Argus - A Distributed Network Intrusion Detection System

Srikanth Kandula, Sankalp Singh
Dept. of Computer Science
Univ. of Illinois at Urbana-Champaign
Urbana, IL, USA
{kandula, ssingh7}@cs.uiuc.edu

Dheeraj Sanghi
Dept. of Computer Science and Engg.
Indian Institute of Technology Kanpur
Kanpur, India
dheeraj@cse.iitk.ac.in

I. Introduction

MODERN 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 of the information transmitted through these networks, the consequence of which, is a higher premium on network security. The availability of valuable information on modern 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 without suffering significant drop in data throughput. They are also extremely vulnerable to denial of service attacks as well as attacks that come from inside the firewall. End-to-end encryption or authentication algorithms are severely hindered by the lack of a public key infrastructure over the Internet. Network-traffic surveillance and intrusion-detection systems (IDS) have a crucial role to play in detecting such network intrusions in real-time. Besides providing sufficient information trace information to track the source of such "attacks", an IDS can possibly ameliorate the effects by executing timely prevention measures such as filtering data from "suspicious" sources.

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, which is the art of detecting attacks that are executed across a network. The complementary problem of detecting host-based intrusions is an equally interesting problem. Several approaches to detect host based intrusions exist [14], [2]. One such approach, expert-BSM[14] uses a security "monitor"' that sifts through system call traces and other operating system audit capabilities attempting to find a match with any of the known intrusion "signatures" that are encoded in its database. Network Vulnerability analysis (SAINT[22], SATAN[23]) is another complementary problem that involves estimation of the efficiency of network security measures by actively probing for known vulnerabilities.

Commercial tools to detect network-based intrusions (NFR[15], NIDES [16], Emerald[7]) as well as several research prototypes (Bro[19]) function by collecting network traffic into records and analyzing these records. Collection of records could be done by installing at every networked host, modules that collect data sent/received from the NIC of the host or by establishing stand-alone systems in each broadcast segment of the network, that collect the data that passes on this segment. Analysis of network data, so-collected, could be done either in real-time or in batch mode. Further there are two major approaches that could be used in the analysis of this data: Misuse based (also referred to as Signature or Knowledge based) or Anomaly based (also referred to as Behavior based or Statistical).

Misuse-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. Such systems usually include specification languages to encode attack signatures and interpreters/compilers that apply the generated attack-signature-filters on the candidate data.

This work was done when the authors were at Indian Institute of Technology Kanpur

Several such systems exist vaying in the ease-of-use, expressiveness, simplicity of the specification languages and the efficiency of their execution environments. The great power of such systems lies in their ability to accurately identify all known attacks by encoding signatures of these attacks. However these systems are insensitive to new attacks and variants of older attacks that are not encoded in the signature database. As a significant amount of programmer effort needs to be spent in encoding the many variants of some more complex attacks, automatic generation of signatures through data mining and other machine learning algorithms has come into vogue.

Alternatively, Anomaly-based systems use large amounts 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. Such an approach has the advantage of requiring less human effort and has a higer sensitivity to to new attacks or those attacks whose signatures have not been encoded. On the flip side, the training data has to be sufficiently exhaustive in order to construct an accurate profile, and there is the possibility of false positives. Further data-mining algorithms that generate profiles from data-sets have time complexity O(n^3) or higher [1]. The Argus architecture has a set of distributed loosely coupled analyzing agents, each of which, could be either knowledge-based or anomaly-based. Such a design choice allows us to achieve a "good" balance exploiting the strengths of both these kinds of systems.

The data collected can be analyzed in a distributed fashion at the point-of-capture or could be dispatched to a centralized authority that is responsible for analysis. Distributed analysis requires significant computational power at each capturing node and may not be able to effectively track correlated attacks unless all the components of the attack occur under the aegis of a single system. Analysis by a centralized authority allows perception of correlated attacks but levies a significant communication overhead, as the collected traffic or corresponding compacted records need to be transported to the centralized authority. Argus has "Managers" that act as stripped-down variants of the centralized authority. While retaining the advantage of distributed analysis, Argus exploits the IDXP[9] framework to enable attack correlation at the managers with limited communication overhead.

[1] n = size of dataset

Batch mode analysis of network traffic implies that the collected traffic records are compacted and stored for analysis at a later time. Such an approach has the advantage that computational resources, that could be better used, are not "wasted" during the actual time of an attack and is appropriate for passive flavors of IDS whose primary function is surveillance. On the other hand, the design objectives of most IDS'es[2] include attack response or real-time reporting of attacks and thus real-time analysis is more popular apart from being interesting and useful. Consequently, Argus performs real-time analysis.

The rest of this paper is organized as follows. In Section II, we describe the multi-agent architecture of Argus and the rationale behind the choice of the individual components involved in this architecture. In Section III, we describe the design and construction of each of the individual components of Argus. Section IV documents the results of the various experiments performed with Argus. Section V briefly describes other efforts at building intrusion detection systems. Section VI finally concludes with an enumeration of avenues for future research in this domain.

II. Architecture

Argus uses a hybrid architecture consisting of both misuse-based agents and anomaly-detection agents that use data mining. We have used Network Flight Recorder (NFR) as the knowledge-based component in Argus. Before we describe the architecture of Argus in detail, we would like to summarize the features of NFR, evaluate its strengths and argue why knowledge-based systems like NFR, and most commercial IDS'es, are not sufficient for effective intrusion detection.

