Argus - A distributed network-intrusion detection system

A Report Submitted
in Partial Fulfillment of the Requirements
for the Degree of
Bachelor of Technology

by
Sankalp Singh
Srikanth Kandula

under the guidance of
Dr. Dheeraj Sanghi

to the
Department of Computer Science & Engineering
Indian Institute of Technology Kanpur
April, 2001

To our parents

Certificate

Certified that the work contained in the report entitled "*Argus - A distributed network-intrusion detection system*", by Sankalp Singh and Srikanth Kandula, has been carried out under my supervision and that this work has not been submitted elsewhere for a degree.

Dr. Dheeraj Sanghi

Abstract

Network-Intrusion Detection deals with identification and containment of network-based attacks. Such systems could be classified based on the medium they listen on (end-hosts or network segments), the timing of information analysis (batch-process the collected data vs. real-time processing) and the type of analysis (knowledge or signature based vs. anomaly based). This project involves the design and construction of an experimental network-based intrusion detection system Argus, that performs a real-time analysis on packet traffic sniffed off the network segment using an appropriate mix of some well-known knowledge based agents and custom-made anomaly agents. Rules for the anomaly agents and signatures of newly found attacks for the knowledge based agents are generated using an implementation of RIPPER [7], a machine learning classifier algorithm. Intrusion Alert Protocol [1] is used as the communication framework to link up these diverse agents and to provide a common interface to other intrusion detection systems. The uniqueness of Argus is in the robustness of its communication framework that offers great versatility in the deployment of its various agents, its ability to detect distributed attacks, allow configuration of real-time response to intrusions and in its enhanced sensitivity to new attacks. Further the framework of this system is highly extensible so as to be able to incorporate different agents with minimum effort. The implementation of IAP is designed so as to conserve the bandwidth required for the transfer of alert information from the analyzers to the managers.

Acknowledgements

We would like to express our deep gratitude to our supervisor Dr. Dheeraj Sanghi, for his guidance and invaluable suggestions throughout the course of this project. It was his enthusiasm, motivation and guidance that saw us through some rough patches during the system building stage of the project. We would also like to thank Dr. Sumit Ganguly for his guidance in the project, especially for his suggestion to integrate NFR as the knowledge-based component of the system and for providing valuable discussion time in the initial stages of the project. We are also thankful to our batchmates for all the cheerful support they have given to us. Finally we would like to thank our parents for their blessings, support and encouragement.

Contents

Abstract **iv**

Acknowledgements **v**

1 Introduction **1**

1.1 Intrusion Detection 1

1.2 Knowledge-based systems vs. Statistical systems 2

1.3 Organization of this report 3

2 NFR: Description, Deployment and Evaluation **4**

2.1 Description 4

2.2 Deployment 5

2.3 Evaluation 5

3 Argus: Design Rationale **8**

4 Details of Individual Components **12**

4.1 IAP: Intrusion Alert Protocol 12

4.1.1 What IAP does? 12

4.1.2 What it does not do? 13

4.1.3 IAP Implementation Specifics 13

4.2 The Data Mining Framework 22

4.2.1 The Data Cleansers 22

4.2.2 The Learning Agents 27

4.2.3 Anomaly detection Agents 32

4.3 Driver for NFR 34
4.4 Managers 39

5 Experiences with Argus **43**
5.1 Testing the IAP Framework 43
5.2 Experiences with the data cleanser 44
5.3 Experinces with the learning agent 45
5.4 "Information Exploration Shootout" runs 46
5.5 Experiences with the NFR Driver 47
5.6 Experiences with the Manager 48

6 Conclusion and Future Work **49**

Bibliography **49**

A Argus Design **52**

B NFR's IDA-Console interaction **53**

C Example of IAP Connection Set-up **54**

Chapter 1

Introduction

1.1 Intrusion Detection

Today's Computer Networks are complex entities that provide a wide-variety of services. The popularity of the Internet, electronic commerce, corporate networks and distributed computing has caused a proliferation not only in the information transmitted through these networks but has also put a premium on security of this information. The increase in the value of information that is available on computer networks has lead to a proportionate increase in the complexity and variety of network intrusions. Classical security mechanisms like firewalls, end-to-end encryption and authentication have their own susceptibilities. Firewalls are unable to look at the data content of packets passing through them and are extremely vulnerable to denial of service attacks as well as attacks that come from inside the firewalls. Thus there is a great need for network-traffic surveillance and intrusion-detection systems as an augmentation to the existing security infrastructure.

Intrusion Detection is the process or the art of detecting inappropriate, incorrect or anomalous activity on a computer system or network of computer systems. This project concentrates primarily on network-based intrusion detection and ignores host-based intrusion detection systems that primarily function through analysis of OS audit trails like sequences of system calls and various logs generated by OS processes. Such an example is the Basic Security Module of Solaris. Broadly speaking, network-intrusion systems would need to collect network traffic into records and then analyse these records. Collection of records could be done by installing modules at each host that collect data sent/received from the NIC of this host or by establishing stand-alone systems, one-per network broadcast segment, that collect the data that passes on this segment. Analysis of data, so-collected, could be done in real-time or in batch mode. Further there are two major approaches that could be used in the analysis of this data: Knowledge/Signature based and Anomaly/Behavior/Statistical based.

1.2 Knowledge-based systems vs. Statistical systems

Knowledge-based systems are expert systems that contain a vast database of signatures of known attacks. Candidate data is pattern-matched with these signatures and a match is flagged as an intrusion of the corresponding type. Thus these systems usually include specification languages to encode attack signatures and interpreters/compilers that apply so generated attack-signature-filters on the candidate data. Several such systems exist and need to be evaluated based on the ease-of-use, expressiveness and simplicity of their specification languages and the efficiency of their execution environments. The great power of these systems lies in their ability to accurately identify all known attacks whose signatures can be encoded. But these systems, by definition are insensitive to new attacks.

Anomaly/ Statistical systems use a large amount of training data to build profiles for normal activity. A mis-match of the candidate data with its corresponding profile is signalled as an intrusion. Clearly this approach has the advantage of requiring less human effort and is also highly sensitive to new attacks or those attacks whose signatures have not been encoded. However it also has several drawbacks. The training data has to be a sufficiently exhaustive set in order to achieve significant accuracy, and there is the possibility of false positives. Further data-mining algorithms that generate profiles from data-sets are usually of O(n^3) or higher (n=size of dataset). Incremental adaptation of these profiles with the growth in data collected is also not possible using the data mining approaches we have studied so far. Clearly the efficient approach, as has been adopted for Argus, appears to be one that uses a mix of both these systems so as to use the strengths of both these systems.

1.3 Organization of this report

The rest of the report is organized as follows. In Chapter 2, we justify our choice of NFR [8] as an off-the-shelf knowledge based component, describe the internals of this system and give the details of our deployment of NFR. This chapter also contains our perspective on the strengths and short-comings of NFR. In Chapter 3, we describe the design of Argus and the rationale behind the design decisions. Chapter 4 consists of detailed descriptions of the individual components of Argus and the logic behind such a design. We report our results and experiences with Argus in Chapter 5. Finally in Chapter 6, we present our conclusions and the extensions that we envisage for Argus.

Chapter 2

NFR: Description, Deployment and Evaluation

2.1 Description

NFR is primarily a knowledge-based system whose three basic components are the IDA, administrative console and the central station. It uses a domain-specific specification language called N-Code to encode attack signatures. N-Code appears to have derived much of its basic structure from PERL. The IDA (Intrusion Detection Appliance) is the heart of NFR containing N-Code encoded attack signatures, the N-Code parser that transforms the attack signatures into suitable filters and the NFR engine which sniffs packets, cleanses and condenses these packets into a form that could be passed through the defined hierarchy of filters. The IDA signals the detection of an attack based on the output of these filters, updates recorders that either measure statistics about the network or store the alert information and runs a gui server that allows administrative consoles, once authenticated, to query the IDA's alert/statistical database and change the IDA's configuration. It is recommended though not necessary that the IDA have two NIC, connecting the IDA to its "managing" network and "monitoring" network respectively. The distinction, if utilized adds much-needed stealth to the IDA - Administrative Console communication. Administrative Consoles are the lightweight managers that administer the IDA and could possibly be located across the Internet from the IDA (proxy-permitting). Central stations are an offshoot of the need for distributed-ness that came as an after-thought to NFR and the need for backward compatibility that caused the administrative console to be able to talk to only a single IDA at a time. They also serve the purpose of giving a unified view of the data collected from multiple IDA's and present a single interface to specify directives to multiple IDA's.

2.2 Deployment

While NFR could be deployed in several architectural configurations (switched, bridged, stand-alone, multiple...), we deployed it in a stand-alone configuration (1 IDA: csews2) with multiple administrative consoles. Ideally however, the IDA needs to be installed on the De-militarized-zone or the sub-network which contains the protected servers, so that it could monitor traffic from both the internal as well as the external networks to any of these servers.

2.3 Evaluation

The strength of NFR is in its pre-bundled n-code modules that are very extensive and comprise almost all known popular attacks. The architecture it uses is also fairly efficient. However we have identified several weaknesses or inadequacies that have been described below.

1. No built-in mechanism to learn new rules, i.e. NFR is fairly static and lacks adaptability. The only mechanism for modifying existing rules or adding new rules is to manually generate code in n-code and manually deploy this code at the IDA through the administrative console. This inadequacy is discernible in the inability of NFR to detect "new attack" signatures and in the difficulty of subsequently coding attack signatures into n-code. This manual coding of the signature would inevitably be a pain-staking effort, but the former is a still more crucial issue and can only be solved by continuous vigilance on part of the manual operator towards new attacks that would sprout up. However

there is a significant time lag between the detection of new attacks and the coding/deployment of corresponding n-code filters.

2. Knowledge-based intrusion detection systems function using thresholds for acceptable limits. As the threshold changing directives are also under manual control, any attack interception that could be done by a tweaking of these parameters proportional to the degree of suspicion is not possible. A continuously learning rule generator that works in isolation of the analyzers and drives each of these analyzers would do away with the above inadequacies.

3. The only Response-to-attack-detection is to trigger real-time alerts on the administrative console or e-mail the alert to specified authority. Most attacks either need alternate real-time proof to be gathered or an active response to be taken, both of which NFR fails to provide. These responses gain in importance as they would do away with the need for real-time human intervention in attack detection.

4. Pattern matching at the various IDA's is done in isolation. Hence an NFR architecture would be vulnerable to distributed attacks, i.e. an attack with several phases may be executed such that no single IDA views all of the individual phases. An instance of this case is when multiple entry-points exist into a network. A Denial-of-Service attack might be carried out on an internal-but-visible target that would escape the isolated IDA's installed on each of the entry points.

