PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic

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

Florida Institute of Technology Technical Report CS-2001-04

Abstract

We describe an experimental packet header anomaly detector (PHAD) that learns the normal range of values for 33 fields of the Ethernet, IP, TCP, UDP, and ICMP protocols. On the 1999 DARPA off-line intrusion detection evaluation data set (Lippmann et al. 2000), PHAD detects 72 of 201 instances (29 of 59 types) of attacks, including all but 3 types that exploit the protocols examined, at a rate of 10 false alarms per day after training on 7 days of attack-free internal network traffic. In contrast to most other network intrusion detectors and firewalls, only 8 attacks (6 types) are detected based on anomalous IP addresses, and none by their port numbers. A number of variations of PHAD were studied, and the best results were obtained by examining packets and fields in isolation, and by using simple nonstationary models that estimate probabilities based on the time since the last event rather than the average rate of events.

1. Introduction

Most network intrusion detection systems (IDS) that use anomaly detection look for anomalous or unusual port number and IP addresses, where "unusual" means any value not observed in training on normal (presumably nonhostile) traffic. They use the firewall paradigm; a packet addressed to a nonexistent host or service must be hostile, so we reject it. The problem with the firewall model is that attacks addressed to legitimate services will still get through, even though the packets may differ from normal traffic in ways that we could detect.

Sasha and Beetle (2000) argue for a strict anomaly model to detect attacks on the TCP/IP stack. Attacks such as *queso, teardrop,* and *land* (Kendall 1999) exploit vulnerabilities in stacks that do not know how to handle unusual data. *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.

Horizon (1998) and Ptacek and Newsham (1998) describe techniques for attacking or evading an application layer IDS that would produce anomalies at the layers below. 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 TTLs that expire between the target and IDS, and so on.

A common theme of attacks is that they exploit software bugs, either in the target or in the IDS. Bugs are inevitable in all software, so we should not be surprised to find them in attacking software as well. For example, in the DARPA intrusion detection data set (Lippmann et al. 2000), we found that *smurf* and *udpstorm* do not compute correct ICMP and UDP checksums. *Smurf* is a distributed ICMP echo reply flood, initiated by sending an echo request (ping) to a broadcast address with the spoofed source IP address of the target. *UDPstorm* sets up a flood between the *echo* and *chargen* servers on two victims by sending a request to one with the spoofed source address of the other.

A fourth source of anomalies might be the response from the victim of a successful attack. Forrest et al. (1996) has shown that victimized programs make unusual sequences of system calls, so we might expect them to produce unusual output as well. In any case, there are a wide variety of anomalies across many different fields of various network protocols, so an IDS ought to check them all.

The rest of the paper is organized as follows. In Section 2, we discuss related work in anomaly detection. In Section 3, we describe a packet header anomaly detector (PHAD) that looks at all fields except the application payload. In Section 4, we train PHAD on attack-free traffic from the DARPA data set and test it with 201 simulated attacks. In Section 5, we simulate PHAD in an on-line environment where the training data is not guaranteed to be attack-free. In Section 6 we compare various PHAD models. In Section 7 we compare PHAD with the intrusion detection systems that were submitted to the DARPA evaluation in 1999. We summarize the results in section 8.

2. Related Work

Network intrusion detection systems like *snort* (2001) or *Bro* (Paxson, 1998) typically use signature detection, matching patterns in network traffic to the patterns of known attacks. This works well, but has the obvious disadvantage of being vulnerable to novel attacks. An alternative approach is anomaly detection, which models normal traffic and signals any deviation from this model as suspicious. The idea is based on work by Forrest et al. (1996), who found that most UNIX processes make (mostly) highly predictable sequences of system calls in normal use. When a server or *suid root* program is compromised (by a buffer overflow, for example), it executes code supplied by the attacker, and deviates from its normal calling sequence.

It is not possible to observe every possible legitimate pattern in training, so an anomaly detector requires some type of machine learning algorithm in order to generalize from the training set. Forrest uses an n-gram model, allowing any sequence as long as all subsequences of length n (about 3 to 6) have been previously observed. Sekar et al. (2000) uses a state machine model, where a state is defined as the value of the program counter when the system call is made, and allows any sequence as long as all of the state transitions have been observed in training. Ghosh et al. (1999) use a neural network trained to accept observed n-grams and reject randomly generated training sequences.

Network anomaly detectors look for unusual traffic rather than unusual system calls. ADAM (Audit Data and Mining) (Barbará, Wu, and Jajodia, 2001) is an anomaly detector 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 the probability that a packet belongs to some class (normal, known attack, or unknown) depends on the *a-priori* probability of the class, and the combined probabilities of a large collection of rules under the assumption that they are independent. ADAM has separate training modes and detection modes.

NIDES (Anderson et. al. 1995), 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 (Neumann and Porras, 1998), which includes host and network based signature detection for known attacks.

Spade is a *snort* (2001) plug-in that detects anomalies in network traffic. Like NIDES and ADAM, it is based on port numbers and IP addresses. 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.

3. Packet Header Anomaly Detection (PHAD)

We developed an anomaly detection algorithm (PHAD) that learns the normal ranges of values for each packet header field at the data link (Ethernet), network (IP), and transport/control layers (TCP, UDP, ICMP). PHAD does not currently examine application layer protocols like DNS, HTTP or SMTP, so it would not detect attacks on servers, although it might detect attempts to hide them from an application layer monitor like *snort* by manipulating the TCP/IP protocols.

