A Machine Learning Approach to Detecting Attacks by Identifying Anomalies in Network Traffic

by

Matthew Vincent Mahoney

A dissertation submitted to the College of Engineering at

Florida Institute of Technology

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

in

Computer Science

Melbourne, Florida

May, 2003

TR-CS-2003-13

A Machine Learning Approach to Detecting Attacks by Identifying Anomalies in Network Traffic

a dissertation by

Matthew Vincent Mahoney

Approved as to style and content

_____
Philip K. Chan, Ph.D.
Associate Professor, Computer Science
Dissertation Advisor

_____
Ryan Stansifer, Ph.D.
Associate Professor, Computer Science

_____
James Whittaker, Ph.D.
Professor, Computer Science

_____
Kamel Rekab, Ph.D.
Professor, Mathematical Sciences

_____
William D. Shoaff, Ph.D.
Associate Professor, Computer Science
Department Head

Abstract

A Machine Learning Approach to Detecting Attacks by Identifying Anomalies in Network Traffic

by

Matthew Vincent Mahoney

Dissertation Advisor: Philip K. Chan, Ph.D.

The current approach to detecting novel attacks in network traffic is to model the normal frequency of session IP addresses and server port usage and to signal unusual combinations of these attributes as suspicious. We make four major contributions to the field of network anomaly detection. First, rather than just model user behavior, we also model network protocols from the data link through the application layer in order to detect attacks that exploit vulnerabilities in the implementation of these protocols. Second, we introduce a time-based model suitable for the bursty nature of network traffic: the probability of an event depends on the time since it last occurred rather than just its average frequency. Third, we introduce an algorithm for learning conditional rules from attack free training data that are sensitive to anomalies. Fourth, we extend the model to cases where attack-free training data is not available.

On the 1999 DARPA/Lincoln Laboratory intrusion detection evaluation data set, our best system detects 75% of novel attacks by unauthorized users at 10 false alarms per day after training only on attack-free traffic. However this result is misleading because the background traffic is simulated and our algorithms are sensitive to artifacts. We compare the background traffic to real traffic collected from a university departmental server and conclude that we could realistically expect to detect 30% of these attacks in this environment, or 47% if we are willing to accept 50 false alarms per day.

Table of Contents

List of Figures

List of Tables

Acknowledgments

I wish to thank my dissertation advisor Dr. Philip K. Chan for guidance through this dissertation, including co-authorship of several papers which were published as a result of this research. I also thank my the dissertation committee, Dr. Ryan Stansifer, Dr. Kamel Rekab, and Dr. James Whittaker.

This work would not be possible without the DARPA/Lincoln Laboratory intrusion detection evaluation data set. I thank Richard Lippmann, who led the development of this data set, for valuable feedback on our work. I also thank the anonymous reviewers of our papers for comments that helped guide our work.

Several other students collaborated on this research under Dr. Chan, and their work is ongoing. Mohammad Arshad is using a clustering approach to outlier detection in the network data to detect attacks. Gaurav Tandon is applying our rule learning algorithm to system call arguments (BSM) in experiments with host based anomaly detection. Anyi Liu also worked in this area. Rachna Vargiya and Hyoung Rae Kim are researching tokenization algorithms for parsing the application payload and identifying keywords.

This work was funded by DARPA (F30602-00-1-0603).

Dedication

To my wife Joan, who put up with my endless hours on the computer, and to my cats Powerball and Dervish, who sat on my lap as I typed, and occasionally tried to help by walking on the keyboard.

Chapter 1

Introduction

Computer security is a growing problem. The Computer Emergency Response Team, or CERT (2003b) reported 82,094 incidents and 4129 new vulnerabilities in 2002. Both of these numbers have approximately doubled each year since 1997. Likewise, the number of web page defacements per year has approximately doubled each year, growing to about 15 per day in 2000, according to www.attrition.org. While much of this growth can be attributed to the growth of the Internet, there is also an increase in the number of incidents per computer. According to the ICSA 1998 Computer Virus Prevalence Survey, the rate of virus infections per computer per month in large North American organizations increased from 0.1% in 1994 to 3% in 1998.

Most vulnerabilities are software errors, a very old problem. For example, both the Morris Internet worm (Spafford, 1988) and the SQL Sapphire worm (CERT, 2003a) exploit buffer overflows, a common type of error occurring in many C programs in which the length of the input is not checked, allowing an attacker to overwrite the stack and execute arbitrary code on a remote server. Both worms spread quickly all over the world and caused widespread damage, but with one major difference. In 1988, patches to fix the vulnerability were developed, distributed, and installed worldwide within a day of the attack. In 2003, the vulnerability was known months in advance and patches were available, but many people had not bothered to install them.

Patches and updated software versions are almost always available soon after a vulnerability is discovered. Unfortunately updating software takes time and computer skills, and sometimes introduces new bugs or incompatibilities. In reality, many people leave their systems insecure rather than try to fix something that already appears to be working. This might explain

why in an audit of U.S. federal agencies by the General Accounting Office in 2000, investigators were able to pierce security at nearly every system they tested (Wolf, 2000).

Even if the software update problem were solved, there would still be a time lag between the development of new exploits and the availability of tests to detect the attack (e.g. virus definition files or firewall rules) or patches to fix the vulnerability. Although patches may be available in a day or so, this would not stop the spread of "flash" worms, which could potentially infect all vulnerable computers on the Internet within a few minutes of release (Staniford, Paxson, & Weaver, 2002). The SQL Sapphire worm is one such example, doubling in population every 8.5 seconds and infecting 90% of vulnerable computers worldwide within 10 minutes of its release (Beverly, 2003; Moore et al., 2003).

Software patches also do not help for the more common case where the victim does not know that his or her computer has been compromised. Attackers may go to great lengths to conceal their backdoors. Websites like www.rootkit.com and www.phrack.org provide tools and describe techniques such as modifying the kernel to hide files and processes or modifying TCP/IP protocols to set up stealth channels to penetrate firewalls.

Furthermore, people are generally unaware that their computers are probed many times per day, either with tools specifically designed for that purpose, such as NMAP (Fyodor, 2003), or by network security tools like SATAN (Farmer & Venema, 1993), which are intended to allow network administrators to test their own systems for common vulnerabilities. Probes often originate from compromised machines, so identifying the source can be helpful to their owners. Thus, it is not enough just to secure our systems. It is also important just to know that a probe or an attack (especially a novel attack) has taken place.

Our goal is to detect novel attacks by unauthorized users in network traffic. We consider an attack to be novel if the vulnerability is unknown to the target's owner or administrator, even if the attack is generally known and patches and detection tests are available. We are primarily interested in three types of remotely launched attacks: probes, denial of service (DOS), and

intrusions in which an unauthorized user is able to bypass normal login procedures and execute commands or programs on the target host. The latter is also known as a remote to local (R2L) attack (Kendall, 1998). Our goal is not to detect viruses, or attacks in which the attacker already has login privileges or physical access and gains root or administrative access (a user to root or U2R attack). Such attacks are easy to conceal from a network sniffer by using a secure shell, and are best detected by monitoring incoming files or the operating system locally.

Our goal is detection, not prevention. We could block suspicious traffic, as a firewall does, but our goal is simply to identify such traffic. This is a difficult problem in the absence of rules to identify such traffic. Although rules to detect many attacks have been developed for network intrusion detection systems such as SNORT (Roesch, 1999) and Bro (Paxson, 1998), our goal is to detect *novel* attacks. By focusing on detection, we can test our algorithms off-line on sniffed traffic.

The normal approach to detecting novel attacks is anomaly detection: modeling normal behavior and signaling any deviation as suspicious. This process generates false alarms, and is one reason the approach is not widely used. Another problem is that the system often cannot help a user, who is typically not an expert in network protocols, decide if an unusual event (say, a UDP packet to port 1434) is hostile or not. In fact, this is the signature of the SQL Sapphire worm, but it could also be legitimate traffic if one were running a server vulnerable to this attack. Nevertheless, an anomaly detection system could help bring unusual events buried in masses of data to the attention of a network administrator, either in real time, or in a forensic analysis of sniffed traffic after something has gone wrong. Thus, our goal is simply to identify the events most likely to be hostile while accepting some false alarms.

1.1. Problem Statement

The problem we are trying to solve is to *detect attacks in network traffic with no prior knowledge of the characteristics of possible attacks.* We assume that a history of attack-free (or

mostly attack-free) traffic is available from the system we are monitoring. The types of attacks we wish to detect are those that *could* be detected in network traffic if we knew what to look for. These are attacks by remote, unauthorized users: probes, DOS, or R2L. We assume that our system will be part of a more comprehensive intrusion detection system (IDS) that also uses hand-coded rules to detect known attacks, and host-based methods (monitoring file system and operating system events) to detect U2R attacks, viruses, and backdoors.

1.2. Approach

Our approach to detecting novel attacks is anomaly detection: using machine learning to generalize from attack-free traffic, with the assumption that events which do not fit the model are likely to be hostile. Currently most network anomaly models are based on source and destination IP addresses and server ports. For example, an IDS might signal an alarm in response to a packet addressed to UDP port 1434 if such packets are normally rare, which would be the case a for system not running a database server. If it were, it might signal an alarm if the source address was unusual for that port. In either case, the IDS would assign an alarm score or confidence level inversely proportional to the probability of the event, based on the average frequency in the past. This approach can detect many port scans and many attacks on servers with trusted clients.

Our approach differs in two respects. First, we model protocols, rather than just addresses and ports. Many attacks exploit bugs in protocol implementations. For example, the Morris worm exploits a buffer overflow vulnerability in *fingerd*, a UNIX based server which tells whether a user is logged in. This attack would not be detected using normal methods (unusual client addresses) because *finger* accepts requests from untrusted clients. However, by modeling the *finger* protocol, we could detect this attack. Normal requests are short one-line commands containing ASCII text, but the exploit is 576 characters long and contains VAX executable code. In addition to application

protocols like *finger*, HTTP (web), and SMTP (email), we also model the transport layer (TCP, UDP, and ICMP), network layer (IP) and data link layer (Ethernet).

Second, our model estimates the probability of an event using the time since it last occurred, rather than average frequency. This model is better suited to bursty (non-Poisson) processes with long range dependencies (Leland et al.; 1993, Paxson & Floyd, 1995). For example, a fast port scan might generate a rapid burst of alarms using a frequency based model, but in a time based model only the first packet would generate a high score, effectively consolidating the alarms.

Because we model a large number of attributes, it is necessary to form conditional rules to constrain the protocols, such as "*if server-port = 80 then word-1 = GET or POST*". We describe an algorithm for generating such rules automatically from a sample of attack-free training data. Many attacks can be detected by events that have *never* occurred before (i.e. *word-1 = QUIT*), but it is also effective to model events that have occurred, perhaps many times, but not recently, for example the first occurrence of *word-1 = POST* in a week. The second model is more appropriate when we do not use explicit training and test periods. We compare these two approaches.

We evaluate our systems on the 1999 DARPA/Lincoln Laboratory IDS off-line evaluation (IDEVAL) data set (Lippmann et al., 2000; Lippmann & Haines, 2000),which simulates several hosts on a local network connected to the Internet under attack by published exploits. Unfortunately the properties that make an IDS sensitive to attacks also make it sensitive to simulation artifacts. For example, the simulation uses different physical machines with the same IP address to simulate some of the attacks and some of the background traffic (Haines et al., 2001). This leads to the unexpected result that many attacks can be detected by anomalies in the TTL, TCP window size and TCP option fields caused by idiosyncrasies of the underlying simulation. Additional artifacts occur in the distribution of client IP addresses and in many application level protocols. We analyze these artifacts in detail by comparing the background traffic with real traffic collected on a university departmental server. We find that by injecting real traffic into the simulation and by some

modification to our algorithms that most simulation artifacts can be removed and the methods we describe remain effective.

1.3. Key Contributions

The key contributions are as follows:

- A time-based model appropriate for bursty traffic with long range dependencies.
- Modeling application protocols to detect attacks on public servers.
- A randomized algorithm that efficiently learns an anomaly detection model represented by a minimal set of conditional rules.
- Anomaly detection without labeled or attack-free training data.
- Removing background artifacts from simulated evaluation data by injecting real traffic.

1.4. Dissertation Organization

The rest of this dissertation is organized as follows. In Chapter 2, we review network security and attacks, intrusion detection, properties of network traffic, and the IDEVAL test set. In Chapters 3 through 6, we introduce four anomaly detection algorithms to illustrate our first four key contributions. These four systems are PHAD (packet header anomaly detector) (Mahoney & Chan, 2001), ALAD (application layer anomaly detector) (Mahoney & Chan, 2002b), LERAD (learning rules for anomaly detection) (Mahoney & Chan, 2002a), and NETAD (network traffic anomaly detection) (Mahoney, 2003).

In Chapter 3, we use PHAD to illustrate time-based modeling. The attributes are the Ethernet, IP, TCP, UDP, and ICMP packet header fields. The model is global. No conditional rules are used, and no distinction is made between incoming and outgoing traffic. An anomaly occurs only if a field has a value never seen training. The anomaly score is proportional to the number of

training packets and the time since the last anomaly, and inversely proportional to the number of allowed values.

In Chapter 4, we use ALAD to illustrate the use of conditional rules to model application protocols. ALAD models inbound client traffic. It combines a traditional model based on addresses and ports with a keyword model of the application payload. For example, it signals an anomaly when the first word on a line in an SMTP or HTTP header is novel.

In Chapter 5, we use LERAD to illustrate a rule learning algorithm. LERAD uses a training sample to generate candidate rules that should generate high anomaly scores, then tests these rules on the full training set, keeping those not likely to generate false alarms. LERAD models inbound client requests, but could be used on any type of data that can be expressed as tuples of nominal attributes.

PHAD, ALAD, and LERAD signal anomalies only when a value is never seen in training. In Chapter 6, we introduce NETAD (which tests inbound client packets with fixed conditional rules) to compare these models with those that assign anomaly scores to previously seen values. This type of model does not require an explicit attack-free training period. We conclude that both types are effective, but attack-free training data should be used if it is available.

In Chapter 7, we analyze the IDEVAL training and background data by comparing it to real traffic collected on a university departmental server. Although the IDEVAL developers took great care to make the simulated Internet traffic as realistic as possible, we find that the simulated traffic is too "clean" and predictable in subtle ways that a good anomaly detection system could root out. This would explain some (but not all) of the attacks we detected earlier, and suggests that the false alarm rates we measured are unrealistically low.

In Chapter 8 we inject real network traffic into the IDEVAL data to better simulate a real environment. This raises the false alarm rate and masks the detection of attacks by artifacts, yielding more credible results, not just on our systems, but also on another network IDS, SPADE (Hoagland, 2000). We conclude that our systems could legitimately detect 20% to 40% of novel

attacks at false alarm rates of 10 to 50 per day, with a tradeoff between detection and false alarm rate. We believe this is an improvement over current methods.

In Chapter 9, we summarize our contributions and discuss limitations and future work. Our contribution is not to build a network anomaly IDS, but to describe the general principles by which one should be built and tested.

Chapter 2

Related Work

Our goal is to distinguish hostile network traffic from normal traffic. Thus, we review hostile traffic, normal traffic, current methods of distinguishing them, and how these methods are evaluated.

2.1. Network Vulnerabilities

There are thousands of known exploits. Kendall (1998) describes a taxonomy of attacks, grouping them into four major categories.

- Probes – testing a potential target to gather information. These are usually harmless (and common) unless a vulnerability is discovered and later exploited.
- Denial of service (DOS) – attacks which prevent normal operation, such as causing the target host or server to crash, or blocking network traffic.
- Remote to local (R2L) – attacks in which an unauthorized user is able to bypass normal authentication and execute commands on the target.
- User to root (U2R) – attacks in which a user with login access is able to bypass normal authentication to gain the privileges of another user, usually root.

We are interested in detecting the first three types, because they generally exploit network protocol implementations. U2R attacks exploit bugs or misconfigurations in the operating system, for example a buffer overflow or incorrectly set file permissions in a *suid root* program. The actions of

an attacker can be easily hidden from a network sniffer by launching the attack from the console or through a secure shell.

Kendall describes many of the attacks we will describe in this section. His analysis does not include self-replicating attacks such as worms or viruses, although they normally propagate by exploiting R2L vulnerabilities. They may also contain DOS attacks as payloads, for example, erasing all files on an infected computer on a certain date.

2.1.1. Probes

Probes gather information to search for vulnerable systems. For example:

- IP sweep – testing a range of IP addresses with *ping* to determine which ones are alive (Kendall, 1998). Another way to gather a list of potential targets is to spoof a zone transfer request to a DNS server, as is done by the *ls* command in NSLOOKUP.
- Port scans – testing for ports with listening servers. Tools such as NMAP (Fyodor, 2003) and HPING (Sanfilippo, 2003) use sophisticated techniques to make scans hard to detect, for example, scanning with RST or FIN packets (which are less likely to be logged), or using slow scans to defeat an intrusion detection system (IDS) looking for a burst of packets to a range of ports.
- Fingerprinting – determining the operating system version of the target based on idiosyncrasies in responses to unusual packets, such as TCP packets with the reserved flags set. This method, implemented by QUESO and NMAP, distinguishes among hundreds of operating system versions using only 7 packets (Fyodor,1998).
- Vulnerability testing – Network administration tools such as SATAN (Farmer & Venema, 1993), SAINT (Kendall, 1998), MSCAN (Kendall, 1998), and NESSUS (Deraison, 2003) test for a wide range of vulnerabilities. These tools serve the dual purpose of allowing network administrators to quickly test their own systems for vulnerabilities, and allowing

attackers to test someone else's system. NESSUS is open source and uses a scripting language and has an extensive library of tests, which is updated as new vulnerabilities are discovered. As of February 2003, NESSUS tests for 1181 vulnerabilities.

- Inside sniffing – An attacker with physical access to a broadcast medium such as Ethernet, cable TV, or wireless, could sniff traffic addressed to others on the local net. Many protocols such as telnet, FTP, POP3, IMAP, and SNMP transmit passwords unencrypted.

Probes (other than sniffers) normally cannot use spoofed source IP addresses because they require a response back to the attacker. However there are a number of methods to make it harder to detect the true source, for example:

- Sending large numbers of spoofed packets in addition to true source addresses, so that the victim will not know which address is the real probe.
- Launching a probe from a compromised host.
- Idle scanning through a zombie (Fyodor, 2002).

Idle scanning, a feature of NMAP, allows an attacker to conceal its address by exploiting any intermediate host (a zombie) that is lightly loaded and that yields predictable IP fragment ID values, as many operating systems do. For example, the ID may be incremented after each packet sent. The attacker probes the zombie on an open port (say, a web server on port 80) to get the current ID, then sends a TCP SYN packet to the target port to be probed, with the spoofed source address of the zombie. The source port is set to 80. The target responds to the zombie on port 80 either with a SYN-ACK packet if the port is open, or a RST if closed The zombie then replies (with a RST) to the target in case of a SYN-ACK (since no TCP connection was open), but does not respond to a RST from the target. The attacker then probes the zombie a second time to see whether the IP ID is incremented by one or two, thus learning whether the port is open or closed.

2.1.2. Denial of Service Attacks

Denial of service attacks can target a server, a host, or a network. These either flood the target with data to exhaust resources, or use malformed data to exploit a bug. Kendall (1998) gives the following examples, all of which are used in the IDEVAL test set.

- Apache2 – Some versions of the *apache* web server will run out of memory and crash when sent a very long HTTP request. Kendall describes one version in which the line "User-Agent: sioux" is repeated 10,000 times.
- Back – Some versions of *apache* consume excessive CPU and slow down when the requested URL contains many slashes, i.e. "GET //////////..."
- Land – SunOS 4.1 crashes when it receives a spoofed TCP SYN packet with the source address equal to the destination address.
- Mailbomb – A user is flooded with mail messages.
- SYN flood (Neptune) – A server is flooded with TCP SYN packets with forged source addresses. Because each pending connection requires saving some state information, the target TCP/IP stack can exhaust memory and refuse legitimate connections until the attack stops.
- Ping of death – Many operating systems could be crashed (in 1996 when the exploit was discovered) by sending a fragmented IP packet that reassembles to 65,536 bytes, one byte larger than the maximum legal size. It is called "ping of death" because it could be launched from Windows 95 or NT with the command "ping –l 65510 *target*".
- Process table – An attacker opens a large number of connections to a service such as *finger*, POP3 or IMAP until the number of processes exceeds the limit. At this point no new processes can be created until the target is rebooted.

- Smurf – An attacker floods the target network by sending ICMP ECHO REQUEST (*ping*) packets to a broadcast address (x.x.x.255) with the spoofed source address of the target. The target is then flooded with ECHO REPLY packets from multiple sources.
- Syslogd – The *syslog* server, which could be used to log alarms remotely from an IDS, is crashed by sending a spoofed message with an invalid source IP address. Due to a bug, the server crashes when a reverse DNS lookup on the IP address fails.
- Teardrop – Some operating systems (Windows 95, NT, and Linux up to 2.0.32) will crash when sent overlapping IP fragments in which the second packet is wholly contained inside the first. This exploits a bug in the TCP/IP stack implementation in which the C function *memcpy*() is passed a negative length argument. The argument is interpreted as a very large unsigned number, causing all of memory to be overwritten.
- UDP storm – This attack sets up a network flood between two targets by sending a spoofed UDP packet to the *echo* server of one target with the spoofed source address and port number of the *chargen* server of the other target.

2.1.3. Remote to Local Attacks

While probes and DOS attacks may exploit TCP/IP protocols, R2L attacks always exploit application protocols to gain control over the target. Kendall describes several attacks, which can be grouped as follows:

- Password guessing – Many users tend to choose weak or easily guessed passwords. An attack could try common passwords such as *guest*, the user name, or no password. If this fails, an attacker could use a script to exhaustively test every word in a dictionary. Any service requiring a password is vulnerable, for example, telnet, FTP, POP3, IMAP, or SSH.
- Server vulnerability – An attacker exploits a software bug to execute commands on the target, often as root. For example, buffer overflow vulnerabilities have been discovered in

sendmail (SMTP), *named* (DNS), and *imap*. Other bugs may allow a command to be unwittingly executed. For example, the PHF attack exploits a badly written CGI script installed by default on an old version of *apache*. The following HTTP command will retrieve the password file on a vulnerable server:

```
GET /cgi-bin/phf?Qalias=x%0a/usr/bin/ypcat%20passwd
```

- Configuration error – An attacker exploits an unintended security hole, such as exporting an NFS partition with world write privileges. One common error is setting up an open X server (using the command `xhost +`) when running a remote X application. The *xlock* attack scans for open X servers, then displays a fake screensaver which prompts the user to enter a password, which is then captured. *xsnoop* does not display anything; it merely captures keystrokes.
- Backdoors – Once a host has been compromised, the attacker will usually modify the target to make it easier to break in again. One method is to run a server such as *netcat*, which can listen for commands on any port and execute them (Armstrong, 2001).

2.1.4. Attacks on the Intrusion Detection System

It is reasonable to expect that if a system is running an IDS, then the IDS might be attacked. This could either be an evasion, such as port scanning with FIN or RST packets, which are less likely to be logged, or a denial of service attack such as *syslogd*.