Even if a rule that matches the attack exists, a distributed execution of individual stages of the attack would pass undetected i.e. if a rule matches an attack that consists of two phases (a) corrupting the ".rhosts" file of a remote machine (b) using the change to exploit weaknesses on that machine, then the observation of the first phase would require a heightened suspicion of activity that would occur later. Later activity if performed through the aegis of a different IDA than the one through which phase (a) had been done, would pass un-noticed. The only alerts that NFR would generate would be that one of its IDA's saw a change in the ".rhosts" file. Further utilization of this exploit cannot be detected by NFR. A distributed architecture that allows for analyzers to exchange information of suspicious events seen would solve the problem to a large extent. Argus uses a mechanism that involves storing of such suspicious information at the manager. Receipt of subsequent warnings would be matched onto earlier warnings that might be relevant or might together constitute a possible attack.

Chapter 3

Argus: Design Rationale

Following is an illustration of the design.

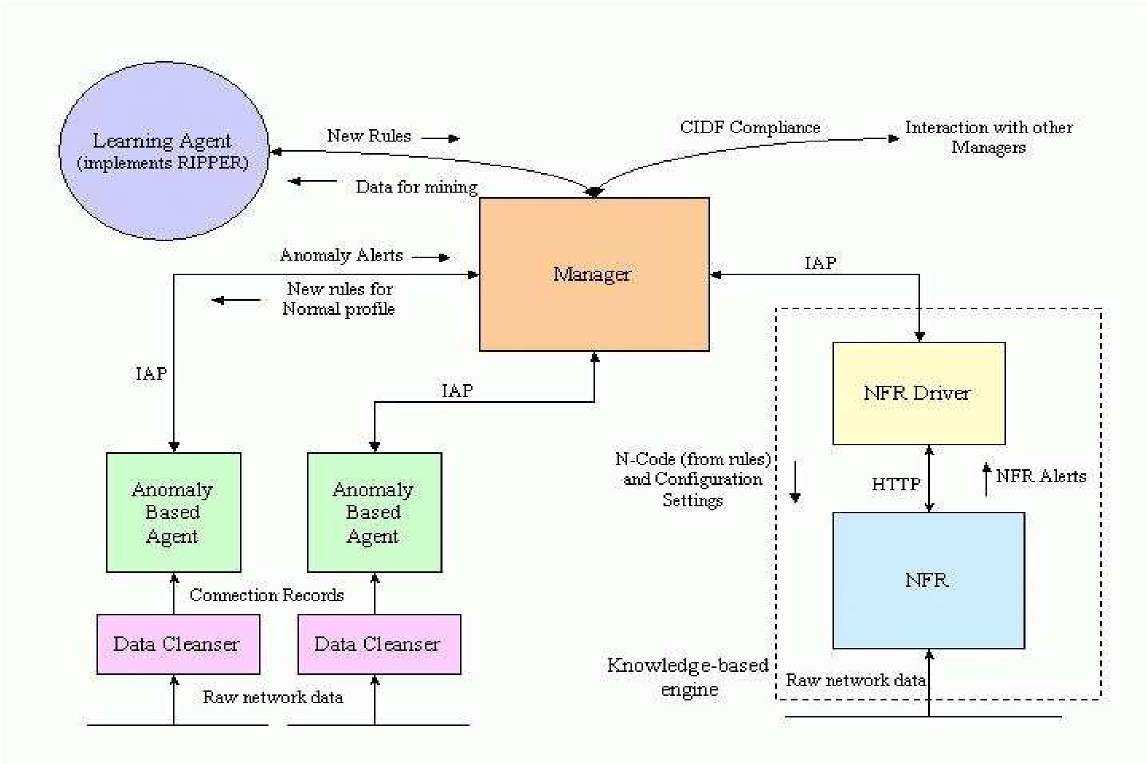

Figure 1: Argus: Design

As we have described above, NFR is purely knowledge-based and suffers from some weaknesses. We have designed a multi-tier architecture that has several components, one of which could be NFR, which would overcome the weaknesses of NFR and provides an intrusion detection system that is much more comprehensive, distributed, scalable, efficient and adaptive. Here we describe the broad design of Argus. For specifics about the choices implemented, look at Chapter 4. A figure of this design is at Appendix-A.

Argus employs an agent-based architecture that has low-level agents, each of which are sufficiently strong, so that the architecture is truly distributed, and not pseudo-distributive in which all the computations are effectively done by the central servers (i.e. load sharing is fairly uniform). A distributed hierarchy of agents is used, with integration of data-mining agents for increasing the adaptability of the system. The data-mining components not only generate rules for matching a *normal profile* but also generate feedback for knowledge-based components, in the form of rules that encode signatures of new attacks, the generation of which would be described in 4.2.2. These rules can then be used to update the rule database of the knowledge-based component like NFR. The output of the anomaly detection agents also serves as a feedback to the data-mining (learning) agents for continuously updating and improving the normal profile. This is described in more detail in 4.2.2.

The lowest level agents in this architecture are data-cleansers, which interface with the physical medium (network) and collect the information to be utilized by the analyzers. The information from the network could be extracted through calls to tcpdump, or through a custom implementation of sniffers using the Berkeley Packet Filter(BPF) library, *libpcap.* This output would need to be condensed into connection records that contain values of the necessary connection features; feature selection is a crucial aspect and our choice would be later explained in 4.2.1. The connection records are then supplied to the higher-level analyzing agents. Connection records may be of different types depending on the choice of connection features that each analyzer agent would need. As an instance, the records for the knowledge-based components, may be a little less comprehensive, while those used by learning agents and anomaly detectors need to contain a lot more temporal and statistical

information.
The Analyzing agents can be one of the following three types.

1. Misuse detectors
2. Anomaly detectors
3. Learning engines

The misuse detectors could be the analyzing parts of knowledge-based intrusion detection agents, like NFR i.e. NFR could be a unit that comprises both a misuse-detector and the corresponding data-cleanser. Use of such off-the-shelf components would need to be encapsulated with lightweight drivers that interface these systems with Argus. For details about the driver construction for NFR, look at 4.3. The interaction between the agents and the manager occurs through IAP as described in 4.1. These misuse detection agents, or as in our case drivers for third-party misuse detection systems, need to have translators that translates the rules that are output from learning engines (see 4.2.2) into filters in the signature specification language (N-Code as described in 2.1).

The learning agents and the anomaly detection agents are closely inter-related in that the latter is an execution environment for rules generated by the former. It is assumed that the learning agents would use a classification algorithm (like RIPPER, see 4.2.2). Learning agents are data-mining components that are trained on the network data. These classification algorithms would be fed with normal network traffic data using which they would generate the set of rules for normal profiles. Rules of normal profiles, thus generated are passed onto the anomaly agents which apply them on suitably cleaned connection records. Learning agents have the ability to generate rules for both anomaly detection agents (rules characterizing normal activity) and knowledge-based agents (rules characterizing some abnormal activity, given sufficient tagged abnormal data, and lots of normal data).

The rules for normal data are dynamically forwarded by the learning agents to the managers, which then distribute them to all the anomaly detection agents. The alerts are forwarded by the anomaly detectors to the managers. If the manager ratifies an alert as an intrusion, the data of the alert is automatically sent to the learning agent that uses it to generate new rules for the knowledge-based component, characterizing this newfound intrusion. The new rules are then distributed to the misuse-detection agents through the manager using the communication framework of IAP as described in 4.1. This provides for adaptability in the system.

The learning agents also regularly get normal traces from the anomaly analyzers, so that the normal profile is kept up-to-date. Depending on the nature of the classifying algorithm in use, it may or may-not-be possible to incrementally update rules that are once generated. In the case wherein incremental updates are not possible, the learning agents can be taught to commence rule learning after a sufficient amount of new normal data has been obtained. Further some form of aging mechanism could be applied to use various sets of rules in concordance with each other, or through the means of meta-classifiers as described in [5]. Meta-classifiers are systems that use several sets of individual classifiers and then learn rules that interpret the set of individual outputs to ascertain the output to be generated. [5] argues why such a system would perform better.

Argus might also be used to perform dynamic deployment and load balancing proportional to the threat perception, as envisaged in [4].

The next higher level in this hierarchy of distributed agents, could be managers or aggregating analyzers. These agents would be responsible for co-ordinating data-flow between the execution agents (both anomaly and misuse) and the learning agents and also for detecting distributed attacks. The latter could be easily done as warnings from the individual systems could be stored here and new warnings could be used to identify matching previous warnings. Further this level could be split up into a multi-level heirarchy of aggregators as exists in the system designed by [4]. Interaction between the managers, higher-level analyzers and the rest of the system is through IAP which would presently be described in 4.1

Chapter 4

Details of Individual Components

4.1 IAP: Intrusion Alert Protocol

IAP (see [1]) is a draft protocol of the Intrusion Detection Working Group in IETF. As the name suggests, it is the result of an effort to provide a standard means of communication between heterogeneous agents and managers over typical Internet scenarios, wherein either the manager or the analyzer could be inside protected networks that prohibit in-coming connections. Atleast one of them has to be in a network that allows in-coming connections directed to this node (alternatively a gateway might accept connections on behalf of an enclosed node). Our decision to use IAP stems mainly from our effort to keep our implementation as conformant to the standards as possible, so that third party components can be plugged in easily for extension.

4.1.1 What IAP does?

IAP is a pseudo-HTTP application layer protocol that uses Transport Layer Security (TLS) as the transport layer protocol in order to ensure security of the message transfers. IDWG also defines a standard XML format for the actual transfer of data known as Intrusion Detection Exchange Format(see [2]). This data could be alert information sent by analyzers to the managers, and occasionally updates sent by the managers to the analyzers.

IAP is different from HTTP mainly in that the connection initiator need not be the client (seeker of information) and responder the server. For example, the analyzer can initiate the connection and then decide to be the sender of alert information. But it allows only one-way communication, i.e. once the sender has been agreed upon, it remains so for the rest of the connection. It also involves an application layer proxy as the manager and the analyzer may be on different sides of a firewall.

4.1.2 What it does not do?

IAP is only a protocol that simply describes the message structures for exchange of information between analyzers and managers in a distributed intrusion detection system. What it does not describe are the semantics and implementation details of this communication. A lot of these issues are left for the implementer to tackle. For example, in a typical scenario, we would require a two-way connection between the analyzer and the manager, one-way for sending the alert information and other for sending the updates (new rule sets etc.). But a given IAP connection would allow for only one-way communication. Also, in the presence of a firewall , it might be quite possible that only the analyzers might be able to open connections with the hosts outside the firewall (which include the managers). These issues add further complexity to the implementation. How our implementation tackles these and other issues is described later in this section. We also describe the inferences we have made and the creation of an entity called the IAP Daemon that is geared to conserve network bandwidth.