An important shortcoming of all anomaly detection systems is that they cannot discern intent; they can only detect when an event is unusual, which may or may not indicate an attack. Thus, a system should have a means of ranking alarms by how unusual or unexpected they were, with the assumption that the rarer the event, the more likely it is to be hostile. If this assumption holds, the user can adjust the threshold to trade off between a high detection rate or a low false alarm rate. PHAD is based on the assumption that events that occur with probability p should receive a score of $1/p$.

PHAD uses the rate of anomalies during training to estimate the probability of an anomaly while in detection mode. If a packet field is observed n times with r distinct values, there must have been r "anomalies" during the training period. If this rate continues, the probability that the next observation will be anomalous is approximated by r/n. This method is probably an overestimate, since most anomalies probably occur early during training, but it is easy to compute, and it is consistent with the PPMC method of estimating the probability of novel events used by data compression programs (Bell, Witten, and Cleary, 1989).

To consider the dynamic behavior of real-time traffic, PHAD uses a nonstationary model while in detection mode. In this model, if an event last occurred t seconds ago, then the probability that it will occur in the next one second is approximated by $1/t$. Often, when an event occurs for the first time, it is because of some change of state in the network, for example, installing new software or starting a process

that produces a particular type of traffic. Thus, we assume that events tend to occur in bursts. During training, the first anomalous event of a burst is added to the model, so only one anomaly is counted. This does not happen in detection mode, so we discount the subsequent events by the factor *t*, the time since the previous anomaly in the current field. Thus each packet header field containing an anomalous value is assigned a score inversely proportional to the probability,

$$\text{score}_{\text{field}} = tn/r$$

Finally, we add up the scores to score the packet. Since each score is an inverse probability, we could assume that the fields are independent and multiply them to get the inverse probability of the packet. But they are not independent. Instead, we use a crude extension of the stationary model where we treat the fields as occurring sequentially. If all the *tn/r* are equal, then the probability of observing *k* consecutive anomalies in a nonstationary model is $(r/tn)(1/2)(2/3)(3/4)...((k-1)/k) = (1/k)r/tn$. This is consistent with the score *ktn/r* that we would obtain by summation. Thus, we assign a packet score of

$$\text{score}_{\text{packet}} = \Sigma_{i \in \text{ anomalous fields}}\, t_i n_i / r_i$$

where anomalous fields are the fields with values not found in the training model. In PHAD, the packet header fields range from 1 to 4 bytes, allowing 2^8 to 2^{32} possible values, depending on the field. It is not practical to store a set of 2^{32} values for two reasons. First, the memory cost is prohibitive, and second, we want to allow generalization to reasonable values not observed in the limited training data. The approach we have used is to store a list of ranges or clusters, up to some limit $C = 32$. If C is exceeded during training, then we find the two closest ranges and merge them. For instance, if we have the set {1, 3-5, 8}, then merging the two closest clusters yields {1-5, 8}. There are other possibilities, which we discuss in section 5, but clustering is appropriate for fields representing continuous values, and this method (PHAD-C32) gives the best results on the test data that we used.

PHAD examines 33 packet header fields, mostly as defined in the protocol specifications. Fields smaller than 8 bits (such as the TCP flags) are grouped into a single byte field. Fields larger than 4 bytes (such as the 6 byte Ethernet addresses) are split in half. The philosophy behind PHAD is to build as little protocol-specific knowledge as possible into the algorithm, but we felt it was necessary to compute the checksum fields (IP, TCP, UDP, ICMP), because it would be unreasonable for a machine learning algorithm to figure out how to do this on its own. Thus, we replace the checksum fields with their computed values (normally FFFF hex) prior to processing.

4. Experimental Results on the 1999 DARPA IDS Evaluation Data Set

We evaluated PHAD on the 1999 DARPA off-line IDS evaluation data set (Lippmann et al. 2000). The set consists of network traffic (*tcpdump* files) collected at two points, BSM logs, audit logs, and file system dumps from a simulated local network of an imaginary air force base over a 5 week period. The simulated system consists of four real "victim" machines running SunOS, Solaris, Linux, and Windows NT, a Cisco router, and a simulation of a local network with hundreds of other hosts and thousands of users and an Internet connection without a firewall. We trained PHAD on 7 days of attack free network traffic (week 3) collected from a sniffer between the router and victim machines (inside.tcpdump), and tested it on 9 days of traffic from the same point (weeks 4 and 5, except week 4 day 2, which is missing), during which time there were 201 attacks (183 in the available data). The attacks were mostly published exploits from rootshell.com and the *bugtraq* mailing list, and are described by (Kendall, 1999), with a

few developed in-house. The test set includes a list of all attacks, labeled with the victim IP address, time, type, and a brief description. Table 4.1 shows the PHAD-C32 model after training.

The model lists all values observed in training, clustered if necessary. The values *r/n* show the number of anomalies that occurred in training (resulting in an addition to the list) out of the total number of packets in which the field is present. It can be observed that port numbers and IP addresses are not among the fields with a small value of *r/n*, and so should not generate large anomaly scores.

