Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks

Matthew V. Mahoney and Philip K. Chan
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901
{mmahoney,pkc}@cs.fit.edu

ABSTRACT

Traditional intrusion detection systems (IDS) detect attacks by comparing current behavior to signatures of known attacks. One main drawback is the inability of detecting new attacks which do not have known signatures. In this paper we propose a learning algorithm that constructs models of normal behavior from attack-free network traffic. Behavior that deviates from the learned normal model signals possible novel attacks. Our IDS is unique in two respects. First, it is nonstationary, modeling probabilities based on the time since the last event rather than on average rate. This prevents alarm floods. Second, the IDS learns protocol vocabularies (at the data link through application layers) in order to detect unknown attacks that attempt to exploit implementation errors in poorly tested features of the target software. On the 1999 DARPA IDS evaluation data set [9], we detect 70 of 180 attacks (with 100 false alarms), about evenly divided between user behavioral anomalies (IP addresses and ports, as modeled by most other systems) and protocol anomalies. Because our methods are unconventional, there is a significant non-overlap of our IDS with the original DARPA participants, which implies that they could be combined to increase coverage.

1. INTRODUCTION

One important facet of computer security is intrusion detection - simply knowing whether a system has been compromised, or if an attack has been attempted. There are two general approaches to this problem: *signature detection*, where we look for patterns signaling well known attacks, and *anomaly detection*, where we look for deviations from normal behavior to signal possibly novel attacks. Signature detection works well, but has the obvious disadvantage that it will not detect new attacks. Anomaly detection has the disadvantage that it cannot discern intent. It can only signal that something is unusual, but not necessarily hostile, thus generating false alarms.

A complete intrusion detection system (IDS) might monitor network traffic, server and operating system events, and file system integrity, using both signature detection and anomaly detection at each level. We distinguish between a *network* IDS, which monitors traffic to and from the host, and a *host* based IDS, which monitors the state of the host. These systems differ in the types of attacks they can detect. A network IDS detects probes (such as port scans), denial of service (DOS) attacks (such as server floods), and remote-to-local (R2L) attacks in which an attacker without user level access gains the ability to execute commands locally. A host based system can detect R2L and user-to-root (U2R) attacks, where an attacker with user level access gains the privileges of another user (usually root). A host based system must reside on the system it monitors, while a network IDS can be physically separated and monitor multiple hosts on a local network. Also, because a network IDS monitors input (and output) rather than state, it can detect failed attacks (e.g. probes).

In this paper, we focus on network anomaly detection, which is essentially the machine learning problem of modeling normal network traffic from a training set. However, the anomaly detection task differs from the classical classification task in machine learning since only one class exists in the training data. That is, in anomaly detection we try to learn the characteristics of one class and determines if an unseen instance belongs to the same class.

Most network anomaly systems such as ADAM [3], NIDES [1], and SPADE [18] monitor IP addresses, ports, and TCP state. This catches user misbehavior, such as attempting to access a password protected service (because the source address is unusual) or probing a nonexistent service (because the destination address and port are unusual). However, this misses attacks on public servers or the TCP/IP stack that might otherwise be detected because of anomalies in other parts of the protocol. Often these anomalies occur because of software errors in the attacking or victim program, because of anomalous output after a successful attack, or because of misguided attempts to elude the IDS. Our IDS has two nonstationary components developed and tested on the 1999 DARPA IDS evaluation test set [9], which simulates a local network under attack. The first component is a packet header anomaly detector (PHAD) which monitors the entire data link, network, and transport layer, without any preconceptions about which fields might be useful. The second component is an application layer anomaly detector (ALAD) which combines a traditional user model based on TCP connections with a model of text-based protocols such as HTTP, FTP, and SMTP. Both systems learn which attributes are useful for anomaly detection, and then use a nonstationary model, in which events receive higher scores if no novel values have been seen for a long time. We evaluate its performance on the DARPA IDS evaluation data set and investigate the contribution of user vs. software anomalies toward detection.

SIGKDD '02, July 23-26, 2002, Edmonton, Alberta, Canada.

The rest of this paper is organized as follows. In Section 2, we discuss related work in network anomaly detection. In 3, we describe nonstationary modeling in general, and then PHAD and ALAD. In 4, we describe the DARPA IDS evaluation data set. In 5, we test our IDS and describe the five types of anomalies found. In 6 we describe our implementation and its run time performance. In 7, we conclude and describe future work.

2. RELATED WORK

Early work in anomaly detection was host based. Forrest et. al. [5] demonstrated that when software errors in UNIX servers or operating system services (*suid root* programs) are exploited in an R2L or U2R attack, that they deviate from the normal pattern of system calls. When compromised, these programs execute code on the behalf of the attacker (usually a shell), rather than the code intended by the developer (such as a DNS name server or print queue manager). Forrest detected these attacks by training an n-gram model (n = 3 to 6) as the system ran normally. More recent work has focused on better models, such as state machines [17], or neural networks [6]. Solaris makes system call information available through its basic security module (BSM) service for this purpose.

Network intrusion detection is typically rule based. It is fairly straightforward to write a rule in SNORT [15] or BRO [12] for example, to reject any packet addressed to a nonexistent host or service, or to write rules restricting services to a range of trusted addresses. However, it is a little more challenging to relieve the network administrator of the task of keeping the rules updated by monitoring the traffic to determine normal usage patterns. However, systems such as ADAM [3], NIDES [1], and SPADE [18] do this. ADAM (Audit Data and Mining) is a combination anomaly detector and classifier trained on both attack-free traffic and traffic with labeled attacks. It monitors port numbers, IP addresses and subnets, and TCP state. The system learns rules such as "if the first 3 bytes of the source IP address is X, then the destination port is Y with probability *p*". It also aggregates packets over a time window. ADAM uses a naive Bayes classifier, which means that if a packet belongs to some class (normal, known attack, or unknown attack), then the probabilities for each condition (such as *IP address = X* and *port = Y*) are assumed to be independent. ADAM has separate training modes and detection modes.

NIDES [1], like ADAM, monitors ports and addresses. Instead of using explicit training data, it builds a model of long term behavior over a period of hours or days, which is assumed to contain few or no attacks. If short term behavior (seconds, or a single packets) differs significantly, then an alarm is raised. NIDES does not model known attacks; instead it is used as a component of EMERALD [11], which includes host and network based signature detection for known attacks.

SPADE [18] is a SNORT plug-in that detects anomalies in network traffic. Like NIDES and ADAM, it is based on port numbers and IP addresses. SPADE estimates probabilities by counting incoming server requests (TCP SYN packets) in a way that favors newer data over old, and assigns high anomaly scores to low probability events. It uses several user selectable statistical models, including a Bayes classifier, and no explicit training period. It is supplemented by SNORT rules that use signature detection for known attacks. SNORT rules are more powerful, in that they can test any part of the packet including string matching in the application payload. To allow examination of the application layer, SNORT includes plug-ins that reassemble IP fragments and TCP streams.

