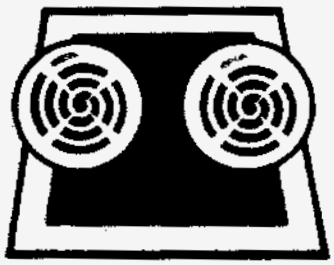
LA-SUB--93-211

DEPARTMENT OF COMPUTER SCIENCE



COLLEGE OF ENGINEERING
UNIVERSITY OF NEW MEXICO

A prototype implementation of a network-level intrusion detection system

by

Richard Heady, George F. Luger, Arthur B. Maccabe,
Mark Servilla, and John Sturtevant

Department of Computer Science
The University of New Mexico
Albuquerque, New Mexico 87131-1386

Technical Report No. CS91-11

DISCLAIMER

A Prototype Implementation of a Network Level Intrusion Detection System*

Richard Heady
George Luger
Arthur Maccabe
Mark Servilla
John Sturtevant

Department of Computer Science
University of New Mexico
Albuquerque, NM 87131

May 15, 1991

Abstract

This paper presents the implementation of a prototype network level intrusion detection system. The prototype system monitors base level information in network packets (source, destination, packet size, time, and network protocol), learning the 'normal' patterns and announcing anomalies as they occur. The goal of this research is to determine the applicability of current intrusion detection technology to the detection of network level intrusions. In particular, we are investigating the possibility of using this technology to detect and react to worm programs.

1 Introduction

Three aspects of network/distributed systems make these systems more vulnerable to attack than independent machines: 1) networks typically provide more resources than independent machines, 2) network systems are typically configured to facilitate resource sharing, and 3) global protection policies which are applied to all of the machines in a network are rare.

Others researchers, such as Lunt [11, 10] and Vaccaro [14], have applied intrusion detection techniques at the machine level of a computer system. In

*This work was supported in part by the Office of Safeguards and Security of the US Dep. of Energy through the Safeguards Systems Group (N-4) of Los Alamos National Laboratory.

contrast, the research project described in this series of reports is aimed at investigating the applicability of intrusion detection techniques to detect network level intrusions. In particular, we are investigating the possibility of developing a system which can detect and react to worm programs. A "worm" program is characterized by the fact that the program moves from one node in a network to another. The Internet worm of November 1988 [12] provided ample demonstration of the fact that computer networks are susceptible to this type of attack.

Our strategy is simple. First, our network security system will learn normal network behavior and store this information in a profile data base. By letting the system learn the local network behavior, we avoid the risk of leading the network security system astray by imposing on it a preconceived notion of intrusive behavior. Second, after the network security system builds a representative profile of the local network the system is allowed to produce a classifying rule set that characterizes the network. This rule set is eventually compiled into an executable image. Finally, the compiled version of the network security system is run in a real-time network environment. If the network behavioral patterns change with time our network security system only requires a period of relearning the new behavior before it is brought back on-line. As such, our network security system is ideal for intrusion detection on any dynamic network.

The first phase of this research project is described in the technical report *The Architecture of a Network Level Intrusion Detection System* [7]. It demonstrated our method of data collection from an Ethernet local area network and explained the importance of data preprocessing. In addition, it proposed the use of a Learning Classifier system and Genetic Algorithm to learn normal network behavior, and to possibly detect anomalous activity indicative of a network level intrusion.

The second phase of this research project is the subject of this report. Along with an overview of our earlier work, this report provides a detailed description of the Learning Classifier system and Genetic Algorithm and how it is currently implemented using MIT Scheme.

2 Background

Protection encompasses the *integrity*, *confidentiality*, and *availability* of the resources provided by a computing system. Historically, protection has been provided in the context of a security model [9]. Security models are based on the concept of an *action* which is applied to a set of resources (frequently called *objects*). Each action can be attributed to an individual user, the *initiator* of the action. A security model specifies which actions are permitted based on the initiator of the action, the objects involved in the action, and the context in which the action is requested. Importantly, every action performed in the computing system must be validated by an implementation of the security model.

There are at least three ways in which a computing system based on the security model approach can be compromised: an incorrect implementation of the model, an inaccurate authentication of the user, or an insider attack.

Any implementation of a security model is at best an approximation of the model. The more complex the model, the more likely it is that there is a discrepancy between the implementation and the model. Any such discrepancy must be viewed as a means by which the integrity, confidentiality, or availability of a resource could be compromised.

The implementation of a security model incorporates an authentication module which is used to identify the individual initiating actions in the system. At best, the authentication module provides a high level of confidence that the individual initiating an action has been correctly identified. Regardless of its complexity, every authentication module can be compromised. When the authentication module is compromised, i.e., an individual is incorrectly identified, the security model no longer provides protection for the resources of the computing system.

Finally, the security model approach does not address the problems associated with an insider attack. It is possible that an individual who has been granted the right to manipulate an object may abuse that right. This possibility is not addressed in most security models. As such, a privileged individual can compromise the integrity, confidentiality, or availability of the resources which he or she has been authorized to manipulate.

Given these difficulties, several researchers have proposed that the traditional security model be augmented with an intrusion detection system [6, 11, 10, 14]. Any set of actions that attempt to compromise the integrity, confidentiality, or availability of a resource is termed an *intrusion*. An *intruder* is the individual or group of individuals who initiates the actions in the intrusion. Intrusion detection systems are based on the belief that an intrusion will be reflected by a change in the 'normal' patterns of resource usage. As such, intrusion detection systems have been developed to monitor specific types of activities and announce anomalies in the behaviors observed. The anomalies announced by an intrusion detection system serve as an indication that an intrusion may be in progress.

If the intrusion detection system bases its monitoring on the actions performed by an individual (as in the IDES system), the monitoring can be viewed as an on-going authentication process. In this sense, the individual's behavior will continue to authenticate his or her identity as long as those activities are within an acceptable variance of the normal behavior for the individual. However, if the activities performed by an individual differ significantly from their normal profile, then there is reason to suspect that an intrusion has occurred.

Like security models, intrusion detection systems are not immune to attack. Because behaviors change over time, intrusion detection systems must be capable of adapting to reflect changes in the actions that they monitor. As such, a careful intruder can 'teach' the intrusion detection system a new behavior pattern which may culminate in invalid access to resources in the system. In this

context the intrusion detection system serves to increase the time it takes to compromise the resources of the system and may increase the probability that the intruder will give up or be caught by alternate mechanisms.

The research project described in this series of reports represents an attempt to apply the techniques associated with intrusion detection to the network level of a computing system. As with the IDES system [11], our system is based on a statistical characterization of normal behavior. Like Wisdom and Sense [14] our system utilizes a Learning Classifier System (LCS) in conjunction with a Genetic Algorithm (GA) to learn which measures yield the best characterization of normal behavior. In addition, our view of the network data flow coincides closely to that of the network security monitor being developed at the University of California, Davis [8].