Field name	r/n	Values
Ether Size	508/12814738	42 60-1181 1182...
Ether Dest Hi	9/12814738	x0000C0 x00105A x00107B...
Ether Dest Lo	12/12814738	x000009 x09B949 x13E981..
Ether Src Hi	6/12814738	x0000C0 x00105A x00107B...
Ether Src Lo	9/12814738	x09B949 x13E981 x17795A...
Ether Protocol	4/12814738	x0136 x0800 x0806 x9000
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 65463...
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 Src	293/12715589	12.2.169.104-12.20.180.101...
IP Dest	287/12715589	0.67.97.110 12.2.169.104-12.20.180.101...
TCP Src Port	3546/10617293	20-135 139 515...
TCP Dest Port	3545/10617293	20-135 139 515...
TCP Seq	5455/10617293	0-395954185 395969583-396150583...
TCP Ack	4235/10617293	0-395954185 395969584-396150584...
TCP Header Len	2/10617293	x50 x60
TCP Flg UAPRSF	9/10617293	x02 x04 x10...
TCP Window Sz	1016/10617293	0-5374 5406-10028 10069-10101...
TCP Checksum	1/10617293	xFFFF
TCP URG Ptr	2/10617293	0 1
TCP Option	2/611126	x02040218 x020405B4
UCP Src Port	6052/2091127	53 123 137-138...
UDP Dest Port	6050/2091127	53 123 137-138...
UDP Len	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 4.1. The PHAD-C32 model after training on week 3.

4.1. Top 20 alarms generated by PHAD-C32

The top 20 scoring alarms are shown in Table 4.2. Eight of these correctly identify an attack according to DARPA criteria, which requires the destination IP address of a packet involved in the attack (either the victim or a response to the attacker), and the time of the attack within 60 seconds. Another three alarms detect *arppoison* from anomalous ARP packets, but since these do not have an IP address, they do not meet the criteria for detection. The other 9 alarms are false positives (FP). The *most anomalous field* column shows which field contributes the most to the anomaly score, the value of that field, and the percentage contribution to the total score. The score is displayed on a logarithmic scale (using $0.1 \log_{10} (\Sigma\ tn/r) - 0.6$) to fit in the range 0 to 1. The machines under attack are on the net 172.16.112-118.xxx.

The router is 192.168.1.1, but since the sniffer is between the router and the local network, only inside attacks on it can be detected.

Date	Time	Dest. IP Addr.	Score	Det	Attack	Most anomalous field
04/06	08:59:16	172.016.112.194	0.748198	FP		TCP Checksum=x6F4A 67%
04/01	08:26:16	172.016.114.050	0.697083	TP	teardrop	IP Frag Ptr=x2000 100%
04/01	11:00:01	172.016.112.100	0.689305	TP	dosnuke	TCP URG Ptr=49 100%
03/31	08:00:28	192.168.001.030	0.664309	FP		IP TOS=x20 100%
03/31	11:35:13	000.000.000.000	0.664225	FP	(arppoison)	Ether Src Hi=xC66973 68%
03/31	11:35:18	172.016.114.050	0.653956	FP		Ether Dest Hi=xE78D76 57%
04/08	08:01:20	172.016.113.050	0.644237	FP		IP Frag Ptr=x2000 35%
04/05	08:39:50	172.016.112.050	0.639134	TP	pod	IP Frag Ptr=x2000 89%
04/05	20:00:27	172.016.113.050	0.628749	TP	udpstorm	UDP Checksum=x90EF 100%
04/09	08:01:26	172.016.113.050	0.626455	FP		TCP Checksum=x77F7 48%
04/05	11:45:27	172.016.112.100	0.626234	TP	dosnuke	TCP URG Ptr=49 100%
03/29	11:15:08	192.168.001.001	0.615842	TP	portsweep	TCP Flg UAPRSF=x01 99%
03/29	09:15:01	172.016.113.050	0.612167	TP	portsweep	IP TTL=44 100%
04/08	23:10:00	000.000.000.000	0.601878	FP	(arppoison)	Ether Src Hi=xC66973 60%
04/06	11:57:55	206.048.044.050	0.601356	FP		IP TOS=xC8 100%
04/05	11:17:50	000.000.000.000	0.597209	FP	(arppoison)	Ether Src Hi=xC66973 60%
04/07	08:39:42	172.016.114.050	0.592460	FP		TCP Checksum=x148C 94%
04/08	23:11:04	172.016.112.010	0.586338	TP	mscan	Ether Dest Hi=x29CDBA 57%
04/08	08:01:21	172.016.113.050	0.583140	FP		TCP URG Ptr=34757 98%
04/05	11:18:45	172.016.118.020	0.581669	FP		Ether Dest Hi=xE78D76 57%

Table 4.2. Top 20 scoring alarms on weeks 4 (3/29-4/2) and 5 (4/5-4/9).

4.2. Attacks detected

By DARPA criteria, PHAD-C32 detects 67 of the 201 attack instances at a rate of 10 false alarms per day (100 total). These are listed in the Table 4.3 (plus *arppoison*, which does not identify the IP address). The *det* column shows the number of detections out of the total number of instances of the attack described. DARPA classifies attacks as probes, denial of service (DOS), remote to local (R2L), user to root (U2R), and other violations of a security policy (Data). The *how detected* column describes the anomaly that led to detection, based on the field or fields that contribute the most to the anomaly score. The numbers in parentheses indicate how many instances are detected with each field when there is a difference in the way that the instances are detected. For example, all of the *dosnuke* attacks all generate two alarms each (URG pointer and TCP flags), but the *ipsweep* attacks generate only one alarm each. Two *ipsweep* attacks are detected by an anomalous value of 253 in the TTL field of the attacking packet, two by an anomalous packet size in the response from the victim, and three are not detected.