ADAM, NIDES, and SPADE only model source and destination addresses and ports and TCP connection state, which catches many attempts to access restricted or nonexistent services. However, there are two reasons that we should monitor the other attributes of the TCP/IP protocols. First, there are many probes and DOS attacks that work by sending malformed or unusual packets to the victim, for example *queso, teardrop*, and *land* [8]. *Queso* is a fingerprinting probe that determines the operating system using characteristic responses to unusual packets, such as packets with the TCP reserved flags set. *Teardrop* crashes stacks that cannot cope with overlapping IP fragments. *Land* crashes stacks that cannot cope with a spoofed IP source address equal to the destination. Attacks are necessarily different from normal traffic because they exploit bugs, and bugs are most likely to be found in the parts of the software that were tested the least during normal use. A strict anomaly model would catch many of these attacks [16], but currently all anomaly models that monitor these attributes are hard coded rather than learned.

Second, an attacker may deliberately use malformed or unusual packets to hide attacks from an IDS application layer. Techniques include the deliberate use of bad checksums, unusual TCP flags or IP options, invalid sequence numbers, spoofed addresses, duplicate TCP packets with differing payloads, packets with short TTL values that expire between the target and IDS, and so on [14]. These techniques exploit bugs in the IDS or incomplete implementation of the protocols. Unfortunately, this is a common problem. For example, Handley et. al. [7] studied four commercial intrusion detection systems and found that none of them reassembled fragmented IP packets, a legal but rarely used feature of the IP protocol.

3. LEARNING NONSTATIONARY MODELS

The goal of intrusion detection is, for any given event *x*, to assign odds that *x* is hostile, e.g.,

odds (x_is_hostile) = P(attack|x) / P(no_attack|x)

By Bayes law, we can write:

P(attack|x) = P(x|attack)P(attack) / P(x)
P(no_attack|x) = P(x|no_attack)P(no_attack) / P(x)

By dividing these equations, and letting odds(attack) = P(attack) / P(no_attack), we have:

odds(x_is_hostile) = odds(attack)P(x|attack) / P(x | no_attack)

We have factored the intrusion detection problem into three terms: odds(attack), the background rate of attacks; P(x|attack), a signature detection model, and 1 / P(x|no_attack), an anomaly detection model. In this paper, we address only the anomaly detection component, 1 / P(x|no_attack). Thus, we model attack-free data, and assign (like SPADE) anomaly scores inversely proportional to the probability of an event based on this training.

Anomaly detection models like ADAM, NIDES, and SPADE are stationary, in that P(x) depends on the average rate of x in training and is independent of time. For example, the probability of observing some particular IP address is estimated by counting

the number of observations in training and dividing by the total number of observations. However, this may be incorrect. Paxson and Floyd [13] showed that many types of network processes, such as the rate of a particular type of packet, have self-similar or fractal behavior. This is a nonstationary model, one in which no sample, no matter how short or long, can predict the rate of events for any other sample. Instead, they found that events tend to occur in bursts separated by long gaps on all time scales, from milliseconds to months. We believe this behavior is due to changes of state in the system, such as programs being started, users logging in, software and hardware upgrades, and so on.

We can adapt to state changes by exponentially decaying the training counts to favor recent events, and many models do just that. One problem with this approach is that we have to choose either a decay rate (half life) or a maximum count in an *ad-hoc* manner. We avoid this problem by taking training decay to the extreme, and discarding all events (an attribute having some particular value) before the most recent occurrence. In our model, the best predictor of an event is the time since it last occurred. If an event x last occurred t seconds ago, then the probability that x will occur again within one second is $1/t$. We do not care about any events prior to the most recent occurrence of x.

In an anomaly detection system, we are most interested in those events that have the lowest probability. As a simplification, we assign anomaly scores only to those events that have *never* occurred in training, because these are certainly the least likely. We use the PPMC model of novel events, which is also used in data compression [2]. This model states that if an experiment is performed n times and r different outcomes are observed, then the probability that the next outcome will not be one of these r values is approximately r/n. Stated another way, the fraction of events that were novel in training is r/n, and we expect that rate to continue. This probably overestimates the probability that the next outcome will be novel, since most of the novel events probably occurred early during training. Nevertheless, we use it.

Because we have separate training data (without attacks) and test data (with attacks), we cannot simply assign an anomaly score of $1/P(x) = n/r$. If we did, then a subsequent occurrence of x would receive the same score, even though we know (by our nonstationary argument) that a second occurrence is very likely now. We also cannot add it to our model, because the data is no longer attack-free. Instead, we record the time of the event, and assign subsequent occurrences a score of $t/P(x) = tn/r$, where t is the time since the previous anomaly. On the first occurrence of x, t is the time since the last novel observation in training.

An IDS monitors a large number of attributes of a message, each of which can have many possible outcomes. For each attribute with a value never observed in training, an anomaly score of tn/r is computed, and the sum of these is then assigned to the message. If this sum exceeds a threshold, then an alarm is signaled.

anomaly score = $\Sigma_i\ t_i n_i / r_i$, where attribute i is novel in training

In the next two sections, we describe two models, PHAD and ALAD. In PHAD (packet header anomaly detection), the message is a single network packet, and the attributes are the fields of the packet header. In ALAD (application layer anomaly detection), the message is an incoming server TCP connection. The attributes are the application protocol keywords, opening and closing TCP flags, source address, and destination address and port number. Our IDS consists of both components running at the same time.

3.1. Packet Header Anomaly Detection (PHAD)

PHAD monitors 33 fields from the Ethernet, IP, and transport layer (TCP, UDP, or ICMP) packet header. Each field is one to four bytes, divided as nearly as possible on byte boundaries as specified by the RFCs (request for comments) that specify the protocols, although we had to combine fields smaller than 8 bits (such as the TCP flags) or split fields longer than 32 bits (such as the Ethernet addresses).

The value of each field is an integer. Depending on the size of the field, the value could range from 0 to $2^{32} - 1$. Because it is impractical to represent every observed value from such a large range, and because we wish to generalize over continuous values, we represent the set of observed values with a set of contiguous ranges or clusters. Each new observed value forms a cluster by itself. If the number of clusters exceeds a limit, C, then we merge the two closest ones into a single cluster. For example, if C = 3 and we have {3-5, 8, 10-15, 20}, then we merge the two closest to form {3-5, 8-15, 20}. For the purposes of anomaly detection, the number of novel values, r, is the number of times the set of clusters is updated.

Table 1. The PHAD model after training on "inside" week 3 of the DARPA IDS data set [9].