A. Background: NFR Description and Evaluation

NFR is primarily a knowledge-based system with three basic components: the IDA, the administrative console and the central station. It uses a specification language called N-Code, similar to PERL in basic structure, to encode attack signatures. The *IDA (Intrusion Detection Appliance)* is the heart of NFR that contains attack signatures encoded in N-Code, 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

[2] Read as Intrusion Detection Systems

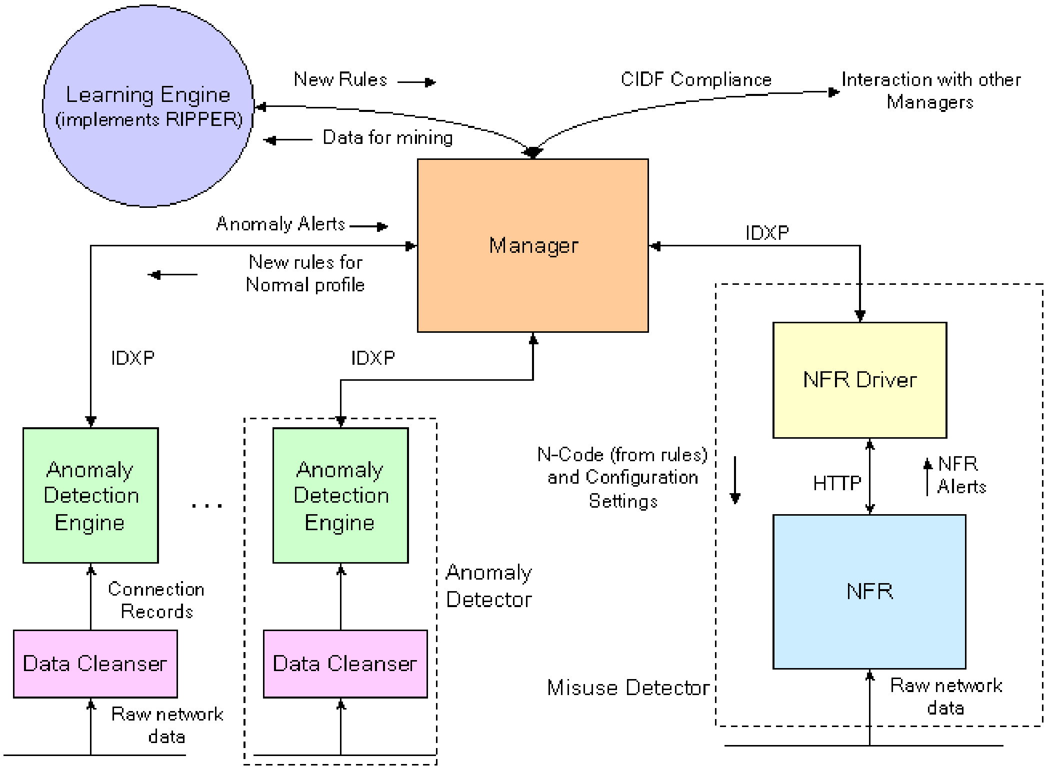


Fig. 1. Argus Architecture

defined hierarchy of filters. The IDA updates its database with alerts based on attacks detected by the filters as well as statistical information about the network and also runs a GUI server that allows administrative consoles, once authenticated, to query the IDA's alerts/statistical information database and change the IDA's configuration. *Administrative Consoles* are lightweight managers that administer the IDA, allowing users to remotely change its configuration, as well as view alerts and statistical information about the network. They could possibly be located anywhere on the Internet (with proper HTTP-Proxy configuration). *Central stations* are an offshoot of the need for distributed deployment, 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 IDAs and present a single interface to specify directives to multiple IDAs.

While NFR could be deployed in several configurations (switched, bridged, stand-alone, multiple to name a few), we deployed it in a stand-alone configuration (single IDA) with multiple administrative consoles.

The strength of NFR is in its pre-bundled N-code modules that are very extensive and contain signatures for almost all known popular attacks. It uses a clever architecture that makes it efficient and responsive. These were the main reasons for our choice of NFR as the knowledge-based component of Argus. However NFR alone is not a panacea. We have identified several inadequacies, which we have tried to address through Argus. Most of these inadequacies also apply to knowledge-based IDS'es in general.

1. There is no built-in mechanism that automatically generates rules, either for new attacks or for more complicated attacks. NFR is thus, fairly static and lacks adaptability. The only mechanism for modifying existing rules or adding new rules is to manually generate N-code signatures and manually deploy this code at the IDA through the administrative console. This manual coding of the signatures of new attacks would inevitably be a tiresome effort, but the inability to

detect hitherto unknown attacks is a still more crucial issue and can only be solved by continuous painful vigilance on part of the system administrator/NFR's human operator towards new attacks that sprout up.
2. Knowledge-based intrusion detection systems function using thresholds for acceptable limits of various system and network parameters. Since these values are also set manually, changing the sensitivity of the IDS according to the currently perceived threat-level automatically is not possible. A continuously learning rule generator that drives the analyzers and a managing component that correlates alert information from distributed detection agents, and in turn changes their sensitivity would do away with the above two inadequacies.
3. The only response to attack detection in NFR is to trigger real-time alerts on the administrative console or e-mail the alert information 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. Signature matching at the various IDAs 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 a target inside the network that would escape the isolated IDAs installed on each of the entry points.

Even if a rule that matches an attack exists, a distributed execution of individual stages of the attack might pass undetected. For example, 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 should cause 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 one of its IDAs seeing a change in a `.rhosts` file. Further utilization of this exploit cannot be detected by NFR. A distributed architecture that allows for analyzers to exchange information about suspicious events would solve the problem to a large extent. Argus uses a mechanism that involves storing of such information at the manager. Receipt of subsequent warnings are matched onto earlier relevant warnings that might together constitute a possible attack.

B. Design of Argus

Figure 1 shows the overall architecture of Argus. 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 is NFR, though it could be any knowledge-based IDS. This architecture attempts to 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. Details of various components and specific choices made in the prototype implementation are described in Section III.

Argus employs an agent-based architecture with the low-level agents having sufficient complexity and strength so that the architecture is truly distributed, and not pseudo-distributive where all the computations are effectively done by the central servers. Hence there is a fairly uniform load sharing. 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 representing a *normal profile* but also generate feedback for knowledge-based components, in the form of rules that encode signatures of new attacks. 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.

The lowest level agents in the 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 and `libpcap`. This output is condensed into `connection records` that contain values of the necessary connection features. Feature selection is a crucial aspect and our methodology is explained in Section III.

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. For 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 analyzer components in a knowledge-based intrusion detection system, like NFR i.e. NFR could be a unit that comprises of both a misuse-detector and the corresponding data-cleanser. Such off-the-shelf components need to be encapsulated with lightweight drivers that interface these systems with Argus. Details about the driver construction for NFR are described in Section III. The interaction between the agents and the manager occurs through *IDXP* (Intrusion Detection Exchange Protocol). These misuse detection agents, or in our case, drivers for third-party misuse detection systems, need to have translators that translate the rules that are output from learning engines into filters in the signature specification language (N-Code for NFR).