4.1.3 IAP Implementation Specifics

An observation we have made is that in web-servers regardless of the request- load, the bottle-neck resource has been proven to be **network-bandwidth**. IAP managers/ IAP Analyzer Agents i.e. whichever side of the alert heirarchy acts as the server for IAP Connection requests(sender of data, not the connection receiver), would experience load that is pretty similar to that of a web-server. Further we make the assumption that it is **likely that more than one managers/ analyzer**

agents could be running on the same system as woud be the case if there is a particular host that performs most of the analysis and if several managers are all located on the same third-party manager-service provider. Thus we relegate both managers and analyzer agents to a sub-ordinate role and instead place IAPDaemons on each host that runs an IAP entity. This is anyway practically necessary as the means for location discovery of the analyzer/manager by the counterpart in a connection are

1. Have an application listening at a common port that responds to queries for location of local agents

2. Pre-configure ports of individual agents with all other counterparts that would need to communicate with this.

3. Route all traffic through a daemon listening on a common port IAP_PORT.

We chose the third option, because this option does not have any significant drawbacks and has the additional qualification that **data that might need to be sent to several agents on the same host will now only need to be sent to the daemon once** (with the option that it be passed on to multiple local agents). The communication between the daemons and the local agents was implemented through a combination of **message queues and shared memory**.

Another valid and useful observation is that the protocol part is **symmetric on both the manager and the sensor/analyzer ends**. Thus this functionality is encapsulated in a single class.

The reason **why message queues were not sufficient** for inter-process communication is that our implementations of both the daemons and the agents are multi-threaded to improve efficiency. We use the **pthread library for implementing threads**. Thus we have multiple threads that could be reading out of these message queues (**two msg_q's one in each direction of data transfer**). The maximum size of a message that could be passed through a message_queue is a constant while there is no such limit on the size of the data packets that could be handled by IAP. The option of **fragmenting a single packet into several messages** each of which is less than the maximum size is not possible as it is very difficult to ensure that all the fragments of a single packet are parsed by the same thread i.e if buffers are not to be used at either end. Hence if the messages that need to be passed are of sizes greater than the maximum size of a message, then the message is put into shared memory and a shared memory id is passed over through the message queue.

Further several proxies may need to be passed on the path between the agent and the manager, i.e. the connection initiator may be behind a firewall that prohibits direct connections to external hosts. Thus we have designed an **application-level proxy** in class IAPProxy. Each agent is presented upon initialization with a proxy-url(proxy ip and port) and a string of no-proxy suffixes/prefixes/exact-matches. This part of the functionality of the agent (proxy-interaction) is much like that of a normal HTTP browser, search for match in no-proxy-string. If found, establish a direct connection. Else forward the packet to the proxy. The details of the individual packet headers and the sequence of packets are in [1]. However, an example of an IAP Connection establishment is presented in the Appendix-C.

▮ *IAP Proxy*

This is the simplest of the three components in our implementation. IAP connection establishment starts with the proxy-setup stage. The proxy receives a packet from the local (i.e. internal network) IAP entity, which contains both the destination address (remote host) and the initiating source address. The proxy then initiates a connection with this remote host and appends a via header informing the remote host that the proxy is handling the connection for the source. Now that we have a socket pair open, the proxy is a transparent forwarding agent here-onwards. The proxy then instantiates two threads each of which blocks on the read of one of the sockets, and writes out the data got onto the other socket.

▮ *TLS and Certificates*

IAP uses Transport Layer Security to establish secure connections for data transfer between the analyzer/sensors and the managers. We implemented this using the openssl library. Further IAP also has the need that each entity that communicates through IAP has to have a X.509 certificate and a Private Key- Public Key pair. Both of these are generated by using command line utilities that interface with the openssl library. The following is the sequence of calls to the openssl utility to generate certificates. Firstly, we need to obtain as many private-public key pairs as we do certificates, i.e. one per each entity. Then we need to create a self-signed certificate for the root certifying authority. We can then use the root's certificate as the start of a heirarchy of certificates by using the root's certificate to sign other certificate requests.

1. **openssl genrsa > key.pem** Generate a new RSA public key - private key pair.

2. **openssl req -new -key key.pem > certreq.pem** Create a new certificate request. This command automatically prompts for values for all the individual fields in a certificate like common_name, organization_name, location etc.

3. **openssl x509 -req -signkey key.pem -in certreq.pem > cert.pem** Uses a key and the given certificate request to self-sign the certificate request. cert.pem now contains a certificate for the Certification authority.

Creation of certificates for non-root entities (which could be certifying authorities in their own domains) follows the same steps 1 and 2. However, in place of 3, we need to use the following **openssl x509 -req -CA cert.pem -in newcertreq.pem -CAKey key.pem > newcert.pem** Here cert.pem and key.pem are the certificate and the RSA key pair of the certifying authority which is one-level above the node whose certificate is being created, in the trust heirarchy. When creating the first certificate using an entity's certificate, i.e. for the first time this entity is acting as a Certification Authority, then one needs to add the option **-CAcreateserial** so as to allow creation of some files that maintain the list of certificates issued by a certifying authority which in turn aid in certificate maintenance chores like preparation of revoked certificate lists, updation of certificates etc..

The following is the sequence of system calls that needs to be called at the TLSClient end, i.e. on the side of a connection that initiates a TLS handshake. Note that the latest version of TLS available TLSv1 is used.

1. **SSL_load_error_strings()** Loads error string specific to SSL.
2. **SSL_library_init()** Initializes the library.
3. **SSL_METHOD* meth = TLSv1_method()** Sets the protocol as TLSv1, as this library also implements protocols like SSLv2 and SSLv3.
4. **SSL_CTX *ctx= SSL_CTX_new(meth)** Gets a context for this protocol. This context object could be used to choose cryptographic algorithms. Defaults would suffice.
5. **SSL_CTX_use_PrivateKey_file(ctx, keyfile, SSL_FILETYPE_PEM)** Initializing the private key to use.
6. **SSL_CTX_use_certificate_file(ctx, cerfile, SSL_FILETYPE_PEM)** Initializing the certificate to use.
7. **SSL_CTX_check_private_key(ctx)** Verifies the validity of the private key, i.e whether this private key matches with the public key present in the certificate.
8. **SSL* ssl=SSL_new(ctx)** Creates a new SSL object from the initialized contex.
9. **SSL_set_fd(ssl, fd)** The TLS connection is bound to fd, a file-descriptor or socket id.
10. **SSL_set_connect_state(ssl)** Sets the state of this SSL object as the initiator of a TLS handshake.
11. **SSL_connect(ssl)** The socket is primed to initiate a TLS handshake request.

Analogous system call sequence needs to be executed at the server end of a TLS handshake. The only difference is in steps 10 and 11 which get modified to **SSL_set_accept_state(ssl, fd)** and **SSL_accept(ssl)** respectively. The TLS protocol only requires that the server end have a certificate, but IAP insists on each entity's having a certificate. The reason for listing out these system calls and command line sequences is that lack of proper documentation caused us to resort to a significant extent of trial and error before landing up with the right sequence. Once the ssl handshake is performed, the ssl object can be read and written to just as if it were the underlying medium. The blocking/un-blocking properties are borrowed from the underlying medium again.

▮ *IAP Agent*

Initialization

Each agent has an agent-id that consists of the local host's ip address and a 16 bit agent-identifier. At initiation, each agent is given a **configuration file**, **files containing its certificate and RSA Private Key**, **a no-proxy string**, **a proxy-url** and **it's unique identifier**. This identifier is corroborated with the identifier present in the c_name of the certificate presented to the agent at initiation time. The configuration file contains a list of remote agent ids and whether or not this agent can initiate a connection with the remote agent. Points of difference between the sensor/analyzer's and manager's version of agents are the facts that their identifiers come from different number spaces and that they manager's configuration file contains only those agents with which the manager needs to establish connections. This is because it is considerably difficult to handle connections with remote agents whose identifiers are not present in the configuration file without re-starting the agent and cases wherein an analyzer can ad-hocly send data to the manager (if it can establish a connection, without prior notice) are deemed to be common and thus supported. Thus at the manager-end the agent may get registration requests from remote agents that can initiate connections with this manager. The manager just adds them on to the list of legal agents with which it would communicate. On the other hand, it is assumed by IAP that the sensor/analyzer need not respond to ad-hoc'ly received connections from managers. Thus reverse registration at the agent end is not supported. The interaction of the agent with the proxy has already been explained in the preceding paragraph.

Data Send

When data needs to be sent or received from a remote agent, it is checked whether the local agent can establish an active connection with the remote agent or not. If it could, then a connection is established according to the procedure shown in Appendix-C, and data is sent out to the remote daemon. This remote daemon then pushes it onto a to-agent message queue with a type-id that would be picked up by the remote agent. This is **snd_active**. Such connections are pooled, as they are usually the most frequently used ones and pooling would improve throughput considerably as the connection establishment stage via a TLS handshake requires several RTT's. The other version of send called **snd_passive** occurs while sending packets to agents on remote machines with whom one cannot establish a direct connection. These packets are then sent onto the local daemon for safe_keeping until the remote agent asks for them.

Data Receive

Similarly there are two versions of read. The **read_active** involves a connection establishment with the remote daemon, upon which the remote daemon replies with all packets that are waiting for this agent from the local agents. Individual sender id's are present in the data packets themselves and multiple local agent-remote agent packets might be passed along in a single daemon- remote agent packet. This call is non-blocking and returns empty if no packets are present at the other end. Connection pooling is implementable easily here too. To implement connection pooling in this situation, the data-producing thread is on the daemon end and would need to block on the queue of the per-remote-agent map entry. However it has been observed that it is more probable that the sensor/analyzer's would initiate connections to the managers. In this case, this connection would be dealing with data from the managers to the analyzers/sensors, which is relatively infrequent. The policy was "Accomodate all possibilities, optimize for the most probable case". Infrequent use would anyway lead to a time-out on the socket. The final remaining case was of **read_passive**. The agent simply picks off data from the to-agent queue whose type_id matches that of the agent. The remote agent id cannot be specified at request time but would be returned to the calling agent through the data.

IAP Daemon

An IAP Daemon runs on each host having any component of Argus. It is required for two reasons. Firstly, an IAP entity is required to listen on the standard IAP port, and can provide a entry point for a remote agent to the local agents. Secondly, as described in 4.1, it is possible that two remote agents wish to communicate and only one of them can initiate a connection (due to firewall etc.). IAP Daemon helps provide a robust commnication system in this scenario.