Attribute	r/n	Allowed Values
Ethernet Size	508/12814738	42 60-1181 1182...
Ether Dest Hi	9/12814738	x0000C0 x00105A...
Ether Dest Lo	12/12814738	x000009 x09B949...
Ether Src Hi	6/12814738	x0000C0 x00105A...
Ether Src Lo	9/12814738	x09B949 x13E981...
Ether Protocol	4/12814738	x0136 x0800 x0806...
IP Header Len	1/12715589	x45
IP TOS	4/12715589	x00 x08 x10 xC0
IP Length	527/12715589	38-1500
IP Frag ID	4117/12715589	0-65461 65462...
IP Frag Ptr	2/12715589	x0000 x4000
IP TTL	10/12715589	2 32 60 62-64 127-128 254-255
IP Protocol	3/12715589	1 6 17
IP Checksum	1/12715589	xFFFF
IP Source Addr	293/12715589	12.2.169.104...
IP Dest Addr	287/12715589	0.67.97.110...
TCP Source Port	3546/10617293	20-135 139 515...
TCP Dest Port	3545/10617293	20-135 139 515...
TCP Seq Num	5455/10617293	0-395954185...
TCP Ack Num	4235/10617293	0-395954185...
TCP Header Len	2/10617293	x50 x60
TCP Flags	9/10617293	x02 x04 x10...
TCP Window Size	1016/10617293	0-5374 5406-10028...
TCP Checksum	1/10617293	xFFFF
TCP URG Ptr	2/10617293	0 1
TCP Options	2/611126	x02040218 x020405B4
UCP Source Port	6052/2091127	53 123 137-138...
UDP Dest Port	6050/2091127	53 123 137-138...
UDP Length	128/2091127	25 27 29...
UDP Checksum	2/2091127	x0000 xFFFF
ICMP Type	3/7169	0 3 8
ICMP Code	3/7169	0 1 3
ICMP Checksum	1/7169	xFFFF

Table 1 shows the result of training PHAD with C = 32 on "inside" week 3 (7 days of attack free network traffic) from the DARPA IDS data set [9]. It shows the name of each attribute

(field in the packet header), the observed values of *n* and *r*, and a partial list of the observed values or clusters. For example, the first line says that out of 12,814,738 packets with an Ethernet size field (all of them in this case), there were 508 cases where the list of clusters had to be updated. Three of these clusters are 42, 60-1181, and 1182. The last two could have been merged, but were not because C was never exceeded. The maximum value of this field is actually 1514.

For most fields, we do not care what they are for, beyond their role in parsing the packet header. We made an exception for the checksum fields by computing them and substituting their computed values. Although our goal is to have the IDS learn as much of the protocol as possible on its own, we felt it was unreasonable for a machine learning program to learn the checksum algorithm. A value of xFFFF is a correct checksum.

There is a wide range of C for which PHAD gives good results. We found in [10] that using C = 32 detects slightly more attacks than using either C = 1000, or using a hash function (with range 1000) and a bitmap. These all work about the same because the fields which generate the highest anomaly scores are those with small *r* ($r < 32$), in which case the models are equivalent (except for hash collisions, which are rare for small *r*).

3.2. Application Layer Anomaly Detection (ALAD)

The second component of our anomaly detection model is the application layer anomaly detector (ALAD). Instead of assigning anomaly scores to each packet, it assigns a score to an incoming server TCP connection. TCP connections are reassembled from packets. ALAD, unlike PHAD, is configured knowing the range of IP addresses it is supposed to protect, and it distinguishes server ports (0-1023) from client ports (1024-65535). We do this because most attacks are initiated by the attacker (rather than by waiting for a victim), and are therefore against servers rather than clients.

We tested a large number of attributes and their combinations that we believed might make good models, and settled on five that gave the best performance individually (high detection rate at a fixed false alarm rate) on the DARPA IDS evaluation data set [9]. These are:

1. **P(src IP | dest IP)**, where *src IP* is the external source address of the client making the request, and *dest IP* is the local host address. This differs from PHAD in that the probability is conditional (a separate model for each local dest IP), only for TCP, and only for server connections (destination port < 1024). In training, this model learns the normal set of clients or users for each host. In effect, this models the set of clients allowed on a restricted service.
2. **P(src IP | dest IP, dest port)**. This model is like (1) except that there is a separate model for each server on each host. It learns the normal set of clients for each server, which may be differ across the servers on a single host.
3. **P(dest IP, dest port)**. This model learns the set of local servers which normally receive requests. It should catch probes that attempt to access nonexistent hosts or services.
4. **P(TCP flags | dest port**). This model learns the set of normal TCP flag sequences for the first, next to last, and last packet of a connection. A normal sequence is SYN (request to open), FIN-ACK (request to close and acknowledge the previous packet), and ACK (acknowledge the FIN). The model generalizes across hosts, but is separate for each port number, because the port number usually indicates the type of service (mail, web, FTP, telnet, etc.). An anomaly can result if a connection fails or is opened or closed abnormally, possibly indicating an abuse of a service.
5. **P(keyword | dest port)**. This model examines the text in the incoming request from the reassembled TCP stream to learn the allowable set of keywords for each application layer protocol. A *keyword* is defined as the first word on a line of input, i.e. the text between a linefeed and the following space. ALAD examines only the first 1000 bytes, which is sufficient for most requests. It also examines only the header part (ending with a blank line) of SMTP (mail) and HTTP (web) requests, because the header is more rigidly structured and easier to model than the body (text of email messages or form uploads). An anomaly indicates the use of a rarely used feature of the protocol, which is common in many R2L attacks.

As with PHAD, the anomaly score is *tn/r*, where *r* different values were observed out of *n* training samples, and it has been *t* seconds since the last anomaly was observed. An anomaly occurs only if the value has never been observed in training. For example, Table 2 shows the keyword model for ports 80, 25, and 21, which are the three ports with the highest *n/r* values.

Table 2. ALAD models for P(keyword | dest port) for ports 80, 25, and 21 after training on "inside" week 3 of the DARPA IDS evaluation data set [9].

Attribute	**r/n**	**Allowed Values**
80 (HTTP)	13/83650	Accept-Charset:
		Accept-Encoding:
		Accept-Language:
		Accept:
		Cache-Control:
		Connection:
		GET
		Host:
		If-Modified-Since:
		Negotiate:
		Pragma:
		Referer:
		User-Agent:
25 (SMTP)	34/142610	(*34 values...*)
21 (FTP)	11/16822	(*11 values...*)

The first line of Table 2 says that out of 83,650 TCP connections to port 80, that only 13 different keywords were observed. These keywords are listed in the third column.

The total score assigned to a TCP connection is the sum of the *tn/r* scores assigned by each of the five components. The keyword model might contribute more than one score because there could be more than one novel keyword.

4. THE 1999 DARPA IDS DATA SET

In 1998 and 1999, DARPA sponsored a project at Lincoln Labs to evaluate intrusion detection systems [9]. They set up a simulated local area network with a variety of different hosts and a simulated Internet connection and attacked it with a variety of published exploits. For each of the two tests, a number of intrusion detection systems were evaluated on their ability to detect these attacks given the network traffic, daily file system dumps, audit logs, and BSM (Solaris system call) logs. Each evaluation had two phases. During the first phase, participants were given data to develop their systems that included both attack

free periods and labeled attacks (time, victim IP address, and description of the attack). During the second phase, about six months later, the participants were given new data sets with unlabeled attacks, some of them new, and were rated by the number of detections as a function of false alarm rate. After the evaluation, the data sets and results were made available to other researchers in intrusion detection.