Attack	Description	Det	How detected
apache2	DOS, HTTP overflow in apache web server	2/3	outgoing TCP window, TCP options, incoming TTL=253
(arppoison)	DOS, spoofed ARP with bad IP/Ethernet map	0/5	Ethernet source address
casesen	U2R, NT bug exploit	1/3	TTL=253
crashiis	DOS, HTTP bug in IIS web server	1/8	TTL=253
dosnuke	DOS, URG data to netbios port crashes NT	4/4	URG ptr, TCP flags
guessftp	R2L, FTP password guessing	1/2	TTL=253
guesstelnet	R2L, telnet password guessing	2/2	TTL=253
insidesniffer	Probe, detected by reverse DNS lookups	1/2	Bad TCP checksum
ipsweep	Probe, tests for valid IP addresses	4/7	outgoing packet size (2), incoming TTL=253 (2)
mailbomb	DOS, mail server flood	2/4	TTL=253

mscan	Probe, tests many R2L vulnerabilities	1/1	Ethernet dest. addr.
named	R2L, DNS buffer overflow	1/3	TTL=253
neptune	DOS, SYN flood	1/4	TTL=253
netbus	R2L, backdoor	3/3	TTL=126
netcat_breakin	R2L, backdoor install	1/2	TTL=126
ntinfoscan	Probe, test for NT vulnerabilities	1/3	TTL=126
pod	DOS, oversize IP packet (ping of death)	4/4	Fragmented IP
portsweep	Probe, test for listening ports	13/15	FIN w/o ACK (3), IP src/dest (1), packet size (1), TTL=36...52 (8)
ppmacro	R2L, Powerpoint macro trojan	1/3	TTL=253
processtable	DOS, server request flood	1/4	IP source address
queso	Probe, stack fingerprint for operating system	3/4	FIN without ACK
satan	Probe, test for many R2L vulnerabilities	2/2	packet size (1), TTL (1)
sechole	U2R, NT bug exploit	1/3	TTL=253
sendmail	R2L, SMTP buffer overflow	1/2	outgoing IP dest addr, incoming IP src addr
smurf	DOS, distributed ICMP echo reply flood	5/5	ICMP checksum (2), IP src addr (1), TTL=253 (2)
teardrop	DOS, overlapping IP fragments	3/3	fragmented IP
udpstorm	DOS, echo/chargen loop using spoofed request	2/2	UDP checksum
warez	Data, unauthorized files on FTP server	1/4	outgoing IP dest. address
xlock	R2L, fake screensaver captures password	1/2	IP source address

Table 4.3. Attacks detected by PHAD.

4.3. Contributions of fields to detection

Table 4.4 lists the attacks detected and false alarms generated by each field. *TP* is the number of detections, *dup* is the number of duplicate detections of an attack already detected by a higher scoring alarm, and *FP* is the number of false alarms generated by the field. *Outgoing* indicates that the attack was detected on a packet responding to the attack.

Field	**TP**	**Dup**	**FP**	**Detected attacks**
Ethernet Size	1	2	1	ipsweep (outgoing)
Ethernet Dest Hi	1	0	6	mscan
Ethernet Src Hi	0	0	7	(arppoison)
IP TOS	0	0	7	
IP Packet Length	2	2	1	teardrop, portsweep, satan
IP TTL	33	8	20	netcat_breakin, netbus, ntinfoscan, dosnuke, queso casesen, satan, apache2, mscan, mailbomb, ipsweep, ppmacro, guesstelnet, neptune, sechole, crashiis, named, smurf, portsweep
IP Frag Ptr	7	0	2	teardrop, pod
IP Source Address	4	1	0	smurf, xlock, portsweep, processtable, sendmail
IP Dest Address	2	1	7	portsweep, warez (outgoing), sendmail (outgoing)
TCP Acknowledgment	0	0	1	
TCP Flags UAPRSF	7	3	2	queso, portsweep, dosnuke
TCP Window Size	0	1	2	apache2 (outgoing)
TCP Checksum	1	0	29	insidesniffer
TCP URG Ptr	3	0	5	dosnuke
TCP Options	2	0	4	apache2 (outgoing)
UDP Length	0	0	5	
UDP Checksum	2	0	0	udpstorm
ICMP Checksum	2	0	0	smurf

Table 4.4. Packet header fields that generate alarms in PHAD.

Fifteen of the 33 fields did not generate any anomalies. These are the lower 3 bytes of the Ethernet source and destination addresses, IP header length, IP fragment ID, IP protocol, IP checksum, TCP source and destination ports, TCP sequence number, TCP header length, UDP source and destination ports and length, and the ICMP type and code fields. Of the 67 attacks, only 8 were detectable by their IP addresses, and none by their port numbers.

The 30 TCP checksum errors are mostly due to fragmented IP packets that fragment the TCP header. This is one of the techniques described by Horizon (1998) to thwart an IDS. There is no legitimate reason for using such small fragments, but no attack is listed by DARPA. We believe that PHAD is justified in reporting this anomaly. The error (accounting for 36% of the total score in most cases) occurs because PHAD makes no attempt to reassemble IP fragments and mistakenly computes a TCP checksum on an incomplete packet. The detection of *insidesniffer* is a coincidence.