3 Overview

The primary assumption of our network security system is that normal network traffic will be characterized by discernible patterns of data flow, and that intrusive behavior will in some way violate those patterns. One approach to designing a network security system is to define network behavior patterns that indicate intrusive or improper use of the network and then look for the occurrence of those patterns. While this may be capable of detecting known varieties of intrusive behavior, it would allow new or undocumented types of attack to go undetected. Furthermore, individual networks may not follow similar patterns of usage. As such, any standardized characterization of a network would be rendered useless.

For these reasons, we have made the decision to build a system which learns normal patterns of network behavior and then detects deviations from those patterns. By employing a learning phase in our network security system, an accurate characterization of the network can be determined regardless of the dynamic nature of network behavior. If the network behavioral patterns change with time our network security system only requires a period of relearning the new behavior before anomalous activity can again be detected. As such, our network security system is ideal for intrusion detection on any dynamic network.

After investigating several models of machine learning we have decided to structure our learning system on the rule-based classifier system and genetic algorithm model developed by John Holland and others [5]. In principle, anomalous events are separated out of a continuous stream of noisy or irrelevant data by a rule-based classifier system. The genetic algorithm effectively narrows the rule space of the learning classifier system.

Conceptually, our network security system is divided into the three modules illustrated in Figure 1. The *network monitoring and preprocessing* module samples the network transmissions to create a valid profile of the network data flow. Currently, this module stores all network information to a data base file. Infor-

mation stored in this profile data base is preprocessed for the learning classifier system. This module will eventually supply the learning classifier system with a continuous stream of network data packets in a real-time environment.

The *learning classifier* module parses the network data packet into discrete pieces of information. Each piece of information affects the rules in the rule-based classifier either directly or indirectly. The end result of the learning classifier system is to learn normal patterns of network traffic and flag deviations from those patterns. In practice, a learning phase must precede the prediction phase of the classifier system.

The *genetic algorithm* module modifies the predictive rules of the learning classifier system to better characterize the network behavioral patterns. During rule evaluation phase, each prediction rule has a strength value that is revised based on its latest prediction of the current state of the network. Using this strength value, the genetic algorithm purges those prediction rules which have a low strength value , and generates new prediction rules from combinations of those rules which have high strength values.

Information flow begins at the monitoring module and continues through to the learning classifier module. After an initial phase of rule evaluation has completed, prediction rule strength information is sent to the genetic algorithm module where genetic operators update the prediction rule set. The new rule set is returned to the learning classifier module where network state classification is resumed. Eventually, as a stable rule set is reached, the genetic algorithm module(3) is removed from the system.

4 Data Sampling

In today's computer networks, data collection may not be a simple task. Data travelling through the cables or airwaves of a network is no more than a stream of bits. Transmitting and receiving stations must somehow piece together each bit to form a coherent message. Many different protocols exist which define how data is to be assembled into a packet of meaningful information. Decoding an information packet requires prior knowledge of the protocol being used on a particular network. When networks do not follow one of the protocols as defined by the International Standards Organization (ISO) decoding the incoming information packet may not be possible.

Furthermore, many computer systems require both specialized software and a priviledged *status* to access the network bit stream. Most often, existing network monitoring software allows network access to some degree, but generally does not provide detailed transport information or allow the collection of packets destined for another machine. In either case, the validity of the network data flow profile would be marginal at best.

Because of the problems involved in data collection, we have constructed our own network monitor. The following section addresses three aspects of network

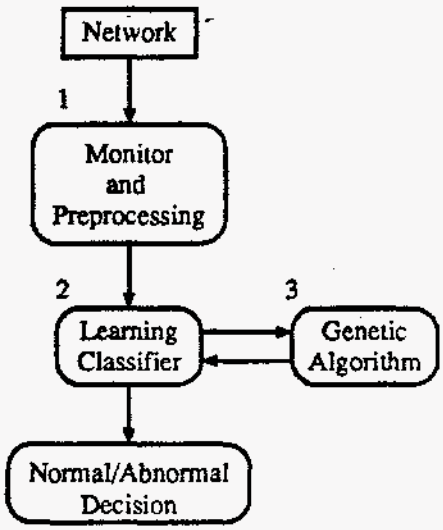


Figure 1: Basic architecture of intrusion detection system.

data collection: 1) the choice of data to be collected, 2) the method of collection, and 3) the preprocessing of data for the learning classifer system.

4.1 A Choice of Data

Since the goal of this project is directed toward intrusion detection at the network level a natural choice of data to collect is the network transmission packet. In particular, we look at the packet generated at the Data Link Control layer of the OSI hierarchy. At this layer, the network packet can be partitioned into *transport information* and *deliverable data* [13]. Transport information generally consists of the source-destination address pair and some type of checksum on which the integrity of the packet is determined. Transport information is added to the packet as defined by the Data Link Control layer protocol and is implemented by hardware in most instances. As such, transport information cannot be directly affected by the user of a network. In other words, transport information is an artifact of the system and not the user. We therefore consider transport information to be *unbiased* data. Unbiased data is simply the information in a network packet that cannot be made deceptive by a fraudulent user.

On the other hand, deliverable data is information which is passed through the higher layers of the OSI hierarchy and generally consists of user information, such as key strokes or large aggregates of text. In general, deliverable data is

encapsulated by various header and trailer information which is required by the OSI layer protocols. Since deliverable data can be accessed either directly or indirectly using the interface software at a specific OSI layer, a fraudulent user can easily modify this information to be deceptive. We therefore consider deliverable data to be *biased* data. It may even be the case that deliverable data is encrypted and cannot be easily interpreted by a monitoring system. For these reasons, we have decided to collect only the transport information part of the network transmission packet and to ignore all other packet information.

4.2 The Network Interface

Since detecting an intrusion is not dependent on the specific method used to monitor packets, any mechanism capable of obtaining a valid data sampling is satisfactory. Currently, we are using a software package that allows monitoring of an Ethernet local area network.

All data collection takes place on two SUN 3 workstations using Lance Ethernet controllers. One workstation receives external network traffic via the Campus Data Communication Network (CDCN). The CDCN is a broadband network and is the backbone along which UNM traffic is handled. The other workstation monitors Ehternet traffic within UNM's Computer Science subnet.

To monitor Ethernet traffic, we use the Network Interface Tap (NIT) facility provided by SUN Microsystems as part of their SUN Operating System network software utilities [1]. At this time, NIT is the only software available on our hardware configurations which allows promiscuous monitoring of the Ethernet network.