The 1999 evaluation includes four real "victim" machines, running unpatched Sun Solaris 2.5.1, SunOS 4.1.4, Linux 2.0.27, and Windows NT 4.0, connected to a simulated local area network and Internet through a real Cisco router over Ethernet. The IDS data includes audit logs and daily dumps of the operating system files on each victim machine, Solaris BSM (system call) logs, and two network traffic sniffer logs, one "inside" between the router and victims, and the other "outside" between the router and the Internet. The first phase consisted of 3 weeks of data. Weeks 1 and 3 contained no attacks, and were made available for training anomaly detection systems. Week 2 contained labeled attack data for testing. The second phase consisted of ten days (weeks 4 and 5) containing 201 instances of 58 different attacks, about half of which were novel with respect to the first phase. Attacks were taken mostly from published sources such as security mailing lists and cracker-oriented web sites. A few were developed in-house for known vulnerabilities for which exploit code was not available.

Eight organizations submitted 18 intrusion detection systems for evaluation. An attack is counted as detected if the IDS correctly identifies the IP address of the victim or attacker and the time of any portion of the attack interval within 60 seconds. The IDS was also required to assign a score to each alarm so that the number of attacks and false alarms could be varied by discarding alarms below a threshold. Duplicate detections of the same attack are counted only once, but every false alarm is counted. The top results reported by [9, Table 6] at a false alarm rate of 10 per day (100 total) are shown in Table 3.

Table 3. Top results of the 1999 DARPA IDS evaluation at 10 false alarms per day [9, Table 6].

System	**Detections**
Expert 1	85/169 (50%)
Expert 2	81/173 (47%)
Dmine	41/102 (40%)
Forensics	15/27 (55%)

The systems were evaluated on the percentage of attacks detected out of those they were designed to detect. None were designed to detect all attacks. Some only examined a subset of the data (e.g. network traffic or file system dumps), or a subset of the victims (e.g. only Solaris BSM data or NT audit logs). Some were designed to detect only certain types of attacks (probes, DOS, R2L, or U2R). Some used only signature detection, and were therefore not designed to detect attacks not present in the training data during week 2, although the top systems used a combination of signature and anomaly detection.

Lippman reports that 77 of the 201 attack instances were poorly detected, in that no system detected more than half of the instances of that type [9, Table 4]. If we take the best results for each of the 21 attack types, then only 15 instances (19%) were detected.

To reduce the complexity of our experiments, we examined only the "inside" network traffic logs. This detects attacks from internally compromised hosts, but misses external attacks on the router. We trained the system on 7 days of attack free traffic from week 3, which consists of 2.9 GB of *tcpdump* files. We tested on 9 days of traffic during weeks 4 and 5, about 6.0 GB of data. One day of inside traffic is missing (week 4, day 2), leaving 189 attacks, although 9 of them (*selfping*, *snmpget*, and *ntfsdos*) generate no evidence in the data we used. This leaves 180 detectable attacks, of which 67 were poorly detected. We did not make use of the labeled attack data from week 2.

5. EXPERIMENTAL RESULTS

We evaluated PHAD and ALAD by running them at the same time on the DARPA IDS evaluation data set and merging the results. Each system was trained on week 3 (7 days, attack free) and evaluated on the 180 detectable labeled attacks from weeks 4 and 5. To merge the results, we set the two thresholds so that equal numbers of alarms were taken from both systems, and so that there were 100 total false alarms (10 per day including the missing day) after removing duplicate alarms. An alarm is considered a duplicate if it identifies the same IP address and the same attack time within 60 seconds of a higher ranked alarm from either system. We chose 60 seconds because DARPA criteria allows a detection to be counted if the time is correctly identified within 60 seconds of any portion of the attack period. Also, to be consistent with DARPA, we count an attack as detected if it identifies any IP address involved in the attack (either target or attacker). Multiple detections of the same attack (that remain after removing duplicates) are counted only once, but all false alarms are counted.

In Table 4 we show the results of this evaluation. In the column labeled *det* we list the number of attacks detected out of the number of detectable instances, which does not include missing data (week 4, day 2) or the three attack types (*ntfsdos*, *selfping*, *snmpget*) that generate no inside traffic. Thus, only 180 of the 201 attack instances are listed.

In the last column of Table 4, we describe the PHAD and ALAD anomalies that led to the detection, prior to removing duplicate alarms. For PHAD, the anomaly is the packet header field that contributed most to the overall score. For ALAD, each of the anomalous components (up to 5) are listed. Based on these descriptions, we adjusted the number of detections (column *det*) to remove simulation artifacts and coincidental detections, and to add detections by Ethernet address rather than IP address, which would not otherwise be counted by DARPA rules. The latter case occurs for *arppoison*, in which PHAD detects anomalous Ethernet addresses in non-IP packets. *Arppoison* disrupts network traffic by sending spoofed responses to ARP-who-has requests from a compromised local host so that IP addresses are not correctly resolved to Ethernet addresses.

The two coincidences are *mscan* (an anomalous Ethernet address, overlapping an *arppoison* attack), and *illegalsniffer* (a TCP checksum error). *Illegalsniffer* is a probe by a compromised local host being used to sniff traffic, and is detectable only in the simulation because it makes reverse DNS lookups to resolve sniffed IP addresses to host names. Because the attack is prolonged, and because all of the local hosts are victims, coincidences are likely.

Table 4. Attacks in the 1999 DARPA IDS data set [9], and the number detected (*det*) out of the total number in the available data. Detections are for merged PHAD and ALAD at 100 total false alarms, after removing coincidences and simulation artifacts (TTL field) and adding detections by Ethernet address (*arppoison*). Attacks listed do not include the 12 attacks in week 4 day 2 (missing data) or 9 attacks that leave no evidence in the inside network traffic (*selfping*, *snmpget*, and *ntfsdos*). Hard to detect attacks (identified by *) are those types which were detected no more than half of the time by any of the 18 original participants [9, Table 4]. Attack descriptions are due to [8].