Ptacek and Newsham (1998) and Horizon (1998) contend that it is not possible for a network sniffer to see exactly the same traffic as the target without an impractical level of knowledge about the target environment. For example, if two TCP segments overlap with inconsistent data, some operating systems will use the first packet, and others will use the second. If the IDS is unaware of which method is used, then an attacker could exploit this to present innocuous data to the IDS while presenting hostile data to the target. Another technique would be to use short TTL values to expire packets between the IDS and the target. Also, if the IDS does not verify IP or

TCP checksums, then an attacker could present the IDS with innocuous packets with bad checksums that would be dropped by the target. Many other exploits could exist, depending on how accurately the IDS implements the TCP/IP protocols. Many do so poorly. Ptacek and Newsham found that out of four commercial systems that they tested, all were vulnerable to some of these attacks, and furthermore, none could properly reassemble fragmented IP packets.

Newman et al (2002) tested one open source and seven commercial network intrusion detection systems costing $2500 to $25,000 on a high speed (100 Mbs.) network Most systems flooded the user with false alarms. Seven of the eight systems crashed at least once during their 31 day test period, often because the logs filled up. This study suggests that many intrusion detection systems would be easy targets of flooding attacks.

Even if the IDS properly reassembled IP packets and TCP streams (handling checksums, retransmissions, timeouts, invalid TCP flags, etc. correctly), it is still possible to elude detection by making small changes to the attack. Consider the PHF attack described in Section 2.1.3. An IDS might detect this by searching for the string "GET /cgi-bin/phf?" or something similar. NESSUS employs a number of features to evade simple string matching, such as the following.

- URL encoding – replacing characters with %XX, e.g. "GET %2F%63%69..."
- Modifying the path, e.g. "GET /foo/../cgi-bin/phf?" or "GET "/cgi-bin/./phf?"
- Replacing spaces with tabs, e.g. "GET<tab>/cgi-bin/phf?". This syntax is nonstandard, but accepted by the target web server.

Thus, it seems that an IDS must not only fully model the IP and TCP protocols of the hosts it protects, with all their idiosyncrasies and bugs, but also application protocols as well. If it does not, then a determined attacker could find a way to evade it.

2.2. Properties of Network Traffic

A network intrusion detection must distinguish between hostile and benign traffic, and must do so quickly to keep up with a high speed network. Depending on whether the IDS uses signature or anomaly detection, it must either model attacks (of which there are thousands) or normal traffic. There are two main challenges for modeling normal traffic (for anomaly detection). First, network traffic is very complex, and second, the model changes over time.

Many internet protocols are described in the Request for Comments (RFC), a set of documents dating back to 1969, which can be found at www.ietf.org. As of Feb. 14, 2003, there were 3317 documents totaling 147 megabytes of text. The set is growing at a rate of about 250 new documents per year. However, the documentation describes how protocols *should* behave, not how the thousands of different clients and server versions actually implement them, with all their idiosyncrasies and bugs. Nor do the RFCs cover every protocol. Some protocols may be proprietary and not documented anywhere.

Bellovin (1993) and Paxson (1998) found that wide area network traffic contains a wide range of anomalies and bizarre data that is not easily explained. Paxson refers to this unexplained data as "crud". Examples include private IP addresses, storms of packets routed in a loop until their TTLs expire, TCP acknowledgments of packets never sent, TCP retransmissions with inconsistent payloads, SYN packets with urgent data, and so on. Bellovin found broadcast packets (255.255.255.255) from foreign sites, ICMP packets with invalid code fields, and packets addressed to nonexistent hosts and ports. Many of these were investigated and found to be not hostile. Instead, many errors were caused by misconfigured routers or DNS servers.

Rather than try to specify the extremely complex behavior of network traffic, one could instead use a machine learning approach to model traffic as it is actually seen. For example, an IDS could be trained to recognize the client addresses that normally access a particular server by observing it over some training period. Unfortunately, research by Adamic (2002) and Huberman

and Adamic (1999) suggest that this approach would fail for some attributes (such as client addresses) because the list of observed values would grow at a constant rate and never be completed no matter how long the training period. Adamic found that the distribution of Internet domains accessed by a large number of HTTP clients (web browsers) has a power law or Pareto distribution, where the r'th most frequent value occurs with frequency $cr^{-1/k}$, where c and k are a constants and k is usually close to 1 (in this case, 1.07). When $k = 1$, the r'th most frequent value occurs with frequency proportional to $1/r$. Zipf (1939) observed this behavior in the distribution of word frequencies in English and several other languages (with $c \approx 0.1$). Since then, power law distributions have been observed in many natural phenomena such as city populations or incomes (Mitzenmacher 2001), and CPU memory accesses (Stone, 1993). We found that many attributes of network traffic collected on a university departmental server have Zipf-like distributions, for example, HTTP and SSH client versions and client addresses in TCP connections (Mahoney & Chan, 2003).

The problem with a power law distributed random variable is it implies that the "normal" set of values cannot be learned no matter how long the observation period. If we make n observations of a Zipf-distributed random variable and observe r distinct values, then the expected number of values occurring exactly once (denoted r_1) is $r/2$. By Good-Turing (Gale & Sampson, 1995), the probability that the next value of any discrete random variable will be novel is $E[r_1]/n$, which for a Zipf variable is estimated by $r/2n$. This implies that r (the size of the vocabulary to be learned) grows without bound. A similar argument can be made for power law distributions where k is not exactly 1.

A confounding problem in traffic modeling is that it is not possible to determine the average rate of many types of events (for example, bytes per second or packets per second for some packet type), regardless of the duration of the sampling period. It had long been assumed that network traffic could be modeled by a Poisson process, in which events are independent of each other. If we measured packet rates, for example, we would find random variations over small time

windows (e.g. packets per second), but these would tend to average out over longer periods (e.g. packets per month).

However, Leland et al. (1993) and Paxson & Floyd (1995) showed that this is not the case for many types of events. If we graphed packets per second or packets per month, they would both show bursts of high traffic rates separated by gaps of low activity. Furthermore, both graphs would look the same. A burst or gap could last for a fraction of a second or for months. The distribution of traffic rates would be independent of time scale. This behavior is a property of self-similar or fractal processes. Leland et al. formalizes the notion of a self-similar process as follows.

- Long range dependency. (For a Poisson process, events separated by a long time interval are independent).
- A nonsummable autocorrelation correlation, such as $1/t$. For a Poisson process, the autocorrelation decays exponentially, e.g. $e^{-t/T}$ for some time constant T.
- A Hurst parameter greater than 0.5.

The Hurst parameter characterizes the self-similarity of a process. It is defined as the rate at which the sample standard deviation of an aggregate process decreases as the aggregate size increases. For example, the aggregate size increases by a factor of M = 60 when going from packets per second to packets per minute. If the sample standard deviation of this process decreases by a factor of M^{1-H}, then the process is said to have a Hurst parameter of H. For a Poisson process, H = 0.5. For a purely self-similar process, H = 1. Leland et al. measured H in the range 0.7 to 0.9 for Ethernet packet rates and byte rates on networks of various sizes, with higher values of H when traffic rates were higher.

These results suggest that for some attributes it is impossible to either learn the full set of possible values of an attribute, or the average rate at which any individual value should be observed. Fortunately all is not lost. Paxson and Floyd found that some events, such as session interarrival times, can be modeled by a Poisson processes. Processes may also have predictable time-dependent

behavior. For example, traffic rates are higher during the day than at night. Also, not all attributes have power law distributions, and it may be possible to distinguish these attributes from the others.

2.3. Intrusion Detection

Intrusion detection systems can be categorized along three dimensions.

- Host or network based – A host based system monitors operating system events and the file system to detect unauthorized use (viruses, R2L or U2R attacks). A network based IDS monitors network traffic to and from one or more hosts to detect remote attacks (usually probes, DOS, or R2L).
- Signature or anomaly detection – A signature (or misuse) detection system searches for patterns or events signaling known attacks. An anomaly detection system signals a possible novel attack in the case of events that differ from a model of normal behavior.
- Hand coded or machine learning – A hand coded system requires the user to specify rules for normal behavior or specific attacks. A machine learning system generalizes from training data, which is either normal or contains labeled attacks.

For example, a virus detection program is host based (it examines files on the computer on which it runs), uses signature detection (pattern matching for known viruses), and is hand coded (using vendor-updated virus definition files). The subject of this dissertation is the exact opposite: network anomaly detection using machine learning to generalize from normal traffic. These methods differ in the types of attacks they detect (U2R or probe, known or novel). A good system will combine different techniques to increase coverage.

2.3.1. Machine Learning

Some intrusion detection systems use machine learning to model attacks (signature detection) or normal behavior (anomaly detection). The general problem of machine learning is

described in (Mitchell, 1997). We are given a set of instances (e.g. network packets or user sessions), some of which have category labels (e.g. normal or hostile). The problem is to assign labels to the remaining instances. In general, an instance is a set of attribute-value pairs (e.g. {port = 80, duration = 2.3 sec.}). An attribute is said to be *nominal* if the values are unordered. For example, port numbers are nominal because port 80 (HTTP) and 79 (finger) are no more related to each other than they are to port 21 (telnet), numeric values notwithstanding. An attribute with ordered values (such as *duration*) is *continuous*. In general, an instance can be represented as a vector of continuous values by assigning elements with values 0 or 1 to each nominal value, e.g. (port21 = 0, port79 = 0, port80 = 1, duration = 2.3).

Machine learning can be applied to either signature or anomaly detection. If we are given training instances labeled both normal and hostile (or labeled with the type of attack), then we are using signature detection. Such data is difficult to obtain. More commonly, we label all traffic as normal (perhaps incorrectly). In this case, any instance far outside the training set is assumed to be hostile. This is the basis of anomaly detection.

The following is a summary of some common machine learning algorithms. There are many others.

- Memorization – We categorize test instances by finding a matching training instance. If no exact match is found (a common problem), then this method fails (or we assume an anomaly).
- Nearest neighbor – We define a distance function between instances, perhaps Euclidean distance between vectors. A test instance is assigned the same category as its nearest neighbors from the training set. If a test instance has no nearby neighbors, it is anomalous.
- Naive Bayes – We use the training data to estimate a set of probabilities, P(category | attribute = value) for each category and each attribute. We then estimate the probability of the category of a test instance by taking the product of the probabilities for each attribute, e.g. P(hostile | port = 80, duration = 2.3) = P(hostile | port = 80)P(hostile | duration = 2.3)..

This approach is called *naive* because we are assuming that the attributes are independent. However this method often works well even when they are not.

- Neural networks – We assign an input neuron (a model of a brain cell) to each vector element and an output neuron to each category. These neurons and possibly other intermediate neurons are connected by weights, w. Given an instance, x, we compute the relative probability y_i that the category is *i* by $y_i = g(\Sigma_j w_{ij} x_j)$, where *g* is a bounding function that limits the output to the range 0 to 1. The network is trained by incrementally adjusting the weights w_{ij} to correctly categorize the training set, i.e. $y_i = 1$ if the category is *i*, and 0 otherwise. By adding intermediate neurons, nonlinear functions of the input can be learned.
- Decision Tree – A nested set of if-then-else rules are constructed of the form "if attribute < value then ...". These rules form a tree, where the leaf nodes specify a distribution of categories, e.g. "P(hostile) = 2%". A tree is constructed recursively using a greedy algorithm by finding the rule that maximizes the separation of categories (information gain) on the portion of the training set at each node.
- RIPPER (Cohen, 1995) finds a rule sequence of the form "if ... then ... else if ... then ... else if ..." to assign categories to a training set. A condition may be a combination of attributes, such as "if port = 80 and duration > 100 sec. then category = hostile, else if ...". It uses a greedy algorithm so that the first few tests apply to most of the training instances, and extends the rule until no more training instances can be categorized.
- APRIORI (Agrawal & Srikant, 1994) finds association rules. It makes no distinction between the category and other attributes. Given a training set, it allows *any* attribute to serve as the label and finds if-then rules that predict it based on other attributes. APRIORI is useful for market basket analysis. For example, does port number predict duration, or does duration predict port number?

2.3.2. Network Signature Detection

We review a representative sample of intrusion detection systems. For a more complete survey, see (Axelsson, 1999).

Network intrusion detection is usually rule based, although the rules may specify the behavior of an attack (signature detection) or rules that specify acceptable traffic (strict anomaly detection). For example, the user could specify that packets addressed to unused ports are not allowed, and list those ports. SNORT (Roesch, 1999) and Bro (Paxson, 1998) are two such systems. Both systems allow rules to be specified using scripts. For example, the SNORT distribution includes the following rule to detect PHF by searching for the string "/phf" in TCP data packets addressed to the normal HTTP port.

```
alert tcp $EXTERNAL_NET any -> $HOME_NET 80 (msg:"IDS128 - CVE-1999-0067 -
CGI phf attempt";flags:PA; content:"/phf";flags:AP; nocase;)
```

Other rule-based systems are components of larger systems that also use host based methods. These include P-BEST (Lindquist & Porras, 1999), which is one component of EMERALD (Newmann & Porras, 1999), and NetSTAT (Vigna & Kemmerer, 1999), which is a component of the STAT suite (Vigna et al., 2000).

2.3.3. Host Based Anomaly Detection

Many of the early anomaly detection systems surveyed by Axelsson (1999) model user behavior, for example, client IP addresses and normal login times. A login from an unusual address or at an unusual time of day would be deemed suspicious. Rules could either be learned or programmed.

DERBI (Tyson et al., 2000) is a file system integrity checker designed to detect signs of an intrusion, such as a backdoor. It can be considered an anomaly detection system in the sense that normal behavior is defined such that operating system files should never be modified.

Forrest et al. (1996) uses the analogy of a computer immune system to apply anomaly detection to *program* behavior in a host-based system by monitoring operating system calls. It was found that servers and operating system components make predictable sequences of operating system calls. When a programs is compromised in an R2L or U2R attack (for example, a buffer overflow), it executes code supplied by the attacker and deviates from the system call sequences observed during training. Forrest et al. used a type of nearest-neighbor approach: an *n*-gram model with $n = 3$ to 6 and would signal an anomaly if a program made a sequence of n calls not observed in training.

Other models are possible. For example, Ghosh and Schwartzbard (1999) trained a neural network with delayed feedback (an Elman network) on system calls to generalize from normal sequences. Schwartzbard and Ghosh (1999) also applied this technique to Windows NT audit logs. The neural networks were trained by labeling attack-free data as normal and by generating random training data labeled as hostile.

NIDES (Anderson et al., 1999) compares short and long term program behavior by comparing vectors of host-based attributes, such as number of files open, CPU time, and so on. The short term behavior (a single event or a small time window) represents the test instance, and the long term behavior (a large time window) represents the training model. As with most practical systems where guaranteed attack-free training data is not available, we just make the assumption that most of the data observed so far is attack free. Obviously if there are attacks in the training data, then repeat attacks of the same type are likely to be missed.

2.3.4. Network Anomaly Detection

Network anomaly detection systems usually monitor IP addresses, ports, TCP state information, and other attributes to identify network sessions or TCP connections that differ from a profile trained on attack free traffic. They are usually combined with signature detectors as components in larger systems.

SPADE (Hoagland, 2000), a SNORT plug-in, is an anomaly detection system which monitors addresses and ports of inbound TCP SYN packets (normally the first packet in a client-server session). By default, it models only the destination (server) address and port, and constructs a joint probability model by counting address/port combinations: P(address, port) = count(address, port) / count(all). If the current packet (included in the counts) has a probability below a threshold, it is deemed anomalous, and an alarm is generated. The threshold is varied slowly in order to keep the alarm rate constant. SPADE also has probability modes that include the source address and port. Unusual source addresses on servers that accept only a small list of trusted clients might indicate an unauthorized user.

ADAM (Barbara et al., 2001a; Barbara et al., 2001b) combines an anomaly detector trained on attack-free traffic with a classifier trained on traffic containing known, labeled attacks. Like SPADE, it monitors TCP connections. In addition to addresses and ports, ADAM also monitors subnets (the first 1-3 bytes of a 4-byte IP address), TCP state flags, the day of the week, and time of day. The anomaly detection component performs offline market basket analysis on attack-free traffic using techniques similar to RIPPER or APRIORI to find conditional rules among these attributes with high support and confidence. During testing, sessions which violate these rules are passed to the second component, a classifier (a decision tree) trained on traffic containing labeled attacks. Sessions which cannot be confidently classified as known attacks or normal are classified as unknown attacks.

eBayes (Valdes & Skinner, 2000), a component of EMERALD (Newmann & Porras, 1999) measures several attributes of a session, such as event intensity, error intensity, number of open connections, number of ports, number of addresses, and distributions of services and connection codes. Unlike SPADE and ADAM, eBayes considers a group of TCP connections within a short interval to be a single session. eBayes maintains a set of probability models, P(attribute = value | category) and uses naive Bayesian inference to assign a category to a session based on the observed attributes, which are assumed to be independent. Like ADAM, categories correspond to known

attacks, normal behavior, and a category for unknown attacks (which requires training data with labeled attacks). Unlike ADAM, eBayes adapts during the test phase in two ways. First, the probabilities are slowly adjusted to reinforce the most likely category, and second, there is a mechanism to automatically add new categories when there is not a good match between the observed values and existing categories.

2.4. Intrusion Detection Evaluation

Ideally an IDS should be evaluated on a real network and tested with real attacks. Unfortunately it is difficult to repeat such tests so that other researchers can replicate the evaluation. To do that, the network traffic would have to be captured and reused. This raises privacy concerns, because real traffic can contain sensitive information such as email messages and passwords. Thus, network traffic archives such as ITA (Paxson, 2002), and the University of New Mexico data set (Forrest, 2002) are sanitized by removing the application payload and some packet header fields, and scrambling IP addresses.

The DARPA/Lincoln Laboratory IDS evaluation (IDEVAL) data sets (Lippmann et al., 2000; Lippmann & Haines, 2000) overcome this difficulty by using synthetic background traffic. The goal of this project was twofold. First, the goal was to test a wide variety of systems (host or network, signature or anomaly, four different operating systems) on a wide range of attacks. The second goal was to provide off-line data to encourage development of new systems and algorithms by publishing a standard benchmark so that researchers could compare systems and replicate results.

Evaluations were conducted in 1998 and 1999. The 1999 evaluation improved on the 1998 evaluation by simplifying the scoring procedure, providing attack-free data to train anomaly detection systems, adding many new attacks, and one new target (Windows NT) to the three UNIX-based targets in 1998.

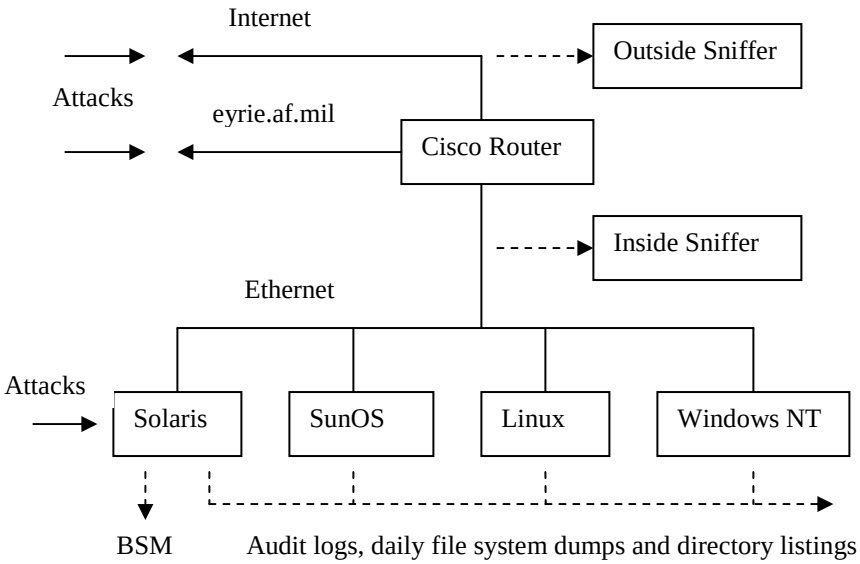

Figure 2.1. The 1999 DARPA/Lincoln Laboratory IDS Evaluation Test Configuration

Figure 2.1 shows the configuration of the 1999 evaluation test bed. A local area network is set up to resemble a portion of a typical Air Force base network. There are four main "victim" machines, running SunOS, Solaris, Linux, and Windows NT. Traffic generators simulate hundreds of other hosts and users running various applications and an Internet connection (Haines et al., 2001). The mix of protocols (HTTP, SMTP, telnet, etc.) and hourly variations in traffic volume are similar to traffic collected on a real Air Force network in 1998. Content is taken from public web sites and mailing lists, or synthesized using English word bigram frequencies.

The evaluation data set is collected from the four victim machines and from two network sniffers, an "inside" sniffer between the router and the victims, and an "outside" sniffer between the router and the Internet. The host based data consists of audit logs and nightly file system dumps and directory listings, in addition to Solaris Basic Security Module (BSM) system call traces. This data

is analyzed off-line to detect attacks. Attacks may originate from either the Internet, from compromised hosts on the local network, or from attackers who have physical access to the victims or the local network. Most attacks are against the four main victim machines, but some are against the network, the Cisco router, or against simulated hosts on the local network.

The 1999 evaluation had two phases separated by about three months. During the first phase, participants were provided with three weeks of data (collected Monday through Friday, 8:00 AM to 6:00 AM local time each day). The second week of data contained 43 labeled instances of 18 attacks which were taken from the Bugtraq mailing list, from published sources such as www.rootshell.com, or which were developed for the evaluation. Many of these attacks are described in Section 2.1. Attacks were sometimes made stealthy or hard to detect, for example, by slowing down a port scan or obfuscating suspicious shell commands. Attacks were labeled with the start time and the name of the victim. The first and third weeks contained no attacks, and could be used to train anomaly detection systems.

During the second phase, participants were provided with two weeks of test data (weeks 4 and 5) containing 201 unlabeled instances of 58 attacks, 40 of which were not in the training data. Participants were required to provide a list of alarms identifying the target address, time, and a numerical score indicating a confidence level in the alarm, and optionally, the type of attack. Participants also provided a system specification which describes the types of attacks their system is designed to detect. Attacks are classified by category (probe, DOS, R2L, U2R), the type of data examined (inside sniffer, outside sniffer, BSM, audit logs, file system dumps, or directory listings), victim operating system (SunOS, Solaris, Linux, or NT), and whether the attack is new (not in week 2). Systems are evaluated by the fraction of attacks detected out of those they are designed to detect at various false alarm rates (say, 10 per day, or 100 total) by ranking the alarms by score and discarding those which fall below a threshold that would allow more false alarms. An attack is counted as detected if there is an alarm with a score above the threshold that identifies the victim address (or any address if there is more than one) and the time of any portion of the attack with 60

seconds leeway before the start of the attack or after the end. Alarms that detect attacks that the system is not designed to detect (out of spec attacks), and duplicate alarms detecting the same attack are discarded without penalty. Any other alarm is considered a false alarm.

Eight organizations participated in the 1999 evaluation, submitting 18 systems. The top four systems reported by (Lippmann et al., 2000) are shown in Table 2.1. The best systems detected about half of the attacks they were designed to detect at a false alarm rate of 10 per day. Subsequent to the evaluation, the five weeks of training and test data were released to encourage research in intrusion detection. The data and truth labels are available at http://www.ll.mit.edu/IST/ideval/.