4.2.1 Network Interface Tap (NIT)

NIT is a facility composed of several *streams* modules and drivers which provide link-level network access. As such, NIT is capable of both reading from and writing to the Ethernet device. NIT performs this service by placing itself between the Ethernet device and a user process. When NIT is initialized as a reading device, it attempts to read the packets that enter the Ethernet device buffer and write them to a stream. When initialized as a writing device, NIT requires the user process to supply an input stream which is then transmitted out onto the network through the Ethernet device. The components which collectively provide this service are the *interface* (nit_if) [2], *packet filter* (nit_pf) [3], and *buffering* (nit_buf) [4] modules.

4.2.2 The Monitor Application Program

The network monitor application is comprised of the NIT interface module and the NIT buffering module. At this time, we do not use the NIT packet filter module.

Functionally, the monitor program polls the read side of the NIT device until a specified time-out occurs. Within the polling loop, packet information is read from the NIT stream device and written to a file for processing at a later time. The relationship between the Ethernet device, the NIT facility, and the monitor application program is demonstrated in Figure 2.

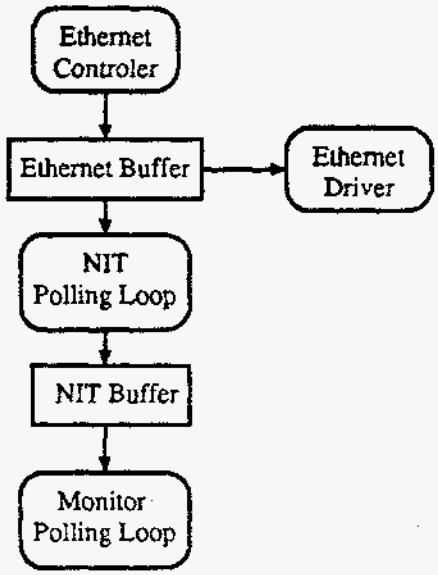


Figure 2: NIT interface facility.

As mentioned earlier, only the transport information portion of the network packet is utilized. All other information is discarded by the monitor application program. In addition to the transport information obtained from the Ethernet packet, NIT prepends a timestamp and a cumulative packet drop count to each packet obtained from the Ethernet device. The collection of information from each packet includes the timestamp (both seconds and microseconds), the drop count, packet size, source-destination address pair, and the overlying network protocol. As such, a total of 30 bytes from each observed packet is written to a file for offline processing.

By using the NIT facility, we have been able to collect on average 90 percent of all packets on the Ethernet network[1]. This results is a collection rate of approximately 160 packets in a one second interval or more than 1 Megabyte of data every four minutes. We feel that this is an acceptable collection percentage when considering the volume of traffic handled on the two monitored networks.

[1] The value of 90 percent was achieved by running the monitor process with root priviledge and a nice priority of "−10".

4.3 Data Preprocessing

Of the data currently saved, only the timestamp value, packet size, source-destination address pair, and network protocol descriptor are used within the learning classifer system. The cumulative packet drop value is of interest only to verify the performance of our monitoring application program. There are two reasons for preprocessing the data.

1. **Data compression**
   In the cases of source-destination addresses, packet sizes, and network protocols the raw data can be compressed without loss of relevant information. This results in data which is easier for the learning classifer system to manipulate and which requires less off-line disk space for storage. A real-time strategy may not require data compression.

2. **Expansion of information**
   In the case of timestamp information, the basic second count provided can be augmented to include contextual information such as *hour of day* and *day of week*. This allows the learning classifier system to build network behavior profiles that are based on human temporal patterns.

5 The Learning Classifier System

The input to the learning classifier system is simply a stream of tuples derived from the network transmission packet described earlier. The tuples specify the source, destination, size, time, and protocol used in an Ethernet transmission. Using this stream of tuples the classifier system must perform two functions. First, it must develop a rule-based model of normal network behavior by observing the tuple sequence. Second, based on the network model and recent tuples it must decide whether the network is in a normal or abnormal state. To perform these two functions the system uses four categories of rules, an historical data base, and an output interface. The relationship between these components is illustrated in Figure 3.

5.1 Rule Categories

1. **Aggregation and event detection rules**

   These rules attend directly to the network transmission tuples. They maintain various statistical counts and may flag significant events. The information produced by these rules is used by the higher level decision making rules and to the data base update rules that build an historical profile of network behavior.

   Examples of this rule class include rules that count the number of packets sent over the network in the last minute, count the number of packets sent

