed.
4. Create two pipes between the parent and child processes, one for sending messages from the parent to the child, and one for messages from the child to the parent. In the child process, the reading end of one pipe is associated with standard input and the writing end of the other one is associated with standard output, therefore establishing the "up" channel of the entity. In the parent process, the corresponding ends of the pipes are stored in an internal parameter for future use, and to be able to receive messages from the new entity.

5. The child process executes the new entity by invoking its `run` method. When it returns, the process terminates.

These steps are illustrated in Fig. 5.

3.4.6.4. Remote execution of an entity. The Monitor class provides the facilities for being able to request the activation of an entity in a remote host. The mechanism used by monitors to activate remote entities is the following:

1. Check if a communication channel to a transceiver or a monitor in the requested host already exists. If so, send to it a message to load the required entity. Otherwise continue with the following steps.
2. Execute in the remote host the Starter program, with the appropriate parameters to tell it from which host it was executed. $AAFID_2$ uses ssh (the Secure Shell) to execute Starter in the remote host.
3. The monitor sets the appropriate flags to indicate that it is waiting for a connection from the host where the Starter was executed, and continues its normal execution.
4. In the remote host the Starter instantiates a transceiver (class PlainTransceiver), then contacts the server port in the monitor, establishes a TCP connection, and redirects its standard input and output to it.
5. The Starter runs the transceiver, which will then communicate with the monitor (through its standard input and output, as redirected in the previous step) to register with it.
6. When the monitor receives the CONNECT message from the newly created transceiver, it identifies it as the one in which an entity has to be started, and sends the appropriate command to it.

These steps are shown in Fig. 6. It is important to notice the following characteristics of this process:

- All entities are started locally (i.e., the transceiver is started locally by the starter, and the new entity is started locally by the transceiver). It is a separate program called Starter which establishes the network connection and does the appropriate redirection of handles. For this reason the network connection is transparent to both the new transceiver and the new entity. The monitor knows about the network connection, because it receives the connection request from the Starter in its server port.
- Once a network connection is established between the monitor and the transceiver, it is kept open. Future requests to start entities in the same host will stop at the first step of the algorithm described before, and simply cause the appropriate START message to be sent.

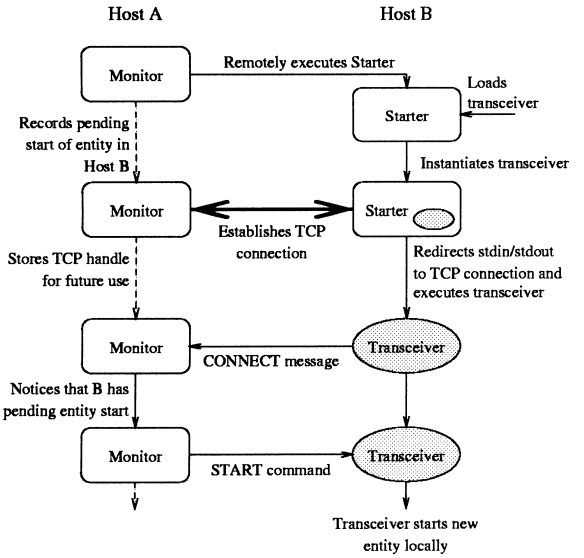

Fig. 6. Execution of a remote transceiver and entity using the Starter program.

3.4.6.5. Requests for missing code. When a controller entity receives a request to start a new entity, it may be that some or all the code necessary to load that entity (for example, its class source file, or a module needed by it) is not present in the local host. In this case, the controller entity sends a NEEDMODULE message up, specifying the name of the missing class. The entity above must find the appropriate code (probably by sending a NEEDMODULE message itself), and then send it down in a NEWMODULE message.

Once the controller entity receives the NEWMODULE message containing the code it needs, it saves it to a local file and tries to load it. If there is still missing code (for example, the code received needs another module that is not present), then the

process is repeated until an entity of the appropriate class can be instantiated.

This mechanism allows the initial installation of $AAFID_2$ in the monitored systems to be limited to the essential classes, and then have all the others (such as agents) deployed as needed.

3.5. Experiences and future development

The $AAFID_2$ prototype has several limitations, and through its testing, we have identified some of them, and obtained other experiences. The main ones we have identified are in the following areas:

Data analysis. No standard mechanism exists for implementing data analysis in the transceivers or in the monitors. The implementation of data analysis mechanisms is complicated by the fact that data coming from a single agent may need to be analyzed in several different ways to look for different problems. Therefore, it is unclear how the messages from the agents have to be processed.