System	In-Spec Attacks	Detected at 10 false alarms per day
Expert 1 (EMERALD)	169	85 (50%)
Expert 2 (STAT)	173	81 (47%)
Dmine (ADAM)	102	41 (40%)
Forensics (DERBI)	27	15 (55%)

Table 2.1. Top results in the 1999 IDS evaluation (Lippmann et al., 2000)

Of the 58 attack types, there were 21 that were classified as poorly detected. None of the 18 systems detected more than half of the instances of any of these attacks. Often these were missed because the systems did not monitor the relevant protocols or because they were new attacks not found in the labeled training data. The attacks are as follows:

- Stealthy *ipsweep* (probe, slow ECHO REQUEST scan for active IP addresses).
- Stealthy *portsweep* (probe, slow port scan, or using stealthy techniques such as FIN scanning).
- *ls_domain* (probe, obtains a list of hosts using a DNS zone transfer).

- *queso* (probe, operating system fingerprinting using malformed TCP packets).
- *resetscan* (probe, port scan using unsolicited RST packets).
- *arppoison* (DOS, disrupting local traffic by forging replies to ARP-WHO-HAS packets).
- *dosnuke* (DOS, crashes Windows by sending urgent data in a NetBIOS request).
- *selfping* (DOS, crashes SunOS by sending an ICMP ECHO REQUEST to self).
- *tcpreset* (DOS, disrupts local traffic by forging RST packets to close TCP connections).
- *warezclient* (DOS, downloading illegal software from a local anonymous FTP server).
- *ncftp* (R2L, FTP client bug exploit).
- *netbus* (R2L, a backdoor server).
- *netcat* (R2L, another backdoor, uses a stealthy DNS channel).
- *snmpget* (R2L, exploits an easily guessed router password).
- *sshtrojan* (R2L, fake SSH server upgrade with login backdoor).
- *loadmodule* (U2R, exploits a trojan shared library to gain root).
- *ntfsdos* (U2R, an attacker with console access copies the disk, bypassing file system protection).
- *perl* (U2R, exploits a bug in a *setuid root* Perl script).
- *sechole* (U2R, exploits a bug in Windows NT).
- *sqlattack* (U2R, a restricted user escapes from a database application to the UNIX shell).
- *xterm* (U2R, gains root using a buffer overflow).

Out of 201 attack instances, 77 are poorly detected. Only 15 instances were detected by any system.

Chapter 3

Time-Based Protocol Modeling

This chapter introduces PHAD (Mahoney & Chan, 2001), a Packet Header Anomaly Detector. PHAD is unusual in two respects among network anomaly detection systems. First, it models protocols rather than user behavior, because many attacks exploit errors in protocol implementations that can be detected by unusual inputs or outputs. Second, it uses a time-based model, which assumes that network statistics can change rapidly in a short period of time. When PHAD sees a burst of unusual events, only the first event of that burst ought to be surprising and reported as an anomaly. This cuts down on false alarm floods.

3.1. Protocol Modeling

Most network anomaly detection systems are designed (perhaps implicitly) to distinguish authorized and unauthorized users. For example, an authorized user would know the environment and not attempt to contact nonexistent hosts or services as a port scan would. Also, servers requiring passwords (telnet, FTP, POP3, etc.) would have a regular set of authorized clients that could be identified by their source IP addresses, and perhaps other idiosyncrasies, such as time of day. Thus, unusual source addresses on these services would hint at unauthorized access. These methods are quite distinct from host based techniques that model *program* behavior. Forrest uses the model of an immune system to detect when a vulnerable server or operating system component is executing code that was not part of the original program, such as a root shell in a buffer overflow attack. Because program code does not change over time, programs tend to make characteristic

sequences of operating system calls. Any novel code executed by these processes is likely to result in sequences that deviate from this pattern.

Program modeling could be used in a network intrusion detection system by monitoring the output of vulnerable servers. Like system call sequences, we would expect a program's output to deviate from normal behavior when it executes novel code. For example, it would be unusual to see a root shell prompt coming from a mail server, as during a *sendmail* buffer overflow attack. Such an attack could not be caught by user modeling, because no unusual ports or addresses are accessed and mail servers will accept any client address without authentication.

Another approach is *protocol* modeling. In section 2.1, we saw that many attacks exploit bugs in protocol implementations. For example, *sendmail*, *imap*, and *named* exploit improper implementation of SMTP, IMAP, and DNS, in which the length of the input is not checked in some cases. *Teardrop* and *ping of death* exploit bad implementations of IP protocol, such that the target crashes when presented with unusual IP fragments that do not reassemble properly. *Queso* is able to identify some operating system versions because of bugs in their TCP implementations which cause them to give nonstandard responses to unusual data, such as TCP packets with the reserved flags set. A common theme of such attacks is that the input is unusual in some sense. There is a reason for this. If the error could be invoked by commonly occurring data, then the bug would have been quickly discovered and fixed.

Another source of protocol anomalies could come from bugs in the attacking code. Just as it is hard for the developer of a server or client and get all the details of a protocol right, it is hard for the attacker too. For example, attacks that spoofs the source address have to be programmed at the IP level using a technique such as raw sockets. This requires the attacker to fill in all of the other IP header fields, such as TTL, header length, checksum, fragmentation pointer, and so on. It is hard to get this right. Not only must the packet satisfy the protocol requirements, but if it is to elude detection, it must also duplicate the idiosyncrasies of the protocols in the target environment. For example, many operating systems generate predictable sequences of IP fragment ID numbers or

TCP initial sequence numbers (a fact exploited by attacks such as *queso*). The same reasoning applies to application protocols. For example, a normal SMTP session starts with a HELO/EHLO handshake. It is not required for an attack like *sendmail*, but if it was omitted, the attack could be detected.

A third source of protocol anomalies could come from attacks on the IDS. Many of the IDS vulnerabilities described in Section 2.1.4 are the result of differences in the IDS's and the target's implementation of TCP/IP. Because of the nature of software testing, differences exposed by the most common cases will be discovered and fixed first. Any remaining inconsistencies will only be exposed by rare events, such as small TTL values, IP fragmentation, inconsistent overlapping TCP segments, and so on.

To summarize, there are five potential ways to detect attacks.

1. Unusual ports and addresses, signs of an unauthorized user (traditional model).
2. Unusual outputs, signs of a successful attack.
3. Unusual inputs that exploit hidden bugs in the target.
4. Unusual inputs due to bugs in the attacking code.
5. Unusual inputs to exploit hidden bugs in the IDS.

3.2. Time Based Modeling

In Section 2.2 we saw that many types of network events occur in bursts separated by gaps, over both short and long time scales. Many network processes tend to be self-similar or fractal rather than Poisson, and to have a nonsummable autocorrelation function such as $1/t$ (where t is time). Events separated by long time intervals are not independent, as in a Poisson model. Instead there is a long range dependency.

Anomaly detection is the identification of rare events. In a frequency-based model, the average rate of events is estimated by counting the number of events and dividing by the

observation period. This is a poor way to model a bursty process because there *is* no average rate. For example, consider the following (bursty) sequence of 20 observations of an attribute: 00000000000000001111. What is the probability that the next observation will be a 1? If we assume that each event is independent, then 1 occurs 4 out of 20 times, so P(1) = 4/20 = 0.2.

However, if the events were independent, it is unlikely that all the "0"s and "1"s would be grouped as they are. (This type of pattern is common in network traffic). Such a sequence is more likely to originate from a process that has a state. At a minimum, the state might represent the previous output, such that the next output repeats with high probability. Without knowing more about the underlying process, we can propose the following model, which we will call the 1/*t* model: the probability of an event is inversely proportional to the time since it last occurred. The last observation of a "0" occurred 5 time units ago, so P(0) ~ 1/5. The last observation of "1" occurred 1 time unit ago, so P(1) ~ 1. Combining these, P(1) = 1/(1/5 + 1) = 5/6. We note that this is almost the same as we would obtain with a frequency based model going back just far enough to avoid probabilities of exactly 0 or 1. In this example, we must go back 5 observations, and observe the value "1" in 4 out of 5 times, for P(1) = 4/5. Thus, if the events really are independent, then the 1/*t* model will not be too far off.

Another possibility to consider is that of novel values. We did not explicitly state in our example that "0" and "1" are the only possibilities. By Good-Turing, P(novel) = E[r_1]/*n*, where r_1 is the number of values occurring exactly once (0 in our example), and *n* is the number of observations (20). However, Good-Turing only applies if the events are independent. If events occur in bursts, then there may be fewer values occurring exactly once, so Good-Turing is probably an underestimate.

The PPMC (prediction by partial match – method C) model proposed by (Bell et al., 1989) for data compression algorithms, does not require that events be independent. PPMC predicts P(novel) = *r/n*, where *r* is the number of observed values. In our example, *r* = 2 and *n* = 20, so P(novel) = 0.1. PPMC assumes that the number of observed values (*r*) grows at a steady rate so

that r/n is constant and independent of n. This is actually the case for attributes with a Zipf distribution, but is usually an overestimate for Poisson processes because $r \geq r_1$.

To apply time-based modeling to anomaly detection with explicit training and test periods, we assign an anomaly score for novel values of 1/P(novel) = tn/r, where n and r are counted during the training period, and where t is the time since the last anomaly. An anomaly may occur during either training or testing, with the difference that if a novel value is observed in training it is added to the set of allowed values, but if it occurs during testing it is not. Note that in our model, P(novel) = $(r/n)(1/t)$, which accounts for both the baseline rate of novel events, r/n, and a time-based model for events occurring outside the set of allowed values, $1/t$. For example, suppose we are given the following training and test sequences:

Training (time 0-19): 00000000000000001111 Test (time 20-24): 01223

During training, we record the set of allowed values {0, 1}, the size of this set, $r = 2$, and the number of observations, $n = 20$. If observations are made at unit time intervals starting at 0, then the last anomaly in training is "1" at time 16. The values "2", "2", and "3" at times 22, 23, and 24 in testing are anomalies because they are not in the training set. The anomaly score of the first "2" is $tn/r = (22\text{-}16)20/2 = 60$. The anomaly scores of the second "2" is $(23\text{-}22)20/2 = 10$. The anomaly score of the "3" is $(24\text{-}23)20/2 = 10$. The anomaly scores of "0" and "1" are 0 because the values occur at least once in training.

The anomaly score of an instance with more than one anomalous attribute is $\Sigma\, tn/r$, where the summation is over the anomalous attributes. It should be noted that there is no theoretical justification for summing inverse probabilities. If the attributes were independent and our probability model is correct, then we should use the product, $\Pi\, tn/r$. If the attributes were fully dependent, we could just select one arbitrarily, or perhaps take the maximum. In reality, the

attributes are neither independent or fully dependent, and we found experimentally that a summation works better in practice than these other possibilities.

We could also assign anomaly scores to values seen one or more times in training. For now we do not. This topic will be addressed in Chapter 6.

3.3. PHAD

PHAD is a simple time-based protocol anomaly detector for network packets. It scores every packet and makes no distinction between incoming and outgoing traffic. It models 33 attributes which correspond to packet header fields with 1 to 4 bytes. Fields smaller than one byte (such as TCP flags) are combined into one byte. Fields larger then 4 bytes (such as 6 byte Ethernet addresses) are split. The attributes are as follows:

- Ethernet header (found in all packets): packet size, source address (high and low 3 bytes), destination address (high and low 3 bytes), and protocol (usually IPv4).
- IP header: header length, TOS, packet size, IP fragment ID, IP flags and pointer (as a 2 byte attribute), TTL, protocol, checksum (computed), and source and destination addresses.
- TCP header: source and destination ports, sequence and acknowledgment numbers, header length, flags, window size, checksum (computed), urgent pointer, and options (4 bytes if present).
- UDP header: source and destination ports, checksum (computed), and length.
- ICMP header: type, code, and checksum (computed).

PHAD computes an anomaly score of $\Sigma\, tn/r$ over the anomalous attributes. n is the number of packets of the type appropriate for each field, e.g. the number of ICMP packets for the ICMP type field.

In order to store potentially large sets of training values (e.g. 2^{32} source or destination addresses), PHAD treats the attributes as continuous and clusters them into a maximum of C = 32

ranges. If the number of ranges ever exceeds C, then PHAD finds the smallest gap between adjacent ranges and merges them, effectively adding the values in the gap to the set of allowed values. For example, given the set {3, 5-10, 14-16} and C = 2, the smallest gap is between 3 and 5-10, so the new set becomes {3-10, 14-16}. *r* is computed as the number of anomalies in training, i.e. the number of times an element is added to the set, not including merges.

The method of approximating large sets is not critical, because it only affects attributes with large *r*, and therefore low scores. PHAD detects about the same number of attacks whether it uses C = 32, C = 1000, or stores a hash (mod 1000) of the value with no clustering.

3.4. Experimental Procedure

PHAD (and all systems to be described later) was tested on the 1999 DARPA/Lincoln Labs IDS evaluation (IDEVAL) data set described in Section 2.4. It was trained on the attack-free inside sniffer traffic from week 3, which contains 7 days of traffic (including two "extra" days). It was tested on weeks 4 and 5, which contains 201 attacks. We used the inside sniffer traffic because it can see inside attacks. However, one day is missing (week 4, day 2). In the evaluation, systems that used this data were not penalized for missing attacks on this day. The IDEVAL truth labels list 177 attacks visible in the inside sniffer traffic after removing this missing day and also removing attacks which do not generate any evidence in the traffic, such as attacks from the console.

We evaluated PHAD using the EVAL3 and EVAL4 programs (Mahoney, 2003b), which are our implementations of the 1999 IDEVAL detection criteria. as described in Section 2.4. PHAD identifies the target by using the destination IP address from the packet header. For an outgoing packet, this would actually be the source address, but the truth labels lists both the source and destination addresses for each attack, so we allowed a match to either. We consider this to be fair because it would be trivial to add knowledge of the home network to PHAD and have it output the appropriate address.

EVAL3 implements an alarm consolidation step prior to evaluation in which duplicate alarms identifying the same target address within a 60 second period are merged into a single alarm by keeping only the alarm with the highest score and discarding the others. This step can reduce false alarms from nearly any system because if there is an attack, the duplicates would be discarded anyway, and if not, only one false alarm is generated. Although the factor *t* in the anomaly score usually prevents consecutive high scoring alarms, it is possible that this could still occur if the anomalies are in different attributes of different packets. Alarm consolidation typically adds several detections to PHAD.

3.5. Experimental Results

EVAL3 computes the number of detections at several false alarm rates by sorting the alarms by descending score and discarding alarms after the false alarm limit is reached. At 100 false alarms (10 per day), PHAD detected 72 of the 201 attacks. PHAD was instrumented so that each alarm identifies the attribute which contributes the most to the anomaly score, the percentage of that contribution, and the anomalous value. An analysis of these results showed that the TTL field was responsible for more detections than any other attribute. We attributed these detections to simulation artifacts. TTL (time to live) is an 8-bit counter which is decremented each time an IP packet is routed. When TTL reaches 0, the packet is discarded in order to prevent infinite loops due to misconfigured routers. Many systems use a fixed TTL value for outgoing packets, such as 128 or 255, so the TTL value often indicates the number of network hops between the source and the sniffer. In the simulation, different physical machines simulating the same host IP address were used to generate some of the background traffic and some of the attacks (Haines et al., 2001). Apparently these machines may have been on different parts of the real network, and PHAD detected this difference.

3.5.1. Attacks Detected

We removed the TTL attribute from PHAD and detected 54 attacks at 100 false alarms with alarm consolidation (by EVAL3), or 48 attacks without consolidation (by EVAL4). The following list groups the 54 attacks by the attribute that contributes the most to its detection. Of these, 32 appear to be detected by features of the attack, 20 detections (marked with *) are not easily explained and might be simulation artifacts, and 2 (marked with **) are coincidental. The pair of numbers after each attribute shows the number of readily explained detections and the total number. For instance, out of the 11 attacks detected by source address, only 1 (*syslogd*) is easily explained.

- IP source address: 1/11 (*syslogd, portsweep*, smurf*, ncftp*, sendmail*, processtable*, xlock*, fdformat*, yaga**)
- TCP flags: 9/9 (*portsweep, queso, dosnuke*)
- IP fragment pointer/flags: 7/7 (*teardrop, pod*)
- IP packet length: 5/5 (*satan, syslogd, portsweep*)
- Ethernet packet size = 52: 4/4 (*ipsweep*)
- ICMP checksum = x0000: 0/3 (*smurf**)
- IP destination address: 0/3 (*portsweep*, warez*, sendmail**)
- Urgent pointer: 3/3 (*dosnuke*)
- TCP options: 0/2 (*apache2**)
- TCP source port: 0/2 (*portsweep**)
- UDP checksum: 2/2 (*udpstorm*)
- TCP checksum: 0/1 (*insidesniffer***)
- Ethernet source address: 0/1 (*insidesniffer***)
- Ethernet destination address: 1/1 (*mscan*, actually *arppoison*)

It is unfortunate that anomaly detection systems such as PHAD are sensitive to simulation artifacts in the IDEVAL data. This topic will be explored in more depth in Chapter 7. The artifacts important to PHAD are as follows:

- TTL, as previously discussed.
- Remote client IP addresses. There are too few (only 29 in week 3) to simulate a diverse range of clients on public services such as HTTP (web), SMTP (mail), and anonymous FTP. This affects detections of incoming packets by source address and outgoing packets by destination address. Although some attacks on private services should be detected this way (e.g. the telnet flood, *processtable*), PHAD makes no distinction between services. *fdformat* and *yaga* are U2R attacks and should not be detected at all. The only detection we consider legitimate is *syslogd*, in which the source address is forged so that reverse DNS lookup fails.
- TCP options. These are highly predictable on single hosts and in the IDEVAL background traffic, but not in real traffic. The anomaly in outgoing *apache2* packets (an HTTP flood) is a maximum segment size option (MSS = 1024), most likely a response to an idiosyncrasy of the attacking host. While a real attack might result in the same response, it is unlikely that it could be detected this way without a lot of false alarms.
- Source port. Only values up to about 33K appear in the IDEVAL background data, as opposed to real traffic (and *portsweep*) which uses the full range of values up to 64K.
- Coincidental detections. Both instances of *insidesniffer* are detected by coincidental false alarms. *Insidesniffer* (a sniffer on the local network) is a prolonged attack (hours) with multiple victims (all local hosts). Thus, any alarm on any host during this time would be counted as a detection.
- Overlapping attacks. The detection of *mscan* by invalid Ethernet address is actually due to a detection of *arppoison*, which occurs simultaneously. In an *arppoison* attack, an attacker

with access to the local network sends forged replies to ARP-who-has packets to disrupt local traffic. This attack cannot be detected directly because ARP packets do not have an IP address as required by IDEVAL criteria. Other *arppoison* detections appear as false alarms.

- *Smurf*. This is one of the few simulated attacks. *Smurf* floods a network with ICMP ECHO REPLY packets by sending ECHO REQUEST (ping) packets to a broadcast address with the spoofed source address of the target. These replies had to be simulated because the broadcast network did not really exist. Real replies would probably not have invalid checksums.

3.5.2. False Alarms

The top 100 false alarms are listed below. They are grouped by the attribute that contributes the greatest fraction to the anomaly score.

- TCP checksum: 32 errors
- IP destination address: 9
- UDP length: 9
- Ethernet source address: 8
- TOS (type of service): 7
- Ethernet destination address: 6
- TCP urgent pointer: 5 (values pointing outside the packet)
- TCP window size: 5
- TCP options: 4
- IP source address: 3
- 2 each: TCP acknowledgment number, TCP destination port, TCP flags (urgent data), IP fragmentation.

- 1 each: Ethernet packet size, IP packet size, TCP sequence number, IP type.

TCP checksum errors account for about a third of the false alarms. There are no checksum errors (IP, TCP, UDP, or ICMP) in any of the training data. Had there been, these false alarm scores might have fallen below the threshold. Other than TCP checksum and TOS, there is not much difference between the set of attributes that detect attacks and those that generate false alarms.

The eight Ethernet source address false alarms and one destination alarms are due to non-IP packets without an IP address. Five of these alarms occur during *arppoison* attacks.

3.5.3. Detection – False Alarm Tradeoff

In the previous sections, we analyzed PHAD at a threshold allowing 100 false alarms (10 per day). In Figure 3.1 we show the effects of varying this threshold in a detection-false alarm (DFA) curve. As the threshold is adjusted, there is a tradeoff between false alarms and missed attacks. As the threshold is lowered, both the number of attacks detected and the number of false alarms increases. However, after 100 false alarms, the number of detections levels off.

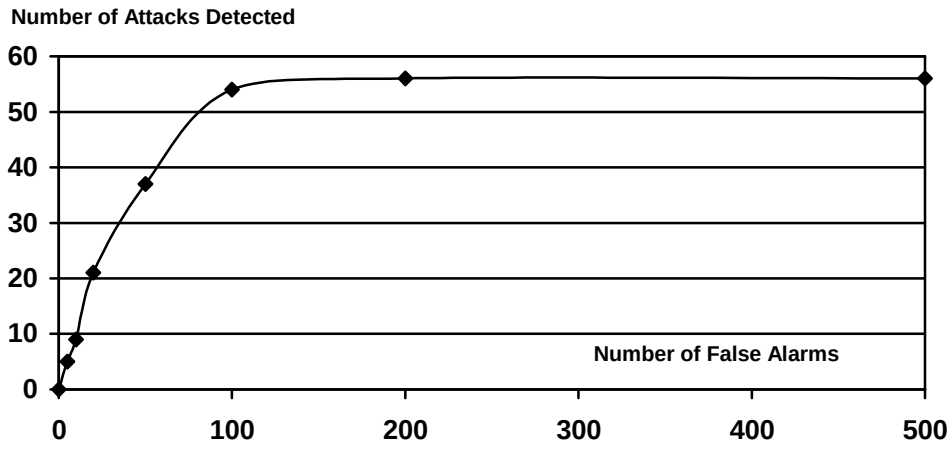

Figure 3.1. PHAD detection-false alarm curve for 0 to 500 false alarms.

3.5.4. Detections by Category

Table 3.1. lists the number and percentage of attacks (in-spec or not, legitimate or not) detected in each category at 100 false alarms. PHAD detects mostly probe and DOS attacks that exploit the protocols that it analyzes, namely those at the transport layer and below. It does poorly on R2L attacks, which generally exploit application layer protocols, as well as probe and DOS attacks on servers. It also misses most U2R and data attacks, which are not easily detected in network traffic by any means.

Among the 77 poorly detected attacks described in Section 2.4, PHAD detects 17: 9 stealthy *portsweep*, 1 stealthy *ipsweep*, 3 *aueso*, and 4 *dosnuke*. The detection rate for these attacks (22%) is not much lower than the overall rate (27%) which suggests that PHAD could be integrated with other systems to fill the gap.

Attack Type	**Detected/Total at 100 false alarms**
Probe	24/37 (65%)
DOS	22/65 (34%)
R2L	6/56 (11%)
U2R/Data	2/43 (5%)
Total	54/201 (27%)
Poorly Detected in 1999 evaluation	17/77 (22%)

Table 3.1. PHAD detections by attack category.