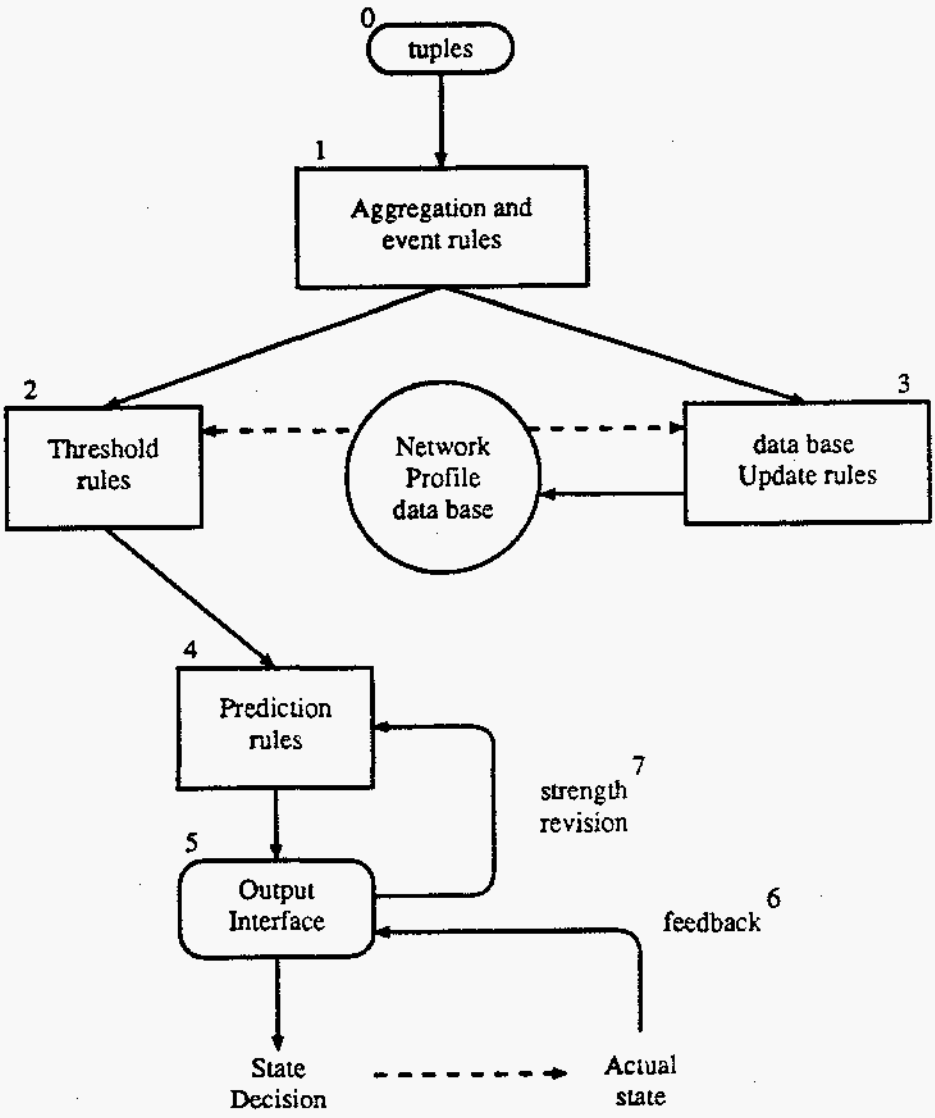


Figure 3: The Learning Classifier System.

between one source-destination machine pair in the last ten minutes, or flag the receipt of a packet from outside the local network.

2. **Data base update rules**

   These rules attend to the values posted by the aggregation and event detection rules. They update the data base that profiles the past network behavior.

   For instance, the standard deviation for the count of packets seen in the last minute might be under consideration as a meaningful measure of the network state. In this case there would be an update rule which attends to the aggregation rule counting the number of packets transmitted in the last minute. When the aggregation rule fires, this update rule would use the new count to update the mean and standard deviation values for packets in the last minute in the data base.

3. **Threshold rules**

   These rules also attend to the aggregation and event detection rules. They embody thresholds on the individual measures to which they attend. They fire with a value of abnormal if the threshold which they embody is exceeded, and with a value of normal otherwise.

   For example, a threshold rule that measures the number of standard deviations away from the mean value of packets transmitted in the last minute will fire abnormal if a certain number of standard deviations is exceeded.

4. **Prediction rules**

   These rules attend to one or more threshold rules. If all of the threshold rules to which they attend fire as abnormal then prediction rules predict the network state to be abnormal. Otherwise they predict the network state to be normal.

   Prediction rules have a strength rating attached to them which reflects their past success in predicting the true network state. Strength ratings are revised based on feedback from the output interface in a manner which will be discussed shortly. Prediction rules are the only rules which are candidates for deletion from the rule base, or for use in rule reproduction, by the genetic algorithm.

   For example, there would be a prediction rule that attends to the threshold rule for monitoring the number of packets sent in the last minute. If the threshold rule fires because the current count exceeded two standard deviations the prediction rule would fire as abnormal. During strength revision, this prediction rule would be rewarded if its prediction of the network state was correct. If its prediction of the network state was incorrect the prediction rule would be punished.

```
WHEN ( evaluation-condition ) DO
        IF ( firing-condition ) THEN
            true-side-effects
        ELSE
            false-side-effects
        ENDIF
ENDDO
```

Figure 4: The standard rule form.

5.2 The Standard Rule Form

All rules are represented in the standard form as seen in Figure 4[2]. The *evaluation-condition* for a rule is a set of one or more conditions that upon satisfaction may indicate that the rule is to be evaluated. For instance, the evaluation-condition for most aggregation rules is simply the receipt of a new network transmission packet. For data base update and threshold rules it is the firing of the aggregation rule on which they depend. For prediction rules it is the firing of any of the thresholds rules on which they depend.

The *firing-condition* for a rule is the circumstance under which it performs a state action. A state action updates information variables that may be either local to the rule or global to all other rules in the system.

To illustrate the use of the standard rule form, consider the monitoring of network packets transmitted in a one minute time period. The set of rules that would perform this task would include: 1) an aggregation rule that would count network packets, 2) a data base update rule that would record count statistics, 3) a threshold rule that would measure and report abnormalities in standard deviations from the count mean, and 4) a prediction rule that would report the network status.

The aggregation rule that counts the number of packets transmitted in the last minute would be evaluated when a new network packet has entered the system. Its firing condition is the completion of a one minute period. This period is determined by information stored in the network packet[3]. If a one minute period has completed the rule executes its true-side-effect of updating a global count variable to the one minute packet count and reinitializing its local count variable to zero. If a one minute peroid has not completed the rule performs its false-side-effect of simply updating its local count variable for the current minute. The rule form for such an aggregation rule is shown in Figure 5.

The data base update rule which attends to the aggregation rule will be

[2]This is true for all rule classes except the prediction rules. Prediction rules adhere to the standard rule execution flow, but are implemented as bit vectors.

[3]Each network packet has a time stamp that can be used to derive a discrete time frame.

```
WHEN ( new packet recieved ) DO
        IF ( one minite has elapsed ) THEN
           plm-count = count
           count = 0
           fire packet-last-minute aggregation rule
        ELSE
           count = count = 1
        ENDIF
ENDDO
```

Figure 5: Rule Form - Aggregation Rule.

```
WHEN ( packet-last-minute aggreagation rule fired ) DO
        IF ( true ) THEN
           plm-samples = plm-samples + 1
           plm-sum = plm-sum + plm-count
           plm-mean = plm-sum / plm-count
           claculate new standard deviation
        ELSE
           no false side effects
        ENDIF
ENDDO
```

Figure 6: Rule Form - Update Rule.

evaluated if the aggregation rule has fired (i.e. if a one minute time period has completed). Its sole purpose is to record count statistics for the number of network packets transmitted in the last one minute period. The rule has a firing condition that is always true, and its true-side-effect updates the profile data base. There is no false-side-effect for this particular rule. The rule form for a packets last minute update rule is shown in Figure 6.

Like the data base rule, the threshold rule will be evaluated if the aggregation rule has fired. When this rule is evaluated the firing-condition determines if the last packet count has exceeded a number of standard deviations away from the count mean. In essence, the firing-condition is always true. The true-side-effect sets a global variable that indicates the standard deviation being monitored has been exceeded, and the false-side-effect sets a global variable that indicates the standard deviation has not been exceeded. the rule form for a threshold rule is given in Figure 7.

The prediction rule attends to the threshold rule and is evaluated if the threshold rule has fired. The firing condition for this rule, like the threshold rule, is always true. It has a true-side-effect of a normal network state prediction

```
WHEN ( packet-last-minute aggreagation rule fired ) DO
        IF ( abs (plm-mean - plm-count) > 0.5 plm-stdev ) THEN
           fire 0.5 threshold rule as true
        ELSE
           fire 0.5 threshold rule as false
        ENDIF
ENDDO
```

Figure 7: Rule Form - Threshold Rule.

and a false-side-effect of an abnormal network state prediction.