Impact. One of the undesirable consequences of implementing $AAFID_2$ in Perl is that it has large resource needs. In an experiment running nine entities (counting agents and filters) on a Sparc Ultra 1 with 128 MB of RAM, each entity consumed an average of 2381.9 kB of memory, or 1.8%. This may not seem exceedingly expensive, but for AAFID to be useful, we need to have a large number of agents running on each host, certainly in the tens and possibly in the hundreds.

Complexity. In some cases, the $AAFID_2$ code has grown more complex than apparently necessary. For example, in our testing, we have not used the automatic distribution of code that is possible with $AAFID_2$, because the whole prototype has been available to all the machines in the network. Having this feature makes the code more complex and difficult to maintain.

Scalability. Although the distribution of tasks between the different AAFID entities aids scalability, there is still the possibility of bottlenecks at the controller entities (particularly monitors, but also transceivers) when there are a large number of hosts being monitored, and a large number of agents in each host. The bottleneck can be both in terms of communication (many entities sending messages to the same monitor) and processing (the monitor having to process the information coming from a large number of transceivers). We have not experienced any of these situations because the $AAFID_2$ implementation has not been tested on a large network, but they could conceivably occur. The AAFID architecture, however, allows for hierarchical deployment of monitors. By arranging monitors in a suitable hierarchical fashion, most scalability problems can be reduced by not allowing each monitor to receive more information than that which it can process without overloading the system in which it is running.

Rigidity of the architecture. We have identified situations where it may be useful to allow more flexible communication between entities in the AAFID architecture, for example to allow one agent to communicate directly with another agent. This might also help avoid the scalability problems mentioned before.

None of these drawbacks invalidates the approach used by the AAFID architecture, because they can be addressed by engineering effort or further research, and because AAFID can be used as a platform for performing the necessary research.

To address these issues and some others, we have determined some points for future work in the $AAFID_2$ prototype:

Reduction modules. We have designed a new entity for the AAFID architecture called *reduction modules*, which is intended to provide flexible data analysis capabilities. A reduction module will be designed to detect certain intrusions, and will consist of a list of agents needed, plus analysis code to execute on the transceivers and the monitors. This allows the user to specify the capabilities required from the intrusion detection system.

Use of threads in Perl. In the current $AAFID_2$ prototype, agents and filters are implemented as separate processes. We are considering making each entity a thread within the corresponding transceiver. This will reduce the resource usage and the impact of $AAFID_2$ on the systems where it runs.

Porting $AAFID_2$ to other architectures. Work is underway to port $AAFID_2$ to Windows NT. The porting effort has provided insight into the

inherent incompatibilities between Windows NT and Unix, but progress has been made, and being able to run $AAFID_2$ under NT will allow us to perform even more experimentation with different intrusion detection techniques.

Embedded sensors. We have also started work on building sensors embedded into the code of the operating system and its programs. This will make it possible to obtain information at the point where it is generated, and will also enable the creation of sensors that are lighter and more resistant.

Advanced architecture capabilities. Some of the capabilities offered by the AAFID architecture (for example, arranging monitors in a hierarchical fashion) are not fully implemented in the latest versions of $AAFID_2$. We plan on fully implementing the architecture in future versions of the prototype.

4. Research directions

The main objective of the AAFID architecture and its prototype is to serve as a platform for research in innovative intrusion detection techniques. As such, it has helped us identify questions that have not been completely answered by past intrusion detection research. We classify these questions in the following areas:

Detection and data requirements. The first step in the design of an intrusion detection system is to determine what it needs to detect. Research questions include:

- How do we express what we want to detect? (determine the detection requirements).
- From the detection requirements, how do we determine which data are needed? (determine the data needs).
- How do we express the data needs?
- How do we verify that the data needs are sufficient to satisfy the detection requirements?

Data collection. Once we know what data are needed, the next step is to collect it. In this area, research questions include:

- Where and how to collect the data? Data may be collected at different points in the system (e.g., application level or kernel level), and different types of data may need to be collected at different points.
- How do we communicate the requirements and the data between the collection and the analysis mechanisms?
- How do we efficiently collect data without disturbing system operation?
- How do we represent and store the data?

Data analysis. Once we have the data, they must be analyzed to detect intrusions. In this area we have the following questions:

- Where do we analyze it? In a central location? At the point of collection?
- How do we analyze it? Different detection requirements may need different types of analysis. However, we do not know how to determine which analysis techniques are better suited for detecting different types of problems or for analyzing different types of data.
- How do we measure "better" in the previous question? It is not clear which criteria can be used to compare analysis techniques.