Type	**Attack and description (* = hard to detect)**	**Det**	**How detected**
Probe	illegalsniffer - compromised local host sniffs traffic	0/2	(1 coincidental TCP checksum error)
Probe	ipsweep (clear) - ping random IP addresses	1/4	1 Ethernet packet size = 52, (1 TTL = 253)
Probe	*ipsweep (stealthy - slow scan)	0/3	(2 TTL = 253)
Probe	*ls - DNS zone transfer	0/2	
Probe	mscan - test multiple vulnerabilities	1/1	1 dest IP/port, flags (1 coincidental Ethernet dest)
Probe	ntinfoscan - test multiple NT vulnerabilities	2/3	2 HTTP "HEAD", 1 FTP "quit", 1 "user", TCP RST, (2 TTL)
Probe	portsweep (clear) - test multiple ports	1/4	1 FIN without ACK, (1 TTL)
Probe	*portsweep (stealthy - slow scan)	2/11	2 FIN without ACK, (5 TTL)
Probe	*queso - malformed packets fingerprint OS	3/4	2 FIN without ACK (1 TTL)
Probe	*resetscan - probe with RST to hide from IDS	0/1	
Probe	satan - test multiple vulnerabilities	2/2	2 HTTP/ 1 SMTP "QUIT", finger /W, IP length, src IP, (TTL)
DOS	apache2 - crash web server with long request	3/3	3 source IP, 1 HTTP "x" and flags, TCP options in reply
DOS	*arppoison - spoofed replies to ARP-who-has	3/5	3 Ethernet src/dest address (non-IP packet)
DOS	back - crash web server with "GET /////..."	0/4	
DOS	crashiis - crash NT webserver	5/7	4 source IP address, 1 unclosed TCP connection
DOS	*dosnuke - URG data to NetBIOS crashes Windows	4/4	3 URG pointer, 4 flags = UAPF
DOS	land - identical src/dest addr/ports crashes SunOS	0/1	
DOS	mailbomb - flood SMTP mail server	3/3	3 SMTP lowercase "mail" (1 TTL = 253)
DOS	neptune - SYN flood crashes TCP/IP stack	0/4	(2 TTL = 253)
DOS	pod (ping of death) - oversize IP pkt crashes TCP/IP	4/4	4 IP fragment pointer
DOS	processtable - server flood exhausts UNIX processes	1/3	1 source IP address
DOS	smurf - reply flood to forged ping to broadcast address	1/5	1 source IP address (2 TTL)
DOS	syslogd - crash server with forged unresolvable IP	0/4	
DOS	*tcpreset - local spoofed RST closes connections	1/3	1 TCP connection not opened or closed
DOS	teardrop - IP fragments with gaps crashes TCP/IP stack	3/3	3 frag ptr
DOS	udpstorm - echo/chargen loop flood	2/2	2 UDP checksum error
DOS	*warezclient - download illegal files by FTP	1/3	1 source IP address
DOS	warezmaster - upload illegal files by FTP	1/1	1 source IP address
R2L	dict (guess telnet/ftp/pop) - dictionary password guessing	3/7	2 FTP "user", 1 dest IP/port (POP3), 1 src IP
R2L	framespoofer - trojan web page	0/1	
R2L	ftpwrite - upload "+ +" to .rhosts	0/2	
R2L	guest - simple password guessing	0/3	
R2L	httptunnel - backdoor disguised as web traffic	0/2	
R2L	imap - mailbox server buffer overflow	0/2	
R2L	named - DNS nameserver buffer overflow	0/3	
R2L	*ncftp - FTP server buffer overflow	4/5	4 dest IP/port, 1 SMTP "RSET", 3 auth "*xxxx*,25"
R2L	*netbus - backdoor disguised as SMTP mail traffic	2/3	2 source IP address, (3 TTL)
R2L	*netcat - backdoor disguised as DNS traffic	2/4	1 src/dest IP, (1 TTL)
R2L	phf - exploit bad Apache CGI script	2/3	2 source IP, 1 null byte in HTTP header
R2L	ppmacro - trojan PowerPoint macro in web page	1/3	1 source IP (and TTL)
R2L	sendmail - SMTP mail server buffer overflow	2/2	2 source IP address, 2 global dest IP, 1 "Sender:"
R2L	*sshtrojan - fake ssh client steals password	1/3	1 source IP address
R2L	xlock - fake screensaver steals password	0/3	
R2L	xsnoop - keystrokes intercepted on open X server	0/3	
U2R	anypw - NT bug exploit	0/1	
U2R	casesen - NT bug exploit	2/3	2 FTP upload (dest IP/port 20, flags, FTP "PWD"), (1 TTL)
U2R	eject - UNIX *suid root* buffer overflow	1/2	1 FTP upload (src IP, flags)
U2R	fdformat - UNIX *suid root* buffer overflow	2/3	2 FTP upload (src IP, flags, FTP "STOR")
U2R	ffbconfig - UNIX *suid root* buffer overflow	1/2	1 SMTP source IP address (email upload)
U2R	*loadmodule - UNIX trojan shared library	0/2	
U2R	*perl - UNIX bug exploit	0/4	
U2R	ps - UNIX bug exploit	0/3	
U2R	*sechole - NT bug exploit	1/2	1 FTP upload (dest IP/port, flags, FTP "STOR"), (1 TTL)
U2R	*sqlattack - database app bug, escape to user shell	0/2	
U2R	xterm - UNIX *suid root* buffer overflow	1/3	1 FTP upload (source IP, dest IP/port)
U2R	yaga - NT bug exploit	1/4	1 FTP upload (src IP, FTP lowercase "user")
Data	secret - copy secret files or access unencrypted	0/4	
Total		**70/180**	**(39%) ; and 23/65 (35%) of hard to detect attacks**

There are 25 attacks detected by anomalous TTL values in PHAD, which we believe to be simulation artifacts. TTL (time to live) is an 8-bit counter decremented each time an IP packet is routed in order to expire packets to avoid infinite routing loops. Although small TTL values might be used to elude an IDS by expiring the packet between the IDS and the target [14], this was not the case because the observed values were large, usually 126 or 253. Such artifacts are unfortunate, but probably inevitable, given the difficulty of simulating the Internet [4]. A likely explanation for these artifacts is that the machine used to simulate the attacks was a different real distance from the inside sniffer than the machines used to simulate the background traffic. We did not count attacks detected solely by TTL.

After adjusting the number of detections in the *det* column, we detect 70 of 180 (39%) of attacks at 100 false alarms. Among the poorly detected attacks [9, Table 4], we detect 23 of 77 (30%), or 23 of 65 (35%) of the 180 detectable attacks in our data set, almost the same rate as for the well detected attacks. This is a good result because an anomaly detection system such as ours would not be used by itself, but rather in combination with other systems such as those in the original evaluation that use signature detection or host based techniques. In order for the combination to be effective, there must be a significant non-overlap, and our results show that. We should also point out that when we developed PHAD and ALAD, we did so with the goal of improving the overall number of detections rather than just the poorly detected attacks.

5.1. Classification of Anomalies

It is interesting to compare the description of each attack in Table 4 with the anomaly that led to its detection. In many cases, the two seem unrelated, or at the very least, the anomaly could plausibly occur in benign traffic. For example, there are several buffer overflow attacks that exploit poorly written programs that use *C* functions such as *gets()* or *strcpy()* to write into a buffer (fixed sized array of characters) without checking the length of the input. An attack will typically contain a long string that overflows the buffer and overwrites the return address on the stack of the target machine. When the executing function returns, it instead jumps to an address supplied by the attacker, typically a string of machine code supplied as part of the same attacking string. The code is executed with the privilege level of the target (often *root*), usually to open a shell or plant a backdoor.