Like all entities in our system, the IAP Daemon is also multithreaded and uses the Pthread library. It has the following major components.

- **Main thread:** The main thread listens on the standard IAP port and spawns a new thread to handle each incoming connection from remote agents.

- **Agents to Daemon message queue:** We use a combination of message queues and shared memory as our IPC mechanism for communication between Daemon and the local agents. This message queue is used to transfer the data which the local agents wish to send to those remote agents with which they cannot initiate a connection. The message type (used for the identification of the message) of messages in this queue is irrelevant and is kept to be 1. The data part of the message contains the 16-bit ID of the sending local agent, the complete ID (IP + Agent ID) of the remote agent for which the packet is meant and the actual data to be sent.

- **Daemon to Agents mesasge queue:** This message queue is used by the daemon to transfer the data it has received from the remote agents to the local agents. The message type of each message is the ID of the local agent for which this message is meant. The data part of the message contains the complete ID of the remote agent which sent this data and the actual datat sent.

- **Send Buffer hashtable:** The daemon maintains a hashtable based send buffer for storing the data queued to be sent to the remote agents. The hashtable uses the complete remote agent ID as the key and a queue data structure as the value, *i.e.* we maintain a send queue for each remote agent for which we have data queued up. Each of these queue objects has an associated mutex and conditional variable.

- **Message queue handler thread:** This thread is created by the Main Thread right when the daemon starts up. The thread repeatedly blocks on the Agents to Daemon Message Queue and as soon as it gets a message, it enqueues the data of the message in the send buffer queue of the destination agent.

- **Connection handling threads:** Each time the main thread gets a connection, it spawns a connection handling thread to handle it. This thread first completes the TLS handshake and complex IAP handshake. During this process, it is informed by the remote agent if it wants the daemon to be the producer of data or consumer of data. If daemon is designated as a *producer*, it looks up the send buffer queue for the remote agent. If it has no data, a reply with Content-length of zero is sent back and the connection is broken. If it contains one or more packets, an IAP PUT packet is constructed with the data packets concatenated in the content part of IAP packet, seperated by a delimiter (“<>CRLF”). The PUT packet is sent, the response of the remote agent is received and the connection is broken. If the daemon is designated as a *consumer*, it receives the entire IAP PUT packet from the remote agent (the Content-length attribute in the packet header is used to determine the end of the packet). The packet is parsed and the content, after removing the header, is obtained. The content is then broken into individual records if it contains multiple records for different local agents, delimited again by “<>CRLF”. Each record is then parsed partially to obtain the local agent ID to which it is addressed. A message for the Agents to daemon message queue is then created. It has the ID of the local destination agent as the message type and the data part contains the ID of the remote sender, alongwith the actual data. The connection is not broken in this case and the thread again begins to listen for another IAP PUT packet after sending an appropriate IAP response.

Handling large data packets in message queues: One major issue is this implementation was how to handle fragmentation when the data to be sent through the message queue was larger than the maximum permissible. One approach could be to fragment the data into multiple messages and reassemble the fragments at the receiving end. But as the receiving end usually employ multiple threads to read messages from the message queue it becomes quite difficult to reassemble the messages gathered by seperate threads. Hence this approach is not suitable. We use a much simpler approach. The message data contains a data-length field. If the actual data is too large to fit in the message, it is stored in a shared memory segment and the id of this segment is passed in the message instead of the data itself. This is implemented as a C union construct. The receiving end just checks to see if the data-length field is larger than the maximum permissible and decides which field on the union (of shared memory id and byte array of data) to use to fetch the data.

4.2 The Data Mining Framework

In this section we describe the Data Mining framework used by ARGUS. We use a rule-based data mining system based on the RIPPER [7] algorithm. The framework comprises of three major components-the Data Cleansers, the Learning Agents and the Anomaly Detection Agents. We now describe each of these components.

4.2.1 The Data Cleansers

- *Feature Selection*

The importance, difficulty and our implementation of feature selection will be discussed here. Features are the predicates on which rules will be framed, i.e. for example let the number of half-open connection requests received in the last second be a feature F. Then rules will be generated using the value for F as a predicate, such as **if (F $\geq$ 5) then SYN_FLOOD**. It is usually a pretty difficult task to identify the correct set of features and addition of features that incorporate temporal and statistical information i.e. features like traffic on the user-ports in the last n seconds, number of un-wanted syn-acks received by httpd in the last n seconds etc., were shown to be very effective in [3]. Thus, we have relied to a large extent on features that have been popularly cited and identify this as a major area of improvement of our system. The features that we chose are:

1. **State of connection establishment** This is a parameter that could be one of the following values, (a)connection rejected: a SYN packet was replied with a RST, (b)connection attempted but not established: a SYN packet was sent but never got a SYN_ACK response, we use the standard value of TCP Connection timeout on linux machines to weed out connection records that are stale for greater than TCP Connection timeout duration, i.e. the last update on this connection record occurred later than TCP Connection timeout duration in the past and (c)un-wanted syn acknowledgement received: a SYN_ACK packet was found on the network that responded to a non-existent SYN

2. **State of connection closure:** This too is a parameter that could take one of several values, (a)normal close: FIN pkt's sent out by both sides and all data packets have been acknowledged, (b)disconnection: either by stale timeout, or by a reconnection attempt on the same $src-port, dest-port, src-ip, dest-ip$ pair, i.e. one of the sides suffered a disconnection but the other side still assumes that the connection is established, (c)abort: one of the host's causes the connection to abort by sending a RST flag across and all data packets are acknowledged and (d)half-closed: only one side of the connection has sent a FIN and the other side has either gone quite for greater than TCP TIME_WAIT timeout duration.

3. **resent rate** The number of bytes that have been resent, control packets count as one byte.

4. **wrong resent rate** The number of bytes that have been wrongly resent, i.e. they were sent even after being acked.

5. **duplicate ACK rate** This counts the number of duplicate acks received.

6. **hole rate** Duplicate ACK's are seen as an indicator that holes have formed at the receiver's end. This estimates the sum of the sizes of the holes, hole-size is approximated as the $seq\_number\_sent\_forward + data\_bytes\_sent\_forward - duplicate\_ack\_seq\_number$. This is an approximation as it only estimates the maximum size of the hole that may be formed as packets received out of sequence are not acknowledged in TCP.

7. **data bytes sent in either direction** Lots of connections with very little data being sent might be probes (if not telnet packets).

8. **percentage of data packets:** The percentage of data packets in this connection.

9. **percentage of control packets:** It can safely be assumed that for "normal" connections of a specific type (determined based on the server port they associate with), these percentages should be fairly constant. Hence a deviation might be an anomaly.

10. **number of connection establishment errors in the last n seconds** Indicative of the type of traffic on the segment, will lead to false alarms in case of a network partition.

11. **all other errors in the last n seconds** Again indicative of the healthiness of the network traffic. These temporal information give a context for each connection record within which it could be analyzed. Contextual information would be necessary as in stray errors might be neglected but many errors within a short span of time are suggestive of something fishy.

12. **connections to designated system services in the last n seconds** Again contribute contextual information.

13. **connections to user applications in the last n seconds** Context, reflective of the type of activity on the network. For corporate networks, a sudden

increase in traffic on user ports can safely be flagged an intrusion (unless a new application has been involved that uses this port).

14. **averages of connection duration and data bytes over all connections in the last n seconds** This can readily be linked up with building temporal profiles for the network, i.e. a large amount of network traffic at an unusual time can easily be flagged as an intrusion.

15. **averages of connection duration and data bytes over all connections to the same destination in the last n seconds** Can possibly identify hot-machines and not-so-hot machines besides building up temporal profiles for traffic to each of these machines. Besides any attempt at intrusion unless it is on very busy ports (or the attacker is very patient) is assured to cause large fluctuations in this parameter

16. **averages of connection duration and data bytes over all connections to the same service** This could be reflective of probes trying to pick out vulnerabilities in known services.

It should be noted that the choice of the above is purely heuristic based. Refinement of these features would require greater insights into individual attacks, but the idea behind statistical anomaly detection is to do away with un-necessary human effort. So an intermediate course must be obtained. Yet, it is our firm belief that more features exist that are intuitively sound as features. It is left to future extensions to pick out these features. It would be really useful if the choice of these features could be automated.

- *Generation of Connection Records*

We initially tried to use the library libpcap, but ran into problems due to lack of sufficient documentation. So the present impelementation runs on top of output generated by tcpdump of tcp packets. The format of the output of tcpdump that is used as input to the data cleanser is **timestamp src_ip:src_port** > **dest_ip:dest_port flags startseq:endseq** [**ack ackval**] [**win winval**] [**urg urgval**] [**options**]. The field **flags** takes some valid combination of S (SYN), F(FIN), P(PUSH) and R(RST). "." is used if there are no flags. **startseq** is the sequence number of the first data byte that is transmitted in this packet, while **endseq** is the sequence number of the last data byte + 1. Values of ack (acknowledgement number, 1+sequence number of the last byte that is received, win (size of the receiving window at the source of this packet), and urg (pointer to the location of urgent data) are used whenever needed. Generation of relative sequence numbers is suppressed.

Each line of the tcpdump is parsed for the above values. A map of connections that are active at any point of time are maintained. Upon parsing of a line of tcpdump, a search is made for an active connection in either the forward (from the connection initiator: client), or the reverse direction (server). Once a match has been found, the TCP state diagram is used to maintain simultaneously the states of both ends of the TCP connection. This state is updated as and when required and values of chosen features as described in 4.2.1 are updated. If no match is found either an erroneous connection is built and de-activated instantaneously or a new active connection record is created, depending on whether the current packet is a valid start for a new connection. Timestamps are stored for several key stages of a connection record, like the time at which it was created, time of last update in either direction etc.. While it is relatively straightforward, this task of maintaining the simultaneous state of both the ends mid-way is a laborious task, as constant attention has to be given to instances where packets could be resent and error conditions have to be handled much more carefully.

Upon closing of the connection, either successfully or erroneously, an active connection record is moved into the last 10 second vector after setting its timestamp for de-activation. We have chosen the value of n as 10 seconds tentatively and it would be interesting to study the effect of the variation of n on the effectiveness of rules generated. Wheneve a new entry is made into the vector containing the connections de-activated in the last n seconds, this vector is made current i.e. connections that have been de-activated at a time that is earlier than n seconds from the present time are moved onto stable storage. After updation of this set of connection records, the existing set of records is used as the basis for calculating values that make up the temporal and statistical i.e. contextual information of the currently de-activated record. Records that have passed into the stable storage are passed onto the anomaly detecting agents.