The simple standard form for the rule base provides a consistent model that can be applied to all four rule classes. In most instances, rules can be generated from a standard input format as described in a later section of this paper, while preserving rule function modularity.

5.3 Strength Revision

Each prediction rule has a *strength* rating which reflects its past success in predicting the true network state. Only prediction rules receive feedback from the environment about their performance, and are the only candidates for deletion or use in reproduction by the genetic algorithm. The separation of the threshold and prediction rule classes eliminates the need for a *bucket brigade* algorithm of the sort found in many classifier systems.

The threshold rules are fed directly by the aggregation and event detection rules. The prediction rules, in turn, are fed directly by the threshold rules. Initially, each threshold rule feeds just one prediction rule which attends only to that threshold rule. In time the genetic algorithm combines the prediction rules to create new prediction rules which attend to more than one threshold rule, and drops the prediction rules corresponding to individual thresholds that have proven to be of little use. As long as a threshold rule feeds at least one active prediction rule it will remain in the system. When there are no longer any prediction rules which rely on a threshold rule it can be purged. Similarly, aggregation and event detection rules and data base update rules remain in the system only as long as they indirectly supply information needed by any existing prediction rule, and are removed when they cease to do so.

When prediction rules fire there are four possible results, depending on the rule prediction and the true network state as shown in Figure 8.

Results 1 and 2 evaluate the degree to which a rule reflects the stability of normal network behavior. Result 1 is rewarded and result 2 is punished. Results 3 and 4 evaluate the degree to which a rule is reactive to abnormal network behavior. Result 3 is punished and result 4 is rewarded. At present

	Prediction	Network State	Result	Reward
1	normal	normal	correct	+1
2	abnormal	normal	incorrect	−1
3	normal	abnormal	incorrect	−5
4	abnormal	abnormal	correct	+10

Figure 8: Possible prediction results.

reward and punishment is carried out according to the following regimen.

1. Rules correctly predicting normal network state have their strength increased by one. This modest reward reflects the fact that for reasonably good rules this will be a common occurrence.

   In addition, a restriction on the reward is added for groups of prediction rules which embody competing single thresholds on a particular measure. For instance, consider the packets transmitted in the last minute measure mentioned earlier. There might be several prediction rules which fire as abnormal depending on whether the latest count exceeded the mean count by more than 2, 3, or 4 standard deviations. If in fact the count never exceeded 3 standard deviations from the mean, we would not want the rule for 4 standard deviations to be rewarded as heavily as the one for 3. The 3 standard deviation rule, being tighter, is clearly more useful. To prevent this situation only the most specific member of such families of related rules are rewarded at any one time. That is, if all three of the 2, 3, and 4 standard deviation rules correctly predict a normal state, only the 2 standard deviation rule will receive the usual reward of having its strength increased by one. The strengths of the other two rules will remain unchanged. Useless prediction rules that embody thresholds so loose as to never be exceeded, therefore, will not become artificially strong. Instead, they will have strengths equal to another group of useless prediction rules, those that are wrong exactly half of the time.

   The above scheme is in effect only during the initial rule evaluation phase when prediction rules correspond directly to threshold families. Application of the genetic algorithm may eliminate the correspondence between the two rule classes, therefore strength revision is based on the performance of individual rules.

2. Rules incorrectly predicting an abnormal state have their strengths decreased by one. This situation indicates a rule does not accurately reflect the stability of normal network behavior.

3. Rules predicting a normal state when anomalous behavior is occurring

have their strengths reduced by five. This situation shows a rule is insufficiently reactive and is therefore penalized fairly heavily.

4. Rules correctly predicting an abnormal network state have their strengths increased by ten. This situation demonstrates the important characteristic of reactivity and is heavily reinforced.

The actual amount of the rewards and punishments for these four situations is likely to change as our work progresses. We expect, however, that the importance attached to each of the situations will be consistent.

5.4 System Running Modes

The network security system can be thought of as operating in three distinct modes as illustrated in Figure 3. In *profile learning* mode only the aggregation and event(1) and the data base update(3) rule classes are engaged. This mode is used to build the statistical profile of normal network activity. Care should clearly be taken not to run this learning mode while the network is operating under anomalous conditions.

In *rule evaluation* mode only the aggregation and event(1), threshold(2), and prediction(4) rule classes are engaged. A statistical profile of the network stored in the data base is assumed, and running the system gathers data on how well the various prediction rules characterize the network state. The success rates of the prediction rules are reflected in their strength values, which are constantly updated by the strength revision algorithm(7).

In *on-line* mode the aggregation and event(1), threshold(2), and prediction(4) rules will again be engaged, but will be assumed to have reached a stable state. Here the goal is to run fast enough to do real-time network monitoring. In this mode the output interface(5) will give a single normal/abnormal decision based on the collective input of the prediction rules, and no strength revision will occur.

The first two modes are the ones which will be used during initial development. The third mode is what is envisaged for a more mature version of the system. Even having achieved such a mature version, however, it will be necessary to periodically run the first two modes in order to adjust to changes in network behavior. The need to rerun the learning modes in a mature system would be indicated by the system's failure to detect attempted intrusions or by a series of false abnormal alarms.

6 The Genetic Algorithm

The individual threshold rules embody various measures of network behavior. The prediction rules can be thought of as hypotheses that combinations of these measures accurately predict network state. Since the size of the prediction rule

space is the power set of the threshold rule set, a relatively small number of measures can give rise to a large number of hypotheses. The function of the genetic algorithm is to explore the space of potential prediction rules without generating all of them.

Each prediction rule has a vector associated with it indicating which threshold rules it depends upon. Each threshold rule has an assigned position in all vectors. If a position in a prediction rule's vector is a 1 then it pays attention to the firing of the corresponding threshold rule. If the position is a 0 the prediction rule does not pay attention to that threshold rule.

The genetic algorithm takes as input the set of prediction rules and their corresponding strengths and vectors. The algorithm purges the rules with weakest strengths from the set, and then uses two genetic operators to generate new rules from the vectors of the remaining stronger rules. A step by step description of the algorithm follows.

1. The incoming set of prediction rules is sorted based on their strengths. Let the number of rules in the set be P.

2. The weakest rules are removed from the rule set. Let the number of rules removed be N.

3. A number of new rules are created from the remaining strongest rules using the crossover operator. Let the number of rules created by crossover be C.