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 [4]). Learning agents are data-mining components that are trained on the network data. These classification algorithms take as input normal network traffic data and generate the set of rules characterizing the normal activity. These rules are then passed to the anomaly agents which apply them on 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, generated using sufficient amount of 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 (connection records) of the alert is automatically sent to the learning agent that uses it to generate new rules for the knowledge-based component, characterizing this new found attack. The new rules are then distributed to the misuse-detection agents through the manager using the communication framework of IDXP. This provides for adaptability in the system.

The learning agents also regularly get normal network traffic data 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 new rule generation 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, with more importance to more recent rule set.

Argus also supports dynamic deployment of agents and load balancing proportional to the threat perception, as envisaged in [11].

The next higher level in this hierarchy of distributed agents, could be managers or aggregating analyzers. These agents are 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 warnings from the individual systems are stored with these agents and new warnings are used to identify similar previous warnings. Further this level could be split up into a multilevel heirarchy of aggregators as exists in the system described in [11]. Interaction between the managers, higher-level analyzers and the rest of the system is through IDXP.

III. Argus Innards

We now describe the details of the various components of Argus, along with the specifics used in the prototype system that we have implemented.

A. *Intrusion Detection Exchange Protocol (IDXP)*

IDXP (earlier Intrusion Alert Protocol, or IAP) is a protocol designed by the Intrusion Detection Working group of the Internet Engineering Task Force (IETF). It is described in the internet draft [9]. As the name suggests, it is the result of an effort to provide a standard means of communication among heterogeneous agents and managers that form an IDS, and among multiple IDS'es over typical Internet deployment scenarios, wherein either the manager or the analyzer could be inside protected networks that prohibit in-coming connections. At least one of them has to be on a network that allows in-coming connections directed to its host (alternatively a gateway might accept connections on behalf of an enclosed node). Our decision to use IDXP 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.

What IDXP provides: IDXP is a pseudo-HTTP application layer protocol that uses Transport Layer Security (TLS) [6] 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, called Intrusion Detection Exchange Format[5]. This data could be alert information sent by analyzers to the managers, and occasionally updates sent by the managers to the analyzers.

What IDXP does not provide: IDXP 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 IDXP connection allows for only one-way communication. Also, in the presence of a firewall, it is quite possible that only the analyzers might be able to open connections with the hosts outside the firewall (which include the hosts running managers). These issues add further complexity to the implementation. Our prototype handles these and other issues as described next. We also describe the inferences we have made that might be benefitial to anyone interested in an implementation of IDXP, including the creation of an entity called the IDXP daemon that is geared to conserve network bandwidth.

A common observation is that in web-servers, regardless of the request-load, network bandwidth usually proves to be the bottleneck resource. Whichever side of the alert hierarchy, Argus Managers or Argus Analyzers, acts as the server for an IDXP Connection (sender of alert/update data, not the entity receiving connection request) would experience load similar to a web-server. Further, we assume that in general more than one manager/analyzer agent could be running on the same host. This would be the case when there is a particular high-end system that performs most of the analysis, or if several managers are located on the same third-party managing-service-provider. Thus we relegate both managers and analyzers to a subordinate role and instead run IDXP daemons on each host that runs an Argus entity. This is also needed for location discovery of various Argus entities. Other possibilities include an application listening on a standard port that reponds to queries about location of local agents, and pre-configured information about ports of individual agents stored with all counterparts that would need to communicate with them. These other possibilities were rather inflexible, and our choice had the advantage that data that is to be sent to multiple agents on the same host will now need to be sent to the daemon once (with proper options set in the message). The protocol is symmetric, hence the daemons on manager and sensor/analyzer ends are identical. All the entities in our prototype were multithreaded for efficiency.

Further, the connection between the agent and the manager might go through one or more proxies, i.e. the connection initiator may be behind a firewall that prohibits direct connections to external hosts. Thus our prototype uses an application-level proxy called IDXP Proxy. Each agent is presented upon initialization with a proxy-url (proxy ip and port) and a string of no-proxy suffixes/prefixes/exact-matches, which lists the hosts to which direct connection can be established. This part of the functionality of the agent (proxy-interaction) is similar to a typical Web browser. The details of the individual packet headers and the sequence of packets are in [9].

IDXP uses Transport Layer Security [6] to establish secure connections for data transfer, and also requires each communicating entity to have an X.509 certificate and a public key-private key pair. We use the OpenSSL[17] library for these tasks.

Each Argus entity has an IDXP Agent component (implemented as a thread in our prototype) that interfaces the agent with the local IDXP daemon. In all the communication, an agent is identified using the host IP and a 16-bit local-agent identifier unique on a given host. Data can be sent to remote agents in two ways: *actively* if a connection to the remote agent can be initiated, or *passively* if that is not possible. In the latter case the messages are sent to the local IDXP daemon for safe-keeping until the remote agent asks the daemon for any outstanding messages for it. Receiving data is also done similarly in two ways: making connections to remote IDXP daemons for outstanding messages, or getting messages from the local IDXP Daemon. Without going into the details, which would be too many to mention here, we would like to mention that the IDXP daemon based IDXP framework helped solve many implementation issues in our prototype. Some details are provided in [24].

B. Learning Agents

The Leraining Agents use a state-of-the-art rule based learning algorithm called RIPPER [4]. We also considered C4.5rules [20] as an alternative, but found our implementation of RIPPER to be faster. The rule generation system works in two modes. In the first mode, it takes as input the connection records of the normal network traffic, and 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 takes as input the connection records of the normal network traffic along with tagged anomalous connection records, and generates rules that constitute the signature for the anomaly. These rules are used to generate new N-Code filters for NFR.

B.1 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.

Each connection record is transformed into a set of attributes. The attributes are either discrete or continuous. 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.

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 used the FOILv6 [21] tool for the rule growing stage in the algorithm. For a more detailed description of our specialization of RIPPER, refer to [24].