One major issue of concern is the over-flowing of buffers that could occur if the output of tcpdump is created at a much quicker rate than the rate at which connections could be cleaned and compacted. This might occur either due to significant increases in the network traffic or due to excessively long-duration connections. It has been observed that random dropping of packets, i.e. tcpdump input lines will adversely affect the quality of the records being generated. Thus a naive approach to this solution was to discard only packets that belong to connections for whom state is not already being maintained, store the connection key i.e. the four tuple of connections whose packets have been discarded and consistently discard all the packets of these connections once we have discarded the first few packets. However the percentage of new connections that are discarded is a parameter that is determined based on load/ buffer-size-in-use . It would be interesting to investigate more elegant approaches of tackling this problem, however it has been observed that no network-based online-connection record generator can be completely free of this problem.

4.2.2 The Learning Agents

The Learining Agents use a state-of-the-art rule based learning algorithm called RIPPER. The rule generation system works in two modes. In the first mode, it is fed with the connection records of the normal network traffic, and it generates rules characterizing the normal traffic profile. These rules are used by the Anomaly Detection Agents for detecting deviations from the normal. In the second mode, the rule generation system is fed with the connection records of the normal network traffic alongwith tagged anomalous connection records, and it generates rules that constitute the signature for the anomaly. These rules are used to generate new N-Code filters for NFR.

■ *Rule generation using RIPPER*

RIPPER is an efficient rule learning algorithm, which produces very low error rates even on noisy data. It is a generalized algorithm and has to be suitably modified when applied on a particular domain. In our case, this domain comprises of connection records of network traffic. We now describe our implementation of RIPPER.

Each connection record is transformed into a set of attributes. The attributes are either discrete or specific. The objective of the rule learning algorithm is to generate rules to classify connection records (specified as sets of attributes) into a set of classes. A configuration file containing the following information is provided:

- The set of classes in the system.
- The set of attributes. For each discrete or nominal attributes, the legal values it can assume are specified. The continuous attributes are just marked as continuous.

The system is provided with the example data set on which to learn in the form of a collection of records, each record containing the values for the various attributes and the class this record belongs to.

We first describe a two-class version of the implementation, and later show how multiple classes are handled. Each record in the example data set belongs to either of the two classes, which we would henceforth refer to as A and B. The rules are generated for the minority class (class having fewer examples in the data set), which we would assume without any loss of generality to be class A. If a record matches a generated rule, it is classified to belong to A. Thus, for the Boolean case, a rule is simply a conjunction of conditions and the complete rule set is a DNF formula.

To build a single rule, the rule learning process goes through the following steps:

1. **Partition the Data:** Examples belonging to both A and B are randomly partitioned into growing sets and pruning sets in a 2:1 proportion. We refer to these sets as GrowA, GrowB, PruneA and PruneB.

2. **Grow a rule:** A rule is grown to overfit the data set. We use a modification of the FOILv6 [10] tool for this purpose. It works over the grow sets and returns a conjunction of conditions. The conditions are of the form $\alpha_n = v$, where α_n is a discrete attribute and v is a legal value it can assume; or $\alpha_c \geq \theta$, or $\alpha_c \leq \theta$, where α_c is a continuous attribute and θ is some value that it assumes in the data set. FOIL begins with an empty conjuction and keeps on adding conditions that maximize its gain criterion, until the rule does not match any example from GrowB.

3. **Prune the rule:** The grown, overfitting rule is then pruned. We delete any final sequence (suffix) of conditions from the rule such that the new pruned rule maximizes the following rule-value metric:

$$v(R,\ PruneA,\ PruneB) = \frac{a\ \text{-}\ b}{a\ +\ b}$$

where R is the pruned rule, a is the number of examples in *PruneA* covered by R and b in the number of examples in *PruneB* covered (incorrectly) by R. The process of deletion is continued until it does not further improve the metric.

After a rule has been found, all the examples in the data set covered by the rule (both of class A and class B) are removed and the above steps from 1 through 3 are repeated. This process continues until the stopping condition is achieved.

The **stopping conditon** is defined as follows. We calculate the total *description length* of the rule set, each time a new rule is added. This is the total number of bits required to encode the rule set in the MDL encoding scheme. The scheme allows identification of a subset of k elements of a known set of n elements using

$$S(n,\ k,\ p) = k log_2 \frac{1}{p} + (n\ \text{-}\ k) log_2 \frac{1}{1-p}$$

bits, where p is known to the other side (the recipient of the encoded message). Given the example data set, we can calculate n, the maximum number of possible rules that could appear in a rule. Thus a rule which is a conjunction of k conditions can be expressed in $|k| + S(n,\ k,\ k/n)$ bits, where $|k|$ is the number of bits required to encode the integer k. This number, multiplied 0.5 times to adjust for possible redundancy in the attributes, forms the description length of this rule. The summation of the description lengths of all the rules in the rule set forms the total description length. The addition of new rules to the rule set stops when this description length is more that 64 bits larger than the smallest description length obtained so far, or when there are no more examples of class A left. The rule set is then simplified by removing the rules, beginning with the last added rule, that increase the total description length.

The rule set thus obtained is **post-processed** for optimization. Suppose our rule set at this stage is $R_1, ..., R_k$. We consider each rule in turn. For the rule R_i, two alternative rules are constructed. The *replacement for* R_i is formed by removing R_i from the rule set, and growing and pruning a new rule in its place. The growing starts from the empty conjunction and pruning is guided so as to minimize the error on the entire rule set (which includes the replacement rule) on the pruning data. The *revision of* R_i is formed in a similar manner except that the *revision* is grown by adding conditions to R_i instead of the empty condition. We then replace R_i by each of the variant rules and find the total description length of the simplified rule set in each case. The case in which the length is lesser is chosen. After the optimization of the rule set is performed, more rules are added to the rule set to cover the examples of class A that may remain uncovered due to removal of rules in the optimization process. The postprocessing process can then be repeated and any number of iterations can be performed. In our implementation, we use two iterations.

Above we described the case for two classes. We handle multiple classes as follows. The classes are ordered in increasing order of prevalence—*i.e.*, the ordering is $C_1, ..., C_k$ where C_1 is the least prevalent class and C_k is the most prevalent. Then, using the Binary version described above, we generate a rule set that separates C_1 from the rest of the classes (with class A being denoted by C_1 and class B being denoted by the remaining classes). Next all instances covered by the learned rule set are removed from the example data set and the Boolean process is repeated with examples of C_2 labeled as examples of A and examples of $C_3, ..., C_k$ labeled as examples of B. This process is repeated until the single class C_k remains, which is used as the default class. This re-ordering of the classes by their prevalence in the example data set is done automatically by the program.

▮ *Rules for normal profile*

As described in the beginning of 4.2.2, each connection record is suitably transformed into a set of attributes, in conformance with the configuration file, for the learning and classification algorithms to work on them. For the mode when the rule learning system has to generate rules characterizing the normal profile, appropriate class and attribute structure is required with the example data set comprising entirely of normal TCP traffic. We use the following classes and attributes in this case.

The set of attributes are the various fields of the connection records, as described in 4.2.1, except from the service (destination port). The destination ports form the classes, with the all the user-defined, non-standard ports being clubbed together into one class. Thus we have classes of the form *ftp*, *telnet*, *non-standard* etc. The rules generated take the fields of a connection record (with the destination port field removed) and predict the destination port. These rules are used by the Anomaly Detection Agents as described in. 4.2.3.

▮ *Rules for attack signatures*

In this case the example data set contains a combination of connection records of normal TCP traffic and some connection records of some anomalous activity. The Boolean case is directly used, with the examples of normal traffic labeled A and example(s) of anomalous traffic labeled as B. The rule set generated act as a signature for the anomalous activity and are used to generate new N-code filters for NFR as described in 4.4.

■ *The internals of a learning agent*

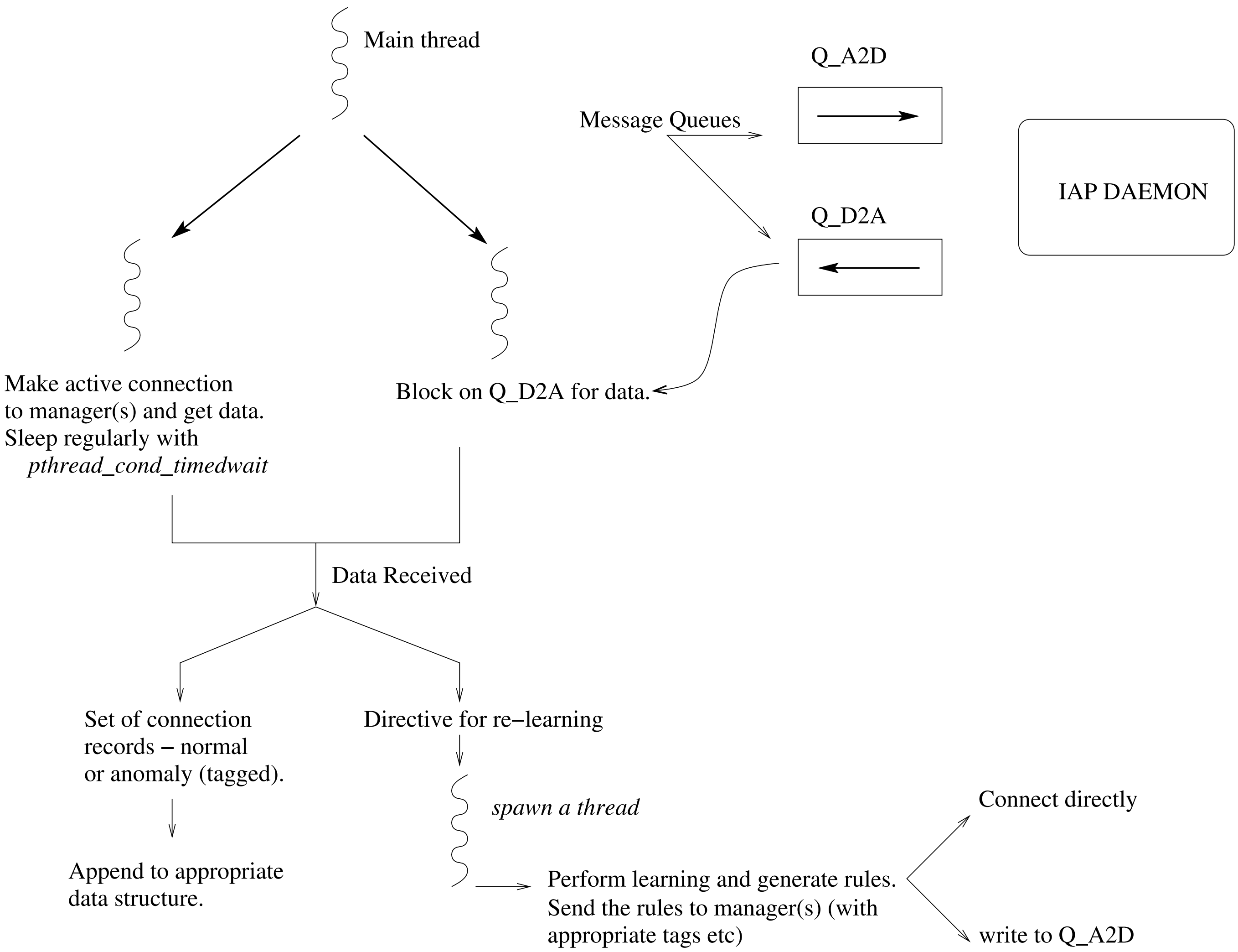

Figure 2: Learning Agent

4.2.3 Anomaly detection Agents

The Anomaly detection agents are the application of the rules produced by the RIPPER based rule generation system implemented as a part of the learning agents. As described in 4.2.2, the rules produced by the learning agent for the purpose of anomaly detection characterize the normal profile. The destination ports serve as the classes and the remaining fields of the connection records serve as the attributes.