4. A number of new rules are also created using the mutation operator. Let the number of rules created by mutation be M.

5. The newly generated rules are given a default strength value. The new prediction rule set is the union of the newly generated rules and the strongest of the incoming rules from which they were generated. We require that $N = C + M$, so the size of the new rule set is P, the same as the size of the existing rule set.

The crossover operator works by choosing two rules probabilistically from the list of the strongest rules. The probability of rule x being chosen is proportional to its strength.

$$\mathrm{P}(x) = \text{strength}(x) \;/\; (\text{sum of all strongest rules strengths})$$

Having chosen two rules, a point within the range of their vector lengths is randomly chosen. A new rule vector is formed by using the portion of the first rule vector up to the crossover point and the portion of the second rule vector after the crossover point. This vector now represents a prediction rule which attends to a new combination of threshold rules. The crossover process is illustrated in Figure 9.

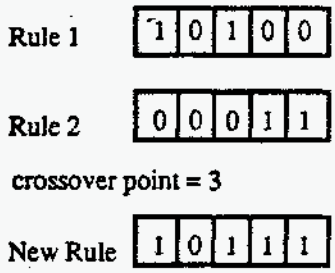


Figure 9: Crossover

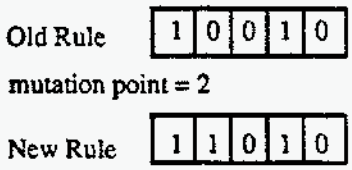


Figure 10: Mutation

The mutation operator probabilistically chooses one rule from the list of strongest rules. It then randomly chooses a vector position and forms a new rule vector by changing the value at that position from a 1 to a 0 or from a 0 to a 1. The mutation process is illustrated in Figure 10.

As each new rule is generated by the genetic operators two checks are made. First, if the result of an operator is a rule vector consisting of all 0's the vector is discarded and another is generated. Clearly such a prediction rule, which attended to nothing, would be useless. Second, each generated vector is compared to the generating list and the new vectors already generated to see if it is a duplicate. If it is then it is also discarded and another is generated.

Having generated a new prediction rule set the network security system will then be run on a substantial amount of tuple transmission data. The default strengths of the new rules will be altered over time by the strength revision algorithm to reflect their utility to the system. Eventually the new prediction rule set and their updated strengths will in turn be submitted to the genetic algorithm.

The assumption is that by combining parts of the strongest rules randomly the genetic algorithm will search the potential rule space more efficiently and more successfully than conventional heuristic methods. The intent is to combine the best features of best first search and random sampling. The algorithm uses the strongest rules to pursue branches in the rule space that have proven to

be successful as does best first search. It also has enough randomness in it operators to help it avoid getting stuck at local maxima.

7 Implementation of a Prototype

The implementation of a prototype network security system was written in MIT Scheme 7.0.0. The decision to use Scheme was based on two conditions. First, as a functional language, Scheme readily supports a prototype programming environment where the modification of source code is simple. That is, elaborate ideas can be implemented without concern for intricate programming details[4]. Second, MIT Scheme can compile source code. Compiled code executes much faster than interpreted code and, may be necessary for adhering to real-time efficiency constraints.

As described earlier, the learning classifier system and genetic algorithm are two distinct phases. The learning classifier system employs static rule types that update the network profile data base or make a prediction as to the state of the network. The genetic algorithm defines a subset of the prediction rule space based on past performance of individual prediction rules. Implementation of the prototype system parallels the learning classifier system and genetic algorithm phases outlined in sections 5 and 6.

7.1 The Learning Classifier System

The structure of the learning classifier system consists of two separate phases. The first phase initializes the rule base such that each rule definition is transformed into an executable form. The second phase implements an interpretive loop. The interpretive loop provides a direct interface between the network transmission tuples and the learning classifier rule base.

7.1.1 Rule Structure

A common rule structure is used for all rules in the rule base. This structure is shown in Figure 11. All rules must contain the first four fields along with some form of action or effect. The remaining items are optional and included in a rule when needed. The *rule name* is a unique identifier that is used to identify the rule in various association lists. The *rule type* is one of aggregation, threshold, or update. The *evaluation condition* dictates when the rule will be evaluated, while the *firing condition* determines if the true or false effect will take place whne the rule is fired. *Locals* are local variables, such as counters, that are used to maintain data specific to a given rule. *Globals-set* and *defaults* must occur in pairs. The *globals-set* are state variables that the rule sets and the *defaults* are the values used to initialize those variables. Notice that each global variable

[4] Such as maintaining linked lists and updating pointer variables.

```
;rule template
(def-rule      ;rule name
  (            ;rule type
  (            ;evaluation condition
  (            ;firing condition
  (:locals
  (:globals-set
  (:defaults
  (:globals-read
  (:true-effects
  (:false-effects
  (:predecessor      ;threshold rules only
)
```

Figure 11: Rule List Template

has a unique rule which is capable of altering the value stored in the variable. *Globals-read* are global variables that the rule uses as input. The *predecessor* field is only used with threshold rules. This information is used to determine the most specific threshold rule when a set of consecutive threshold rules all fire with the same value.

7.1.2 Initialization of the Rule Base

Rules are stored in a rule data base file and are in the form of a Scheme *list*. Each rule list may be viewed as rule construction information. Executable rules are produced by a Scheme *macro*. The *macro* extracts the rule construction information from the rule list and inserts that information into the appropriate fields of the standard rule form described earlier. When all applicable fields in the standard rule form have been defined, the *macro* constructs a *closure* and the executable rule body is placed in a list based on rule type. Each list forms a sub-partition of the rule base and includes the rule categories 1) aggregation and event detection, 2) data base update, and 3) threshold. Rule lists for the packets last minute example are displayed in figures 12, 13, and 14. Figure 12 shows the file list for the packets last minute aggregation rule. Figures 13 and 14 display the rule lists for the attending data base update and threshold rules.

In most cases, the order in which rules are fired is not important. However, there may be data dependencies among the aggregation and event detection rules. In particular, one rule may use values determined by another rule in its firing condition or in its rule body. The macro performs a topological sort on the aggregation and event detection rules based on the globals-read and globals-set fields to insure that the firing order maintains the proper data dependencies.