B.2 Rules for Normal Profile

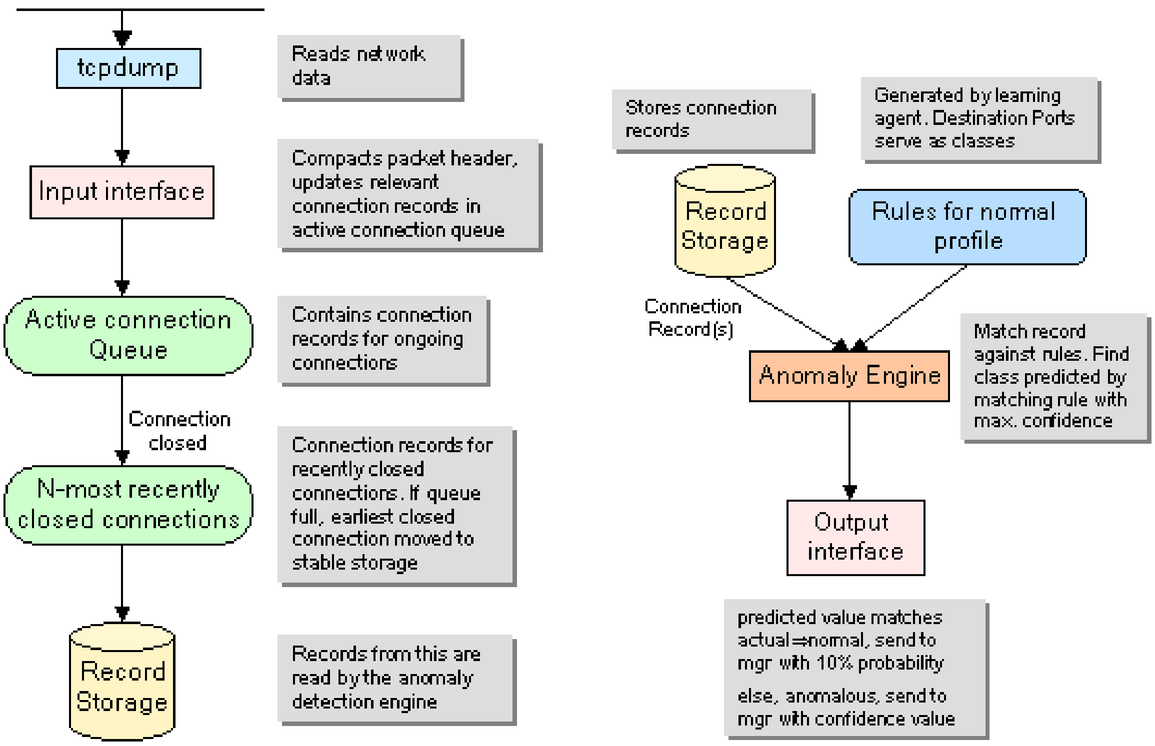
As described earlier, each connection record is suitably transformed into a set of attributes for the learning and classification algorithms to work on them. For the mode when the rule learning system has to generate rules characterizing the normal profile, appropriate class and attribute structure is required with the example data set comprising entirely of normal TCP traffic. The set of attributes are the various fields of the connection records except from the service (destination port). The destination ports form the classes, with 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 class (destination port). A confidence factor based on the number of test cases handled correctly by a rule is associated with each rule. These rules are then used by the Anomaly Detection Engines. This confidence factor is used to arbitrate when more than one rules match a given connection record.

B.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 examples of normal traffic labeled as belonging to one class, say *normal*, and example(s) of anomalous traffic labeled as belonging to the other class, say *anomalous*. The rule set generated act as a signature for the anomalous activity and is used to generate new N-code filters for NFR.

C. The Data Cleansers

Data cleansers, shown in Figure 2(a), are the low level agents that are responsible for the construction of TCP connection records. Argus' Anomaly agents currently handle attacks that use TCP at the transport layer. We believe that very similar extensions could be incorporated to handle attacks at the level of network protocols and other transport layer protocols and that most modern attacks use TCP at the transport



(a) Working of the Data Cleanser

(b) Working of the Anomaly Detector

Fig. 2.

layer. We first describe the features we selected to adequately represent TCP connection information and then describe a tcpdump based implementation that constructs connection records, consisting of the desired features, from tcpdump output.

C.1 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

$$\boxed{\textit{if } (F \geq 5) \textit{ then SYN\_FLOOD}}.$$

It is usually a pretty difficult task to identify the correct set of features, especially features that incorporate temporal and statistical information such as data traffic, in bytes, on the user-ports in the last n seconds, number of un-wanted syn-acks received by httpd in the last n seconds etc. The following is a description of the set of features that we found to be effective after experimentation.

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 pkts 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 ACKs 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 a TCP connection is usually very specific to the type of application that runs on the specific standard port.
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 stray errors might be neglected but many errors within a short span of time are suggestive of something fishy going on in the network.
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 installed 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. 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 motivated by experiments. Refinement of these features would require greater insights into individual attacks, but the idea behind statistical anomaly detection is to do away with unnecessary or exorbitant human effort. So an intermediate course must be struck. While, it is our firm belief that the above set of features is not the complete set of "sound" features, experiments show that they perform "fairly well".

C.2 Generation of Connection Records

The present impelementation compacts tcp packet header information generated by tcpdump executing in the mode wherein relative sequence numbers are suppressed.

Each line of the tcpdump is parsed for the required values. A map of connections that are active at any point of time are maintained by simultaneously updating the state of both ends of the TCP connection whenever a packet belonging to this connection is seen by tcpdump. The required values are stored for each active connection.

When a connection is closed, either successfully or erroneously(timeout, RST), the connection record is timestamped and transferred from the active queue to one which stores connections completed in the last 'n' seconds.

Whenever 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 engines as shown in Figure 1.

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 many 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 simplistic solution was devised that discards packets belonging to connections for whom state is not already being maintained whenever the buffer usage *approaches* the maximum available. The connection key of such an "ignored" connection, i.e. the four tuple of connections whose packets have been discarded, is stored and all later packets of these connections are discarded. As this is a "droptail" approach, the percentage of new connections that are discarded is directly proportional to

$$\boxed{\frac{\text{network load}}{\text{available buffer size}}.}$$

D. Driver for NFR

We use an off-the-shelf knowledge based system NFR [15] 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.