The anomaly agents receive the rule set from the manager and store this in the form of a file. The configuration file describing the class and attribute structure present on the learning agent's host is present on the anomaly agent's host also. The classification process goes through the following steps.

1. Connection records are obtained by a call to the data cleansing module.

2. Each connection record is then matched against each rule of the rule set. Each rule in the rule set has an associated fraction called its *confidence value*. In case of multiple matches, matching rule with the highest confidence value is selected.

3. The rule is then applied to the connection record and the class, which in this case is the destination port, predicted by the rule is obtained. If the predicted value matches with the actual value, the connection record is termed normal. It is sent to the manager tagged as normal data with a probability of 10%. If the predicted and the actual values mismatch, the connection record is marked anomalous and is sent to the manager alongwith the confidence value of the rule used for classification.

4. The above steps are repeated for new connection records.

▮ *The internals of an anomaly detection agent*

Figure-3 describes the detailed internal implementation of an anomaly agent. Like all other components of Argus, it is fully multithreaded.

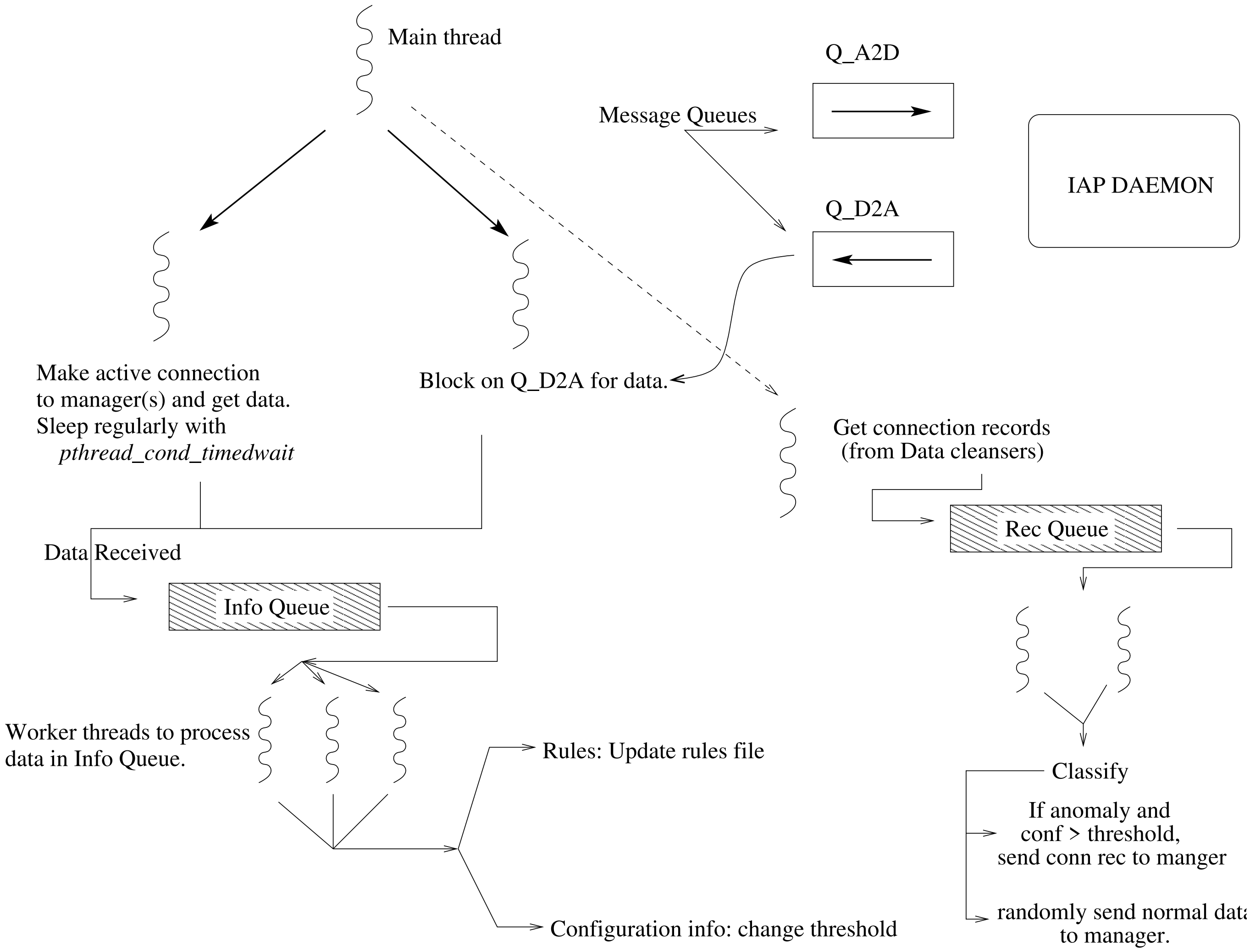

Figure 3: Anomaly Detection Agent

4.3 Driver for NFR

We use an off-the-shelf knowledge based system NFR [8] that contains its own connection record creator, an attack signature database and a rule application engine (misuse agent). To integrate NFR into Argus, we needed to build a driver that interfaces Argus with NFR. The functionality that the driver had to handle was

1. Transfer NFR alert information from NFR's IDA to the manager of Argus.
2. Forward the manager's directives to NFR's IDA. These directives could be the N-code for new attack signatures or threshold-changing instructions.

However, despite persistent requests, NFR has refused to give us access to their source code or sufficient information about their protocol of communication between the IDA and manager. To summarize, the problems in achieving the abovementioned functionality were:

1. IDA-Console communication goes encrypted. This encryption mechanism was not known.
2. Protocol for this communication was not known.

Luckily, the NFR distribution consisted of Perl scripts that present a text-based interface to NFR and also consisted of functions (in binary) that perform the required encryption and decryption. This seemed to solve the problem but brought an additional constraint that the driver be coded in Perl. We overcame this constraint using a Perl add-on module ExtUtils::Embed that provides for embedding perl in C/C++. This module allows us to initialize the perl interpreter inside a C/C++ program and thus the job is done without sacrificing the power of C/C++. Our final implementation uses only the relevant functions that encrypt/decrypt and have most of their functionality implemented in C++. The relevant details about embedding follow:

The following is a brief background on how-to embed Perl in C/C++ along with our choices of calls Perl executable is made up of two parts: a library of core Perl routines (libperl.a on Unix systems) and a simple driver file, perlmain.c, containing main(), which, stripped of portability necessitated aspects is:

```
#include < EXTERN.h >
#include < perl.h >
staticPerlInterpreter *my_perl;
int    main(intargc, char **argv, char **env)
{
    my_perl = perl_alloc();
    perl_construct(my_perl);
    perl_parse(my_perl, xs_init, argc, argv, env);
    perl_run(my_perl);
    perl_destruct(my_perl);
    perl_free(my_perl);
}
```

perl_alloc and perl_construct create an interpreter object. perl_parse initializes this object, parses the command-line parameters provided to it via argc and argv, calls an initialization routine, xs_init, to load other extensions (or to at least initialize the dynamic loader), and finally parses the script provided as part of the command line. perl_run executes the script (name present in argv and initialized in the interpreter object my_perl by perl_parse) . Finally, perl_destruct and perl_free shut down and deallocate the interpreter.

To embed perl, we need to clone this file and take care that the necessary modules are loaded by suitably defining xs_init. The module ExtUtils::Embed allows us to automatically generate xs_init for the corresponding extensions that we may need to use. Actually, the module NFR was being loaded dynamically by perl's DynaLoader module itself, so xs_init only involved initializing this dynamic loader. The required encryption, decryption functions could be directly called from the C/C++ program using the following methods,

perl_call_argv(char *sub, I32 flags, char **argv);

This call is available in the standard Perl distribution, and calls a subroutine (name pointed to by sub) with an array of string arguments terminated by NULL (argv).

However no provision for returning results is present. The only currently implemented value for the flag is G_DISCARD that asks Perl to silently discard all returned results. Therefore this call is only marginally useful, mainly in testing.

perl_call_va (char *sub, [char *type, arg,]* ["OUT",] [char *type, arg,]* NULL);

This call is a convenient means for passing a null-terminated list of "typed" parameters to a Perl subroutine and to collect the returned results into a list of parameters (similar to printf and scanf). The only possible values for type argument are i, s and d (integer, string, double). The string OUT begins a list of return parameters, which are pairs of type specifiers and appropriately typed pointers. String output parameters are copied into the buffers supplied, which consequently should have enough space to absorb the returned strings. The list is basically a set of pointers and the types of the variables that these pointers refer to (needed as the values need to be copied from and to the pointer specified locations). This call has been most often used. Other allied functions that allow manipulation of global variables in the perl program include get_int, set_int, get_string, set_string, get_double and set_double that have been used as required.

Now that issue 1 was solved, we embarked on a sniffing based analysis of the IDA-Administrative console communication protocol. We started with the few perl scripts that came pre-bundled. These perl scripts were designed to query the IDA for certain information like summaries of the alerts seen today, register the client with the IDA to receive alerts from the IDA etc. However the communication from the administrative console to the IDA that also allows for deployment of code at the IDA and manipulation of the IDA's attack-specific threshold variables was still unknown. Though the last resort of automatic n-code generation followed by a daily manual deployment of the code was identified, lack of real-time control over the threshold variables and the code caused us to undertake a simultaneous stimulation and sniffing of network traffic so as to understand this communication. The following is our present understanding of this protocol.