The macro also generates secondary information lists which contain internal

```
(def-rule A1-packets-last-minute                    ; name, set true if firing.
  aggregation                                       ; rule type
  #t                                                ; evaluation condition
  (>=tuple-sec (+ start-time 60))                   ; firing condition

  (:locals ((plm-count -1) (start-time -1)))        ; local var, init value pairs
  (:globals-set (A1-plm-count) )                    ; count of packets last minute
  (:defaults (-1))
  (:globals-read (tuple-sec))                       ; global vars attended to

  (:true-effects                                    ; true side effects
    (
      ; make last minute count available
      (set! global (set-alist 'A1-plm-count plm-count global))

      ; show rule firing
      (set! aggregate-fired
      (set-alist 'A1-packets-last-minute #t aggregate-fired))

      ; reset locals for next minute count
      (set! plm-count 0)
      (set! start-time tuple-sec)
    )
  )

  (:false-effects                                   ; false side-effects
    (
      (set! plm-count (+ plm-count 1))              ; update count
    )
  )
)
```

Figure 12: Packets Last Minute Aggregation Rule List

```
(def-rule U1-A1-plm-count                                  ;rule name
  update                                                   ;rule type
  ( get-alist 'A1-packets-last-minute aggregate-fired)     ;evaluation condition
  #t                                                       ;firing condition

  ; db values read, set: U1-plm-samples, U1-plm-sum, U1-plm-squares-sum,
  ; U1-plm-mean, U1-plm-stdev.
  (:globals-read ( A1-plm-count ))
  (:true-effects
  (
  ; get current data base values
  (let*(  (tmp-samples (+ (get-alist 'U1-plm-samples dbval) 1))
          (tmp-count (get-alist 'A1-plm-count global))
          (tmp-sum (+ (get-alist 'U1-plm-sum dbval) tmp-count))
          (tmp-mean (/ tmp-sum tmp-samples))
          (tmp-squares-sum (+ (get-alist 'U1-plm-squares-sum dbval)
                              (* tmp-count tmp-count)))
       )

  ;calculate new values and store in the data base
  (set! dbval (set-alist 'U1-plm-samples tmp-samples dbval))
  (set! dbval (set-alist 'U1-plm-sum tmp-sum dbval))
  (set! dbval (set-alist 'U1-plm-mean tmp-mean dbval))
  (set! dbval (set-alist 'U1-plm-squares-sum tmp-squares-sum dbval))
  (set! dbval (set-alist 'U1-plm-stdev (sqrt (- (/ tmp-squares-sum tmp-samples)
                                  (* tmp-mean tmp-mean))) dbval))
  (set! update-fired (set-alist 'U1-A1-plm-count #t update-fired))
  )
  )
  )
)
```

Figure 13: Packets Last Minute Update Rule List

```
(def-rule T1-plm-thresh-0.5                                    ;rule name
  threshold                                                    ;rule type
  (get-alist 'A1-packets-last-minute aggregate-fired)          ;evaluation condition
  #t                                                           ;firing condition
  (:globals-read (A1-plm-count))
  (:vector-size (1))
  ; db-values read : U1-plm-mean U1-plm-stdev
  (:true-effects
    (
      (let* ((base-addr (get-alist 'T1-plm-thresh-0.5 thresh-base-addr))
             (offset 0)
             (index (+ base-addr offset)))
        (cond
          (
            (> (abs (- (get-alist 'A1-plm-count global)
                (get-alist 'U1-plm-mean dbval)))
                     (* (get-alist 'U1-plm-stdev dbval) 0.5))
                (bit-string-set! threshold-result index)
            )
          )
          (set! pred-firing-list (cons index pred-firing-list))
          (bit-string-set! threshold-fired index)
        )
      )
    )
)
```

Figure 14: Packets Last Minute Threshold Rule List

information required by other parts of the learning classifier system and the genetic algorithm to the prediction rule set.

Initialization of the prediction rule set is accomplished by the construction of a list of *bit-strings*. Each bit-string has an initial bit position set to 1 that corresponds to a unique threshold rule.

7.1.3 Parameterized Rules

Threshold rules may be parameterized. Parameterization of the threshold rules allows a small set of threshold rules to cover a large rule space. Take, for example, a set of threshold rules that attend to eight different standard deviations from the mean for each half hour in a given 24 hour period. This rule space is defined by an 8 by 48 matrix or 384 total rules. By parameterizing the threshold rule on the half hour variable, we are able to reduce the 384 separate rules to only 8 rules, one for each of the standard deviation measures. As a result, a savings in both the execution time and the physical size of the rule base was realized. Currently, we have only used this capability to parameterized threshold rules on half hour time slices.

7.1.4 The Interpretive Loop

The interpretive loop provides two integral functions to the learning classifier system. First, network transmission tuples are introduced into the system with each iteration of the loop. Second, the rule base is allowed to interact with the global variables which profile the current state of the network during each iteration of the loop.

Prior to invoking the interpretive loop, the network transmission tuple file is opened as an *input-port*. Each iteration of the loop is marked by the introduction of a new network transmission tuple into the system. The components of the tuple are bound to the global variables.

After each tuple component is bound to the appropriate global variable rule class members are evaluated to determine if they should fire. Depending on the operation mode the system, certain rule classes are not evaluated.

As outlined earlier, there are three modes that the learning classifier system can operate in. These modes are "Profile Learning", "Rule Evaluation" and "On-line". The actions that occur during each of the modes are described below.

Profile Learning
: In "Profile Learning" mode only the aggregation and event, and the data base update rule classes are evaluated . It is during this mode that variables in the network profile data base are updated.

Rule Evaluation
: In "Rule Evaluation" mode only the aggregation and event, threshold, and

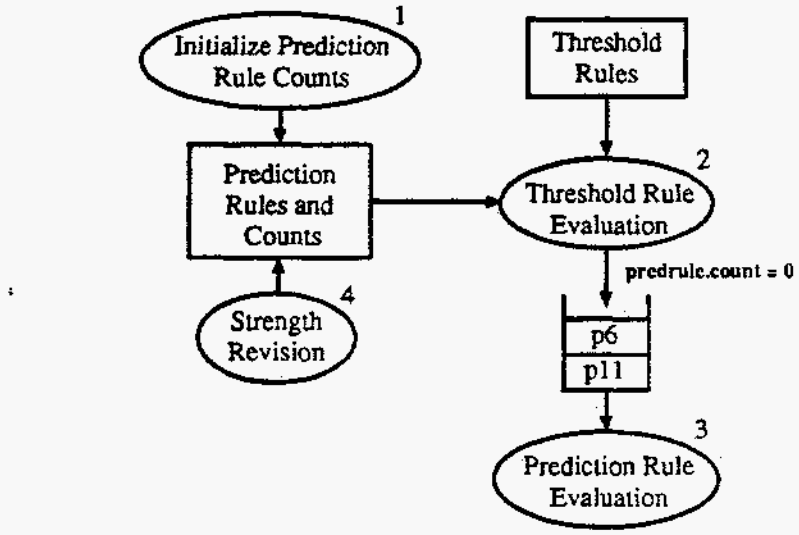


Figure 15: Prediction rule evaluation scheme.

prediction rule classes are evaluated. Prediction rules have their strengths revised based on their latest prediction of the current network state.