The TTL field generates 33 detections of 19 types of attacks, and 20 false alarms. TTL is an 8-bit counter (0-255) which is decremented with each router hop until it reaches zero, in order to prevent infinite routing loops. Most of the detections and all of the false alarms due to TTL result from the anomalous values 126 or 253, which are absent in the training data. We believe that this is not realistic. The 12 million packets in the DARPA training set contain only 8 distinct TTL values (2, 32, 60, 62-64, 127-128, 254-255), but in real life, we observed 80 distinct TTL values in one million packets collected on a department web server over a 9 hour period. It is possible that an attacker might manipulate the TTL field to thwart an IDS using methods described by Horizon (1998) or Ptacek and Newsham (1998), but these techniques involve using small values in order to expire packets between the target and the IDS. A more likely explanation is that the attacks were launched from a real machine that was 2 hops away from the sniffer in the simulation, but all of the other machines were at most one hop away. It is extremely difficult to simulate Internet traffic correctly (Floyd and Paxson, 2001), so such artifacts are to be expected. When we run PHAD-C32 without the TTL field, it detects 48, instead of 67, attack instances.

4.4. Attacks not detected

Table 4.5 lists all attacks not detected by PHAD. All of the undetected attacks except *land, resetscan,* and *syslogd* either exploit bugs at the application layer, which PHAD does not monitor, or are *U2R* or *data* attacks where the attacker already has a shell. In the latter case, the attacks are difficult to detect because they require interpretation of user commands, and could be hidden by running them from a local script or through an encrypted SSH session. A better way to detect such attacks might be to use anomaly detection at the system call level, for example, as described by Forrest et al. (1996).

Attack	**Description**	**Det**
anypw	U2R, NT backdoor	0/1
back	DOS, HTTP overflow	0/4
dict	R2L, password guessing using dictionary	0/1
eject	U2R, Solaris buffer overflow in suid root prog.	0/2
fdformat	U2R, Solaris buffer overflow in suid root prog.	0/3
ffbconfig	U2R, Solaris buffer overflow in suid root prog.	0/2
framespoofer	R2L, hostile web page hidden in invisible frame	0/1

guesspop	R2L, password guessing to POP3 mail server	0/1
httptunnel	R2L, backdoor (uses HTTP to evade firewalls)	0/3
imap	R2L (root), buffer overflow to IMAP server	0/2
land	DOS, IP with source = destination crashes SunOS	0/2
loadmodule	U2R, SunOS, set IFS to call trojan suid program	0/3
ls	Probe, download DNS map	0/3
ncftp	R2L, FTP exploit	0/5
netcat	R2L, backdoor	0/2
ntfsdos	Data, copy NT disk at console	0/3
perl	U2R, Linux exploit	0/4
phf	R2L, Apache default CGI exploit	0/4
ps	U2R, Solaris exploit using race conditions	0/4
resetscan	Probe, test for listening ports	0/1
secret	Data, unauthorized copying	0/5
selfping	DOS, Solaris, *ping localhost* from shell crashes	0/3
snmpget	R2L, Cisco router password guessing	0/4
sqlattack	U2R, escape from SQL database shell	0/3
sshtrojan	U2R, fake SSH login screen captures password	0/3
syslogd	DOS, Solaris, crash audit logger w/ spoofed IP	0/4
tcpreset	DOS, sniff connections, spoof RST to close	0/3
xsnoop	R2L, intercept keystrokes on open X servers	0/3
xterm	U2R, Linux buffer overflow in suid root prog.	0/3
yaga	U2R, NT	0/4

Table 4.5. Attacks not detected by PHAD.

Land crashes SunOS by sending a spoofed IP packet with identical source and destination addresses, but PHAD misses it because it only looks at one field at a time. *Resetscan* probes for ports by sending a RST packet on an unopened TCP port, but PHAD misses it because it doesn't track TCP connection states, and doesn't know that the connection is not open. *Syslogd* crashes the *syslogd* server on Solaris by sending a packet with a spoofed source IP that cannot be resolved to a hostname. If the threshold is increased from 10 false alarms per day to 30, PHAD does indeed detect all four instances of this attack, two by their source IP address, and two by an anomalous IP packet length (of 32).

4.5. Run-time overhead in time and space

Our implementation of PHAD-C32 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.

5. Attacks in the Training Data

In a practical system, known attack-free training data will not always be available as it is in the DARPA set. If the training data contains attacks, then the anomalies that it generates will be added to the model of "normal" traffic, so that future attacks of the same type will be missed. We could use a shorter

training period to reduce the probability of training on an attack, but this could also reduce the set of allowed values, resulting in more false alarms. We performed experiments to answer two questions. First, how much is performance degraded by using a shorter training period? Second, how much is performance degraded due to the masking effect, where training on one attack type masks the detection of other attacks.

To answer the first question, we reduced the training period from 7 days (week 3, days 1-7), to 1 or 5 days. To answer the second question, we ran PHAD in on-line mode. For each test day, we used the previous day's traffic for training. On all but the first test day, the training data therefore contained attacks. The results are shown in Table 5.1 (for 10 FP/day).