3.5.5. Implementation and Run Time Performance

PHAD was implemented with about 400 lines of C++. On a Sparc Ultra-60 with a 450 MHz 64-bit processor, PHAD processes 2.9 gigabytes of training data and 4.0 gigabytes of test data in 364 seconds (310 user + 54 system). This is about 95,900 packets per second, or 23 seconds per simulated training/test day. Training and testing speeds are approximately equivalent.

On a Compaq Presario with a 750 MHz AMD Duron processor running Windows Me, PHAD runs in 15 minutes on the same data. The primary limitation appears to be disk speed, although no tests were done to confirm this.

Memory requirements are negligible. PHAD allocates about 4K bytes of memory to store the anomaly model.

Source code is available at (Mahoney, 2003b).

3.6. Summary

PHAD introduces two innovations in network anomaly detection: time based modeling and protocol modeling. Time-based modeling regulates the flood of alarms that would otherwise be caused by bursts of anomalous events. Protocol modeling allows PHAD to detect four of the five attack categories described in Section 3.1. Examples of each are given below.

- Unusual outputs as symptoms of a successful attack: *arppoison* causes victims to send packets with the wrong Ethernet addresses.
- Unusual inputs to exploit bugs: Attacks that exploit IP fragmentation (*teardrop, ping of death*) are detected by the presence of fragments, which are normally rare. *Dosnuke*, which crashes Windows by sending urgent data to the NetBIOS port, is detected by the URG flag and urgent pointer. Urgent data is a rarely used feature of TCP.
- Bugs in the attack: The initiating packet in *udpstorm* has an incorrect checksum. The checksum is apparently not verified, but failing to set it correctly allows the attack to be

detected. (The actual storm is missed). *IPsweep* could also be hidden by using larger packets (a parameter to *ping*).

- Attacks on the IDS. Some of the *portsweep* probes are FIN scans, which are used because FIN packets are less likely to be logged. (The attack is detected because the ACK flag is not set).

No attacks are detected (legitimately) by user modeling, the dominant form of detection in most other systems. PHAD cannot detect attacks by source address because it makes no distinction between public and private services. It does not detect attacks by server port because it makes no distinction between server and client traffic. Also, PHAD detects (legitimately) only probes and DOS attacks. It does not detect R2L attacks because these normally exploit application protocols, which PHAD does not examine. We address these shortcomings in the next chapter.

Chapter 4

Application Layer Modeling

This chapter introduces ALAD (Mahoney & Chan, 2002b), an Application Layer Anomaly Detector. ALAD differs from PHAD in two respects. First, it models the application payload of TCP connections to detect attacks on servers. Most R2L attacks, which PHAD misses, are of this form. Second, ALAD uses conditional models, such as P(source address | destination port), rather than global models such as P(source address). This allows ALAD to model public and authenticated services separately in order to detect anomalous client addresses only on those services where novel addresses are not expected. Conditional models could be applied to arbitrary combinations of attributes.

4.1. ALAD

ALAD has two stages, one to reassemble inbound client TCP packets into streams, and a second stage to perform anomaly detection on these reassembled streams. We examine only inbound TCP traffic from clients to servers (identified by a low numbered destination port). Most R2L attacks, as well as probes and DOS attacks on applications, target servers. Although there is useful information in the server's response, we examine only the request to the server because four of the five sources of anomalies described in Section 3.1 occur in input rather than output. Also, very few attacks in PHAD (just *arppoison*) were detected by monitoring output. Restricting the data set to inbound client traffic greatly reduces the load on the IDS.

We consider the following six attributes of reassembled client TCP streams.

- Source (remote client) IP address.
- Destination (local server) IP address.
- Destination (server) port number (which identifies the protocol).
- TCP flags of the first packet and last two packets, e.g. "SYN/FIN-ACK/ACK".
- Application keywords (the first word on a line, delimited by white space).
- Application arguments (the rest of the line, delimited by a linefeed).

The first three attributes should allow us to detect user anomalies. TCP flags should allow us to detect unusual states, such as connections that are never closed. Protocol modeling takes place in the application payload. The application protocols are assumed to be text based, with lines of the form "keyword arguments", and ending with a linefeed, which is often true. For example, in the line "GET / HTTP/1.0" , the keyword would be "GET" and the argument would be "/ HTTP/1.0". For HTTP and SMTP, we model only the header, which is delimited by a blank line. For all protocols, we examine no more than the first 1000 bytes.

ALAD uses the same time-based model as PHAD, anomaly score = tn/r for novel values and 0 for any value observed at least once in training. For a conditional model such as P(source address | port), a separate t, n, r, and set of observed values is maintained for each value of the condition (i.e. for each port). n is the number of times the condition was met, r is the number of values for that condition (addresses for that port), and t is the time since the last anomaly for that condition. ALAD computes the anomaly score for a TCP stream by summing the anomaly score tn/r over all of the attributes for each complete line of text, then summing those scores to get the final score.

It is not immediately clear which combinations of attributes should be modeled and which should be conditions. Any combination of one or more attributes may be in the consequent, and any of the zero or more remaining attributes may be in the antecedent. In general, if there are m attributes, then there are $\Sigma_{0 \le i < m} 2^i \, m!/(i!(m-i)!)$ possible combinations, which grows exponentially

with m. For $m = 6$, there are 665 possible rule forms. Rather than test all of these rule forms, we hand pick combinations for testing based on the assumption that arguments depend on keywords and that the other attributes depend on the server address and/or port.

Once we have selected a small set of rule forms, we can combine them in at least two ways. One way is to add the associated anomaly scores, as previously mentioned. Another is to use each rule form by itself in a separate IDS, then run them in parallel and merge the alarms. In this mode, we would want to set each IDS threshold so that they all produce the same number of alarms, then consolidate the duplicate alarms when two or more systems identify the same target at the same time. Our approach is to find a small number, k, of rule forms that do well individually in terms of detecting attacks, then to exhaustively test all $2^k - 1$ possible mergers of these systems. It is not always optimal to merge all the systems, because if we merge two systems that detect the same attacks then no new attacks are discovered but we could double the number of false alarms. However, if two systems detect different attacks in roughly equal numbers, then there is a possibility that the merged combination could improve on both components.

We build our final system by adding together the anomaly scores of those rule forms from an optimal merge. This is not optimal, because merging and adding do not give the same results, although they are similar. However, we use this approach rather than exhaustively test all combination of sums because evaluating combinations of merges can be done quickly. To evaluate merged systems, we sort the alarms by score, consolidate duplicates, and evaluate equal numbers of alarms from each system in a round-robin fashion until the false alarm threshold is reached. Evaluation run time is dominated by sorting the alarms by score, which only has to be done once for each system. If we exhaustively tested summations rather than merges, then each combination would require a sort operation.

4.2. Experimental Results

We evaluated ALAD on the same data as PHAD. It was trained on week 3 of the 1999 IDEVAL inside sniffer traffic, and tested on weeks 4 and 5. Results were evaluated with EVAL3 at 100 false alarms (10 per day). We found $k = 11$ rule forms that detect between 13 and 44 attacks, as shown in Table 4.1. Next we used EVAL3 to exhaustively test all $2^{11} - 1 = 2047$ merged combinations of these 11 outputs. The optimal merged combination was found to consist of 5 of the top 6 rule forms as shown in Table 4.1, detecting 63 attacks. When we replace the merge with a summation, ALAD detects 60 attacks.

PHAD and ALAD detect sufficiently different attacks that their results can be merged. PHAD detects 54 attacks by itself, but the merger with ALAD detects 73.

Rule Form	Attacks detected at 100 false alarms
1. P(client address | server address)	44
2. P(client address | server address and port)	42
3. P(keyword | server port)	34
4. P(TCP flags | server address)	29
5. P(arguments | keyword)	26
6. P(server address and port) (unconditional)	26
7. P(server port | server address)	25
8. P(TCP flags) (unconditional)	23
9. P(arguments | port, keyword)	23
10. P(TCP flags | server address and port)	13
11. P(server address | client address)	13
ALAD = models 1 + 2 + 3 + 4 + 6	60
PHAD	54
PHAD + ALAD	73

Table 4.1. Attacks detected by ALAD rule forms on IDEVAL inside weeks 3-5 at 100 false alarms.

The attacks detected by ALAD are best understood by examining the results from individual rule forms rather than their combination.

4.2.1. Client Address Detections

Rule forms 1 and 2 (client address) detect almost the same attacks. Form 2, which is more specific (modeling each server address/port combination separately, rather than modeling each server address) detects fewer attack instances (42 vs. 44) but more attack types. As expected, most of these are attacks on servers rather than the TCP/IP stack, as in PHAD. As with PHAD, we mark detections that seem unlikely to occur in real traffic with an asterisk. The detections are as follows:

- Probes: *mscan, ntinfoscan, satan*. These test a wide range of server vulnerabilities.
- DOS: *apache2** (HTTP), *crashiis** (HTTP), *mailbomb** (SMTP), *warezclient** (anonymous FTP), *warezmaster** (anonymous FTP, form 2 only), *arppoison** (ARP, form 2 only). Unfortunately the most likely explanation for the detection of these attacks on public services is that the range of legitimate addresses is unrealistically small in the IDEVAL simulation. *Arppoison* is probably coincidental.
- R2L: *dict* (password guessing), *ftpwrite** (form 2 only), *netbus**, *netcat** (backdoors using DNS/TCP), *phf** (HTTP), *ppmacro** (an emailed trojan), *sendmail** (SMTP), *sshtrojan*. Only *sshtrojan* (a backdoored server) uses an authenticated protocol (SSH).
- U2R: *casesen**, *eject**, *fdformat** (form 2 only), *ffbconfig**, *xterm**, *yaga**. These are most likely cases of detecting the FTP upload of the exploit code or the shell session that executes the attack. We could not expect these attacks to be detected if they were executed by authorized users.

4.2.2. TCP State Detections

Rule forms 4, 8, and 10 detect three kinds of anomalies.

- Locally initiated FTP data connections on port 20, identified by an initial SYN-ACK rather than SYN. Attacks are detected this way only because the anonymous FTP server is never

used to upload files in training. Uploads include U2R exploit code: *casesen**, *eject**, *fdformat**, *sechole**, and *xterm**, and the FTP server exploits *satan, warez** and *ftpwrite*.

- Unclosed connections in response to a DOS attack, indicated by the absence of a trailing FIN or RST. These include *apache2*, *crashiis*, and *tcpreset*.
- Reset connections. Detects *ntinfoscan*. It is unusual for a client to open a connection and then close it with a RST packet.

Rule form 4, which models each server address separately and is used in ALAD, detects all types of anomalies. The unconditional model (8), detects FTP uploads almost exclusively, and would probably be of little use in a more realistic environment. The most specific rule form (10), which models by address/port combination, detects mostly unclosed or reset connections.

Out of 60 false alarms for rule form 4, 53 are for outgoing sessions (FTP uploads) 5 are client initiated RST packets, and 2 are unclosed connections. For the unconditional model (8), there are 59 uploads and one RST. For the most specific model (10), there are 25 false alarms distributed as follows: port 22 (SSH): 6 RST by the client, 3 unclosed (no FIN), 2 unopened and unclosed.(no SYN or FIN); for port 23 (telnet): 2 RST and one unopened and unclosed; for port 80 (HTTP): 8 unopened (no SYN), 2 unclosed, and one RST. All of these distributions are similar to the hostile traffic.

4.2.3. Keyword Detections

Rule form 3 models keywords by server port number. Although every server port is modeled, all attacks are detected on ports 21 (FTP), 23 (telnet), 25 (SMTP) or 80 (HTTP), which makes up most of the background TCP traffic. The attacks detected are as follows:

- *apache2*, an HTTP flood: an invalid command "x" on port 80 in 1 of 3 instances.
- *casesen** (U2R): "PWD" and "STOR" on port 21 (uploading the exploit code), or "QUIT" on port 25 (possibly emailing the code with a nonstandard mail client).

- *crashiis*, eject*, fdformat*, ftpwrite, mscan, sechole*, warez, xterm*: "STOR" on port 21 (FTP upload).
- *framespoofer**: "Content-Transfer-Encoding:" on port 25, commonly found in email headers but apparently absent in the training traffic. This attack uses an email to direct the victim to a website.
- *ftpwrite*, insidesniffer**: "QUIT" on port 25. Possibly coincidental because neither attack uses email.
- *mailbomb* (an email flood): "mail" on port 25. Normally this command is upper case.
- *ncftp* (an FTP client exploit): "PWD" or "RSET" on port 21. "PWD" (print working directory) should be common.
- *ntinfoscan*: "HEAD" on port 80, a valid HTTP command but not used by older browsers, "quit" and "user" on port 21, unusual but valid use of lower case.
- *phf*: a null byte on port 80 in one instance, not valid, but not part of the attack either.
- *processtable**: "^Iboundary="KAA04098.922893276/169-215-104" on port 25, a coincidental detection because the attack does not use email. Most (60%) of the top 100 false alarms are of this form.
- *satan*: "QUIT" on ports 23, 25 and 80, "user" on port 21,
- *sendmail**: "Sender:" on port 25, should be common in email headers.

The top 100 false alarms are distributed as follows.

- 73 on port 25 (mail): 59 of the form "^Iboundary="..."" (used to delineate attachments), 9 "QUIT", 5 others.
- 17 on port 21 (FTP): 14 "STOR" (upload) and 3 "SYST^@" (null byte appended).
- 6 on port 23 (telnet): various commands.
- 3 on port 80 (HTTP): 1 "HEAD", 2 with null bytes.
- 1 on port 515 (printer): "^C517".

4.2.4. Keyword Argument Detections

Although ALAD does not include a model for keyword arguments, this rule form by itself (conditioned on keyword) detects 26 attacks. Most are detected in SMTP arguments, but would probably be missed in an environment receiving email from many sources.

- *ffbconfig*, netbus*, sshtrojan*, xterm**: EHLO arguments, e.g. "EHLO ppp5-213.att.net.hk", identifying the client from which the exploit code is emailed.
- *mailbomb**: "mail from:<asdfg@hotlips.com>" identical in all three instances.
- *ps**: "MAIL From:<suzannac@marx.eyrie.af.mil> SIZE=1989".
- *sendmail**: "Content-Type: text/plain; charset=us-ascii".
- *ntinfoscan**: "HELO hobbes.eyrie.af.mil".

One FTP attack is detected.

- *ncftp*: "LIST -d y2kfix" in all 4 instances. *y2kfix* is a program containing the exploit.

The following HTTP attacks are detected.

- *apache2*: "User-Agent: sioux". The attack repeats this line thousands of times to slow down the web server.
- *phf**: "Accept: application/applefile, application/x-metamail-patch, ...", a novel HTTP client, probably would be missed.
- *portsweep**, *queso**: "User-Agent: Mozilla/4.08 [en] (WinNT; I)", coincidental, neither attack uses HTTP. This exact string makes up 28% of the false alarms.

Of the top 100 false alarms, 59 occur in HTTP commands, mostly in the arguments to User-Agent, Accept, and Content-Type. Usually these values are fixed for a given client. Most of the remaining false alarms are in SMTP, mostly in the "EHLO" and "Received:" fields, which contain the sender's address.

4.2.5. Server Address/Port Detections

Rule form 6 unconditionally models server address/port combinations, which ought to detect probes that access unused ports, such as portsweep. However, the surprising results is that very few of the attacks detected are probes. Instead, the following are detected.

- *casesen*, crashiis*, eject*, fdformat*, ftpwrite*, sechole*, warez**: port 20 (FTP uploads).
- *ncftp*, xterm**: port 21 (FTP). These attacks would only be detected if the FTP server was otherwise unused. One *ncftp* instance is also detected on port 113 (authentication).
- *netcat, netcat_breakin*: port 53 (DNS). This backdoor uses DNS/TCP as a stealth channel to penetrate firewalls. However all of the normal DNS traffic is UDP.
- *guesspop*: port 110, guessing passwords on an unused POP3 server.
- *satan*: ports 20 and 70, probing for FTP and gopher servers.
- *mscan*: ports 20, 21, and 111, probing FTP and the portmapper service.

There are 37 false alarms. Of these, 28 are on port 20 (FTP data), 3 on ports 21 and 113, and one each on ports 22, 139, and 1023. Rule form 8 (port given server address) gives similar results for both attacks and false alarms. In both cases the attacks and false alarms have similar distributions with no easy way to distinguish them.

4.2.6. Detection – False Alarm Tradeoff

Figure 4.1 shows the DFA curve for ALAD using the optimal combination of rule forms (1, 2, 3, 4, and 6). As with PHAD, most attacks are detected at a threshold allowing 100 false alarms (10 per day), although the number of detections continues to rise slowly, from 60 up to 72 at 500 false alarms.

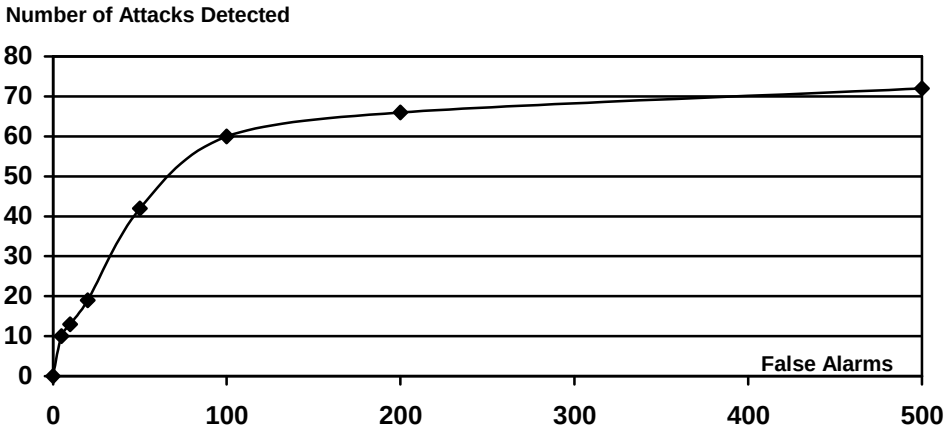

Figure 4.1. ALAD detection-false alarm curve for 0 to 500 false alarms.

4.2.7. Detections by Category

Table 4.2 lists detections by category for the optimal combination of rule forms. The detection rate is highest for R2L attacks, which normally exploit the application layer. In addition, most of the DOS attacks (all but one *tcpreset*) and probes (all but one coincidental *insidesniffer*) exploit application protocols as well. ALAD detects more R2L attacks but fewer probes than PHAD.

The poorly detected attacks (from the 1999 evaluation) include *ncftp, netbus, netcat, sechole*, *sshtrojan*, *tcpreset*, *warezclient*, and *xterm*. ALAD detects 18% of these, compared with 30% of attacks in general.

Attack Type	Detected/Total at 100 false alarms
Probe	6/37 (16%)
DOS	19/65 (29%)
R2L	25/56 (47%)
U2R/Data	10/43 (23%)
Total	60/201 (30%)
Poorly Detected in 1999 evaluation	14/77 (18%)

Table 4.2. ALAD detections by category.

4.2.8. Implementation and Run Time Performance

ALAD was implemented in two parts: a 400 line C++ program to reassemble TCP packets into streams, which is input to a 90 line Perl program. TCP reassembly of 6.9 GB of training and test data (inside weeks 3-5) takes 17 minutes on the 750 MHz PC described in Section 3.5.5. The output of this program is two text files: 20 MB of training data and 40 MB of test data. The Perl script processes this data in 60 seconds. Source code is available at (Mahoney, 2003b).

4.3. Summary

ALAD introduces the following new concepts.

- Modeling text-based protocols at the application layer. Keywords can be conditioned on server port, or arguments on keywords.
- Modeling using conditional rule forms. A rule form consists of a set of rules, one for each possible value in the condition.

- Merging alarms. This seems to work best when both systems are equally strong but detect different kinds of attacks. However, the technique we used is off-line because it would require setting thresholds in advance to produce the same number of alarms from each system.

ALAD detects each of the five categories of anomalies described in Section 3.1.

- Unauthorized users: *sshtrojan* by source address, *mscan* and *satan* by destination port.
- Unusual data to exploit bugs: "QUIT" from *satan*, RST packets from *ntinfoscan*.
- Bugs in the attack: lowercase commands in *mailbomb* and *ntinfoscan*, garbage data in *phf*.
- Evasion: DNS/TCP traffic from *netcat*.
- Symptoms from the victim: missing FIN packets from *apache2, crashiis*, and *tcpreset* indicating interrupted TCP connections.

Attacks and false alarms usually have similar distributions, regardless of the attribute. There is no obvious way to distinguish them to improve the results. However there are a few exceptions, such as the "^Iboundary=..." keywords in SMTP, which makes up most of the keyword false alarms. Each keyword is unique because there is no space to separate what should be the argument. This could be fixed by limiting the length of a keyword.

A shortcoming of ALAD is that it requires an ad-hoc approach to selecting rule forms. We address this problem in the next chapter.

Chapter 5

Learning Conditional Rules

This chapter introduces LERAD (Mahoney & Chan, 2002a), Learning Rules for Anomaly Detection. It differs from ALAD in that it generates rules from arbitrary combinations of nominal attributes, eliminating the need to select rules in an ad-hoc fashion. Rules have the general form "if $A_1 = v_1$ and $A_2 = v_2$ and ... and $A_k = v_k$ then $A_{k+1} \in V = \{v_{k+1}, v_{k+2}, ..., v_{k+r}\}$", where the A's are attributes and v's are values. LERAD selects those rules from this huge rule space that would generate the highest anomaly scores, i.e. those that have high *n* and low *r*, where *n* is the number of training instances that satisfy the antecedent ($A_1 = v_1$ ... and $A_k = v_k$, $k \geq 0$), and $r = |V|$, the number of allowed values in the consequent. This goal is different from algorithms such as RIPPER (Cohen, 1995) or APRIORI (Agrawal & Srikant, 1994), which have the goal of finding rules that predict a single value in the consequent with high probability (i.e. high confidence). Although all three algorithms find rules with high support (large *n*), the goal of LERAD is to find rules with small *r*, regardless of the distribution within V.

For example, one rule might be "if *port* = 25 then *word1* ∈ {"HELO", "EHLO"}". Unlike ALAD, there need not be a rule for every value of *port*. During training, LERAD counts *n*, the number of instances where *port* = 25, and records V, the set of values for *word1*. During testing, if LERAD observes an instance where port = 25 but *word1* is not "HELO" or "EHLO", then it assigns an anomaly score of *tn/r* (as with PHAD and ALAD) where $r = |V| = 2$ and *t* is the time since the rule was last violated. The total score assigned to a test instance is the sum of the anomaly scores assigned by each rule in the rule set for which the antecedent is satisfied but the consequent is not.

5.1. Rule Learning

The challenge of generating good rules (high *n/r*) is searching the huge rule space, which grows exponentially with the number of attributes. RIPPER and APRIORI use greedy deterministic algorithms that gradually add conditions to the antecedents while satisfying the constraint of high support and confidence. LERAD differs in that it uses a randomized algorithm to generate candidate rules, then tests them on increasingly large subsets of the training data, discarding redundant rules and those that do not satisfy the constraint of high *n/r*. The steps are as follows:

1. Randomly generate rules with *n/r* = 2/1 on pairs of training instances.
2. Discard redundant rules in favor of those with higher *n/r* on a larger training sample, S.
3. Discard rules that perform poorly on the full training set (where *r* increases near the end).