On-line

In "On-line" mode the aggregation and event, threshold, and prediction rule classes are evaluated. To date, all testing is performed off-line where the network transmission tuple data is read from a historical data file rather than an active network. It is assumed that prediction rules will be in a stable state while in this mode and will not require their strengths to be revaluated.

In both the rule evaluation and on-line modes evaluation of the prediction rule set occurs. To avoid evaluation of the entire rule set, each prediction rule member maintains a count of how many threshold rules it attends. When a threshold rule fires, it decrements the count value for each prediction rule that depends on it. When the count value reaches zero for any prediction rule, that rule is placed on a prediction rule firing list. The prediction rule evaluation scheme is illustrated in Figure 15.

As the network security system is still in the prototype phase, all three modes of operation produce user output. Profile learning mode emits the current data base values. Rule evaluation mode emits the network state predictions and individual prediction rule strengths. On-line mode emits the network state

predictions. Currently, all output is reported to the standard output interface[5] and to separate data files.

7.1.5 Strength Revision

Strength revision is performed in accordance with the current state of the network as defined by a network state variable, and occurs only while in the Rule evaluation mode. Implementation of the strength revision is based on the conditional expression *cond*. After all the prediction rules have fired (i.e., make a prediction as to the network state) the strength revision function is called. It is at this time that the *cond* expression is evaluated and the strength value of each prediction rule that had fired is revised as described earlier.

7.2 Implementation of the Genetic Algorithm

Implementation of the Genetic Algorithm requires two *assocation lists* and a number of small helper functions which together perform the genetic operations. The association lists are produced as a result of the rule base initialization and provide a method of data base query. The search key in both association lists are individual prediction rule names. The data field in one association list consists of rule strengths while the other association list maintains each of the prediction rule's dependence vector. Each dependence vector is implemented as a *bit string*. The association list which holds the prediction rule strengths is sorted according to individual strengths. A predetermined number of these rules are purged from the list. The corresponding set of rules is also purged from the dependence vector list.

As mentioned earlier, there are two changes which take place to the prediction rule's dependence vector. The first change occurs by using a crossover operator. The crossover operator is implemented as a probabalistic function which takes as an argument the remaining prediction rule list and returns a rule pair upon which the crossover is applied. The crossover function divides each of the rule's dependence vectors at a random location and produces two new vectors by swapping and concatenating the halves. By utilizing a similar probabalistic function, the mutation operator selects a single rule from the remaining prediction rule list and randomly changes a single value in the dependence vector.

The number of new prediction rules generated is equal to the number of rules purged from the older rule list. These new rules are inserted directly into the prediction rule list without any user interaction. As such, the network security system can be brought back into full "On-line" operation immediately after applying the genetic algorithm module.

[5]The green phosphor CRT.

8 Summary and Future Work

Considerable work has been done in the area of computer system protection. For the most part, however, that work has concentrated on protection at the level of individual machines and operating systems. The widespread existence of computer networks, combined with events such as the Internet worm of November, 1988, demonstrate the need to address protection issues at the network level as well. The focus of our research is to determine the feasibility of network level monitoring to protect network resources from attack.

Our initial goal is to build an off-line prototype system capable of learning normal patterns of network use and flagging departures from those patterns of normality. Such a system will permit verification of the hypothesis that intrusive attacks are in fact detectable as deviations from a rule-based profile of normal behavior. In addition, the prototype will allow us to establish whether the behavior profile eventually reaches a state of statistical stability. Our belief is that after an initial learning phase during which both rules and the confidence factors attached to them vary rapidly a state of equilibrium will be reached. At that point the system will need to be flexible enough to adjust to genuine incremental changes in normal network behavior, but not so flexible as to allow an intruder to gradually teach the system to accept new behavior which opens the network to attack.

Our short term objective is to continue the work on the current prototype. We will expand the prototype by adding new rules to the rule base. We feel that source/destination pairings will prove to be useful in characterizing network behavior. To implement the rules needed to capture source/destination parings, we will need to extend the parameterization of the threshold rules to include internal variables such as source and destination identifiers. We also need to evaluate the prototype on two different levels. To refine our statistical characterization of network behavior, we need to evaluate the statistical distrubutions produced by the prototype. To avoid over learning, we will evaluate the amount of learning that must take place before the classifying rule set stablizes. Finally, we hope to construct a meaingful graphical user inferface for the network monitoring system.

Our longer term research goals include moving from an off-line to an on-line system in order to provide real-time network level protection. The move to an on-line system will in turn raise the issue of developing appropriate reactions to detected intrusions. Attempts to lessen the impact of detected intrusions may include delaying or ignoring communications involving the suspected participating nodes.

References

[1] *Sun Microsystem's Operating System* 4.0. Section nit (4P): Protocols.

[2] *Sun Microsystem's Operating System* 4.0. Section nit_if (4M): Devices and Network Interfaces.

[3] *Sun Microsystem's Operating System* 4.0. Section nit_pf (4M): Devices and Network Interfaces.

[4] *Sun Microsystem's Operating System* 4.0. Section nit_buf (4M): Devices and Network Interfaces.

[5] L. B. Booker, D. E. Goldberg, and J. H. Holland. Classifier systems and genetic algorithms. *Artificial Intelligence*, 40:235–282, September 1989.

[6] Dorothy E. Denning. An intrusion-detection model. In *IEEE Symposium on Security and Privacy*, pages 118–131. IEEE, 1986.

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

[8] L. Todd Heberline, Gihan V. Dias, Karl N. Levitt, Biswanath Mukherjee, Jeff Wood, and David Wolber. A network security monitor. In *IEEE Symposium on Security and Privacy*, pages 296–304. IEEE, 1990.

[9] Carl E. Landwehr. Formal models for computer security. *ACM Computing Surveys*, 13(3):247–278, September 1981.

[10] T. F. Lunt and R. Jagannathan. A prototype real-time intrusion-detection expert system. In *IEEE Symposium on Security and Privacy*, pages 59–66. IEEE, 1988.

[11] T.F. Lunt, R. Jagannathan, R. Lee, S. Listgarten, D.L. Edwards, P.G. Neumann, H.S. Javitz, and A. Valses. Ides: The enhanced prototype. Technical report, SRI International, October 1988.

[12] Eugene F. Spafford. The internet worm: Crisis and aftermath. *Communications of the ACM*, 32(6):678–687, June 1989.

[13] A. S. Tanenbaum. *Computer Networks*. Prentice-Hall, Englewood Cliffs, N. J., second edition, 1988.

[14] H. S. Vaccaro and G. E. Liepins. Detection of anomalous computer session activity. In *IEEE Symposium on Security and Privacy*, pages 280–289. IEEE, 1989.