Training set	**Detections**
Days 1-7 (no attacks)	67
Days 1-5	66
Day 1	55
Day 2	51
Day 3	55
Day 4	51
Day 5	53
Day 6	64
Day 7	34
Average of days 1-7	52.3
Previous day (on-line with attacks)	35

Table 5.1. Attacks detected by PHAD using various training sets

First, we observe almost no loss in going from 7 training days to 5, but a significant 22% loss on average in going to one day. Second, we observe a loss of 17 detections, or 33% when we use the previous day's data as training for the next in on-line mode, compared to using one day of attack free training data. Out of 201 total attacks, there are 59 types (tables 4.3 and 4.5), so each attack type occurs on average 3.4 times during the 10 day test period, or 0.34 times per day. If we assume that each attack has a 34% chance of having occurred on the previous day, then the observed loss (33%) suggests that there is little or no masking effect. It is possible that there is a masking effect, but it is compensated by using more recent (better) training data in on-line mode than in the other 1-day tests. But if this is so, we should observe improved performance as we go from day 1 to day 7 of the training data, and no such trend is apparent.

6. Tuning PHAD

We investigated a number of ideas and performed experiments with PHAD to improve its performance on the DARPA evaluation data set, obtaining the best result with the variation we denote PHAD-C32 described in the previous sections. The results for the other versions are shown in Table 6.1. We describe the idea behind each variation briefly.

For each variation, we also applied a postprocessing step where we remove duplicate alarms (within 60 seconds of each other) that identify the same IP address. This step almost always improves the results because it reduces the number of false alarms (which would be counted twice) without reducing the number of detections (which would be counted only once). Also, in the rare case of tie scores, we rank

them in order of the time since the previous alarm indicating the same IP address with the longest intervals first. For instance, if there are alarms at times 5, 6, and 8 minutes, then we rank them 5, 8, 6 (intervals of 5, 2, and 1 minute) and discard 6 because it occurs within one minute of 5. Time-ranking has the same effect as the factor *t* in the PHAD-C32 score *tn/r*.

For PHAD-C32, postprocessing increases the number of detections from 67 to 72, the best result we could obtain from any system we tried.

PHAD variation	**Detections**	**Postprocessed**
PHAD-C, values stored in C = 32, 1000 clusters, score = tn/r	67, 64	72, 70
PHAD-H, values hashed modulo H = 1000	64	67
PHAD-B, one byte fields, all values stored (H = 256)	19	not tested
PHAD-S, stationary model, score = 1/p based on counts, H = 1000	8	27
PHAD-KL, stationary KL divergence of 60 sec. windows, H = 1000	16	16
PHAD-KL0, KL without fields where all H = 1000 values > 0	22	38
PHAD-Kt, score = time t since last anomaly in nearest of K = 16, 32, 64 clusters in 34-dimensional packet space	28, 38, 14	28, 38, 14
PHAD-Kt/r, score = t/r of nearest of K = 16, 32 clusters, containing r elements	38, 46	39, 47
PHAD-Kt/a, Score = t/a of nearest of K = 16, 32, 64 clusters expanded *a* times during training	35, 53, 12	37, 53, 12
PHAD-1H, PHAD-H1000 but score = t/r (not tn/r)	58	62
PHAD-2H, H = 1000 over all pairs of fields, score = t/r	56	62
PHAD-C32-TTL, PHAD-C32 without TTL field	48	54

Table 6.1. Number of detections for various PHAD versions.

6.1. Increasing number of clusters

PHAD-C32 stores the observed values for each field in a list of 32 ranges or clusters. When C is increased to 1000, we have a more accurate representation of the training data, but lose some generalization when the training set is small. The models differs only when r > 32, which means that it affects only low scoring fields. In our experiments, the postprocessed detection rate decreased slightly from 72 to 70 postprocessed.

6.2. Storing hashed values

PHAD-H stores a set of H = 1000 hashes (mod H) of the observed values. This is faster to look up ($O(1)$ vs. $O(\log C)$), but loses the ability to generalize over continuous fields (such as packet size). There is a slight decrease in detection rate from 72 to 67. In our implementation, we did not see a noticeable increase in speed.

6.3. One byte fields

A simplified version of PHAD is to parse the header into 1 byte fields and simply store the set of 256 possible values. However, the detection rate was severely degraded from 64 to 19 (not postprocessed), suggesting that it is worthwhile to find "meaningful" boundaries in the data.

6.4. Stationary models

PHAD makes no distinction between a training value that occurs once and a value that occurs millions of times. It is a nonstationary model: the probability of an event depends only on the time since it last occurred. We built a stationary model, PHAD-S, in which the probability of an event depends on its average rate during training, and independent of recent events. Specifically, if a value is observed r times in n trials, then we assign a score of $1/p$, where $p = (r + 1)/(n + H)$, where there are H possible (hashed) values. This is the Laplace approximation of the probability p, allowing for nonzero probabilities for novel values. The result (8 detections) suggests that a stationary model is a poor one. This model generates a large number of tie scoring alarms, so postprocessing (which makes the model nonstationary) increases the number of detections to 27.

All of the PHAD models so far rate individual packets independently, but we know that many attacks can be recognized by a short term change of rate of some event over many packets (for example, a flooding attack). PHAD-KL assigns a score to a field over a set of packets in a 60 second time window based on how much the distribution of values differs from the distribution in training. The normal way to measure the divergence of some probability distribution q (the 1 minute window) from a reference distribution p (the training data) is the Kullback-Liebler divergence, $D(q \parallel p) = \Sigma_i\, q_i \log q_i/p_i$. p_i is measured using a Laplacian approximation as before, but $q_i = r_i /n$ where the i'th value is observed r_i times out of n during the one minute window. We do not use Laplacian estimation for q because otherwise a window with no packets would generate a uniform distribution and a large KL score, rather than a score of 0). PHAD-KL is a stationary model over the window, but the detections increase from 8 to 16 compared to considering single packets.