We might expect a buffer overflow attack to generate anomalies in the form of long strings of executable code where we would normally expect to find short strings of text. However, we do not have the appropriate attributes to detect this type of anomaly. Usually it is something else that makes the attack stand out. For example, *ncftp* and *sendmail* are caught because they use perfectly legal, but seldom used keywords, or the keyword is lower case when most clients use upper case. Many of the U2R attacks (which a network IDS would normally miss) are detected because the attacker uploads the exploit program using an FTP server that is normally used only for downloads. Other attacks are detected because the source IP address is new to the host or server under attack.

Even when the anomaly is related to the attack, it usually does not tell us much about it. For example, *portsweep* and *queso* are detected by identical anomalies, a TCP FIN packet (request to close connection) without an accompanying ACK flag set. Normally ACK would be set in every packet except the first one in a TCP connection (a SYN packet, or request to open) to acknowledge the previous packet. However, the reasons for the anomalies are quite different. *Portsweep*, which probes for open ports, uses FIN packets to prevent the connection attempt from being logged by the target. *Queso* tests the response by the target to unusual or malformed packets to determine what operating system it is running.

We can classify anomalies into five categories:

1. Learned signatures - attempts to exploit bugs in the target
2. Induced anomalies - symptoms of a successful attack.
3. Evasion anomalies - attempts to elude the IDS.
4. Attacker errors - bugs in the attacking program.
5. User behavior - unexpected client addresses.

First, a security vulnerability is an error, whether in software design, coding, or system configuration. Attackers exploit these errors. Because it is impossible to test software completely, some errors will always be discovered in software after it has been delivered and put to use. The errors that are least likely to be discovered and patched are those that occur least often in normal traffic. Thus, the input required to invoke the error is likely to be unusual. We call such input a *learned signature* anomaly. Examples include the urgent data in *dosnuke* or the fragmented IP packets in *pod* and *teardrop*. Most of the time, the IDS learns only a part of the signature. For example, it does not learn that the urgent data must be directed to the NetBIOS port, or that the IP fragments must have gaps or reassemble to more than 64K bytes.

Second, sometimes the anomalous input is missed, but we can observe anomalous behavior from the target after a successful attack. This is similar to Forrest's host based anomaly detection technique, except that the symptoms are observed in the output of the target, rather than in the system calls that it makes. Examples include the unusual TCP options generated by *apache2* and the anomalous destination Ethernet addresses resulting from *arppoison* victims. We call these *induced* anomalies.

The third type of anomaly we discovered is one in which the attacker tries to exploit errors in the IDS to hide the attack, for example, FIN scanning by *portsweep* to prevent server accesses from being logged. If the attempt backfires, as it does in this case, we call it an *evasion* anomaly.

Fourth, the attacker might introduce arbitrary variations in the data, which we can consider to be errors in the attacking software. Examples include garbage in the *apache2* and *phf* attacks, the use of lowercase commands in *dict*, *sendmail*, and *ntinfoscan*, and the UDP checksum error in *udpstorm*. We call these *attacker error* anomalies. Most provide no clues as to the nature of the attack.

Finally, *behavioral* anomaly detection models users rather than software. Most of the U2R attacks are discovered because the exploit software is uploaded on an FTP server normally used for downloads. Many R2L attacks are discovered because the client IP address is not one of the usual users.

Table 5 shows the number of attacks detected in each anomaly category, based on the anomaly description. The totals are more than 100% because some attacks are detected by more than one type of anomaly. For example, all three *apache2* detections have an unusual client IP address, but one also has the unusual keyword "x" (an error), and another induces unusual TCP options

in the reply. It is not always clear whether an anomaly is an error or part of the signature, so we use the principle that if the attack could be easily changed to hide it, then it is an error. For example, one *apache2* attack is a malformed HTTP request:

```
x
User-Agent: sioux
User-Agent: sioux
User-Agent: sioux
```
... (repeated thousands of times)

But the other instances, which do not generate the same anomaly, replace "x" with a normal "GET / HTTP/1.1". The attack succeeds either way, so we consider "x" to be an error instead of part of the signature.

Table 5. Detected attacks classified by the type of anomaly, and the fraction of the 70 total detected attacks detected this way.

Anomaly	**Det/70**	**Attacks Detected**
Learned Signature	24 (34%)	PROBE: ipsweep, mscan, 2 ntinfoscan, 3 queso, 2 satan; DOS: crashiis, 4 dosnuke, 4 pod, 3 teardrop; R2L: ncftp, 2 sendmail
Induced	5 (7%)	DOS: apache2, 3 arppoison, tcpreset
Evasion	3 (4%)	PROBE: 3 portsweep
Attacker Error	10 (14%)	DOS: apache2, 3 mailbomb, 2 udpstorm; R2L: 2 dict, phf; U2R: yaga
User Behavior	38 (54%)	PROBE: mscan; DOS: 3 apache2, 5 crashiis, mailbomb, processtable, smurf, warazclient, warezmaster; R2L: dict, mailbomb, 4 ncftp, 2 netbus, 2 netcat, 2 phf, ppmacro, 2 sendmail, sshtrojan; U2R: 2 casesen, 2 fdformat, ffbconfig, sechole, xterm, yaga

5.2. Analysis of False Alarms

Table 6 shows the causes of the 100 false alarms in the evaluation. A few alarms were generated by more than one anomaly, so the total is more than 100.

Table 6. The 100 top false alarms detected by PHAD and ALAD.

Anomaly	**False alarms**
TCP source IP address	35
Keyword (7 SMTP, 4 FTP, 3 auth, 2 HTTP)	16
TTL (time to live, simulation artifact)	9
TCP checksum (simulation artifact)	8
Outgoing TCP connection on server port	7
TOS (type of service)	7
Urgent data pointer or URG flags	7
Bad TCP connection (3 no SYN, no FIN, RST)	5
Destination address/port	5
Packet size (Ethernet, IP, UDP)	3
Other (2 IP fragments, 2 TCP options)	4

From table 6 we can see that the types of anomalies that generate false alarms are generally the same types that detect attacks. For example, source and destination addresses and outgoing TCP connections (FTP upload) which are responsible for the behavioral detections (about half of the total) are also responsible for almost half of the false alarms. Novel keywords, which detect most attacker errors (14% of the total) are responsible for 16 false alarms, about the same percentage. The errors occur in mostly the same protocols as well, SMTP, FTP, and HTTP. This is unfortunate, because it gives us no easy way to distinguish between real attacks and false alarms.