LERAD makes two passes through the training data, one to sample training instances in steps 1 and 2, and a second pass to fully train the rules in step 3.

5.1.1. Generating Candidate Rules

The first step in LERAD is to generate a pool of candidate rules. This is done by randomly selecting pairs of instances from the training set and finding rules that satisfy both instances. Such rules can be found whenever one or more attributes have the same value in both instances. When this happens, one attribute becomes the consequent and any subset of the remaining attributes can become the antecedent. For example, suppose we are given the following two training examples.

port = 80	word1 = GET	word2 = /	word3 = HTTP/1.0
port = 80	word1 = GET	word2 = /index.html	word3 = HTTP/1.0

There are three matching attributes, *port*, *word1*, and *word3*. Some of the 12 possible rules with *n/r* = 2/1 are:

- word1 = "GET"
- if word3 = "HTTP/1.0" then port = 80

- if port = 80 and word1 = "GET" then word3 = "HTTP/1.0"

In general, if there are *k* matching attributes, then there are *k* possible consequents and 2^{k-1} possible subsets of the remaining attributes to form the antecedent, allowing for $k2^{k-1}$ possible rules. In practice, LERAD picks a random subset of these rules as candidates. There are many ways to do this, but LERAD uses the following algorithm:

- Randomly select a sample S training instances
- Repeat L times
    - Randomly select a pair of training instances from S
    - Randomly order the *k* matching attributes in a sequence, but not more than k_{max}
    - Generate *k* rules using 1 through *k* attributes, making the first one the consequent and the others the antecedent.

Figure 5.1. LERAD candidate rule generation algorithm.

For example, if the matching antecedents of the pair above were picked in the order *word1*, *port*, *word3*, then the rules would be:

- word1 = "GET"
- if port = 80 then word1 = "GET"
- if port = 80 and word3 = "HTTP/1.0" then word1 = "GET"

Experimentally, we find it makes little difference whether the random pairs are selected from S or from the full training set, even with |S| as small as 20. We use |S| = 100. The set S is also used in step 2. We select L = 1000 pairs, again experimentally finding little improvement after a few hundred. We stop at $k_{max} = 4$ matching attributes because rules that make the final rule set rarely have more than 2 conditions in the antecedent, and many have none.

5.1.2. Removing Redundant Rules

The second major step in LERAD is to remove rules that do not give us any new information about a small sample training set, S. When two rules predict the same values, we keep the one with the higher *n/r* (when trained on S), or in case of a tie, the one with fewer conditions in the antecedent. For example, suppose S is as follows:

port = 25 word1 = HELO word2 = pascal

port = 25 word1 = HELO word2 = hume

port = 80 word1 = GET word2 = /

And suppose we have the following rules, sorted by descending *n/r*:

1. if port = 25 then word1 = "HELO" (n/r = 2/1)
2. word1 = "HELO" or "GET" (n/r = 3/2)
3. if word2 = "pascal" then word1 = "HELO" (n/r = 1/1, 1 condition)
4. if word1 = "GET" and word2 = "/" then port = 80 (n/r = 1/1, 2 conditions)

Rule 1 predicts *word1* in the first two training instances. Rule 2 is not redundant because it predicts *word1* in the third training instance, which was not predicted by a previous rule. Rule 3 is redundant because the only value it predicts was also predicted by rule 1 (and rule 2). Thus, rule 3 is removed. Rule 4 is not redundant because none of the previous rules predicted *port* in the third training instance. The algorithm is as follows:

- Sort the candidate rules by descending *n/r* on S, or by ascending size of the antecedent in case of ties.
- For each rule
    - For each sample in S
        - If the sample satisfies the antecedent, then mark the consequent value in S unless already marked
    - If no new values in S could be marked, then remove the rule

Figure 5.2. LERAD redundant rule elimination algorithm.

LERAD uses |S| = 100. Experimentally, values between 20 and several thousand work well. Very large values can result in too few rules being found redundant and slightly reduce the number of detections.

5.1.3. Removing Poorly Performing Rules

The third major step in LERAD is to remove any remaining rules that are likely to generate a lot of false alarms, based on their behavior towards the end of training on the full training set. The best attributes for anomaly detection are those whose distribution is stable over time, for example, the set of client addresses seen on a POP3 server where the same people log in every day to retrieve their mail. A poor attribute would be one in which the potential set of values is very large and the values seen each day varies, for example, the set of client addresses seen on a web server which could be viewed by millions of people. In this case only a small subset of addresses would be seen during training, and we would expect new values to appear in testing, resulting in false alarms.

Fortunately it is easy to distinguish these two cases from the training data. We simply count the number of novel values are added to the set V of allowed values near the end of the training period, for example, the last day. Had this been during the test period, all of these new values would be false alarms, since we know that there are no attacks in training. We would expect this false alarm rate to continue during the test period, so if the rate is high, we remove the rule. LERAD uses the following algorithm:

- Train all rules from step 2 on the full training set.
- If *r* increases at all during the last T_{val} percent of the training data, then remove the rule.

Figure 5.3. LERAD rule validation algorithm.

We found that T_{val} = 10% maximized accuracy on the IDEVAL test set.

5.1.4. Alarm Generation

Once LERAD generates a set of rules on the training data and fixes *n/r* for each rule, the test data is evaluated. If a test sample meets the conditions of a rule antecedent but the consequent is not one of the allowed values, then LERAD adds *tn/r* to the anomaly score for that sample, where *t* is the time since the last anomaly for that rule, either in training or testing. We found experimentally that using a sample count for *t* rather than real time gives slightly better results (more attacks detected in IDEVAL). This could be due to gaps in the training and test data from 6:00 to 8:00 AM each day and on weekends. Using real time would inflate the scores of anomalies seen on Monday mornings.

5.2. Experimental Evaluation

5.2.1. Experimental Data and Procedures

LERAD models 23 attributes of inbound TCP client data streams. TCP is first reassembled as with ALAD. All attributes are nominal. The attributes are as follows:

- Date.
- Time of day (to the nearest second).
- Source (remote) IP address as 4 1-byte attributes.
- Last 2 bytes of the destination (local) IP address as 2 1-byte attributes. (The first 2 are fixed from the home network, 172.16.x.x).
- Source port.
- Destination port.
- TCP flags of the first, next to last, and last packet, as 3 attributes.
- Log base 2 of the duration in seconds, truncated to an integer.
- Log base 2 of the length (number of application data bytes sent) truncated to an integer.
- First 8 words of the payload. Words are delimited by white space or the non-ASCII characters x80 to xFF and truncated to 8 characters.

Although it makes no sense to include the date or time as attributes, we included them as an implementation convenience and to test the robustness of the rule learning algorithm.

A second version of LERAD includes inbound client UDP and inbound ICMP packets in order to detect attacks that use these protocols. The attributes are the same except that the first TCP flag attribute is replaced with "UDP" or "ICMP" and the remaining flags are blank. To reduce traffic load, UDP destination ports 53 (DNS), 123 (NTP) and 1024 and above (clients) are excluded.

LERAD was tested on the same data as PHAD and ALAD: trained on inside sniffer traffic from week 3 of the 1999 IDEVAL data, and tested on weeks 4 and 5. The TCP data has 35,456

training instances and 178,100 test instances. The data with UDP and ICMP has 68939 training instances and 937,334 test instances. The results were evaluated with EVAL3 at 100 false alarms. Because LERAD uses a randomized algorithm, results were averaged over 5 runs with different seeds.

5.2.2. Experimental Results

LERAD with TCP detects 114 to 119 attacks (average 117). A typical run generates 1000-1200 candidate rules, reduced to about 70-85 after the redundancy test, and to about 55-70 final rules. With UDP and ICMP, LERAD generates slightly more rules in steps 2 and 3 (typically 100-125, 70-80). However it detects fewer attacks than TCP alone, 108 to 115 (average 112). There are few UDP and ICMP attacks compared to TCP, and the additional data adds false alarms.

The following analysis is for one typical run of LERAD with UDP and ICMP which detects 111 attacks. There are 69 rules, with *n/r* ranging from 34906/1 to 34887/297 (listed in Appendix A). *n* ranges from 3521 to the maximum of 68,939. The average number of conditions in the antecedent is 0.84. There are 11 rules with no antecedent, and 2 with the maximum of 3 conditions in the antecedent. Attributes appear in the antecedent/consequent of the following number of rules: payload 16/22, TCP flags 12/15, source address 8/10, destination address 8/7, destination port 5/7, duration 3/6, length 1/5, source port 2/0, date 0/0, time 0/0. All parts of multi-part attributes such as addresses (4 bytes), payload (8 words), and flags (3) appear in at least once consequent, and most but not all appear in at least one antecedent.

LERAD is instrumented to indicate which rule contributes the most to an alarm score. Usually a single rule contributes at least half of the total anomaly score, but the fraction is occasionally as low as 20%. We say that this a rule generates the alarm or detects an attack. Of the 69 rules, 31 generate alarms that detect at least one attack (at 100 false alarms), and 25 rules generate at least one false alarm. These rules overlap substantially. There are 18 rules that detect both attacks and false alarms, 13 that detect attacks without false alarms, and 7 that generate false

alarms with no detections. Of the 111 attacks detected, 34 (31%) are detected by "good" rules that generate no false alarms. Of the 100 false alarms, and 18 (18%) are generated by "bad" rules that detect no attacks. Whether a rule is good or bad does not depend on *n/r*, except possibly at the high and low ends. The median rank of good rules (on a scale of 1 to 69, with the highest *n/r* being 1), is 33 for good rules and 31 for bad rules. However, out of the top 3 ranking rules (1, 7, and 12) that generate alarms, all are good. Out of the bottom three rules (62, 65, and 68), one is good and two are bad.

The greatest number of attacks detected by a single rule is 20. This rule has no antecedent: "SA3 = 172 196 197 194 195 135 192 152" (n/r = 68,939/8, ranked 29'th). SA3 is the first byte of the source IP address. Such detections are probably due to an artificially low number of client addresses, as discussed previously, because it makes no distinctions between public and private services. This rule also generates the most false alarms, 18.

If we disregard detections by source address, then we are left with 75 detections by 26 rules. No rule detects more than 7 attacks. The detected attacks can be grouped by the anomalous attribute:

- Destination port detects *ls_domain* and named on port 53 (both DNS attacks), *portsweep* on ports 19 (chargen) and 143, *udpstorm* on port 7 (echo port, which it exploits), and *ftpwrite* on port 515 (printer).
- Destination address: *neptune* (SYN flood) conditioned on port 520 (route), and *portsweep* by accessing an unused address.
- Flag1 (first packet): *apache2*, *eject*, *tcpreset*. The first TCP flags are ACK-PSH, when they should be SYN, indicating a connection open before the sniffer was started. Probably coincidental.
- Flag 2 (next to last packet): *dosnuke* by URG-ACK-PSH, *queso* by 1-0-SYN. Both are attack signatures.

- Flag 3 (last packet): back and *netbus* by the absence of FIN, *ntinfoscan* by RST, *portsweep* and *queso* by FIN without ACK.
- Length: *apache2*, *back*, *netbus*, and *warez* by unusually long payloads, *phf* by an unusually short payload.
- Duration: *arppoison*, *back*, *casesen*, *crashiis*, *guest*, *ntfsdos*, *ntinfoscan*, *secret*, *teardrop*. *guest* (testing for a guest account) is unusually short; the others are long. *ntfsdos* is a console attack that generates no traffic, but because it requires a reboot, it sometimes results in hanging TCP connections. Many of the other attacks may have been detected this way.
- Payload: The most common anomaly is an empty string, omitting details irrelevant to the attack (*back*, *crashiis*, *mscan*, *ncftp*, *phf*). *apache2* ("User-age") and *ncftp* ("/etc/hos") are detected by strings in the exploit. *mailbomb* is detected by lowercase SMTP commands. *sendmail* is detected by an opening "MAIL" rather than "HELO" or "EHLO". *satan* is detected by an SMTP null byte. The other detections are *guessftp*, *guesspop* (bad passwords), *guesstelnet* (a null byte), *insidesniffer* (coincidental), *ipsweep*, *smurf* (binary data), and *netcat_breakin*. ("ver").

5.2.3. Detection – False Alarm Tradeoff

Figure 5.4 shows the DFA curve averaged over 5 runs of LERAD with different random number seeds. Like PHAD and ALAD, the number of detections increases rapidly up to about 100 false alarms and then levels off. However, the total number of detections at all false alarm levels is almost twice as high.

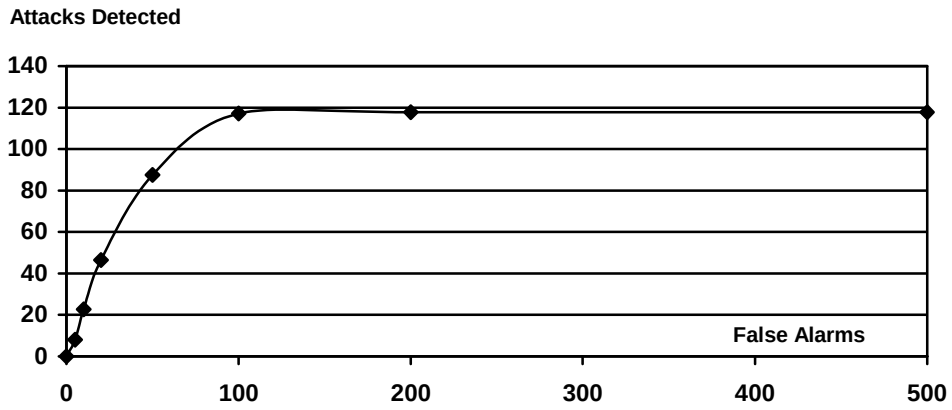

Figure 5.4. LERAD detection-false alarm curve for 0 to 500 false alarms.

5.2.4. Detections by Category

Table 5.1. lists the range and average number of attacks in each category detected by the 5 runs of LERAD in the previous section. Although each run generates a different rule set, there is very little variation between runs. LERAD does about equally well for probes, DOS, and R2L attacks, which are exactly those types for which a network IDS is best suited. Recall that ALAD was poor at detecting probes, even though it examines the same data.

The poorly detected attacks (from 1999) detected by LERAD are: *arppoison, dosnuke, ls_domain, ncftp, netbus, netcat, ntfsdos, perl, portsweep, queso, resetscan, sechole, sqlattack, sshtrojan, tcpreset, warezclient*, and *xterm*. LERAD does almost as well at detecting these attacks (52% of them are detected) as detecting attacks in general (58%).

Attack Type	Detected – Range over 5 runs	Average/Total at 100 FA
Probe	23-24	23.6/37 (64%)
DOS	36-39	37.8/65 (58%)
R2L	35-36	35.4/56 (63%)
U2R/Data	17-22	20.6/43 (48%)
Total	114-119	117.2/201 (58%)
Poorly Detected	39-41	40.0/77 (52%)

Table 5.1. LERAD detections by category at 100 false alarms.

5.2.5. Implementation and Run Time Performance

LERAD is implemented in three parts. It first uses the same 400 line C++ program as ALAD to reassemble TCP. It reduces 6.9 GB of tcpdump files into 60 MB of TCP streams in a text format in 17 minutes on the 750 MHz PC described in Section 3.5.5. The second part is a 26 line Perl script that reads this data and constructs database tables of training and test instances. It runs in 5 seconds producing 4.4 MB of training data and 17 MB of test data as two text files. LERAD is a 470 line C++ program that reads these tables and produces a list of alarms in 23 seconds. Source code is available at (Mahoney, 2003b).

5.3. Summary

LERAD introduces an algorithm for finding good rules for anomaly detection – those with high *n/r* – given a training set. It is a significant improvement over ALAD, which models mostly the same attributes, but uses an ad-hoc approach to selecting conditional rules. The LERAD algorithm is randomized. The idea is to generate a pool of candidate rules suggested by matching

attributes in pairs of training samples, remove redundant rules in favor of those with high *n/r*, and remove rules that generate false alarms on an attack-free validation set. LERAD was demonstrated on network session attributes, but in theory the algorithm could be applied to any set of nominal attributes. It is robust enough to eliminate useless attributes such as date and time from the rule set.

Our implementation of the algorithm is reasonably fast. Most of the run time is in reading the 6 GB of raw packet data to reassemble TCP streams, which takes about 10-15 minutes of mostly disk I/O time on our 750 MHz PC. The actual program, implemented in C++ and Perl, takes about 1 minute, a little slower than PHAD or ALAD.

LERAD has some limitations. One is that it requires two passes through the training data. We cannot sample just the start of the data because the bursty nature of network traffic would make any small window unrepresentative. This problem is unsolved. However, another limitation – which is shared with PHAD, ALAD, and any system that uses an anomaly score of *tn/r*, is that it requires attack-free training data in the first place. In reality, we must train the system on whatever traffic is available, which we assume to be *mostly* attack free. This suggests another approach – flagging rare but previously seen values so that if an attack occurs during training that subsequent instances are not completely masked.

Chapter 6

Continuous Modeling

This chapter introduces NETAD (Mahoney, 2003a), a Network Traffic Anomaly Detector. Unlike PHAD, ALAD, and LERAD, which detect only novel events, NETAD also classifies non-novel events as anomalous if they are sufficiently rare and have not occurred recently. This type of model is more suitable for the realistic case where we must update the anomaly model continuously on live traffic to keep up with changes in the network and we cannot guarantee that the training data is attack-free.

6.1. Modeling Previously Seen Values

A fundamental problem in anomaly detection is modeling, or estimating the probability of events, because it is assumed that as the probability decreases, the likelihood that the event is hostile increases. PHAD, ALAD, and LERAD all use an anomaly score of the form $1/P_fP_t$, where P_f and P_t are frequency and time-based probability estimates, respectively. The frequency based component is $P_f = r/n$, the average rate of anomalies during the entire training period. (A value is anomalous in training when it is seen for the first time). The time based component is $P_t = 1/t$, the average rate of anomalies over the shortest possible history for which the rate is not 0, i.e. back to the last anomaly.

We can apply the same approach to modeling values which have been previously observed in normal data. If the value i occurs n_i times out of n training instances, then its average frequency is $f_i = n_i/n$. We let $P_f = f_i$ be our frequency based model. If the value i last occurred t_i seconds ago, then its average rate since that event is $1/t_i$. We let $P_t = t_i$ be our time based model. Combining these as before, we let our anomaly score be $1/P_fP_t = t_i/f_i = t_in/n_i$.

The next problem is to combine the novel and non-novel models into a single model. Unfortunately the problem is not as simple as selecting the appropriate model depending on whether the event is novel or not. We need to know the absolute (not relative) probability of novel events so that the two models can be weighted appropriately. This problem, known as the *zero frequency problem* has been studied extensively in the data compression literature (Bell et al., 1989; Witten & Bell, 1991). For frequency-based models (where events are assumed to be independent), a common approach is to add some small value $\varepsilon > 0$ to the observed counts, i.e. $P_f = (n_i + \varepsilon)/(n + R\varepsilon)$ for some small constant, $\varepsilon > 0$, and R is an estimate of the (possibly unknown) size of the set of possible values. For $\varepsilon = 1$, we have Laplace's estimate, which assumes *a-priori* a uniform distribution over all possible probability distributions of a random variable with R possible values. However, Cleary & Teahan (1995) found that smaller values of ε often give better predictions for many types of data. Some of the best compression programs, such as PPMD (Shkarin, 2002) and RK (Taylor, 2000) adapt the novel event probability to the data using second level modeling.

A second approach is to maintain two models, one for novel events ($n_i = 0$), such as $score = tn/r$, and another model for non-novel events ($n_i > 0$), such as $score = t_i/f_i$. This approach requires that the two models be appropriately weighted according to the novel event probability. At a minimum, we should ensure that if a value i occurs a second time, it should receive a lower anomaly score than if it occurs for the first time. This means that we should choose a weight W such that $Wtn/r > t_i/f_i = t_i n/n_i = t_i n$ when $n_i = 1$. Factoring n, we have the requirement $Wt/r > t_i$. Now if this is a "good" rule (which we can distinguish from "bad" rules by using an attack-free validation set as with LERAD), then all of the possible values in normal traffic will be seen early in the training period, which implies $t \approx n$ (using unit time per instance). Also, if there are r possible values and these are distributed fairly uniformly (true for "good" rules), then on average, each value is seen with probability $1/r$, so $t_i \approx r$ on average. Making these substitutions into $Wt/r > t_i$, we now have the requirement $Wn/r > r$ or $W > r^2/n$. While r^2/n is small for "good" rules where r is small, it may not

be for "bad" rules, which as we discussed in Section 2.2, are quite common in real traffic. In particular, we have $t << n$ (recent novel events), $t_i >> r$ (non-uniform distribution of values, e.g. Zipf), and large r, all of which tend to make $Wtn/r < t_i/f_i$ unless W is sufficiently large.

6.2. NETAD

NETAD is a network traffic anomaly detector. It models single packets like PHAD, uses ad-hoc conditional rules like ALAD, and rule validation like LERAD. Its main contribution is in modeling non-novel values.

NETAD uses two stages, a filtering stage to select the start of inbound client sessions, and a modeling phase. The attributes are simply the first 48 bytes of the IP packet, which are considered to be nominal attributes with 256 possible values. For most packets, these attributes include all of the header information and a portion of the application payload. For normal TCP data packets, there are 40 bytes of header information and the first 8 bytes of payload. TCP streams are limited to packets containing the first 100 bytes, which is normally one data packet, so under normal circumstances NETAD only sees what would be the first 8 bytes of the TCP stream if it were reassembled.

The filtering stages removes 98% to 99% of the traffic, greatly reducing the load on the modeling stage and passing only the type of traffic most likely to contain evidence of attacks, i.e. unsolicited inbound traffic. We assume that attacks can be detected quickly, using only the first few packets of a long session. Thus, filtering removes the following packets:

- All non-IP packets (e.g. ARP), because an alarm needs an IP address to identify the target.
- All outgoing packets.
- All TCP streams that begins with SYN-ACK (i.e. the response to a local client).
- UDP packets to port number higher than 1023 (i.e. the response to a local client).

- TCP packets with sequence numbers more than 100 past the initial sequence number (i.e. after the first 100 bytes of incoming client data).
- Packets addressed to any address/port/protocol combination (TCP, UDP, or ICMP) after the first 16 packets in 60 seconds (to limit bursts of UDP or ICMP traffic).

The last two filters use hash tables of size 4K to look up destination addresses, ports, and protocols. This fixed-memory design thwarts memory exhaustion attacks against the IDS, but a small number of packets may be dropped due to hash collisions. Ideally, the hash function would need to be secret (selected randomly) to prevent collisions from being exploited in an evasion attack.

The second stage of NETAD models nine types of packets, for a total of 9 x 48 = 432 rules. The rules have the same form as in LERAD, in that the antecedent is a conjunction of conditions of the form *attribute* = *value*, where each attribute is one packet byte. The nine models represent commonly used (and commonly exploited) protocols. The rules were selected because they give good results experimentally.