Reaction. Assuming we have successfully analyzed the data, the question remains of how to react when problems are detected. And this area includes the following three points:

- How do we present the results? Ultimately, a human user must be able to see the results of the analysis and take a decision. But if the intrusion detection system is not capable of presenting its results in a form that is understandable to the user, then its whole purpose is defeated.
- Which control structure to use? In a distributed intrusion detection system, some components control others, and the user must be able to exert control to react to detected problems. Research is needed to determine which control structures are best for different situations.
- Should the intrusion detection system react automatically? It is possible for the intrusion detection system to react automatically to certain problems to try to contain or stop the damage. However, automatic reaction creates the possibility of further problems when false positives occur. Further research is needed to determine under which situations it is safe to react automatically, and which types of reaction are appropriate.

These questions are only some of the most relevant open research areas in intrusion detection. It is a young field, and we are only starting to understand some of the problems involved.

References

[1] J.S. Balasubramaniyan, J.O. Garcia-Fernandez, E. Spafford, D. Zamboni, An architecture for intrusion detection using autonomous agents, Technical Report 98-05, COAST Laboratory, Purdue University, May 1998.

[2] K.A. Bradley, S. Cheung, N. Puketza, B. Mukherjee, R.A. Olsson, Detecting disruptive routers: a distributed network monitoring approach, in: Proceedings of the 1998 IEEE Symposium on Security and Privacy, May 1998.

[3] J.M. Bradshaw, An introduction to software agents, in: J.M. Bradshaw (Ed.), Software Agents, AAA1 Press/MIT Press, Cambridge, MA, 1997, pp. 3–46 (Chapter 1).

[4] S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, J. Rowe, S. Staniford-Chen, R. Yip, D. Zerkle, The design of GrIDS: a graph-based intrusion detection system, Technical Report CSE-99-2, Department of Computer Science, University of California at Davis, Davis, CA, January 1999.

[5] T. Christiansen, Tom's object-oriented tutorial for Perl manual page included with the Perl 5 distribution, April 1998.

[6] Cisco Systems, Cisco netranger. Web page at `http://www.wheelgroup.com/univercd/cc/td/doc/pcat/nerg.htm`, accessed in May 2000.

[7] M. Crosbie, B. Dole, T. Ellis, I. Irsul, E. Spafford, IDIOT – Users Guide, COAST Laboratory, Purdue University, 1398 Computer Science Building, West Lafayette, IN 47907-1398. `ftp://coast.cs.purdue.edu/pub/COAST/papers/IDIOT/IDIOT_Users_Guide.ps`, September 1996.

[8] M. Crosbie, E. Spafford, Defending a computer system using autonomous agents, Technical Report 95-022, COAST Laboratory, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, March 1994.

[9] M. Crosbie, E. Spafford, Defending a computer system using autonomous agents, in: Proceedings of the 18th National Information Systems Security Conference, October 1995.

[10] M. Crosbie, G. Spafford, Active defense of a computer system using autonomous agents, Technical Report 95-008, COAST Group, Department of Computer Sciences, Purdue University, West Lafayette, IN 47907-1398, February 1995.

[11] D.E. Denning, D.L. Edwards, R. Jagannathan, T.F. Lunt, P.G. Neumann, A prototype IDES – a real-time intrusion detection expert system, Technical Report, Computer Science Laboratory, SRI International, 1987.

[12] D.E. Denning, P.G. Neumann, Requirements and model for IDES – a real-time instrusion detection system, Technical Report, Computer Science Laboratory, SRI International, August 1985.

[13] D. Farmer, W. Venema, Computer forensics analysis class handouts. Web page at `http://www.fish.com/forensics/`, accessed in May 2000, August 1999.

[14] W.M. Farmer, J.D. Guttman, V. Swarup, Security for mobile agents: issues and requirements, in: Proceedings of the 19th National Information Sytems Security Conference, vol. 2, National Institute of Standards and Technology, October 1996.

[15] R. Heady, G. Luger, A. Maccabe, M. Servilla, The architecture of a network level intrusion detection system, Technical Report, University of New Mexico, Department of Computer Science, August 1990.

[16] L. Heberlein, G. Dias, K. Levitt, B. Mukherjee, J. Wood, D. Wolber, A network security monitor, in: Proceedings of the IEEE Symposium on Research in Security and Privacy, May 1990.

[17] J. Hochberg, K. Jackson, C. Stallings, J.F. McClary, D. DuBois, J. Ford, NADIR: an automated system for detecting network intrusion and misuse, Computers and Security 12 (3) (1993) 235–248.

[18] S.A. Hofmeyr, An immunological model of distributed detection and its application to computer security, Ph.D. thesis, University of New Mexico, May 1999.

[19] IEEE, IEEE Journal on Selected Areas in Communications (Special issue on Secure Communications), May 1989.

[20] G.H. Kim, E.H. Spafford, The design and implementataion of Tripwire: a file system integrity checker, in: J. Stern (Ed.), The Second ACM Conference on Computer and Communications Security, ACM Press, Fairfax, VA, November 1994.