A few false alarms are not responsible for any detections, in particular TCP checksums and TOS (type of service). We thought it suspicious that no checksum errors whatsoever occur in training (Table 1), but many occur during testing. On inspection, we found that the errors are the result of short IP packets that fragment the TCP header. (These also generate IP fragment anomalies, but the checksum contributes the most to the score). There is no good reason for creating such short fragments, except to elude the IDS [14], but no attack is labeled. Perhaps the DARPA team intended to make the problem more difficult, or perhaps they just made an error and misconfigured one of the hosts with a small MTU (message transmission unit). We also found another example of unusual traffic, TCP streams composed of one byte packets with every other packet missing. Again, this occurs in the test data and not in the training data, but there is no attack (and no detection by PHAD or ALAD).

5.3. Coverage of PHAD and ALAD

From Table 4 we can classify each attack by whether it was detected by PHAD, ALAD, or both. The results are shown in Table 7.

Table 7. Contributions of PHAD and ALAD to the 70 attack detections, grouped by attack category.

Type	**Total Detected**	**By PHAD only**	**By ALAD only**	**By Both**
Probe	12/37	7	4	1
DOS	32/59	17	15	0
R2L	17/49	0	17	0
U2R/Data	9/35	0	9	0
Total	**70/180**	**24**	**45**	**1**

There is almost no overlap between PHAD and ALAD. The only attack detected by both is one instance of *satan*, in which PHAD detects an anomalous IP packet length and ALAD detects novel HTTP, SMTP, and *finger* keywords. PHAD detects mostly probes and DOS attacks that exploit lower level network protocols: *ipsweep, portsweep, queso, arppoison, dosnuke, pod, smurf, teardrop, udpstorm.* ALAD detects user behavior anomalies (all U2R and some R2L), and attempts to exploit errors in application servers. These include most R2L attacks, as well as DOS attacks aimed at servers (*apache2, crashiis, mailbomb, processtable*), and probes for server vulnerabilities (*satan, mscan, ntinfoscan*).

We can combine PHAD and ALAD to improve coverage because they work in different ways to detect different attacks, but the merge is not perfect. By themselves, PHAD detects 54 attacks when the TTL field is excluded (for both detections and false alarms), and ALAD detects 59. However, when they are merged, we must raise the alarm thresholds in order to keep the total false alarms to 100 (10 per day). This results in some true detections being discarded, so the total is only 70 (or 73 excluding TTL false alarms), not 54 + 59 = 113.

5.4. Overlap with SPADE

From Table 4 we note that we detect 23 of the 77 hard to detect attacks from the original 1999 evaluation, whereas the original participants detected at most 15 of these. Also, since some of these systems detect more attacks than we do (Table 3), we see that there is a significant non-overlap between them and PHAD/ALAD. As we noted in section 5.3, it is this property of non-overlap that allows PHAD and ALAD to be combined with each other to increase total coverage.

In this section, we evaluate SPADE [18], which was not one of the original DARPA participants, to see whether it overlaps PHAD/ALAD. SPADE is a user behavioral anomaly detector which models port numbers and IP addresses on incoming TCP SYN (request to open) packets, similar to ALAD. However, since SPADE is intended for actual use, there is no explicit training period. It continues training throughout the entire run period and assigns an anomaly score to each TCP connection based on the distribution over the entire run (but with greater weight given to recent history). Thus, when a second instance of an attack occurs, it is likely to receive a lower score due to being trained on the first instance.

To test SPADE, we ran version 092200.1 as a plugin to SNORT 1.7 [15] on the same data used for PHAD/ALAD ("inside" weeks 3, 4, and 5) in each of SPADE's 4 user selectable models (Table 8). All other parameters were set to their default values. SPADE, like PHAD/ALAD, reports an anomaly score, so after discarding alarms during week 3, we set a threshold allowing 100 false alarms (10 per day) during the attack period.

Table 8. Attacks detected by SPADE at 10 false alarms per day. Attacks not detected by PHAD/ALAD are shown in **bold**.

SPADE Model	**Detections/180**
0. Bayes approximation of P(src IP/port, dest IP/port)	7 (1/3 apache2, 1/5 smurf, 1/4 **perl**, 1/1 mscan, 1/5 arppoison, 1/2 **illegalsniffer**, 1/4 **syslogd**)
1. P(src IP/port, dest IP/port)	1 (1/1 mscan)
2. P(src IP, dest IP/port)	8 (1/4 **ps**, 1/4 **neptune**, 1/2 eject, 1/4 yaga, 1/4 **perl**, 1/3 fdformat, 1/4 queso, 1/1 mscan)
3. P(dest IP/port)	7 (as in (2) except yaga)

SPADE models 0, 2, and 3 give the best performance. In each of these models, 3 out of the 7 or 8 detections are not detected by PHAD/ALAD. Again, we have a significant non-overlap, suggesting that using a combined IDS would result in better coverage than any system by itself.

It is important to stress that we cannot compare PHAD/ALAD directly to SPADE or to the original DARPA participants with regard to the number of attacks detected. SPADE lacks an explicit training period. Preliminary experiments with PHAD suggest that under similar conditions it would miss about half of the attacks it now detects. Also, we used a fixed threshold rather than the many variations of adaptive thresholds offered by SPADE, some of which might have improved the results. We also cannot compare PHAD/ALAD to the DARPA participants because their evaluation was blind (no access to the test data), because they were designed to detect different sets of attacks with different data, because some use signature detection, and because their evaluation criteria, although similar to ours, was not identical.

6. IMPLEMENTATION

Our implementation of PHAD processes 2.9 gigabytes of training data and 4.0 gigabytes of test data in 364 seconds (310 user + 54 system), or 95,900 packets per second on a Sparc Ultra 60 with a 450 MHz 64-bit processor, 512 MB memory and 4 MB cache. The overhead is 23 seconds of CPU time per simulated day, or 0.026% at the simulation rate. The wall time in our test was 465 seconds (78% usage), consisting of 165 seconds of training (77,665 packets per second) and 300 seconds of testing (73,560 packets per second). The PHAD model uses negligible memory: 34 fields times 32 pairs of 4-byte integers to represent the bounds of each cluster, or 8 kilobytes total. The program is about 400 lines of C++ code.

ALAD was implemented in two parts, a 400 line C++ program to reassemble TCP packets into streams, and a 90 line Perl script to analyze them. Reassembly of 6.9 GB of data on a 750 MHz PC with 256 MB memory running Windows Me took 17 minutes, although no attempt was made at optimization, as this part only had to be run once prior to developing the second part. The output of part 1 is 20 MB of training data and 40 MB of test data as two text files. Because of the 99% reduction in data, the Perl script which implements ALAD runs in only 60 seconds on the same PC.

ALAD stores all attributes and values (in Perl hashtables), but there is no need to do so. Memory could be reduced to a few kilobytes by storing only the attributes with large *r/n* and hashing their associated values.

7. CONCLUDING REMARKS

We investigated network anomaly detection and saw that most systems are rule based with the possible exception of IP addresses and ports. We proposed extending the adaptive model to other parts of the protocol, and described two techniques for doing so. PHAD is a primitive packet model that knows very little about the network or protocols it is modeling, or about which fields might prove to be useful. ALAD combines traditional user modeling of TCP services (ports and addresses) with a simple generic text-based model of application protocols. Both models are nonstationary. They assume that the probability of an event depends on the time since it last occurred, regardless of any prior occurrences. We believe (although we cannot show) that this model is superior to one based on frequency counts. We also saw that memory requirements are small, because we only need to model very low probability events. We saw that modeling both user behavior and protocols (for evidence of software errors) increases coverage to almost twice that of user modeling alone.