- All IP packets (no antecedent).
- All TCP packets (if protocol = TCP (6))
- TCP SYN (if TCP and flags = SYN (2))
- TCP data (if TCP and flags = ACK (16))
- TCP data for ports 0-255 (if TCP and ACK and DP1 (destination port high byte) = 0)
- telnet (if TCP and ACK and DP1 = 0 and DP0 = 21)
- FTP (if TCP and ACK and DP1 = 0 and DP0 = 23)
- SMTP (if TCP and ACK and DP1 = 0 and DP0 = 25)
- HTTP (if TCP and ACK and DP1 = 0 and DP0 = 80)

The anomaly score assigned to a packet is the sum of the anomaly scores reported by each of the 432 rules. Individual rules can be scored in several ways.

1. Novel values only. Score = tn/r, where t is the time (packet count, training or test) since an anomaly was last observed for this rule, n is the number of training packets satisfying the antecedent, and r is the number of values seen at least once in training (1 to 256).
2. Validation weighed novel values. Score = tn_a/r, where n_a is the number of packets satisfying the antecedent *from the last training anomaly to the end of training*. This has the effect of giving greater weight to rules that generate no false alarms near the end of training, but without introducing a parameter as in LERAD. In LERAD, this technique was found to have the same effect as using the empirically determined optimal size validation set (10%).
3. Fast uniformity detection. Score = $tn_a(1 - r/256)/r$. This gives less weight to rules that generate most of the 256 possible byte values, which has the effect of discovering and removing uniform distributions more quickly.
4. Non-novel values. Score = $t_i n/(n_i + 1)$, where t_i is the time (packet count, training or test) since the value i was last seen, and n_i is the number of times i was seen in training. It reduces to $t_i n$ for novel events and t_i/f_i (with a Laplace approximation of f_i) for non-novel events.
5. Weighed model. Score = $t_i n/(n_1 + r/W)$, where $W = 256$ is an experimentally determined weight emphasizing novel events. It reduces to $Wt_i n/r$ for novel events and approximately t_i/f_i for non-novel events.
6. NETAD combined model. Score = $tn_a(1 - r/256)/r + t_i n/(n_1 + r/W)$, combining scoring functions 3 and 5.

6.3. Experimental Results

NETAD was first tested on the same data as PHAD, ALAD, and LERAD, trained on inside sniffer week 3 and tested on weeks 4-5 (177 detectable attacks) of the 1999 IDEVAL data set. It

was evaluated at false alarm rates from 20 to 500 (2 to 50 per day) using EVAL3 with alarm consolidation for each of the six anomaly scoring functions described in Section 6.2. Because we know from the PHAD experiments that the TTL field contains a simulation artifact, it was removed from NETAD.

The results are shown in Table 6.1. The combined scoring function (6) detects the most attacks at 100 false alarms, but either of the two components (3 or 5) do well by themselves. The weighted function (5) gives better results than any model that considers only novel events, especially at low false alarm rates. Rule validation improves the results somewhat. Fast uniformity detection has a small benefit.

NETAD scoring function	**20 FA**	**50 FA**	**100 FA**	**500 FA**
1. tn/r (novel values only)	56	78	104	141
2. tn_a/r (novel values with validation)	57	89	118	149
3. $tn_a(1 - r/256)/r$ (fast uniformity detection)	60	92	120	149
4. $t_i n/(n_i + 1)$ (non-novel values)	33	52	81	130
5. $t_i n/(n_i + r/256)$ (weighted)	78	115	127	142
6. $tn_a(1 - r/256)/r + t_i n/(n_i + r/256)$ (3 + 5)	66	97	132	148

Table 6.1. Attacks detected by NETAD at 20 to 500 false alarms for each of six scoring functions.

NETAD is instrumented to indicate all bytes that contribute to at least 10% of the total anomaly score. Out of 132 attacks detected using combined scoring function 6 at 100 false alarms, the number of attacks and false alarms detected by each field is distributed as follows. The total

attacks is more than 132 because some attacks are detected by more than one field. Detections that do not appear to be "legitimate" are marked with an asterisk.

- Source address: 72 (anypw*, apache2*, arppoison, crashiis*, eject*, ffbconfig*, guessftp, guesstelnet, guesspop, guest, imap*, insidesniffer*, ipsweep*, ls_domain*, mailbomb*, ncftp*, netbus*, netcat_setup*, perl*, pod*, portsweep*, ppmacro*, processtable, ps*, satan, sechole*, secret*, smurf, sqlattact*, sshtrojan, syslogd, tcpreset*, warez*, warezclient*, xlock, xsnoop, xterm*, yaga*), 43 false alarms.
- Packet size/TCP header size: 21 (back*, land, named*, pod, portsweep, queso, smurf), 10 false alarms.
- Application data: 20 (back, land, named, neptune, portsweep, queso, sendmail, udpstorm), 16 false alarms.
- TCP window size: 18 (apache2*, casesen*, ls_domain*, neptune, netcat, netcat_breakin*, ntinfoscan, phf*, portsweep, resetscan), 13 false alarms.
- Destination address: 11 (guesstelnet, mscan, ncftp, netbus, perl*, portsweep, xterm*), 8 false alarms.
- IP fragmentation: 8 (insidesniffer*, pod, teardrop), 9 false alarms.
- TCP flags: 5 (portsweep), 3 false alarms.
- Destination port: 4 (guesspop, imap, ls_domain), 1 false alarm.
- Urgent data: 4 (dosnuke), 0 false alarms.
- TOS: 2 (ftpwrite*), 3 false alarms.
- Source port: 6 false alarms.
- TCP checksum (not computed), 1 false alarm.

As with ALAD and LERAD, the majority of detections are by source address, and many of these attacks are on public services and would probably be missed if the background traffic simulated the distribution of remote client addresses more realistically. TCP window size and TOS also appear to

be simulation artifacts, similar to TTL, in that many of the detected attacks manipulate only higher level protocols. The detection of *insidesniffer* is probably coincidental.

Nevertheless, there are many detections that we could consider legitimate. For example, *sendmail* is detected because the SMTP payload starts with "MAIL" instead of "HELO" or "EHLO". *dosnuke*, which exploits a NetBIOS bug in handling urgent data, is detected by the urgent pointer field. *portsweep* is detected by TCP flags (FIN scan, no ACK), destination address (probing inactive addresses), unusually small packet sizes (no TCP options, which are usually present in a SYN packet), and window size (set to an arbitrary value by the attack code). Even if some of these anomalies are masked by background traffic, there are enough anomalies to make some detections likely.

6.3.1. Detections by Category

Table 6.2 lists the attacks detected by NETAD (scoring function 6) by category. NETAD scores lowest in detecting U2R attacks, which is to be expected for a network IDS. Like PHAD, which also models single packets, NETAD scores highest for probes.

The poorly detected attacks from the 1999 evaluation detected by NETAD are: *arppoison, dosnuke, ipsweep, ls_domain, ncftp, netbus, netcat, perl, portsweep, queso, resetscan, sechole, sqlattack, sshtrojan, tcpreset, warezclient,* and *xterm*. Like PHAD, ALAD, and LERAD, NETAD scores slightly lower for these attacks, detecting 57% of them, compared to 66% in general.

Attack Type	Detected/Total at 100 false alarms
Probe	32/37 (86%)
DOS	43/65 (66%)
R2L	38/56 (68%)
U2R/Data	19/43 (44%)
Total	132/201 (66%)
Poorly Detected in 1999 evaluation	44/77 (57%)

Table 6.2. NETAD detections by category.

6.3.2. Detection – False Alarm Tradeoff

Figure 6.1 shows the DFA curve for NETAD using scoring function 6. As with PHAD, ALAD, and LERAD, the number of detections rises rapidly up to around 100 false alarms, then tends to level off.

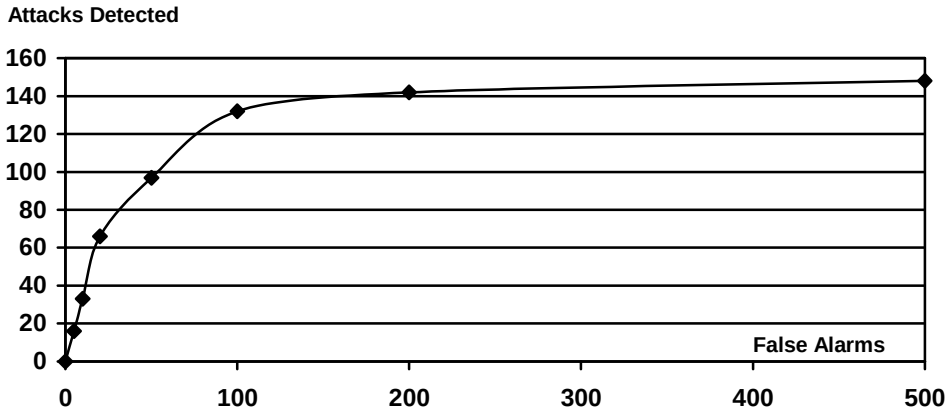

Figure 6.1. NETAD detections at 0 to 500 false alarms.

6.4. Unlabeled Attacks in Training

The purported benefit of modeling non-novel values is that it is more appropriate for a realistic environment where we lack attack-free training traffic and where network statistics change over time. In this case, we train and test simultaneously – we use all of the past traffic up to the previous packet to train the anomaly model being used to evaluate the current packet. After evaluation, we add this packet to the training model. This method should be less effective at detecting attacks because attacks are added to the "normal" training model, masking similar attacks in the future. Also, attacks near the beginning of the data are likely to be missed because the model is not sufficiently trained.

To test these two effects separately (masking and undertraining), we first ran NETAD (using each of the six scoring functions) on weeks 3, 4, and 5 as before, but left NETAD in training mode throughout the entire period, while continuing to output alarms during the attack period (weeks 4 and 5). This tests the masking effect. Second, we ran NETAD on weeks 4 and 5 only, with no training data prior to the start of the attack period. This tests the undertraining effect in addition to masking. This mode is the most realistic scenario.

For both modes, we measured the number of attacks detected at 100 false alarms. These are shown in the columns labeled W3-5 (training on weeks 3-5) and W4-5 (training on weeks 4-5). For comparison, we show the original results from the previous section where we trained NETAD on week 3 and froze the model (W3).

For all six scoring functions, masking and undertraining result in fewer detections than W3. The percent difference averaged over 20, 50, 100, and 500 false alarms is shown in the two columns labeled D3-5 and D4-5. For example, using scoring function 6, NETAD in mode W3 detects 66, 97, 132, and 148 attacks at 20, 50, 100, and 500 false alarms, respectively (see last row of Table 6.1). In mode W3-5, it detects 47, 80, 111, and 120 attacks at these levels. This is 71%, 82%, 84%, and 81% (average 79.5%) as many attacks in mode W3-5 as mode W3. In mode W4-5, NETAD detects

41%, 55%, 58%, and 78% (average 58%) as many attacks as in mode W3. The results for all six scoring functions is shown in Table 6.3.

NETAD scoring function	**W3**	**W3-5**	**D3-5**	**W4-5**	**D4-5**
1. tn/r (novel values only)	104	88	83%	65	64.5%
2. tn_a/r (novel values with validation)	118	84	74%	67	59%
3. $tn_a(1 - r/256)/r$ (fast uniformity detection)	120	94	75.5%	71	58.5%
4. $t_i n/(n_i + 1)$ (non-novel values)	81	77	98.5%	41	59%
5. $t_i n/(n_i + r/256)$ (weighted)	127	82	85%	82	69%
6. $tn_a(1 - r/256)/r + t_i n/(n_i + r/256)$ (3 + 5)	132	111	79.5%	76	58%

Table 6.3. NETAD attacks detected by scoring function in weeks 4-5 at 100 false alarms when trained on week 3 (W3), weeks 3-5 (W3-5), or weeks 4-5 (W4-5). Percentage of attacks detected in modes W3-5 (D3-5) or W4-5 (D4-5) compared to mode W3 averaged over 20, 50, 100, and 500 false alarms.

From Table 6.3 we see that the masking effect (D3-5) and combined masking and undertraining effects (D4-5) are more severe for scoring functions 1, 2, and 3 (novel values only) than scoring functions 4 and 5 (novel and non-novel modeling). This is the behavior that we expect, given our assumption that modeling non-novel values is more appropriate for continuous training. In fact, in the absence of attack-free training data (W4-5), function 5 beats the combined function 6, of which it is a component, just as it did for low false alarm rates in W3.

Another effect of continuous training is that rule validation (function 2) hurts rather than helps (compared to function 1). The idea of validation is to reduce the effects of rules that generate false alarms during an attack free validation period, but this makes no sense when such data is

lacking because we might instead be removing a rule that detects a genuine attack. Indeed, during training, $n_a = t$, so scoring function 2 is really t^2/r.

6.5. Implementation and Run Time Performance

NETAD is implemented as two programs, a 200 line C++ program to filter packets, which is input to NETAD, a 290 line C++ program. The filter program reads 6.9 GB of tcpdump files (inside weeks 3-5) in about 15 minutes on the 750 MHz PC described in Section 3.5.5. It outputs 37 MB of training data and 72 MB of test data as new tcpdump files. NETAD reads these files and outputs a list of alarms in 30 seconds. Source code is available at (Mahoney, 2003b).

6.6. Summary

NETAD introduces a model which can be used in an environment where attack-free training data is not available and the model is trained continuously to keep up with changing statistics while simultaneously detecting attacks. One way to accomplish this is by adding a component of the form t_i/f_i for non novel values i, where t_i is the time since it was last seen and f_i is the average frequency so far. For novel values, we can use Wt_in/r, where W is a weight (we used 256), n is the number of instances satisfying the rule antecedent, and r is the number of values observed at least once. This was approximated by scoring function 5, which results in more detections on the IDEVAL test data than the other five scoring functions we tried, including tn/r. It is also more resistant to the masking and undertraining effects caused by the lack of attack-free training data.

Analysis of the detected attacks raises some nagging questions about the validity of the IDEVAL data and the four systems we tested on it. Some attacks seem to be detected by legitimate features of the attack, while others seem to be detected by simulation artifacts. We address these questions in the next chapter.

Chapter 7

A Comparison of Simulated and Real Network Traffic

In chapters 3 through 6 we evaluated PHAD, ALAD, LERAD, and NETAD on the IDEVAL data set. As we mentioned in Section 2.4, the background network traffic on which these systems are trained is synthesized. It is difficult to do this right (Floyd & Paxson, 2001). Anomaly detection evaluation depends critically on the accuracy of this background traffic because it is used to construct a model of "normal". Although great care was taken to ensure that the data is realistic, our detailed analyses of detected attacks suggests that some simulation artifacts may have crept in.

- PHAD detects more attacks by the TTL field than any other (until we removed the rule). Most of the anomalous TTL values are smaller by 1 than the trained values. According to the technical report describing the traffic synthesis (Haines et al., 2001), hostile and background traffic was generated on two different real machines simulating the same IP address. Although the exact configuration was not described, our observations are consistent with a configuration in which the attack simulator was further from the sniffer by one router hop than the background simulator.
- ALAD, LERAD, and NETAD detect a large number (about half) of attacks by source address anomalies. These include attacks on public services (web, mail, and DNS), where novel addresses are to be expected. According to the technical report, simulated client

traffic was generated by randomly selecting one of only 10 IP addresses. This number is clearly unrealistic.

- ALAD and LERAD detect many U2R attacks because an anonymous FTP server was used to upload the exploit code, and no uploads ever occur in training.
- NETAD detects several attacks by anomalies by packet size, TCP header size, and TCP window size fields, including some application protocol exploits which should not affect these fields.
- PHAD detects no IP, TCP, UDP, or ICMP checksum errors in over 12 million packets of training data.

Indeed, it is surprisingly easy to detect attacks in the IDEVAL data using anomaly detection. In (Mahoney & Chan, 2003) we describe a simple anomaly detector called SAD that examines only one byte of inbound TCP SYN packets and outputs an alarm (with score = 1) when it observes a value never seen in training (limited to one alarm per minute). When trained on the 1999 IDEVAL inside sniffer week 3 and tested on inside weeks 4 and 5, SAD detects 71 out of 177 attacks with 16 false alarms by examining just the low order byte of the source address. This result is competitive with the top systems in the original 1999 evaluation, some of which used a combination of signature and anomaly detection techniques on both host and network data. Many different SAD bytes give good results, in particular the other source address bytes, TTL, IP packet size, TCP header size, TCP window size, TCP options, and the high byte of the source port. Many of the detected attacks are application protocol exploits or U2R attacks, which should not influence these values. Similar results can be obtained when SAD is trained on attack-free traffic in week 1 and tested on a subset of the evaluation attacks in week 2, data which was provided in advance to the eight original IDEVAL participants in order to develop their systems.

We address the problem of possible simulation artifacts by collecting real traffic from a university departmental server and injecting it into the IDEVAL data. We use this approach rather than creating a whole new data set because it allows us to use the rich set of labeled attacks rather

than simulate new ones. Most of these attacks use actual exploit code available from published sources and real target machines running real software. Thus, the attacks should not be subject to the same kind of simulation artifacts as the background traffic. Our approach is to "fix" the training and background traffic by making it appear as if the real server and the real Internet traffic visible to it are part of the IDEVAL network. When necessary, we modify the IDS to ensure that it is unable to defeat this mixing by modeling the simulated and real traffic separately. To test whether this strategy is successful, we evaluate the attacks detected by the modified IDS for legitimacy on mixed traffic compared with simulated traffic.

This chapter is organized as follows. In Section 7.1 we describe the environment from which the real traffic was collected. In 7.2 we analyze the real data and compare it with the simulated IDEVAL training data. In 7.3 we describe how the mixed evaluation data set is constructed. In 7.4 we describe how PHAD, ALAD, LERAD, and NETAD are modified to force all of the rules to be trained on real data. In 7.5 we compare these systems on the simulated and mixed evaluation data sets. In 7.6 we conclude. Most of the work described in this chapter is from (Mahoney & Chan, 2003).

7.1. Traffic Collection

Our goal was to collect network traffic from an environment similar to the IDEVAL network, but practical considerations raise many barriers. Sniffing traffic on a network used by others without their knowledge raises many privacy and security issues. A sniffer captures email, web surfing habits, and passwords of unsuspecting users. Obtaining permission to collect data limits our choice of environments. A sniffer also raises performance issues, as it can collect huge amounts of data, which must be stored securely. Care must be taken that the sniffer does not also generate traffic visible to itself, for example by storing the data on a remote host via FTP or NFS while the sniffer is active.

7.1.1. Environment

We collected traffic from a university departmental server, www.cs.fit.edu. Like the IDEVAL network, the server is connected by Ethernet to a number of local machines and to two routers, one to a larger local area network (eyrie.af.mil or fit.edu) and another router to the Internet. The server is a Sun running Solaris, like the victim host *pascal* in IDEVAL. The server hosts a website with several thousand pages maintained by several faculty members via UNIX shell accounts. There is also an SMTP server for sending mail and POP3 and IMAP servers for receiving mail. In all these respects, the server resembles the victim hosts on the IDEVAL network (although *pascal* does not run a web server). However there are many differences.

- The real traffic was collected in 2002, as opposed to 1999. During this time new operating system, server, and client versions were released, new protocols were introduced, and others updated.
- Security is tighter. Our server runs behind a firewall, unlike the IDEVAL network. Also, there is no *telnet* server. Communication is by SSH and secure FTP. (However POP3 and IMAP passwords are not encrypted).
- Our server is on an Ethernet switch rather than a hub. Only traffic addressed to the server (other than multicast or broadcast) is visible.
- Some remote IP addresses are dynamically assigned using DHCP and may change daily. All IP addresses in the IDEVAL network are static.
- The real traffic contains many protocols not present in the IDEVAL background traffic. Among these are undocumented network protocols (in addition to IP and ARP), the transport protocols OSPFIGP, IGMP and PIM (in addition to TCP, UDP, and ICMP), and application payloads such as NFS, RMI, portmap, and several others that do not use well known port numbers.

Another problem is that real training/background data may contain unlabeled attacks. In one informal experiment using the author's dialup Windows PC, probes (usually to a single port) were observed about once per hour, a rate higher than in the IDEVAL test data. The server firewall undoubtedly filters many probes, but cannot block attacks on open ports. We found (by manual inspection) about 30 suspicious HTTP requests in 600 hours of traffic. Two of these are shown below.

```
GET /MSADC/root.exe?/c+dir HTTP/1.0
```

This is a probe for a backdoor dropped by the Code Red worm.

```
GET /scripts/..%255c%255c../winnt/system32/cmd.exe?/c+dir
```

This probe appears to exploit a URL decoding bug in IIS. The string "%25" decodes to "%". A second decoding of "%5c" decodes to "\" which IIS treats the same as "/". Furthermore, Windows treats a double slash (e.g. ..\\..) like a single slash (..\..). Of course, this exploit has no effect on a UNIX machine.

7.1.2. Data Set

We collected daily traces collected from Sept. 30 through Dec. 13, 2002. Each trace is 2,000,000 packets, starting at 12:01 AM local time and ending about 10 to 15 hours later. Packets are truncated to 200 bytes. For our analysis, we used only traffic collected on Monday through Friday (like IDEVAL) for the 10 weeks from Sept. 30 through Oct. 25 and Nov. 4 through Dec. 13. We skipped one week because no data was available on one of those days.

To reduce the data load in our analyses, we filtered the data set using the first stage of NETAD, as described in Section 6.2. This preserves only the beginning of inbound client to server

requests, which is the traffic of most interest. This filtering reduces the data from 100,000,000 packets to 1,663,608 packets (1.6%). Similar filtering on IDEVAL inside week 3 reduces the data from 12,814,738 packets to 362,934 packets (2.8%). The difference in data reduction is due mainly to a lower percentage of UDP in the real traffic. For our analysis, we use both attack-free weeks 1 and 3 from the inside sniffer, which contain a total of 658,801 packets after filtering.

7.2. Comparison with Real Traffic

We are interested in evaluating the realism of the background traffic in the IDEVAL data set with respect to our anomaly detection algorithms. For algorithms that use a scoring function of tn/r for novel values (e.g. PHAD, ALAD, and LERAD), there are at least four conditions under which our algorithms would seem to work well using the IDEVAL data but then fail in practice, assuming that the attack simulations are realistic. These are as follows:

1. A value that is anomalous in simulation appears in real background traffic, resulting in the detection being missed.
2. The number of training instances, n, is smaller in real traffic due to insufficient training data.
3. The number of values observed in training, r, is larger in real traffic due to greater variation in the protocols.
4. r grows at a faster rate in real traffic due to changing network statistics or insufficient training time to observe all possible traffic sources, resulting in more false alarms. If the algorithm uses rule validation, this would instead result in more "bad" rules being discarded.

In addition, algorithms that model non-novel values using $t_i/f_i = t_i n/n_i$ (e.g. NETAD) could be affected by smaller n (condition 2), or larger n_i (higher frequency of anomalous value i in normal traffic, similar to condition 1).

We can measure n and r (conditions 2 and 3) using any of our anomaly detection algorithms. In addition, a large value of r would imply that fewer values are likely to be reported as anomalies (condition 1), and could imply a faster growth rate of r (condition 4). However, the growth rate can be approximated more accurately if we measure it near the end of the training period. We define three statistics that estimate the rate of growth of r.