[21] S. Kumar, Classification and detection of computer intrusions, Ph.D. Thesis, Purdue University, West Lafayette, IN 47907, 1995.

[22] T.F. Lunt, R. Jagannathan, R. Lee, S. Listgarten, D.L. Edwards, P.G. Neumann, H.S. Javitz, A. Valdes, Development and application of IDES: a real-time intrusion detection expert system, Technical Report, SRI International, 1988.

[23] T.F. Lunt, A. Tamaru, F. Gilham, R. Jagannathan, P.G. Neumann, H.S. Javitz, A. Valdes, T.D. Garvey, A real-time intrusion detection expert system (IDES) – Final Technical Report, Technical Report, SRI Computer Science Laboratory, SRI International, Melno Park, CA, February 1992.

[24] Merriam-Webster, Intrusion Merriam-Webster OnLine: WWWebster Dictionary. `http://www.m-w.com/dictionary`, accessed on 16 May 1998.

[25] B. Mukherjee, T.L. Heberlein, K.N. Levitt, Network intrusion detection, IEEE Network 8 (3) (1994) 26–41.

[26] P.A. Porras, P.G. Neumann, EMERALD: Event monitoring enabling responses to anomalous live disturbances, in: Proceedings of the 20th National Information Systems

Security Conference, National Institute of Standards and Technology, 1997.
[27] T.H. Ptacek, T.N. Newsham, Insertion, evasion, and denial of service: eluding network intrusion detection, Technical Report, Secure Networks, January 1998.
[28] E. Siever, D. Futato, Perl Module Reference, vol. 1&2, O'Reilly, Sebastopol, CA, included in the Perl Resource Kit, November 1997.
[29] S.R. Snapp, J. Brentano, G.V. Dias, T.L. Goan, L.T. Heberlein, C. Lin Ho, K.N. Levitt, B. Mukherjee, S.E. Smaha, T. Grance, D.M. Teal, D. Mansur, DIDS (distributed intrusion detection system) – motivation, architecture, and an early prototype, in: Proceedings of the 14th National Computer Security Conference, Washington, DC, October 1991.
[30] S.R. Snapp, S. Smaha, D.M. Teal, T. Grance, The DIDS (distributed intrusion detection system) prototype, in: Proceedings of the USENIX Summer 1992 Technical Conference, San Antonio, TX, June 1992.
[31] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, D. Zerkle, GrIDS: a graph based intrusion detection system for large networks, in: Proceedings of the 19th National Information Systems Security Conference, vol. 1, National Institute of Standards and Technology, October 1996.
[32] K.M.C. Tan, D. Thompson, A.B. Ruighaver, Intrusion detection systems and a view to its forensic applications, Technical Report, Department of Computer Science, University of Melbourne, Parkville 3052, Australia. http://www.securityfocus.com/data/library/idsforensics.ps.
[33] L. Wall, T. Christiansen, R.L. Schwartz, Programming Perl, 2nd ed., O'Reilly, Sebastopol, CA, September 1996.
[34] G.B. White, E.A. Fisch, U.W. Pooch, Cooperating security managers: a peer-based intrusion detection system, IEEE Network (1996) 20–23.

Eugene H. Spafford is a professor of Computer Sciences at Purdue University, the University's Information Systems Security Officer, and is Director of the Center for Education Research Information Assurance and Security. CERIAS is a campus-wide multi-disciplinary Center, with a broadly-focused mission to explore issues related to protecting information and information resources. Spaf has written extensively about information security, software engineering, and professional ethics. He has published over 100 articles and reports on his research, has written or contributed to over a dozen books, and he serves on the editorial boards of most major infosec-related journals. Dr. Spafford is a Fellow of the ACM, Fellow of the AAAS, senior member of the IEEE, and is a charter recipient of the Computer Society's Golden Core award. Among other activities, he is chair of the ACMs US Public Policy Committee, a member of the Board of Directors of the Computing Research Association, and is a member of the US Air Force Scientific Advisory Board. He regularly serves as a consultant on information security and computer crime to law firms, major corporations, US government agencies, and state and national law enforcement agencies around the world. More information may be found at http://www.cerias.purdue.edu/homes/spaf. In his spare time, Spaf wonders why he has no spare time.

Diego Zamboni is a Ph.D. student at Purdue University, where he is working in CERIAS in Intrusion Detection research. He obtained his M.S. in Computer Science from Purdue University. Previously he obtained his bachelor's degree in Computer Engineering from the National Autonomous University of Mexico, where he was in charge of the security for the Unix machines at the Supercomputing Department. He also established the University's Computer Security Area, one of the first Computer Security Incident Response Teams in Mexico.