While NFR's source code was available to the research community until a few years back, it is no longer available. Hence we faced the following problems:
1. Communication between the IDA and the Administrative Console(Manager in NFR) is encrypted. This encryption mechanism was not known.
2. Message Exchange Protocol for communicating with the IDA was not available.
We instead used Perl libraries that were distributed with NFR and consisted of functions (in binary) that perform the required encryption/decryption and message exchange. The IDA communication code in Perl was embedded into the C/C++ system of Argus using the Perl add-on module ExtUtils::Embed. This module allows function calls to native Perl methods from within a C/C++ program as a Perl interpreter is initialized prior to the start of the program. Our final implementation of the NFR diver uses only the relevant PERL functions that encrypt/decrypt and has most of its functionality implemented in C++.

The above-mentioned scripts only allowed for NFR's alerts to be relayed to external agents. NFR also provides for a GUI-based, remote deployment of new attack signatures to the IDA but this could not be automated as the requisite libraries were not available. Thus new attack signatures encoded in N-Code are currently stored on the NFR driver and need to be manually inserted into NFR through the GUI console. We validate the correctness/efficiency of the rule generator by manually installing the generated signatures during experimentation.

E. Anomaly Detection Agents

The Anomaly detection agents, an example of which is shown in Figure 2(b), are the application of the rules produced by the RIPPER based rule generation system implemented as a part of the learning agents. 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 an encoded (and compressed) rule set from the manager and store this in the form of a file. A configuration file describing the class and attribute structure that is shared between the anomaly agent and the learning agent is used to decode and encode the rule set respectivley. 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.

F. Managers

This component of Argus (Figure 3) embodies most of our novel ideas. The various execution agents be they anomaly-based or misuse-based send alert information to the manager. The manager is also initialized like an ordinary IDXP agent but with slight differences and is multi-threaded. Additionally, the manager stores a database mapping an intrusion type to the set of attack response handlers to be executed whenever an instance of this intrusion is observed.

The uniqueness of Argus' manager lies in the type of data that is reported to the manager and in the handling of this data.

The following cases are handled:
1. Receive registration request from remote agents: The internal data structures are updated and fresh rule sets are passed out in case of anomaly agents.
2. Alert information received from NFR: An alert informing the detection of this attack is dispatched. This could take the form of a graphical pop-up display, or an e-mail alert. The corresponding attack response handler is identified based on the type of the intrusion detected. The alert information is stored in a system-wide anomaly queue, appropriately tagged with the source of detection in order to aid in correlation with later attacks.
3. Alert information from the anomaly based agents is received: The alert information is stored in the anomaly queue along with the associated confidence value.
4. Rules received from the learning agent: If these rules are for the anomaly based agents, they are passed onto the anomaly agent in verbatim, along with a directive requesting the agent to update its rule set. The anomaly agent updates its effective rule set upon receipt of such a directive from the manager. If the rules are for NFR, the manager translates between RIPPER's clause based rule structure to equivalent N-Code filters and passes on the N-code filter to the NFR driver. Generation of N-code from the rules of the learning agent assumes that a one-to-one map exists between the features in the connection records and features that could be identified by NFR. If this were not the case, then the base filter set of NFR needs to be augmented appropriately.

The following is the functionality of the tagger component of the manager that reads input into the anomaly queue. If
1. The source of the alert is NFR: Alert data is matches with pre-existing information to detect the possibility of a distributed attack or a continuation of an earlier exploit. If such is the case, the relevant alert is triggered and the corresponding response handler is called.
2. The source of the alert is anomaly-based agent: The system administrator is asked to validate the alert information and is presented with a detailed trace and the confidence level of the mis-match. If the administrator tags it as an alert, this tagged connection record is passed to the learning agent which would generate rules for the signature of this attack for the knowledge-based agent. Notice that this is an instance of a previously unknown attack (or variant) being detected. The corresponding signature is automatically generated and deployed into the database of known signatures. The only supervision required is to validate and tag the alert information. Tagged attacks are also matched with pre-existing alert information to detect distributed/correlating attacks, and the information is stored for matching with later attacks. If the alert is tagged as an error, the counter of false alarms is updated. The rate of false alarms currently observed in the system is used as a parameter to decide the amount of "normal" data that needs to be passed onto the learning agent, the frequency with which the learning agent needs to re-generate rules for normal data and the frequency with which the anomaly based systems need to age their rules. For instance, all of the above mentioned factors are directly proportional to the false alarm rate, with appropriate damping factors. Thus the anomaly agents update their rule sets and age the older rule sets with a greater frequency when they perceive a

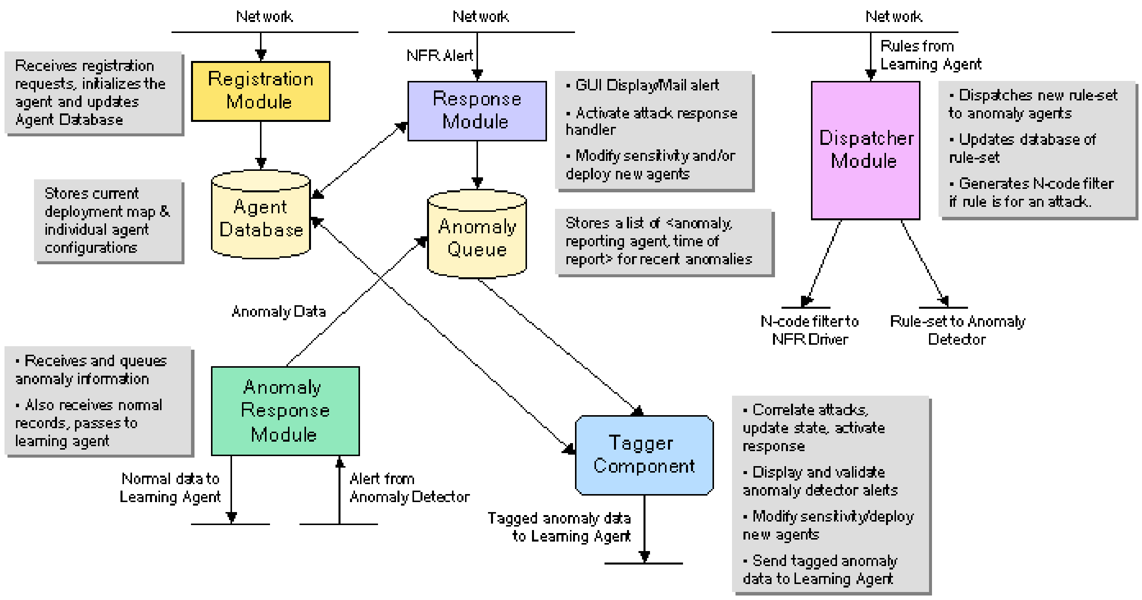


Fig. 3. Working of the Manager

higher false alarm rate.