The IDA runs a pseudo-HTTP server which replies to client requests. A typical communication is illustrated in Appendix-B. The client starts of with a query for the time at the IDA's end. The value returned for this is a time-variant piece of information, that along with the user's password is passed as input to the encryption initiator on the client-end. Use of this information is like using random variables so as to guard against replay attacks. The encryption initiator function returns a Ticket and an object that encapsulates the key to be used for later encrypted communication. The ticket is surmised to be some combination of the user's password that is encrypted with a key that is keyed by the time-variant piece of information. The IDA, upon receiving this ticket, and perhaps having stored the initial information that it sent out to the client can now decrypt the ticket to get the user's password, validate it and then generate the key independent of the client. When the server validates the password, the communication shifts from its present clear-text mode to the encrypted mode.

The packets now follow an HTTP encapsulated in HTTP mechanism. The outer header is clear-text which specifies the mime type of the data and the ticket while the encrypted data contains the actual communication containing HTTP packet. This inner header is a cgi-bin directive like POST /cgi-bin/<command> that invokes the corresponding code at the server-side. Several values for the <command> have been experimented with, but it has been slow-going as each command involved its own sub-protocol based on the arguments it wants.

We abandoned this analysis settling on the last resort of manual deployment of the N-code rules into NFR and would like to stress once again the fact that multiple requests to NFR Inc. for documentation of the IDA-Console protocol have not been responded. The present implementation has been clearly illustrated in Figure-4.

■ *The internals*

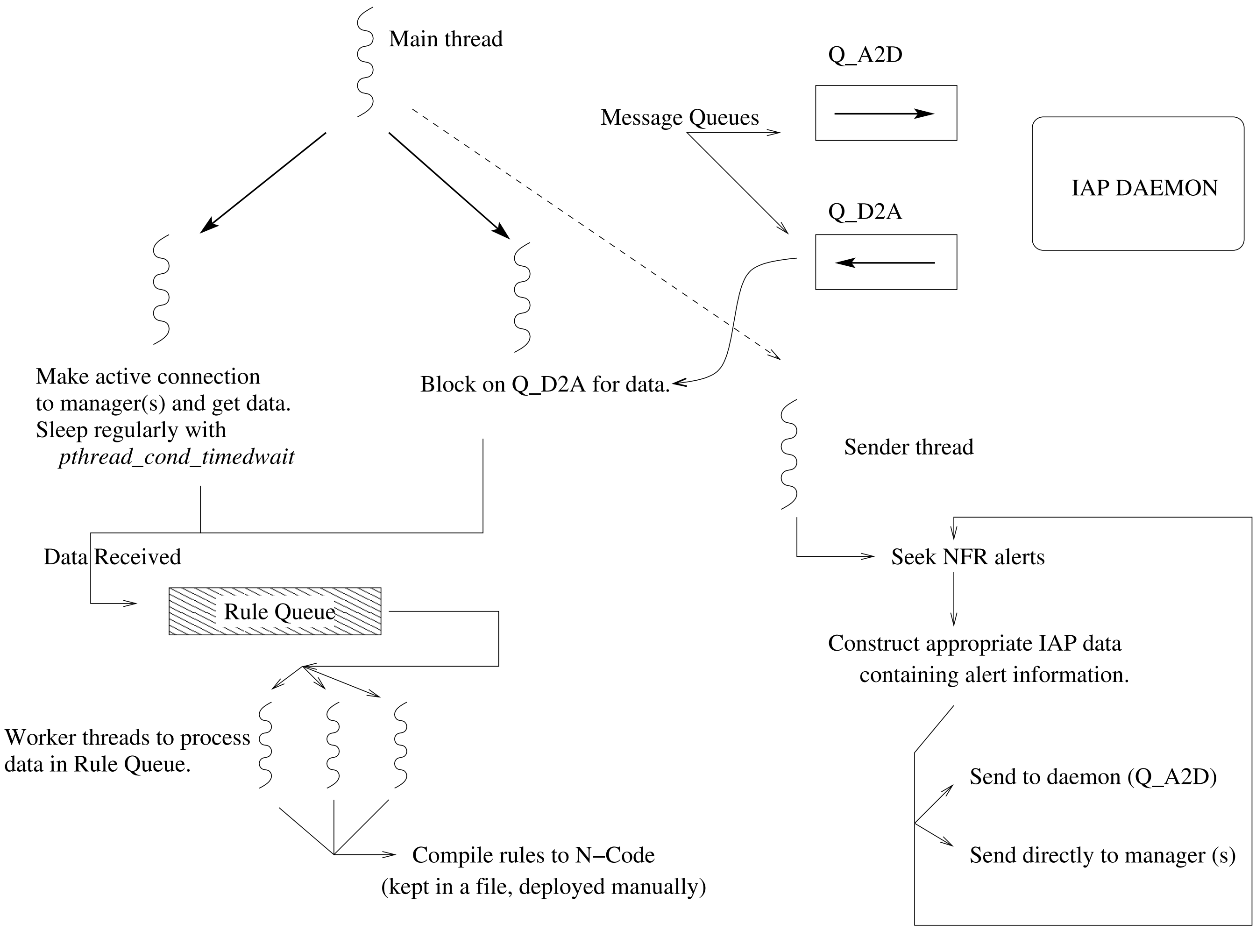

Figure 4: NFR Driver

4.4 Managers

This is the component wherein most of our novelty lies. The various execution agents be they anomaly-based or knowledge-based send alert information to the manager. The manager is also initialized like an ordinary IAP agent but with slight differences as already pointed out in 4.1.3. The general structure, i.e. one of two distinct receive threads receiving data from the analyzers and putting it into the receive queue, several worker threads processing the information present in the receive queue and prescence of a send-queue that is serviced by sender threads is similar to those of the other agents. The uniqueness comes in the type of data that arrives and in the handling of this data.

The following are the presently handled case:

1. **alert information received from NFR**: Display the detection of this attack. Call the corresponding response handler based on the type of the intrusion detected. Store alert information in the anomaly queue for correlation with later attacks.

2. **receive registration request from remote agents**: update the data structure containing the list of known attacks.

3. **rules received from the learning agent**: If these rules are for the anomaly based agents, pass them on to the agent. The anomaly based agents update their effective rule set upon receipt of a directive from the manager. If the rules are for NFR, generate the corresponding N-Code filter and pass it on to the NFR driver. Generation of n-code from the rules of the learning agent assume that a one-to-one map exists between the features in the connection records and features that could be identified by NFR. If this is not the case, then a modification of the base filters of NFR would needs to be done, but the lack of NFR's source code prevents us from doing so. The conversion of rules to N-code uses ideas similar to those in [6].

4. **alert information from the anomaly based agents is received**: Store alert info in the anomaly queue.

The following is the functionality of the tagger thread which waits for fresh input into the anomaly queue. If

1. **the source of the alert is NFR**: Matches this data with other pre-existing

alert information to detect the possibility of a distributed attack or a continuation of a minor earlier exploit. If such is the case, the relevant alert is triggered and call the corresponding response handler.

2. **the source of the alert is anomaly-based agent**: Prompt the user to tag the connection record. If the user tags it as an alert, pass-on the tagged connection record to the learning agent so as to enable it to generate rules for the signature of this attack for the knowledge-based agent. Also match it onto pre-existing alert information to detect distributed/correlating attacks, and store the info for matching with later attacks. If the alert is tagged as an error, update counters that record the number of false alarms. The rate of false alarms could be used as a parameter in deciding how much "normal" data should be passed onto the learning agent, how frequently the learning agent needs to re-generate rules for normal data and how frequently the anomaly based systems need to age their rules.

The handlers that are called by the tagger thread would involve actions in specific response to the attack. For Ex. blocking of an external IP address at the firewall etc. Further these handlers could also re-assign values of relevant thresholds in the anlayzers. For Ex. Detection of several intrusions in NFR is threshold-based, relevant threshold values may be changed and directives sent to the driver. For the anomaly-based systems, one relevant threshold is the confidence of mis-match, which could be suitably changed.

▮ *The internals*

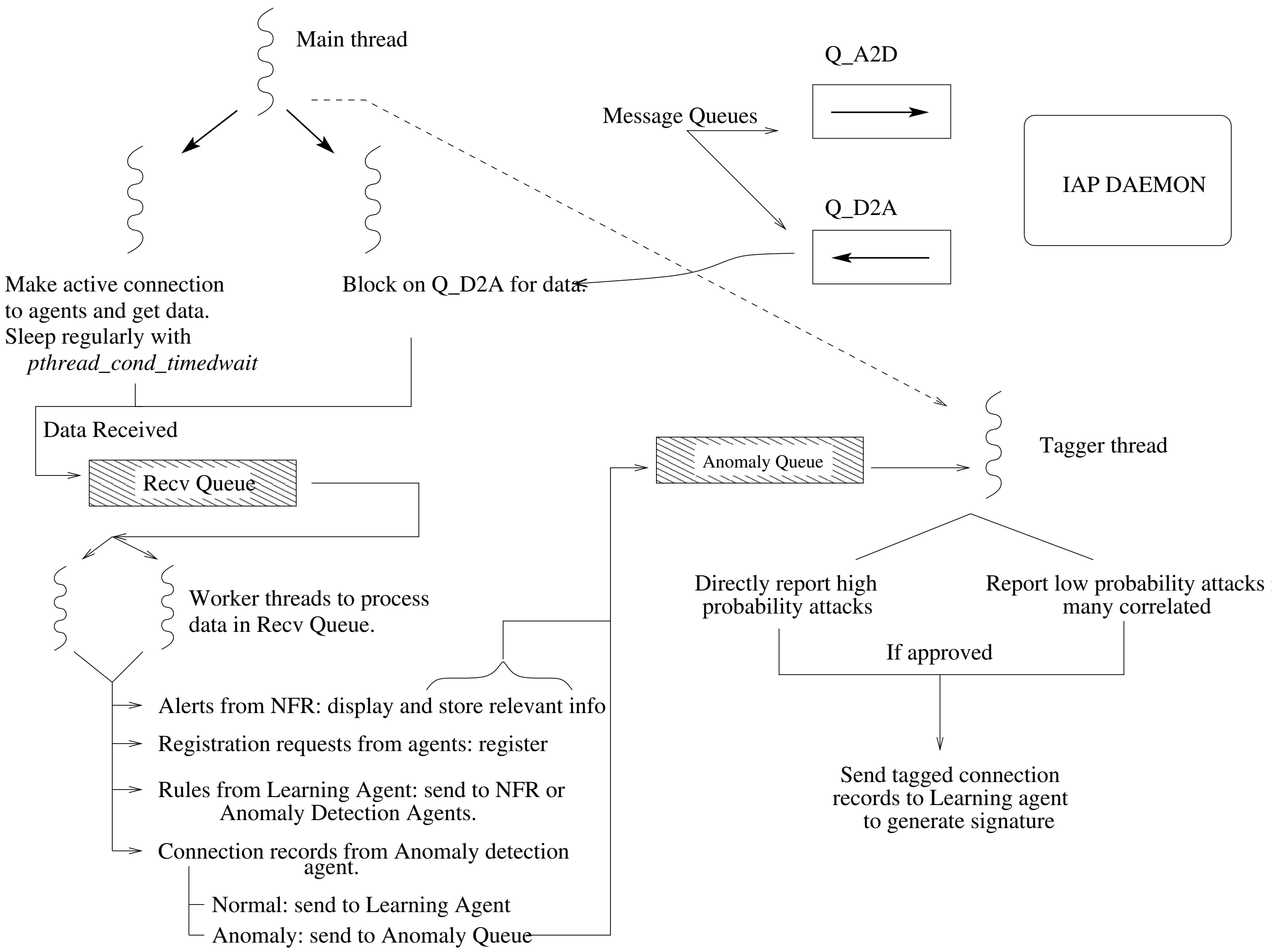

Figure 5: Manager

Chapter 5

Experiences with Argus

We have successfully built and integrated the individual components of Argus. The following is a description of the test-bed and our experiences with Argus.