- r_1 is defined the fraction of values (out of r) that occur exactly once in training. r_1 is a Good-Turing estimate of the probability that the next value will be novel (i.e. a false alarm), assuming that each value is independent. If this is not the case (e.g. for bursty traffic with long range dependencies), then r_1 is an underestimate.
- r_h is defined the fraction of values (out of r) seen for the first time in the second half of the training data. Thus, the probability of a novel value over this period is estimated at $2r_h$
- r_t is defined as the fraction of training time needed to observe half of all the values.

For example, given the sequence ABCDAB, then $n = 6$, $r = 4$, $r_1 = 2/4 = 0.5$ (C and D occur once), $r_h = 1/4$ (D occurs only in the second half), and $r_t = 1/3$ (the time to observe 2 of the 4 values, A and B). In general, "good" rules have large n, and small r, r_1, r_h, and r_t. The distribution of values can be learned quickly, and it does not change over time. If the values have a Zipf distribution (which is often the case), then this is a "bad" rule. In the worst case, r grows at a constant rate, indicated by $r_h = r_t = n/2$. These two cases are illustrated in Figure 7.1. If a rule is "bad", it will either generate a lot of false alarms or be removed by rule validation. If not removed, it will likely miss some attacks (because the value is more likely to be seen in training) or generate a smaller anomaly score (because n/r is smaller).

In all of our measurements, we use packet count in lieu of real time to calculate r_h and r_t. This removes any bias due to gaps between traces in either the simulated or real data. For example, to calculate r_h, we count all values that occur for the first time in the last 329,400 out of 658,801 simulated packets. The data in Section 7.2 also appears in (Mahoney & Chan, 2003).

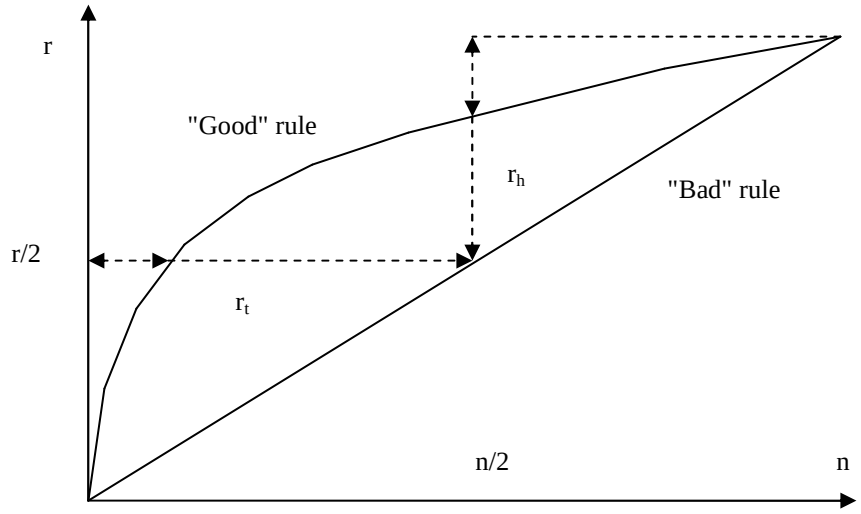

Figure 7.1. Growth in the number of unique values (r) over time for "good" and "bad" rules with the same *n* and *r*. Small values of r_h (new values in the second half of training) and r_t (time to learn half the values) are indicators of good rules.

7.2.1. Comparison of All Filtered Packets

We first compare the training traffic (inside sniffer weeks 1 and 3) with the 10 week data set, both after filtering. In most of the attributes we examined, the rate of anomalies is higher in the real traffic, as indicated by higher values of r, r_1, r_h and r_t (listed as four consecutive values in Table 7.1), even after taking into account the larger *n* of the real data set. Where the difference is significant (a somewhat subjective judgment), the higher values are highlighted in italics. These fields include the Ethernet source address, TTL, TOS, TCP options, UDP destination port, and ICMP type.

r, r_1, r_t, r_h	**Simulated**	**Real**
Ethernet source address	8, 0, 0, .00001	*76, .01, .11, .03*
IP source address	1023,.26, .71, .73	27632, .08, .53,.53
IP destination address	32, 0, 0, .0002	1, 0, 0, 0
TCP header size	2, 0, 0, .000003	*19, .16, .05, .024*
ICMP types	3, 0, 0, .001	*7, .14, .14, .16*
TTL	9, 0, .1, .00002	*177, .04, .12, .023*
TOS	4, 0, 0, .0003	*44, .07, .64, .53*
TCP destination port	*8649, .35, .66, .65*	32855,.001, .002, .3
TCP flags	8, 0, 0, .00002	13, 3, 0, .00009
TCP options 4 bytes	2, 0, 0, .00002	*104, .22, .31, .18*
UDP destination port	7, 0, 0, .0001	*31, .52, .55, .45*

Table 7.1. Comparison of r, r_1, r_h and r_t for nominal attributes of inside sniffer weeks 1 and 3 (simulated) with 10 weeks of real traffic after filtering (real).

The following binary events occur only in the real traffic (Table 7.2): fragmented IP packets (with the "don't fragment" flag set), TCP and ICMP checksum errors, nonzero bits in TCP reserved fields and reserved flags, and nonzero data in the urgent pointer when the URG flag is not set. These events are present even after removing TCP packets with bad checksums.

Percent	**Simulated**	**Real**
Packets	n = 658,801	n = 1,663,608
IP options	None	None
IP fragments	0	*0.45%*
Don't fragment (DF)	52% set	90% set
DF set in fragment	No fragments	100% bad
IP checksum	No errors	No errors
TCP checksum	No errors	*0.017% bad*
UDP checksum	No errors	No errors
ICMP checksum	No errors	*0.020% bad*
TCP reserved flags	Always 0	*0.093% bad*
TCP reserved field	Always 0	*0.006% bad*
Urgent data, no flag	None	*0.022% bad*

Table 7.2. Comparison of binary attributes of inside sniffer weeks 1 and 3 (simulated) with 10 weeks of real traffic after filtering (real).

For all continuous attributes (Table 7.3.) we measured, the range is higher in real traffic. This includes packet size, UDP payload size, TCP header size, urgent pointer, and window size. However it is difficult to judge the significance of these differences based on range alone.

Range	Simulated	Real
IP packet size	(38-1500)	(24-1500)
TCP window size	(0-32737)	(0-65535)
TCP header size	(20-24)	(20-48)
Urgent pointer	(0-1)	(0-65535)
UDP packet size	(25-290)	(25-1047)

Table 7.3. Comparison of continuous attributes of inside sniffer weeks 1 and 3 (simulated) with 10 weeks of real traffic after filtering (real).

Most attributes are less predictable in real traffic than in simulation. However the situation is opposite for TCP ports. The rate of novel values is lower in the real traffic. Most of the simulated TCP ports are high numbered FTP data ports negotiated during FTP sessions. The real traffic has a much lower rate of FTP sessions. Also, some real ports may be blocked by the firewall.

7.2.2. Comparison of TCP SYN Packets

In Table 7.4 we compare inbound TCP SYN packets in the simulated and real traffic. This exposes some potential artifacts that were not apparent in the larger set of all filtered packets. The most striking difference is in IP source addresses. The number and rate of novel addresses is thousands of times higher in real traffic than in simulation. This is *not* the case when UDP and ICMP traffic (or outbound TCP) is included.

Other differences include TCP options (which determine packet size and TCP header size) and window size. Every inbound TCP SYN packet uses the exact same four TCP option bytes, which set the maximum segment size (MSS) to 1500. In reality, the number of options, their order, and the option types and values varies widely.

Window size (used to quench return traffic) is allowed to range from 0 to 65535. The full range of values is seen only in real traffic. Simulated traffic is highly predictable, always one of several values. A difference in range is also observed in source ports (selected randomly by the client) and high numbered destination ports (often negotiated). One other type of anomaly seen only in real traffic is a nonzero value in the acknowledgment field, even though the ACK flag is not set.

Attribute	**Simulated**	**Real**
Packets, n	50650	210297 + 6 errors
Source address r, r_1, r_h, r_t	29, 0, .03, .001	*24924, .45, .53, .49*
Destination address, r	17	1 (163.118.135.1)
Source port, r	13946 (20-33388)	45644 (21-*65534*)
Destination port, r	4781 (21-33356)	1173 (13-*65427*)
IP packet size, r	1 (44, 4 option bytes)	*8 (40-68)*
TCP options, r	1 (MSS=1500)	*103 in first 4 bytes*
Window size, r	7 (512-32120)	*523 (0-65535)*
TCP acknowledgement	Always 0	*0.02% bad*

Table 7.4. Comparison of simulated and real inbound TCP SYN packets (excluding TCP checksum errors).

7.2.3. Comparison of Application Payloads

We compare HTTP requests in the simulated data (weeks 1 and 3) with 10 weeks of real traffic. Because the real packets were truncated to 200 bytes (usually 134-146 bytes of payload), we examine only the first 134 bytes in both sets. Table 7.5 summarizes the differences we found.

Inbound HTTP Requests	**Simulated**	**Real**
Number of requests, n	16089	82013
Different URLs requested, r, r_1	660, .12	*21198, .58*
HTTP versions, r	1 (1.0)	2 (1.0, 1.1)
Commands (GET, POST...), r	1 (GET)	*8*
Options, r	6	*72*
User-agents, r, r_1	5, 0	*807, .44*
Hosts, r	3	13

Table 7.5. Comparison of HTTP requests in simulated traffic (inside weeks 1 and 3) and 10 weeks of real traffic.

There are two simulated web servers (hume and marx). However, the one real web server receives more traffic and has more web pages. The distribution of real URLs is approximately Zipf, consistent with findings by Adamic (2002). A characteristic of a Zipf distribution is that about half of all values occur exactly once. The simulated URLs are distributed somewhat more uniformly. Many of the singletons are failed requests which were simulated by replacing the last 4 characters of the file name with *xxxx* (e.g. "GET /index.xxxx HTTP/1.0").

There is a huge disparity in the number of user-agents (client types). The simulated traffic has only five, all versions of *Mozilla* (Netscape or Internet Explorer). Real web servers are

frequently accessed by search engines and indexing services. We found the top five user-agents in the real data to be (in descending order) *Scooter/3.2*, *googlebot/2.1*, *ia_archiver*, *Mozilla/3.01*, and *http://www.almaden.ibm.com/cs/crawler*. They also have a Zipf distribution.

The only simulated HTTP command is GET, which requests a web page. The real traffic has 8 different commands: GET (99% of requests), HEAD, POST, OPTIONS, PROPFIND, LINK, and two malformed requests, *No* and *tcp_close*,. There is also a much wider variety of options, although some of these are due to the introduction of HTML/1.1. Nevertheless there is wide variation in capitalization and spacing. In the simulated traffic, HTTP options invariably have the form *Keyword: value*, with the keyword capitalized, no space before the colon and one space afterwards. This is usually but not always the case in real traffic. Furthermore, we occasionally find spelling variations, such as *Referrer:* (it is normally misspelled *Referer:*) or the even more bizarre *Connnection:* with three n's. Some keywords are clearly malformed, such as *XXXXXXX:* or ~~~~~~~:. A few requests end with a linefeed rather than a carriage-return and linefeed as required by HTTP protocol. Finally there are some requests which are clearly suspicious, as mentioned previously.

We look only briefly at SMTP (mail) and SSH (secure shell). These are the only other TCP application protocols besides HTTP that exist in sufficient quantity in both data sets to do a useful comparison. Like HTTP, we once again find that real traffic is "messy", high in benign anomalies. Table 7.6 summarizes the results.

Inbound Request	Simulated	Real
SMTP requests, n	18241	12911
First command, r	2	*7*
HELO hosts, r, r_1	3, 0	*1839, .69*
EHLO hosts, r, r_1	24, .04	*1461, .58*
No initial HELO or EHLO	0	*3%*
Lower case commands	0	*0.05%*
Binary data in argument	0	*0.1%*
SSH requests, n	214	666
SSH versions, r, r_1	1, 0	*32, .36*

Table 7.6. Comparison of inside sniffer weeks 1 and 3 with 10 weeks of real inbound SMTP and SSH requests.

A normal SMTP session starts with HELO or EHLO (echo hello), but these are optional. In the simulated traffic, every session starts with one of these two commands. However, about 3% of real sessions start with something else, usually RSET, but also QUIT, NOOP, EXPN, or CONNECT. About 0.2% of real commands are lower case. One command (EXPN root) is suspicious.

The number of simulated remote hosts sending and receiving mail (arguments to HELO and EHLO) is clearly unrealistic. This is also reflected in the small number of source IP addresses in general. The simulated traffic has one malformed command, an EHLO with no argument. The real traffic does too, and a variety of other malformed arguments, including binary strings (1-21

bytes, probably too short to be a buffer overflow). The host name arguments are roughly Zipf distributed, with over half appearing only once.

An SSH session opens with the client version string. The simulated traffic uses a single client version. In real traffic there are many versions, again Zipf distributed.

7.3. Summary

In this chapter we collected real network traffic and compared it with the simulated training and background traffic in the IDEVAL data set. We concluded that many attributes have a wider range of values (*r*) in real traffic, and that the range grows more rapidly. This means that an anomaly detection system will learn many more "bad" rules in real traffic. This should result in masked detections, lower anomaly scores, and higher false alarm rates, or if rule validation is used, fewer rules. Among the worst offenders are remote client addresses, TTL, TCP options, TCP window size, and application payload keywords and arguments. A large percentage of the attacks detected by PHAD, ALAD, LERAD, and NETAD are detected by these attributes.

However, we cannot conclude that these algorithms would not work. Our analysis of the data suggests that there are still some "good" rules, although not as many as the IDEVAL simulation would suggest. Our analysis does not reveal whether the additional values that appear in real traffic are the same ones that would appear in an attack, so we cannot say whether the alarm score would be zero or just smaller. All of our algorithms adapt to data with bad rules either by reducing the alarm score (larger *r*) or by rule validation. If there are any good rules, then these algorithms should find them, just as they are very good at finding simulation artifacts. Although many attacks appear to be due to simulation artifacts, others appear to be due to legitimate features of the attack. In the next chapter, we address the question of how many of these attacks would actually be detected if this real traffic had been used in the evaluation.

Chapter 8

Evaluation with Mixed Traffic

In Chapter 7 we found strong evidence that the IDEVAL data would not reliably predict the performance of some network anomaly detection algorithms because the synthesized data appears to be too predictable. Unfortunately there is no reasonable alternative data set, due to the great expense of producing this type of data. We would prefer to "fix" the background and training data, if possible, then use it to test our algorithms with the original labeled attacks. Thus, our goal is to answer two questions.

1. Can the IDEVAL data be "fixed" by injecting real traffic?
2. Would PHAD, ALAD, LERAD, and NETAD work on real traffic?

We approach both questions injecting real background traffic into the IDEVAL data to make it appear as if there was a real host receiving real Internet traffic during the evaluation. To answer the first question, we evaluate PHAD, ALAD, LERAD, NETAD, and another network anomaly detection system, SPADE (Hoagland, 2000), on this mixed traffic and test whether more of the detections are "legitimate". If the answer is yes, then the results of these tests will answer our second question.

We propose to add real traffic to the IDEVAL data to make it appear as if it were being sent and received during the simulation. We believe it is not necessary to remove the simulated background traffic because the combination should be similar (in the statistical sense of Section 7.2) to the real traffic alone. To see this, let A_S be the set of values of attribute A seen in simulation up

to the present time, and let A_R be the corresponding set of values seen in real traffic. Then the set of values A_M seen in merged traffic would be at all times:

$$A_M = A_S \cup A_R$$

Note that the *r* statistic for attribute A_S, which we denote r_S is simply $|A_S|$. Likewise, we define $r_R = |A_R|$ and $r_M = |A_M|$. Therefore, we have at all times:

$$\max(r_S, r_R) \leq r_M \leq r_S + r_R$$

In cases where we suspect r is an artifact, we have $r_S << r_R$, and therefore $r_M \approx r_R$, so removing the simulated traffic would have little effect. Furthermore, because this is true at all times, r_M and r_R would have similar growth rates.

A problem can occur when A_R is too small or empty, i.e. there is little or no real traffic of types where A is defined to mix with the simulation. In this case, $r_M \approx r_S$, and the artifact, if there is one, would not be removed. One such example is the destination address of incoming traffic, where there are $r_S = 16$ simulated hosts and $r_R = 1$ real host. We are unable to test whether the destination address is an artifact in the simulation (although we have no reason to believe that it would be). Other untestable attributes are those of FTP and telnet payloads, because there is little FTP and no telnet traffic in our real data. (Remote login and FTP are available only via the SSH protocol).

We wish to evaluate network anomaly detection systems on mixed data. Our approach is as follows. First, we analyze the system to determine which attributes are monitored. Then we test the simulated and real data to determine which attributes are present in the simulation, but absent or rare in the real data. Then we modify the system to ignore these attributes.

8.1. Data Preparation

For our mixed traffic, we use the same large, filtered data set as described in Section 7.2. We have 579 hours of traffic, which is more than enough to mix into the 146 hours of traffic in inside sniffer week 3 plus the 198 hours in weeks 4 and 5. We mix the traffic in a 1:1 ratio, i.e. one hour of simulated traffic is mixed with one hour of real traffic. Other ratios would be possible by stretching or compressing the real traffic, but we do not do this.

We mix the traffic to make it appear as if all of the collected data occurs during the simulation. We do this by adjusting the time stamp of the first real packet to match the time stamp of the first simulated packet, then maintain the relative times of the other real packets, excluding gaps in the two collections. This is illustrated in Figure 8.1. Time reads from left to right.

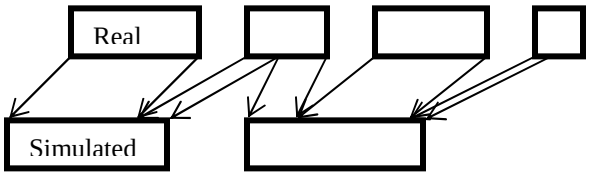

Figure 8.1. Mapping real time into simulation time when there are gaps in collection in both data sets.

The real traffic consists of 50 traces, divided into 10 weeks. We mix these into weeks 3 (training), 4, and 5 (test) of the inside sniffer data to prepare three mixed data sets, which we label A, B, and C as shown in Table 8.1. Prior to mixing, both the simulated and real traffic are filtered as described in Section 7.1 to pass only truncated and rate limited inbound client to server requests. In particular, the simulated packets are truncated to 200 bytes so that they are indistinguishable from the real packets. We denote the unmixed data (after filtering) as set S.

Set	Training data	Test data
S	IDEVAL inside week 3	IDEVAL inside weeks 4-5
A	S + real weeks 1-3	S + real weeks 4-7
B	S + real weeks 4-6	S + real weeks 7-10
C	S + real weeks 7-9	S + real weeks 1-4

Table 8.1. Mixed data sets used for evaluation. All data is filtered.

8.2. Algorithm Preparations

In this section we describe how we modify PHAD, ALAD, LERAD, NETAD, and SPADE to meet the requirement that it not test any attributes where $r_R << r_S$. We can do this by determining if there are any rules that would be conditioned on mostly simulated data and removing them. For SPADE, we modify the input data rather than the algorithm.

8.2.1. PHAD Modifications

Recall that PHAD is a time-based global model of packet header fields. If any packet (inbound or outbound, client or server) displays a value never seen in training, then PHAD assigns a score of $\Sigma\ tn/r$, where t is the time since the previous anomaly, n is the number of training packets, and r is the number of allowed values, and the sum is over all of the anomalous attributes. There are 34 attributes corresponding to the various 1 to 4 byte fields in the Ethernet, IP, TCP, UDP, and ICMP packet headers. The conditions for these fields to be present, is that the packet be of the corresponding type. Therefore, if all of these packet types exist in the real data, then no modification is necessary. If there are any packet types that PHAD tests for and which exist in the simulated but not the real data, then we would have to remove all of the attributes for that packet.

For example, if there were no real ICMP packets, then we would remove the ICMP type, ICMP code, and ICMP checksum fields.

Table 8.2 shows the number of packets for each type tested by PHAD in sets S, A, B, and C. There is a significant increase in A, B, and C over S for all packet types, indicating the addition of a significant number of real packets. Thus, no modification is needed to PHAD. However we still expect PHAD to give a different result on set S than on the original data because the packets are filtered.

Packet Type	**S**	**A**	**B**	**C**
Ethernet	362,934	789,504	813,011	740,479
IP	362,934	789,504	813,011	740,479
TCP	170,435	407,858	482,351	395,693
UDP	186,051	325,742	318,761	325,002
ICMP	6448	55,904	11,899	19,784

Table 8.2. Number of packets of each type tested by PHAD in filtered IDEVAL inside week 3 (S) and in mixed sets A, B, and C.

8.2.2. ALAD Modifications

Recall that ALAD models inbound client TCP streams and that the optimal combination of 11 rule forms was found to be the following five:

- P(client address | server address)
- P(client address | server address and port)
- P(TCP flags | server address)
- P(server address and port)

- P(keyword | server port)

The first three of these rule forms are conditioned on the server (local) address, which differs in the simulated and real traffic. Each rule form consists of two sets of rules, one which models traffic on the simulated servers, and one set on the real server.

The fourth rule is unconditional, and therefore makes no distinction between simulated and real traffic. However, because there is only one real server address, we know that any novel addresses must come from the simulation. We can test the port number by itself, however. Therefore, we modify the first four rules to remove the server address, so that the modified ALAD uses the following rule forms:

- P(client address)
- P(client address | server port)
- P(TCP flags)
- P(server port)
- P(keyword | server port)

In addition, we must modify the TCP reassembly algorithm to work with the filtered and truncated packets, and do so consistently with the simulated and real traffic. Recall that packets are truncated to 200 bytes (header plus 134 to 146 payload bytes), and that only packets containing the first 100 bytes of the sequence are passed. This means that we cannot capture the closing TCP flags as before. Instead, we let the TCP flag attribute be the sequence of flags for the first three packets. Also, to avoid gaps in the TCP payload, only the first 134 bytes of the first TCP data packet are used in the reassembled stream (instead of the first 1000 bytes).

Of all the TCP protocols, only SSH, SMTP, and HTTP (ports 22, 25, and 80) exist in significant quantities in both the simulated and real traffic. Therefore, the rules that are conditioned on server port are restricted to these three ports.

8.2.3. LERAD Modifications

Recall that LERAD is a time-based model, like PHAD and ALAD, assigning a score of $\Sigma\ tn/r$ to novel attribute values, summed over the rules. It creates conditional rules of the form

$$\text{if } A_1 = v_1 \text{ and } A_2 = v_2 \text{ and ... then } A_{k+1} \in \{v_{k+1}, v_{k+2}, \ldots v_{k+r}\}$$

where the A_i are attributes and the v_i are values. The rules are randomly selected such that they are always satisfied in training and have high n/r.

LERAD models inbound TCP streams from client to server. The attributes are date, time, single bytes of the source and destination address, source and destination ports, TCP flags of the first, next to last or last packet, duration, length, and the first 8 words in the application payload.

There are many potential rules that could exclude real traffic, for example "if destination address = *pascal* and destination port = FTP then ...". It would be error prone to manually modify LERAD to avoid such rules. Instead, we experimented with modifying LERAD to record the number of simulated and real training instances (by using the destination address to distinguish them) that satisfy the rule antecedent, then weight each rule by the fraction of real traffic when computing the anomaly score. In other words, we use the scoring function tn_R/r, where n_R is the number of training instances from the real data. The set of r allowed values can still come from either source. This should have the effect of removing rules that depend only on the simulated traffic.