The attack response handlers that are called by the tagger component are defined specific to the type of the intrusion and involve actions in specific response to the attack. For instance, a response handler might necessitate the blocking of an external IP address at the firewall when a SYN_FLOOD attack is detected with the source at that IP address. Further these handlers could also re-assign values of relevant thresholds in the analyzers. For instance, Detection of several intrusions in NFR is threshold-based, relevant threshold values may be changed by sending appropriate directives to the NFR driver. For the anomaly-based systems, one relevant threshold is the confidence associated with a rule, which could be suitably changed based on the success of the result obtained by an application of the rule.

IV. Experimental Results

We successfully built and tested a prototype system with the implementation choices mentioned in Section III. We now describe our experimental setup, experiences with Argus and some results obtained.

A. Guarding against SAINT

SAINT[22] is a network vulnerability analysis tool for the UNIX platform. When configured with a `distance` in hop count and an integer `probe level`, it checks all possible hosts within `distance` hops away from the host running SAINT for all vulnerabilities that are associated with a probe level equal to or less than the `probe level`. The higher the probe level, the greater is the potential damage that the vulnerability could lead to. Alternatively, explicit lists of hosts that are to be checked for vulnerabilities (or not) are also accepted as input by SAINT. The objective of this experiment was to verify the effectiveness of our anomaly detection engines.

The testbed for this sub-class of experiments consisted of the internal network of the Department of Computer Science at Indian Institute of Technology Kanpur, that consisted of a set of subnets linked through switches. To avoid overloading the network, we had SAINT attempt to perform a vulnerability analysis on a small set of workstations, running one of Linux-2.2, Solaris 8 or Microsoft Windows 2000. NFR's IDA needs to be installed on a dedicated machine and was installed on one of these switched segments, the one that contained the various server machines. One anomaly agent was initialized in each switched segment. A learning agent and a manager were initialized on high-end Linux machines. Finally, a set of dormant anomaly agents were spread at random throughout the network, each consisting of a daemon waiting for the "deployment directive" from the manager. We believe that this is a typical installation of Argus on the internal network of a reasonably large or-

Confidence Threshold	0.4	0.6	0.7	0.8	0.9
Attacks Detected%	98	98	97	95	78
False Alarm%	2.1	1.26	0.73	0.50	0.01

Fig. 4. Anomaly Detection Agents: Results

ganization.

Since anomaly detection agents in our prototype implementation were only TCP aware, we compared the number of distinct attacks identified by the anomaly detection engines with the number of TCP based attacks that SAINT employs. Some of these attacks are TCP Syn based port scanners and TCP SYN Flood attacks. For a complete list of such attacks, refer to [22]. As mentioned in Section III, anomaly detection engines apply rules to connection records and identify mis-matches. Corresponding to each mismatch the confidence estimator of the rule that generated it, is attached. An anomaly detection engine alerts the manager of a mis-match only if the confidence value associated with the mismatch is greater than the threshold. By modifying the value of this threshold, we could exercise a fine degree of control on the percentage of attacks observed versus the number of false alarms. When a reported mis-match is tagged to be "normal" at the manager, the associated confidence value associated with this rule is decremented by a small value. Further, the more the number of false alarms the faster is the rate of updation and installation of new profiles, and aging of existing profiles. The normal profiles were trained by having the anomaly detection engines run for a few minutes before starting SAINT. Figure 4 shows the variation of percentage of TCP based attacks detected by the anomaly detectors and the percentage of alerts that were false-alarms with the sensitivity of the anomaly detection engines.

At low thresholds, the false alarm rate is considerably high and the percentage of attacks detected is also very high. Increase in the threshold decreases both these rates. However the system becomes unstable at very high confidence thresholds because most of the rules generated by RIPPER have confidence values in the 40-90% range. Thus as seen in the table, we empirically observe that the optimal performance occurs at a confidence level of 0.8. At this level, the anomaly detection engines detect nearly 95% of the attacks with as low as 0.5% false alarms. One should note these values are rather domain-specific. For this reason, we did not try to optimize the values, and just try to present a trend.

B. The Case of The Missing Signatures!

The objective of this experiment was to demonstrate the ability of Argus to automatically generate signatures for hitherto unseen attacks. The alerts generated by various agents (anomaly detectors as well as misuse detectors) are stored and temporally correlated by the manager(s). Anomalies generated in the temporal vicinity of an alert from a misuse detector are assumed to be referring to the same event. This is a first-cut approximation because we need a greater amount of shared knowledge between the misuse agent and the anomaly agent in order to filter out anomalies that are already captured by attack signatures.

For the purpose of this experiment, we wrote an application that performs distributed TCP SYN flood attack on a host. This host was on the switched segment on which both NFR and an anomaly agent were present. We removed the N-code filter corresponding to this attack from NFR. Thus no misuse agent alerts were generated while anomaly alerts were generated due to the presence of a rule (for the normal profile) that bounds the number of half-SYNs present in the network. These alerts were hence classified as corresponding to a hitherto unseen attack and were forwarded to the learning agent for signature generation. Upon receipt of the ruleset characterizing the attack from the learning agent, the N-code translator in the manager generated an N-code signature. This signature was sent to the NFR driver where it was saved for deployment. It was manually deployed into NFR at a later time. Continuing the SYN Flood attack after deployment elicited alerts from NFR, though the name of the attack was not SYN Flood, since the tagger did not tag it with that name.

Hence we were able to dynamically and automatically generate a signature for a "new" attack. Manually detecting this attack and coding an N-code filter for it would have been a very cumbersome task which was greatly automated and simplified by Argus.