A variant, PHAD-KL0, is to remove those fields which do not contribute to the other models, specifically those fields in PHAD-H1000 where all H possible values are observed. PHAD-KL should be able to collect a distribution (and therefore, useful information) from these fields, so we might expect the number of detections to drop in PHAD-KL0. Instead, the detections increase from 16 to 22 (16 to 38 postprocessed), suggesting that these fields probably added more noise than useful data.

6.5. Examining combinations of fields

Every version of PHAD so far looks at each field in isolation, but we know that some fields may or may not be anomalous depending on the values of other fields. For example, a TCP destination port of 21 (FTP) is only anomalous when the destination IP address is that of a host not running an FTP server. We would like a model that allows port 21 only in some range of IP addresses, and possibly restricts other fields at the same time. PHAD-K clusters all training packets into K clusters, each defined by a lower and upper bound on each field, or equivalently, a 34-dimensional volume (33 fields plus time) in packet-space, where each field is a dimension. The clustering algorithm is to add each packet to the packet space as a cluster of size 1^{34} (1 by 1 by 1...) and if the number of clusters exceeds K, to combine the two closest ones, where the distance is defined as the increase in volume that results from enclosing them both in a new cluster. The goal of this algorithm is to keep the model specific by minimizing the total cluster volume. Missing field values (such as the UDP checksum in a TCP packet) are assumed to be 0.

There are three variants of PHAD-K, depending on how the score is computed for packets that fall outside of all clusters. In the PHAD-Kt variant, the score is t, where t is the time since the last anomaly in the nearest cluster (defining distance as before). This is a nonstationary model, and it assumes that

consecutive anomalies are likely to be close together, and therefore closest to the same cluster. We tested K = 16, 32, and 64 clusters, with the best result of 38 detections at K = 32.

PHAD-Kt is based on the assumption that the expected rates of anomalies near each cluster are initially equal. We can do better than that, by measuring the anomaly rate during training. We first estimate this rate by counting the number of packets, *r*, in each cluster, and assigning a packet score of *t/r*. This was tested with K = 16 and 32, and detects 47 attacks with K = 32. A better way to measure the anomaly rate is to count the number of times a cluster was expanded (due to an anomalous training event). A new packet gets a count of 1. If two clusters have counts of c_1 and c_2, then when merged the count is $c_1 + c_2 + 1$. This method, PHAD-Kt/a, was tested with K = 16, 32, and 64, with a best result of 53 detections at K = 32.

PHAD-K variants are computationally expensive (O(K) to find the closest cluster by linear search), and no better than PHAD-C or PHAD-H in the number of detections. PHAD-2H considers only pairs of fields, rather than combining them all at once, storing a hash (mod H = 1000) of both fields in a set of size H. There are (34)(33)/2 = 560 such pairs, plus the original 34 fields. In PHAD-H and PHAD-C, we use a score of *tn/r*, where *n* is the number of times the field is observed. Since we now have two fields with possibly different *n*, we have to eliminate *n* from the score. To test the effect of doing this, we modified PHAD-H to PHAD-1H (with H = 1000) using the score *t/r*, where there are *r* allowed values, and the last anomaly was *t* seconds ago. The effect of removing *n* is to effectively assign higher scores to the fields of seldom-used protocols such as UDP and ICMP. This baseline detects 62 attacks postprocessed (compared to 67 for PHAD-H). Extending this to PHAD-2H with 560 field pairs results in no change, 62 detections.

6.6. PHAD without TTL

We mentioned in Section 4.3 that many of the detections (and false alarms) are due to anomalies in the TTL field, which is probably a simulation artifact. When we remove this field (PHAD-C32-TTL), the number of detections drops from 72 to 54 postprocessed.

6.7. Conclusions

To summarize, our experiments suggest that:

- Nonstationary models (C, H, 1H, 2H, K), where the probability of an event depends on the time since it last occurred, outperform stationary models (S, KL), where the probability depends on the average rate during training.
- There is an optimal model size (C32, Kt32, Kt/r32, Kt/a32) which performs better than models that overfit the training data (C1000, Kt64, Kt/a64) or underfit (Kt16, KT/r16, KT/a16).
- Models that consider fields individually (C, H) outperform models that combine fields (2H, K), but these fields should be larger than one byte (B). (This could be an example of overfitting/underfitting).
- Models that treat values as continuous (C) outperform discrete models (H).
- Postprocessing by removing duplicate alarms or ranking by time interval never hurts.

7. A Note on Results from DARPA Participants in 1999

Lippmann et al. (2000) reports that the detection rates for the best 4 of 18 systems evaluated on the DARPA set in 1999 had detection rates of 40% to 55% on the types of attacks that they were designed to detect (Table 7.1). However, we caution that unlike the original test, we had access to the test data during development and did our own unofficial evaluation of the results. Also, the original participants could only categorize their systems according to the type of attacks they are to detect (probe, DOS, R2L, U2R, data), operating system (SunOS, Solaris, Linux, NT, Cisco), and type of data examined (inside or outside traffic, BSM, audit logs, and file system dumps). We use a finer distinction when we classify attacks by protocol.