Out of 180 attacks in the DARPA IDS evaluation data set, PHAD and ALAD detect 70 (39% recall), with 100 false alarms (41% precision). Although these models use pure anomaly detection, they perform almost as well (by our measure) as systems that combined both signature and anomaly detection in the 1999 blind evaluation, even though anomaly detection by itself is a harder problem because the attack signatures are unknown More importantly, there is a significant non-overlap between our IDS and other systems, so that they can be combined to increase coverage. We detect 23 (by our measure) of the 77 poorly detected attacks in the 1999 evaluation, compared to at

most 15 for the original evaluation participants. Likewise, there is a non-overlap with SPADE, which detects attacks that we miss.

We found five categories of anomalies. By decreasing frequency, these are:

- User behavior anomalies, as detected by traditional systems, for example FTP uploads of U2R exploits on a server normally used only for downloads.
- Attempts to exploit bugs in poorly tested features, such as the malformed IP fragments of *pod* and *teardrop*.
- Bugs in the attacking code, such as UDP checksum errors in *udpstorm*, or using lowercase text in *sendmail* and *dict.*
- Induced anomalies in the target after a successful attack, such as the unusual TCP options in reply to *apache2*.
- Unsuccessful attempts to elude the IDS, such as FIN scanning by *portsweep*.

Compared to signature detection, anomaly detection, which is a harder problem, has lower detection rates and higher false alarm rates, and hence less practical. Also, unlike signature detection, anomalies often appear completely unrelated to the attacks that generated them, which makes it difficult to display alarm messages that even an expert in network protocols could understand. Devising more sophisticated attributes for the learning algorithm could enhance alarm explanation as well as detection rates.

We have made some progress on one aspect of the anomaly detection problem, that of modeling. However, there are still two outstanding problems. First, there is the combinatorial explosion problem of combining attributes. In ALAD, we use combinations of the form P(x,y) (e.g. P(dest IP, dest port)), and P(x|y) (e.g. P(src IP|dest IP)), but these were *ad-hoc*. We *knew* that certain combinations would give better results. Our goal is for the IDS to figure out which combinations to use from among the exponential number of possibilities. We are currently developing a system called LERAD (LEarning Rules for Anomaly Detection) which addresses this issue. It quickly finds good (high *n/r*) candidate conditional rules using a small sample of the training data, and it detects more attacks than our current system, but it is currently an off-line algorithm, requiring two passes over the training data.

Second is the parsing problem. We had to hard code rules into PHAD (field sizes and offsets) and ALAD (words, lines, and headers) to parse the input into attributes. Our goal is for the IDS to learn new protocols as they are encountered, but our system would fail unless the new protocol had the same syntactic structure as the existing ones. Ideally, the system should learn the lexical and syntactic structure. This problem remains unsolved.

Acknowledgments

This research is partially supported by DARPA (F30602-00-1-0603). We also thank the anonymous reviewers for their comments.

References

[1] Anderson, Debra, Teresa F. Lunt, Harold Javitz, Ann Tamaru, Alfonso Valdes, "Detecting unusual program behavior using the statistical component of the Next-generation Intrusion Detection Expert System (NIDES)", Computer Science Laboratory SRI-CSL 95-06 May 1995. http://www.sdl.sri.com/papers/5/s/5sri/5sri.pdf

[2] Bell, Timothy, Ian H. Witten, John G. Cleary, "Modeling for Text Compression", ACM Computing Surveys (21)4, pp. 557-591, Dec. 1989.

[3] Barbará, D., N. Wu, S. Jajodia, "Detecting Novel Network Intrusions using Bayes Estimators", First SIAM International Conference on Data Mining, 2001, http://www.siam.org/meetings/sdm01/pdf/sdm01_29.pdf

[4] Floyd, S. and V. Paxson, "Difficulties in Simulating the Internet." IEEE/ACM Transactions on Networking Vol. 9, no. 4, pp. 392-403, Aug. 2001. http://www.icir.org/vern/papers.html

[5] Forrest, S., S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes", Proceedings of 1996 IEEE Symposium on Computer Security and Privacy. ftp://ftp.cs.unm.edu/pub/forrest/ieee-sp-96-unix.pdf

[6] Ghosh, A.K., A. Schwartzbard, M. Schatz, "Learning Program Behavior Profiles for Intrusion Detection", Proceedings of the 1st USENIX Workshop on Intrusion Detection and Network Monitoring, April 9-12, 1999, Santa Clara, CA. http://www.cigital.com/~anup/usenix_id99.pdf

[7] M. Handley, C. Kreibich and V. Paxson, "Network Intrusion Detection: Evasion, Traffic Normalization, and End-to-End Protocol Semantics", Proc. USENIX Security Symposium, 2001.

[8] Kendall, Kristopher, "A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems", Masters Thesis, MIT, 1999.

[9] Lippmann, R., et al., "The 1999 DARPA Off-Line Intrusion Detection Evaluation", Computer Networks 34(4) 579-595, 2000.

[10] Mahoney, M., P. K. Chan, "PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic", Florida Tech. technical report 2001-04, http://cs.fit.edu/~tr/

[11] Neumann, P., and P. Porras, "Experience with EMERALD to DATE", Proceedings 1st USENIX Workshop on Intrusion Detection and Network Monitoring, Santa Clara, California, April 1999, 73-80, http://www.csl.sri.com/neumann/det99.html

[12] Paxson, Vern, "Bro: A System for Detecting Network Intruders in Real-Time", Lawrence Berkeley National Laboratory Proceedings, 7'th USENIX Security Symposium, Jan. 26-29, 1998, San Antonio TX, http://www.usenix.org/publications/library/proceedings/sec98/paxson.html

[13] Paxson, Vern, and Sally Floyd, "The Failure of Poisson Modeling", IEEE/ACM Transactions on Networking (3) 226-244, 1995.

[14] Ptacek, Thomas H., and Timothy N. Newsham, "Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection", January, 1998, http://www.robertgraham.com/mirror/Ptacek-Newsham-Evasion-98.html

[15] Roesch, Martin, "Snort - Lightweight Intrusion Detection for Networks", Proc. USENIX Lisa '99, Seattle: Nov. 7-12, 1999.

[16] Sasha/Beetle, "A Strict Anomaly Detection Model for IDS", Phrack 56(11), 2000, http://www.phrack.org

[17] Sekar, R., M. Bendre, D. Dhurjati, P. Bollineni, "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors". Proceedings of the 2001 IEEE Symposium on Security and Privacy.

[18] SPADE, Silicon Defense, http://www.silicondefense.com/software/spice/