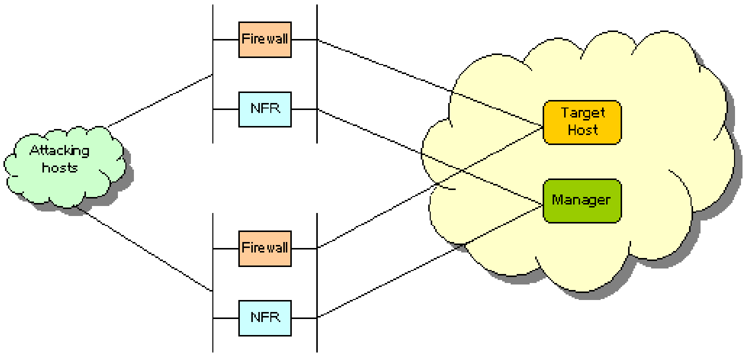


Fig. 5. Testbed for Attack Correlation

C. Attack Correlation

The objective of this experiment was to demonstrate the ability of Argus to correlate the alert information reported by different analyzing agents, and consequently detect distributed attacks.

The testbed for this subclass of experiments is shown in Figure 5. It consisted of a network with multiple entry points and NFR IDAs installed on the DMZ at each point of entry. Both the target of the attack and the Argus manager were present on hosts inside the network.

NFR's thresholds for various attacks are remotely configurable. These can be adjusted to change NFR's sensitivity. We did a distributed SYN Flood attack on the target host from a set of source hosts on the outside. The rates of the half-SYNs flowing through each entry point was lower than NFR's threshold. Hence no inidividual NFR IDA generated an alert, but simply "warnings". These warnings were correlated by the manager, as they were clustered together (they were warnings of the same kind and they targeted the same host on the internal network). This caused the manager to generate a correlated alert that characterizes the distributed attack.

D. Real-time Attack Response

The testbed for this experiment was the same as the one for the Attack Correlation experiment described above. In addition, we had a firewall at each gateway, and an *attack response handler* was initialized at the manager for response to the SYN Flood attack. This handler updates the filtering rules at each of the firewalls so as to block all traffic from the attack sources, whose IP address figured in the warnings.

When the manager detected a distributed SYN Flood attack, it executed the attack response handler for this attack. The handler caused each firewall to block traffic from the suspicious hosts, stopping the SYN Flood attack.

E. Robustness of The Architecture and Sample Rules

In our prototype implementation, each host running one or more Argus components has an Argus daemon that receives and sends information on behalf of the component. Each Argus component has a unique component-id (CID) and public, private key pairs which are used to communicate securely with other components. All communication is addressed to and from CIDs and a global system-wide map of CID to IP address is maintained within reach of each daemon. This shared-daemon implementation is light-weight and having communication addressed to the components allows the components to move seamlessly throughout the network. Each agent could either establish active connections to the remote daemon (to send or receive data), or passively wait for the remote agents to contact this agent by leaving messages at the local daemon. This allows us to have dormant daemons spread throughout the network, allowing us to quickly grab more samples from the network if necessary, especially to start agents that are previously unknown to an attacker or a malicious insider. The robustness of the architecture was tested by simulating faults in the counterpart agents when agent communication was underway. Of special importance, was the need to keep the manager tolerant to malicious packets directed at it.

Some of the sample rules generated by the learning agent for normal traffic are shown in Figure 6.

R1: if cls=13 **and** bytesF$\leq$233 **and** dataP>29 **then** class=23
R2: if cls=13 **and** bytesF>233 **and** dataP$\leq$14 **then** class=3128

Fig. 6. Sample rules for normal profile

Here *cls* denotes the how the connection was closed and the value 13 denotes that the TCP connection was closed correctly. *bytesF* are the number of bytes that were sent in the forward direction i.e., from the sender of the first SYN of the connection to the receiver. In case of symmetrical initiation of TCP connections, the tie is broken arbitrarily in the order in which tcpdump observes these events. *dataP* is the number of data packets that are sent in one normal connection. Similarly *contrP* is the number of control packets in a connection. Finally the *class* of a connection is the destination port (service type) of the first SYN packet of the connection. Thus, the above three rules can be paraphrased as follows:
R1: A connection that had very few data bytes transmitted in the forward direction, a large number of data packets (each containing little data) and was closed normally, most likely is a telnet connection.
R2: A connection that had few data packets, but involved a large amount of data transfer within these data packets is most likely a HTTP connection (Squid proxy server running on Port:3128). Why? A normalized HTTP transfer is about 5kB as a new HTTP connection is initiated for each request.

Note that a confidence measure is also associated with each rule but was ignored above for the sake of clarity.

V. Related Work

Network intrusion detection has been a research problem for some time now and several experimental and commercial solutions have been proposed.

Helmer et. al. [11] proposes an artificially intelligent (AI) agent architecture for detecting intrusions. This work is similar to Argus in the common design goal of light-weight distributed agents. However, Argus differs in both the number and variety of agents and leverages IDXP in its communication framework. While Argus has two sets of data collecting and analyzing agents, [11] proposes a plethora of agents dedicated to analyzing specific sources of information. Argus uses unique agents called Managers to provide attack response, distributed attack correlation and updation of both signatures and profiles.

Cannady [3] proposes a neural network based approach to detect spurious activity. Our approach is complementary to this in that we use machine learning algorithms in our Anomaly/Learning agents. Further the extensibility of Argus, allows us to interface with such neural networks based agents through IDXP.

Lee et. al. [13], [12] devise automatic methods for constructing intrusion detection models. This work is contemporary to Argus and is closest to it in that the same machine learning algorithm, RIPPER is deployed. However the main focus of Argus is in the design and implementation of an efficient distributed agent architecture and communication framework, while [12] focuses primarily on improved data-mining based methods to automatically generate new attack signatures. Unlike [12], Argus also emphasizes correlation of attacks and timely attack response.

Vigna et. al. [25] proposes a state transition network based approach to detect network intrusions. The idea is to have a "State transition database" that specifies a set of actions corresponding to each intrusion scenario. The "actions" specify the data to be collected or analyzed is at the current state before conditionally proceeding to a successor state in the state transition diagram. At any point of time whether an intrusion has been detected is an attribute of the current "state" in which the system lies. This method allows greater expressiveness in encoding attack signatures and is easy to use, however, it is primarily a knowledge-based approach that relies on a known set of attack signatures and hence lacks adaptability.

NFR[15], Emerald[7] and Bro[19] are other expert systems that, primarily, perform knowledge based intrusion detection. Argus shares the same design goals of real-time attack response and extensibility as these systems but also exploits the complementary advantages of anomaly based agents and data-mining algorithms, namely greater sensitivity to newer attacks and automatic construction of complicated signatures.