5.1 Testing the IAP Framework

The NFR IDA was installed on a network segment that contained an equal mix of both Linux and Windows clients. Each agent, i.e. the NFR driver, the learning agent and the manager agent, was tested in two different scenarios i.e. those in which the agent could establish connections with its counterpart (active connections to the remote daemon) and those in which it leaves the data for a remote agent at the local daemon itself (changing the agent configuration files). The IAP framework was proven to work correctly in both these scenarios. IAP proxy was installed on one of the machines and was tested by setting it as the proxy for communication between agents in two different networks (the cc network and the cse network). We also tested with messages of different sizes and different amounts of data being present at the daemons. Messages were explicitly constructed of sizes larger than the maximum size of the messages that go through message queue to test the implementation of the daemons (shared memory). The robustness of the system was tested by simulating faults in one of the counterpart agents while agent communication was underway. An issue of optimization was in collection of data from remote agents that is stored at remote daemons. Alert throughput (i.e. the ratio of the bytes of alert information transmitted to the total bytes transmitted on the network) was recognized as a measure of efficiency of the framework. This has been vastly improved by two subtle changes:

- Increasing the time lag between repeated connection initiations to remote daemons seeking data, and
- Remote daemons aggregate all the alert information that has been obtained for this agent and send it over as a single IAP Packet upon request.

5.2 Experiences with the data cleanser

The data cleanser which basically maintains the state of a TCP connection by looking at traffic on the network segment was tested for both normal and anomalous data. Hand-constructed tcpdump input was used to exercise the segments of the code that were not exposed during these test runs. By re-directing the tcpdump output of a normal network segment as the output of the data-cleanser, we tested it's performance on normal data. Later we used a vulnerability analysis tool SAINT to generate spurious traffic on the network segment and thus tested it on anomalous data. It has been observed that the data on most of the network segments in IIT Kanpur is mainly http traffic to and from the Squid proxy, the next closest contenders were NFS, POP, TELNET and FTP in that order. It has been observed that in the initial stages of the start of the cleanser, most packets belong to connections whose establishment packets have not seen by the cleanser. Invariably these packets get tagged as errors. We modified the cleanser to store ad-hoc packets in the initial stages, i.e if it has seen less than a 1000 tcpdump lines, tentatively at first and later upgrading it to a valid connection upon receipt of one packet from either end point.This vastly reduced the false alarms.

5.3 Experinces with the learning agent

To generate sufficient amount of normal network traffic for the learning-agent initialization, we got the data-cleansers off-line and dumped the connection records into a file. The learning agent genertes rules for classifying a connection record into a class (the destination port) based on all the remaining attributes of this record. An initial rule-set was created by causing the learning agent to learn off-line from this connection record file, with appropriate configuration files containing the specification of the attributes and their types contained in the connection record. This rule set was then used in getting the data-cleanser-enabled-anomaly agent online as this set constituted the base-rule set of this agent. As has been noted above, the mis-classification rate of the anomaly agent is low when the network segment is experiencing primarily HTTP-Squid traffic. Simulating telnet/ftp traffic caused it's mis-classification rate to increase, and thus more alarms were sent to the manager. This is the proper action for an anomaly agent as such simulated data was not present in the training set and so was ab-normal. However abnormal data once tagged by the manager as normal was sent to the learning agents, these agents used this new normal data to improve their existing rule set, thus regardless of the initial training data, sufficient exposure to a particular network segment would enable our anomaly agent to tune itself. This has been seen in the consistent drop in the rate of mis-classifications per minute over an extended session of activity. As has been implied above, we assume the prescence of human element at the manager who tags the abnormal information coming from the anomaly agent as normal or abnormal. This tagging could be done in batch-mode too.

Following are some sample rules generated on normal data found in the Hall-I intranet.

if cls=13 && bytesF$\leq$233 && dataP>29 **then** class=23

ifcls=13 && bytesF>233 && dataP$\leq$14 **then** class=3128

if bytesF>233 && contrP>14 && contrP$\leq$73 **then** class=3128 **if** srcPort>1025 **then** class=1025

We chose all ports less than 1024 as valid classes and then clubbed all the others into one class of user-port connections. Squid was treated as a special case. The variables in the above and their explanations are cls(state of close of the connection: 13 refers to the forward stream being closed earlier than the reverse stream), bytesF(the number of bytes in the forward direction i.e. from source to destination), dataP(the number of data packets), contrP(the number of control packets), srcPort(the source port of the connection).

5.4 "Information Exploration Shootout" runs

To generate concrete results on the working of our anomaly-agent, learning-agent combine, we have obtained a set of tcpdump data available at "http://iris.cs.uml.edu". This set of data was used as part of an "Information Exploration Shootout", a comparison of then-existing intrusion detection systems. The data set consists of processed tcpdump output of packet headers, received at a gateway machine. 3 of these output files were of artificially generated network intrusion traffic, while the fourth of these was on normal network traffic. The files contained each tcpdump packet header on a separate line as a set of comma separated values.

The data cleanser was again brought off-line and was modified to read these files. The generated connection records were then fed into the off-line learning agent. We trained the learning agent on about 70% of normal data and the rules were fed into the anomaly agent. The percentage of mis-classifications, i.e. the situations wherein rules identify the destination port of a connection to be something different from the actual one, was a measure of the abnormality of the network traffic. The anomaly agent was then fed with input from the whole of all the four files. Table: 6 shows the percentage mis-classification obtained on each of these files. As the input data distinguishes between incoming connections and outgoing connections, we present our output on each of this type of connections differently. As can be seen, there is a small but sufficient margin between the score on the normal data and the least of the scores on the anomaly data. These ideas have been motivated by [3] but our results are inferior to those obtained by this author, which could be attributed to their greater care in choosing connection features.

% misclassification ↓	Data ⟶	Base	Intrusion 1	Intrusion 2	Intrusion 3
	in–coming	4.97%	8.04%	6.91%	12.1%
	out–going	4.03%	4.94%	5.83%	5.16%

Figure 6: Percentage mis-classifications on sample tcpdump output.

5.5 Experiences with the NFR Driver

Interfacing NFR with Argus, is one-way done, i.e. alerts generated at the NFR are transfered to the Manager. This has been tested by causing the vulnerability analysis tool SAINT to target machines on the network segment that the IDA was on. At different rates of alert generation, it has been proven that NFR's alerts are accurately relayed to the manager. However, On the reverse direction we have faced problems owing to the ignorance of the NFR IDA-Console communication protocol. Further transfer of learning agent generated rules into N-Code is not possible in the current installation as we are unable to map several features into N-Code without modifying some of NFR's internal modules. The trade-off was between a choice of simple yet inefficient features for the anomaly agents so as to aid rule-building and an involved but efficient set of features that cannot be readily translated into N-Code. We chose the latter and demonstrated the in-principle feasibility of the feature selection to N-code translation on simpler features. Argus, presently, dumps both signatures and threshold values on the driver end.

5.6 Experiences with the Manager

The relaying functionality of the manager, i.e. normal data from anomaly to learner, abnormal data from anomaly: tagged and sent to either the learning agent or ignored based on the real-time response at the manager end, signatures sent to the NFR driver has been tested at varying rates of each stream. The framework for alert-specific response specification has been tested, but more work needs to be done on specifying these responses. Further the structure for alert correlation has been implemented but the actual alert correlation is very domain-specific and work needs to be done on this.

Chapter 6

Conclusion and Future Work

Argus has been designed with a broad perspective in mind. Several loose-ends still remain in the implementation which have been pointed out during the course of this report. These loose ends are more on the lines of simplistic or naive solutions being used in scenarios wherein it would be both interesting and enlightening to try out different solutions or experiment with different values. We thus believe that several extensions to Argus are possible which would be interesting to take up.

Bibliography

[1] D. Gupta. IAP: Intrusion Alert Protocol. IETF, Intrusion Detection Working Group, Internet-Draft 2001.

[2] H. Debar, M. Huang, D. J. Donahoo. Intrusion Detection Exchange Format Data Model. IETF, Intrusion Detection Exchange Format Data Model, Internet-Draft 2000.

[3] W. Lee, S. J. Stolfo. Data Mining Approaches for Intrusion Detection. In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[4] G. Helmer, J. S. K. Wong, V. Honavar and L. Miller. Intelligent Agents for Intrusion Detection. In *Proceedings, IEEE Information Technology Conference*, Syracuse, NY, September, 1998, pp. 121-124.

[5] W. Lee, S. J. Stolfo. A data mining framework for building intrusion detection models. In *1999 IEEE Symposium on Security and Privacy*, Oakland, CA, May 1999.

[6] W. Lee, C. T. Park, S. J. Stolfo. Automated Intrusion Detection using NFR: Methods and Experiences. In *Workshop on Intrusion Detection & Network Monitoring*, Santa Clara, CA, April 1999.

[7] W. W. Cohen. Fast effective rule induction. In *Machine Learning: The 12th International Conference*, Lake Tahoe, CA, 1995.

[8] Network Flight Recorder 5.0, NFR Security Inc 2001. *http://www.nfr.net*

[9] Transmission Control Protocol, RFC 793.

[10] J. R. Quinlan, R M. Cameron-Jones. FOIL: Amidterm report. In Pavel B. Bradzil, editor, *Machine Learning: ECML-93*, Vienna, Austria, 1993. Sprngler-Verlag. Lecture notes in Computer Science # 667.

Appendix A

Argus Design

Appendix B

NFR's IDA-Console interaction

Appendix C

Example of IAP Connection Set-up