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

Table 7.1. Official detection rates (out of the total number of attacks that the system is designed to detect) at 10 FA/day for top systems in the 1999 DARPA evaluation (Lippmann et al. 2000, Table 6).

PHAD-C32 with postprocessing unofficially detects 72 of the 201 attacks in the DARPA evaluation. If we consider only *probe* and DOS attacks, and exclude the attacks in the missing data (week 4 day 2), then PHAD-C32 detects 59 of 99 instances, or 60%. It detects at least one instance of 17 of 24 *probe* and DOS types, and 29 of 59 types altogether. Of the 30 types missed, only 3 (*resetscan*, *land*, and *syslogd*) produce anomalies at the transport layer or below, and one of them (*syslogd*) is detectable at a lower threshold. Twelve of the 29 attack types that PHAD-C32 detects are detected solely by the TTL field, which we suspect is a simulation artifact. But if we remove this field, only *neptune* would be added to the list of attacks that we could reasonably detect. In contrast to the firewall model, only 8 instances (6 types) of attacks are detected by their IP addresses, and none by their port numbers.

Lippmann reports that several types of attacks were hard to detect by every system tested. PHAD-C32 outperforms the official results on four of these: *dosnuke*, *queso*, and the stealthy (slow scan) versions of *portsweep* and *ipsweep*. These are summarized in Table 7.2. PHAD is relatively immune to slow scans, since it examines each packet in isolation rather than correlate events over an arbitrary time period.

Attack	**Best detection (Lippmann 2000)**	**PHAD-C32 (unofficial)**
stealthy ipsweep	0/3	1/3
stealthy portsweep	3/11	11/11
queso	0/4	3/4
dosnuke	2/4	4/4

Table 7.2. Unofficial detection rates by PHAD-C32 for some hard to detect attacks.

8. Concluding Remarks

Intrusions generate anomalies because of bugs in the victim program, the attacking program, or the IDS, or because the victim generates anomalous output after an attack. We have seen examples of all four types of anomalies:

1. Strange input to poorly tested software, e.g. the corrupted packets used by *teardrop*, *pod*, *dosnuke*.
2. Strange data from poorly written attacks, e.g. bad checksums in *smurf* and *udpstorm*.
3. Strange data used to hide an attack from layers above, e.g. FIN scanning by *portsweep*.
4. Strange responses from the victim, e.g. unusual TCP options generated by *apache2*.

We proposed an anomaly detection algorithm (PHAD) based on examining packet header fields other than just the normal IP addresses and port numbers. We found that these fields play only a minor role in detecting most attacks. PHAD detects most of the attacks in the DARPA data set that involve exploits at the transport layer and below, including four that eluded most of the systems in the original evaluation.

We experimented with a number of models, and the best results are obtained by nonstationary models that examine each field and each packet separately. PHAD-C32 gets slightly better results than C1000, probably because it avoids overfitting the training data, and better than H1000 because it is able to generalize to reasonable values for continuous variables. All three get similar results because they are equivalent for small *r*, those fields that contribute most to the anomaly score. They outperform stationary models (S, KL, KL0), and slightly outperform models that consider combinations of fields (2H, Kt, Kt/r, KT/a). PHAD-C32 is among the simplest models tested. It incurs low time and space overhead.

PHAD was relatively more successful in detecting four of the hard-to-detect attacks identified by Lippmann et al. (2000). This indicates that PHAD could cover an attack space different from the other detectors. Hence, a combination of the detectors could increase the overall coverage of detectable attacks.

All of the PHAD models require no *a-priori* knowledge of any attacks and very little *a-priori* knowledge about the protocols they model. The models are learned from data which can have different characteristics in diverse environments. Moreover, new models can be learned and adapt to new traffic characteristics in the same environment. However, PHAD is not intended to be a standalone IDS. It should be component in a system that includes signature detection for known attacks, system call monitoring for novel R2L and U2R attacks, and file system integrity checking for backdoors.

As expected, we observed some degradation in PHAD's performace when attacks are present in the training data. One remedy is to employ misuse signature detectors to filter out known attacks from the data before supplying the data to PHAD for training. Another approach is to make PHAD more noise-tolerant (attacks, usually in small amounts, in the presumably attack-free data can be considered as noise). Currently, PHAD does not consider the frequency of each value observed during training, however, values of low frequency could be noise.

PHAD-C32 already detects most attacks in the protocols it examines. We suspect that in order to yield any significant improvements, it will be necessary to examine the application layer. There are systems that do this now, but they are mostly rule-based, matching strings to known attacks. We have detected a small number of attacks using anomaly models based on n-gram statistics in some preliminary

experiments, but there are great variety of application layer protocols, some far more complex than TCP/IP, and there is much work to be done.

Acknowledgments

This research is partially supported by DARPA (F30602-00-1-0603). We thank Richard Lippmann and Richard Kemmerer for comments on an earlier draft of this paper.

References

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

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

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

Floyd, S. and V. Paxson, "Difficulties in Simulating the Internet." To appear in IEEE/ACM Transactions on Networking, 2001. http://www.aciri.org/vern/papers.html

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

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

Horizon, "Defeating Sniffers and Intrusion Detection Systems", Phrack 54(10), 1998, http://www.phrack.org

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

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

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

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

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

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

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.

snort ver. 1.7 (2001), http://www.snort.org