Pal et. al. [18] is representative of a different class of solutions that argue the case of intrusion tolerance, namely the need for building systems that can tolerate and survive malicious attacks. Specifically, this paper proposes a policy based approach that allows applications to execute correctly even in corrupt environments and proposes several obstacles in the path of an attacker that significantly decrease the amount of damage that can be performed immediately after a successful intrusion. This proposal is complementary to Argus.

Foster et. al. [10] presents a security architecture for grid computing that some contend would be the way of the future. Grid computing envisages highly distributed deployment of resources which could be accessed and used from anywhere else in the grid. A greater onus on network security exists in a grid-like environment as more sensitive information wades through the network. We contend that Argus is uniquely suited for this purpose when deployed as follows: the Grid would be divided into multiple administrative divisions which would be policed by one or more managers. Inter-manager communication would be acheived through the standard Intrusion Detection Message Exchange Format[5]. Alerts or Intrusion Detection Messages that are created by analyzing agents would be limited to the adminstrative domain of occurrence, thereby ensuring scalability. The architecture of Argus also provides for customization, in that the "profiles" of normal activity would vary with the adminstrative domain. Profile- Customization decreases the number of false positives considerably and thereby decreases the network traffic.

VI. Conclusion and future work

The construction of Argus demonstrates that a mix of anomaly detection agents and knowledge based agents performs better than an architecture consisting of only one variety of agents. To the best of our knowledge Argus is the first intrusion detection system that conforms to the emerging IDXP standard and uses a mixture of agents. We believe that intrusion detection systems of the future would comprise a secure, peer-to-peer infrastructure of heterogenous independently administered systems that communicate through IDXP and exchange alert information conforming to the Intrusion Detection Message Exchange Format Data Model [5]. While sophisticated languages such as N-Code, p-BEST that encode attack signatures exist, there exists little support for encoding attack responses. We believe that the intrusion detection systems of the future would also be equipped with real-time response capabilities and there is much to do before response-specification languages of an equivalent sophistication become available. Argus uses shell/perl/TcL scripts for this purpose. Of great importance is the integration of existing network control methods such as SNMP, within the attack response specification primitives. Intrusion Detection systems should also make better use of the available ICMP information.

Acknowledgments

We are very grateful to Sumit Ganguly at Indian Institute of Technology Kanpur for his helpful advice and positive criticism.

References

[1] M. Almgren and U. Lindqvist, *Application-Integrated Data Collection for Security Monitoring*, From *Recent Advances in Intrusion Detection (RAID 2001)*. Springer, Davis, California. October, 2001. Pages 22-36.

[2] M. Bernaschi, E. Gabrielli and L. V. Mancini, *Operating System Enhancements to Prevent the Misuse of System Calls*, ACM 7th CCS, 2000.

[3] J. Cannady, *Artificial Neural Networks for Misuse Detection*, 21st NISSC, 1998.

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

[5] D. Curry and H. Debar, *Intrusion Detection Message Exchange Format Data Model and Extensible Markup Language (XML) Document Type Definition*, IETF, Intrusion Detection Working Group, Internet-Draft 2001.

[6] T. Dierks and C. Allen, *The TLS Protocol, Version 1.0*, Network Working Group, RFC-2246, Jan 1999.

[7] Event Monitoring Enabling Responses to Anomalous Live Disturbances (EMERALD), Security Research International. *http://www.sdl.sri.com/projects/emerald/*

[8] S.T. Eckmann, G. Vigna, and R.A. Kemmerer, *STATL: An Attack Language for State-based Intrusion Detection* Journal of Computer Security, 2001.

[9] B. Feinstein, G. Matthews and J. White *The Intrusion Detection Exchange Protocol (IDXP)*, IETF, Intrusion Detection Working Group, Internet-Draft 2002.

[10] I. Foster, C. Kesselman, G. Tsudik and S. Tuecke, *A Security Architecture for Computational Grids*. In *Proceedings of 5th ACM Conference on Computer and Communications Security Conference, pp. 83-92*, 1998.

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

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

[13] W. Lee, S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop and J. Zhang, *Real Time Data Mining-based Intrusion Detection*, In *Proceedings of DISCEX II*. June 2001.

[14] U. Lindqvist and P. A. Porras, *eXpert-BSM: A Host-based Intrusion Detection Solution for Sun Solaris*, From *Proceedings of the 17th Annual Computer Security Applications Conference (ACSAC 2001)*. IEEE

Computer Society, New Orleans, Louisiana. December 10-14, 2001. Pages 240-251.

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

[16] Next-Generation Intrusion Detection Expert System(NIDES), Security Research International. *http://www.sdl.sri.com/projects/nides/*

[17] The OpenSSL Project *http://www.openssl.org*

[18] P. Pal , F. Webber, R. E. Schantz, J. P. Loyall, R. Watro, W. Sanders, M. Cukier and J. Gossett, *Survival by Defense-Enabling*, In *Proceedings of the New Security Paradigms Workshop (NSPW 2001)*, Cloudcroft, NM, September 2001.

[19] V. Paxson, *Bro: A System for Detecting Network Intruders in Real-Time*, In *Proceedings of the 7th USENIX Security Symposium*, San Antonio, TX, January 1998.

[20] J. R. Quinlan *C4.5: programs for machine learning*, Morgan Kaufmann, 1994.

[21] J. R. Quinlan and R M. Cameron-Jones, *FOIL: A midterm report*, In *Machine Learning: ECML-93*, Vienna, Austria, 1993. Springler-Verlag. Lecture notes in Computer Science # 667.

[22] SAINT Vulnerability Analysis Tool, *http://www.wwdsi.com/saint/*

[23] Security Administrator Tool for Analyzing Networks, *http://www.fish.com/satan/*

[24] S. Singh and S. Kandula, *Argus - A distributed network intrusion-detection system*, B. Tech Thesis, Department of Computer Science and Engineering, Indian Institute of Technology Kanpur, India. May 2001.

[25] G. Vigna and R. Kemmerer, *NetSTAT: A Network-based Intrusion Detection Approach*, in *Proceedings of the 14th Annual Computer Security Application Conference*, Scottsdale, Arizona, December 1998.