In practice, this modification had practically no effect. Only a small fraction of the rules, less than 5%, were affected significantly (removing more than 80% of the training data). When we compared the original and modified LERAD on mixed sets A, B, and C, both versions usually detected the same number of attacks (about 30) at 100 false alarms. The difference, if any, was at most one detection.

Although we did not modify LERAD, we did modify the TCP stream reassembly algorithm as with ALAD. The three TCP flag attributes are for the first three packets, or blank if there are less than three packets after filtering. The word attributes (up to 8) are extracted only from the first 134 bytes of the first data packet.

8.2.4. NETAD Modifications

Recall that NETAD models nine types of packets: IP, TCP, TCP SYN, and TCP ACK for all ports, ports 0-255, telnet, FTP, SMTP, and HTTP. NETAD already uses filtered traffic, so all that remains is to test whether the nine packet types are present in sufficient quantities in the real traffic. As can be seen from Table 8.3, two are not – telnet and FTP – so we remove them and modify NETAD to test only the seven remaining types.

Packet Type	S	A	B	C
IP	362,934	789,504	813,011	740,479
TCP	170,435	407,858	482,351	395,693
TCP SYN	29,263	83,660	79,686	80,520
TCP ACK	56,923	141,426	124,214	132,041
ACK 0-255	50,356	90,002	96,284	90,243
ACK FTP	4668	4719	4743	4726
ACK telnet	13,947	13,947	13,947	13,947
ACK SMTP	20,944	28,392	29,290	28,363
ACK HTTP	8458	31,020	33,382	31,556

Table 8.3. Number of packets of types modeled by NETAD in the filtered inside sniffer training traffic from IDEVAL week 3 (S) and in mixed sets A, B, and C.

8.2.5. SPADE Modifications

SPADE (Hoagland, 2000) is a network anomaly detection plug-in to SNORT (Roesch, 1999). It models ports and addresses of inbound TCP SYN packets (requests to servers). It uses a pure frequency based model, in which the joint probability of a combination of ports and addresses depends only on the number of times that the combination was observed (including the current packet) divided by the total number of observations. The anomaly score is inversely related to the probability. There is no explicit training period. Every packet is tested, then added to the training model.

SPADE has four probability modes, which can be selected by the user.

0. 1/P(SA, SP, DA)P(SA, SP, DP)/P(SA, SP)

1. 1/P(DA, DP, SA, SP)
2. 1/P(DA, DP, SA)
3. 1/P(DA, DP) (default)

where SA, DA, SP, and DP are the source and destination IP addresses and port numbers. All four models depend on the destination address (DA), which distinguishes the simulated and real traffic. Rather than modify these models (which we could do; it is open source), we modify the input data. For each real inbound TCP SYN packet, we randomly replace the destination address with one of the four main victim machines (pascal, marx, hume, or zeno). Thus, to SPADE, it appears as if the real servers (HTTP, SMTP, etc.) are running on these victim machines rather than on a separate host.

8.3. Evaluation Criteria

We described a procedure for making it appear to an IDS that it is monitoring real traffic in the IDEVAL data. This requires injecting the traffic into the simulated data (adjusting time stamps), and possibly modifying or removing rules from the IDS so that it makes no distinction between simulated and real traffic. We wish to test whether this procedure works. To do this, we compare the output of the IDS on simulated and mixed traffic. If the procedure works, then we would expect that only "legitimate" detections appear in the IDS output on real traffic, and that detections that appear to be due to simulation artifacts would not. We already identified suspicious detections in our analyses of previous results, for example, detections by TTL or detections of attacks on public servers by source address. In (Mahoney & Chan, 2003) we use the following criteria to decide whether a particular anomaly legitimately detects an attack.

- Source address is legitimate for denial of service (DOS) attacks that spoof it, or if the attack is on an authenticated service (e.g. telnet, auth, SSH, POP3, IMAP, SNMP, syslog,

etc), and the system makes such distinctions (i.e. conditioned on server port number). FTP is anonymous in the IDEVAL data, so we consider it public.

- Destination address is legitimate for probes that scan addresses, e.g. *ipsweep*.
- Destination port is legitimate for probes that scan or access unused ports, e.g. *portsweep*, *mscan, satan*. It is debatable whether it is legitimate for attacks on a single port, but we will allow them.
- TCP state anomalies (flags, duration) are legitimate for DOS attacks that disrupt traffic (*arppoison, tcpreset*), or crash the target (*ntfsdos, dosnuke*).
- IP fragmentation is legitimate in attacks that generate fragments (*teardrop, pod*).
- Packet header anomalies other than addresses and ports are legitimate if a probe or DOS attack requires raw socket programming, where the attacker must put arbitrary values in these fields.
- Application payload anomalies are legitimate in attacks on servers (usually R2L attacks, but may be probes or DOS).
- TCP stream length is legitimate for buffer overflows.
- No feature should legitimately detect a U2R (user to root) or Data attack (security policy violation).

In (Mahoney & Chan, 2003), we evaluated PHAD, ALAD, LERAD, and NETAD using the EVAL implementation of the 1999 IDEVAL detection criteria (Mahoney, 2003b). This differs from EVAL3, which we had been using, in three minor respects.

- If an alarm occurs during two overlapping attacks, then EVAL counts both as detected. EVAL3 counts only the attack listed first in the IDEVAL truth labels.
- The alarm IP address must be the target and not the source. The IDEVAL truth labels contain both source and target addresses. EVAL3 allowed a match to either. This change

affects only PHAD on the original unfiltered data because no other system sees remote destination addresses.

- EVAL does not count out-of-spec detections. A detection is out-of-spec if it is not one of the 177 attacks that the IDEVAL truth labels lists as having evidence in the inside sniffer traffic. EVAL3 counts these. The difference is typically about 2%. It is possible to detect an out-of-spec attack by coincidence or if it overlaps an in-spec attack. In a few cases, attacks that generate no traffic (e.g. *ntfsdos*) and are labeled as such, can still be detected because the IDS detects interrupted TCP connections when the target is rebooted.

8.4. Experimental Results

We tested the modified PHAD, ALAD, LERAD, and NETAD on simulated set S and mixed sets A, B, and C as described in Section 8.1. We evaluated the results with EVAL at 100 false alarms. Alarms were consolidated using AFIL (Mahoney, 2003b). The results are shown in Table 8.4. For comparison, the number of detections for the unmodified systems (but using AFIL/EVAL rather than EVAL3) are also shown.

For sets S and C, we manually inspect each detection and classify it as legitimate or not according to the criteria described in Section 8.3. The results are presented in the form of a fraction, legitimate/total, and a percentage. We analyze set C because it appears to be the most representative of the three mixed sets. For all four systems, the number of attacks detected using C falls between the numbers for A and B.

System	Original Total	S Legit/Total	A Total	B Total	C Legit/Total
PHAD	39	31/51 (61%)	17	31	19/23 (83%)
ALAD	57	16/47 (34%)	11	17	10/12 (83%)
LERAD	109	49/87 (56%)	29	30	25/30 (83%)
NETAD	129	61/128 (48%)	38	46	27/41 (67%)
SPADE-2		3/6 (50%)	2	1	1/1 (100%)

Table 8.4. Number of attacks detected at 100 false alarms (measured using EVAL) using the original IDEVAL inside sniffer week 3-5 data, data after filtering (S), and after injecting real traffic (A, B, and C). For S and C, the number and percentage of detections judged legitimate is shown. SPADE is evaluated in mode 2 at 200 false alarms.

For each of the four systems, the fraction of legitimate detections is higher in set C than in set S. This suggests that the technique of injecting real data and removing rules dependent on simulated data effectively removes simulation artifacts. In (Mahoney & Chan, 2003), a similar result was also obtained with SPADE (Hoagland, 2000), and with PHAD with the TTL field active. Most of the attacks detected by TTL on set S were absent in set C.

In the following sections, we summarize results for PHAD (without TTL), ALAD, LERAD, and NETAD for sets S and C.

8.4.1. PHAD Results

PHAD detects 51 attacks (plus 3 out of spec) on set S and 23 attacks (plus 1 out of spec) on set C. These are grouped by the attribute that contributes the greatest fraction of the anomaly score. An asterisk indicates the detection is not legitimate according to our criteria.

- Ethernet destination address: *mscan** in S and C. The anomaly is most likely caused by an overlapping *arppoison* attack against another IP address, which is missed.
- Ethernet source: *insidesniffer** in S. Coincidental.
- Ethernet packet size: *ncftp** in C.
- ICMP checksum: 5 *smurf** in S. This is due to a bug in the attack, one of the few attacks that was simulated.
- IP destination: *portsweep*, 2 in S and 1 in C; 3 *ncftp** in S and C; *guesstelnet** in S. Since there is only one real IP destination address, this attribute cannot be tested.
- IP fragment ID: *neptune* in S. Although unlikely, it is legitimate by our criteria because *neptune* is programmed at the IP level to spoof the source address.
- IP fragment pointer: *pod*, 4 in S and 2 in C, *teardrop*, 3 in S and C, *insidesniffer** in S (coincidental). The others exploit IP fragmentation.
- IP source address: only in S: 2 *portsweep**, *neptune*, *back**, *xlock**, *syslogd*, *ncftp**, *processtable**, *sendmail**. *neptune* and *syslogd* are legitimate because they spoof the source address. Although *processtable* attacks an authenticated service, it is not legitimate because the rule is not conditioned on port number.
- TCP checksum: *apahce2** in S (probably coincidental).
- TCP flags: 5 *portsweep* in S and 4 in C, 3 *queso* in S and 2 in C, 3 *dosnuke* in C. All are legitimate. *portsweep* and *queso* are detected by FIN without ACK. *dosnuke* is detected by the URG flag.
- Urgent pointer: 4 *dosnuke* in S and 1 in C.
- TCP window size: 1 *portsweep* in S and 2 in C, *ntinfoscan* in S. Although window size is probably an artifact, these are legitimate by our criteria because both attacks are programmed at the IP level.
- UDP checksum: *udpstorm* in S.

- UDP destination port: all in S: 2 *satan*, 2 *portsweep*, 1 *udpstorm*.
- UDP length: *syslogd* in S, *satan* in C.

8.4.2. ALAD Results

ALAD was modified to remove the destination address, telnet, and FTP from rule conditions. The modified ALAD detects 47 attacks in S and 12 in C, all of them in-spec. They are grouped by attributes that contribute at least 10% of the anomaly score. Some attacks are therefore listed more than one. A single detection may be legitimate for one attribute and not for another (marked with *). A detection must have one legitimate attribute to be classified as legitimate overall.

- Source address: 43 in S, 9 in C. In S: 2 *apache2**, *arppoison**, *casesen**, 5 *crashiis**, *fdformat**, *ffbconfig**, 2 *guessftp**, *guesspop*, *guesstelnet**, *mailbomb**, *mscan*, 2 *ncftp**, 2 *netbus**, 2 *netcat**, *netcat_breakin**, *netcat_setup**, 3 *ntinfoscan**, 2 *phf**, 1 *ppmacro**, 2 *satan**, 2 *sendmail**, *sshtrojan**, 3 *warezclient**, *warezmaster**, *xterm**, 2 *yaga**. In C: *guessftp**, *mailbomb**, *ncftp**, 2 *netcat**, *netcat_breakin**, *satan*, 2 *sendmail**. Except *satan*, the source address scores are lower in C. All of the detections in C are by at least one other attribute. In S, *guesstelnet* is detected on port 80 (not 23) and *sshtrojan* on port 25 (not 22), so these are not legitimate (probably coincidental).
- TCP flags: 3, all in S: *loadmodule**, 2 *sendmail**. Two anomalies are due to connections without an initial SYN, and one due to SYN-ACK-ACK.
- Destination port: 2 *netcat* in S and C, *netcat_breakin* in S and C, 2 *satan* in S and 1 in C, *mscan* in S, *guesspop* in S. All are legitimate by our criteria. *netcat* is detected on port 53 (DNS) because it uses TCP rather than the usual UDP. *satan* is detected on port 70 (gopher), *mscan* on port 111 (portmap), and *guesspop* on 110 (POP3). We consider

guesspop legitimate although it would only be detected if the POP3 server were running but never used.

- Keyword: 3 *mailbomb* in S and C by "mail", 2 *sendmail* in S and C by "MAIL", 2 *ntinfoscan* on S only by "HEAD" on port 80. All are legitimate by our criteria.

8.4.3. LERAD Results

LERAD was not modified except for the TCP reassembly algorithm, which was adjusted to deal with filtered and truncated packets as with ALAD. In one test run of LERAD, it detects 87 attacks on set S (plus 2 out of spec), and 30 (all in spec) on set C. These are grouped by the one attribute that contributes the largest portion of the anomaly score. Non-legitimate detections are marked with an asterisk.

- Source address: 27, all in S: anypw*, casesen*, crashiis*, dict, fdformat*, ffbconfig*, 2 guessftp*, 2 guesstelnet, 3 guest, netbus*, netcat_setup, 2 perl*, 2 ps, sechole*, sqlattack*, sshprocesstable, sshtrojan, warezclient*, warezmaster*, xterm*.
- Destination address: 6 in S and C: *guesstelnet*, mscan*, 4 *ncftp*. There is no difference probably because only one new destination address is added by the real traffic.
- Destination port: 14 in S, 11 in C. On both: 2 *ftpwrite*, 2 *ls_domain*, 3 *named*, 2 *netcat*, *netcat_breakin*, *satan*. On S only: *guesspop*, *imap*, *neptune*. *ftpwrite* is detected on port 513 (*rsh*) which is not available in the real traffic. *satan* is detected on port 70 (gopher). The others are detected on TCP port 53 (DNS), normally a UDP service. POP3 and IMAP mailbox service traffic is found in the real traffic but not in the IDEVAL background.
- TCP flags: In S and C: 4 *dosnuke* (URG flag set). In S only: 3 *back**, *insidesniffer**, *loadmodule**, *sendmail**.
- Length: in S: *netbus**, *ppmacro**. In C: *sendmail* (a buffer overflow).
- Duration: *insidesniffer** in S (coincidental).

- Payload: 8 in C: *back* (word 1 = "G"), 2 *imap* (word 3 = """, a quote character), 2 *ntinfoscan* (word 3 = "EOEBFECNFDEMEJDB"), *phf* (word 1 = "G"), *satan* (word 1 = "QUIT" on port 80), *sendmail* (word 3 = "root@calvin.worl"). 29 in S: 3 *apache2, back, 6 crashiis, guesstelnet, imap,* 3 *mailbomb,* 3 *ntinfoscan,* 3 *phf, portsweep, queso, satan, sendmail,* 4 *yaga**. The *yaga* (U2R) detections include a *crasiis* to reboot the target (after a registry edit) which is detected by the absence of word 3, normally the HTTP version. The *back* and *phf* anomalies ("G") could be due to the HTTP GET command being split among packets in an interactive telnet session to the HTTP port. These would be missed using a better TCP reassembly algorithm.

The difference between set S (87 detections) and the original results (117 detections) is mainly due to the loss of trailing TCP data. This affects the closing flags, length, duration, and part of the payload. On the modified data, LERAD is unable to detect broken TCP connections which are often a sign of a DOS attack.

8.4.4. NETAD Results

NETAD was modified to remove telnet and FTP rules. The modified NETAD detects 129 (plus 3 out of spec) attacks on S, and 41 (plus one out of spec) on set C. The detections below are grouped by attributes that contribute at least 10% of the anomaly score. Non-legitimate detections are marked with an asterisk.

- Source address: 65 in S, 3 in C. Most are not legitimate. On S: *anypw**, *apache2**, *arppoison**, 4 *crashiis**, 2 *eject**, *fdformat**, 2 *guessftp**, 2 *guesstelnet**, 2 *guest**, *imap**, *insidesniffer**, 3 *ipsweep**, *ls_domain**, *mailbomb**, *ncftp**, 3 *netbus**, *netcat_setup**, 2 *perl**, *pod**, 3 *portsweep**, *ppmacro**, *processtable**, 2 *ps**, *satan**, *sechole**, *secret**, 4 *smurf**, *sqlattack**, *sshtrojan**, 4 *syslogd, tcpreset**, 3 *warezclient**, 3 *xlock**, 3 *xsnoop**, 2 *xterm**, 2 *yaga**. On C: *ncftp**, *xlock**, *xsnoop**. Only source address forgeries (*syslogd*)

can be considered legitimate because the modified NETAD does not have any rules conditioned on private ports (*telnet* and FTP were removed).

- Destination address: 8 in S, 10 in C. This cannot be tested because C adds only one address. In S: *guesstelnet*, mscan,* 4 *ncftp*, netbus*, portsweep*. In C: 2 additional *netbus**.
- Destination port: 8 in S, 2 in C: In S: *guesspop*, 2 *imap*, 2 *ls_domain*, 2 *netcat*, 1 *satan*. In C: 2 *satan* (port 70, gopher).
- IP fragmentation: 7 in S and C: In S: 4 *pod*, 3 *teardrop*. In C: 3 *pod*, 3 *teardrop*, 1 *insidesniffer** (coincidental).
- TCP flags: 5 in S, 12 in C. In S: 5 *portsweep* (FIN scan). In C: 5 *portsweep*, 3 *queso* (FIN), 4 *dosnuke* (URG).
- TOS: 2 in S and C: 2 *ftpwrite**.
- Payload: 22 in S, 6 in C. In S: 3 *back, casesen*, ffbconfig*, land*, 3 *named*, neptune*, 6 *portsweep*, 3 *queso*, 2 *sendmail, udpstorm*. In C: *back* ("E" in first byte), 3 *named* (8-bit ASCII), 2 *sendmail* ("A" in "MAIL"). The payload anomalies in *neptune, portsweep*, and *queso* in S are actually due to the absence of TCP options (an artifact) where the payload would normally appear. The *back* anomaly is probably due to TCP fragmentation of the HTTP "GET" command.
- IP length/TCP header length: 23, all in S: 2 *back*, ffbconfig*, land, 3 named, neptune, 2 pod*, 6 *portsweep*, 3 *queso*, 4 *smurf*. Although many of these attacks produce unusually large or small packets, there is probably enough natural variation in real packet size to mask these detections.

8.4.5. SPADE Results

We evaluated SPADE version 092200.1 within SNORT version 1.7 Win32 in each of the four user selectable probability modes, with all other SNORT rules turned off and with all other SPADE parameters set to their defaults. SPADE has a variable threshold that adapts over a period of hours to regulate alarms, but we used the raw anomaly score reported by SPADE instead. As mentioned, SPADE does not have an explicit training period. Instead it tests each packet based on the joint frequency of the attributes in all previous packets. We ran SPADE on inside sniffer weeks 3-5 as a single data stream and reported all alarms during weeks 4-5. Because SPADE was found to perform poorly on our data, we evaluated it at a threshold of 200 false alarms.

SPADE Detections at 200 False Alarms	**S**	**A, B, C**
0: P(SA, SP, DA)P(SA, SP, DP)/P(SA, SP)	6	6, 6, 7
1: P(DA, DP, SA, SP)	1	0, 0, 0
2: P(DA, DP, SA)	6	2, 1, 1
3: P(DA, DP) (default)	8	9, 8, 7

Table 8.5. SPADE detections at 200 false alarms on filtered IDEVAL weeks 3-5 (S) and on mixed sets A, B, and C.

Modes 0 and 1 include the source address, which does not normally contain meaningful information, as it is picked arbitrarily by the client. The attacks detected in mode 0 on set S are *insidesniffer, syslogd, mscan, tcpreset, arppoison,* and *smurf.* All but *mscan* are probably coincidental because none of them generate TCP SYN packets and most are prolonged attacks with multiple targets.

Modes 2 and 3 show the effect that we would expect if source address were an artifact, but not the destination address or port, as Chapter 7 suggests. The six attacks detected by SPADE in mode 2 (1/P(DA, DP, SA)) on S are *guest*, *syslogd*, *insidesniffer*, *perl*, *mscan*, and *crashiis*. By our previously mentioned criteria, three of these are legitimate: *guest* because it attacks a private service, *syslogd* because it forges the source address, and *mscan* because it accesses unused ports. We do not count *insidesniffer* (no TCP SYN packets), *perl* (U2R), or *crashiis* (a public server). On sets A, B, and C, SPADE detects only *mscan* (and *portsweep* on A), both of which can be legitimately detected by the destination port. Thus, the effect of injecting real traffic is to increase the percentage of legitimate detections from 50% to 100%. There is no effect in mode 3 because we did not identify the destination address or port as being affected by artifacts.

8.5. Results Analysis

In this section, we analyze the combined results for each of our systems. The top two systems are NETAD, which tests packets, and LERAD, which tests TCP streams. All of the principles outlined in this paper are implemented in these two systems. We can think of LERAD as an enhanced version of ALAD with rule learning and validation. We can think of NETAD as an enhanced version of PHAD with conditional rules, filtering, and modeling of non-novel values. These two systems cover all of the attributes modeled by the other two, e.g. packet fields and TCP streams.

We had, up to this point, been evaluating our systems by the number of attacks detected, whether or not it was reasonable to detect them. There are 201 attacks in weeks 4 and 5 of the IDEVAL data, but only 189 when you subtract the 12 in the missing day of inside sniffer traffic (week 4, day 2). If you consider only those attacks for which there is evidence in the traffic

according to the IDEVAL truth labels, then there are 177. Occasionally we detected a few others, due to coincidences, overlapping attacks, or unanticipated side effects of the attack.

In the 1999 evaluation, participants could classify their systems as to the types of attacks they were designed to detect, so they were not penalized for missing out-of-spec attacks. Participants categorized their systems not only by the data they examined, but also by the category of attack: probe, DOS, R2L, U2R, or data. It is difficult for a network anomaly detection system to detect U2R attacks, where the attacker with user access gains the privileges of another user (usually root). These attacks exploit operating system flaws using input data which is not directly observable on the network. It might be possible to observe exploits in a telnet session or an FTP upload, but the attack could easily be hidden using encryption. Also, it is difficult to detect data attacks, which are violations by authorized users of a written security policy for which the IDS has no knowledge, for example, copying or transmitting secret but unprotected data files.

Therefore, a reasonable specification for a network anomaly detection system is that it should detect probes, DOS, and if it examines the application payload, R2L attacks. In the inside sniffer traffic, there are 148 such attacks. In Figure 8.2, we show the detection-false alarm (DFA) graphs for NETAD and LERAD on sets S (simulated, modified), and C (mixed with real traffic). Recall that C gives the median number of detections out of A, B, and C at 100 false alarms for all four systems. At 100 false alarms, LERAD detects 72 (49%) in-spec attacks on set S, and 30 (20%) on set C. NETAD detects 111 (75%) on set S and 41 (28%) on set C. If we are willing to tolerate more false alarms, then we could detect 69 attacks (47%) in the mixed set C using NETAD at 500 false alarms (50 per day), while LERAD levels off at 34 (23%). Remember that these results use modified algorithms to reduce the effects of simulation artifacts. LERAD and NETAD would probably detect more legitimate attacks in real traffic if we had used more of the TCP payload for LERAD or included telnet and FTP rules for NETAD.

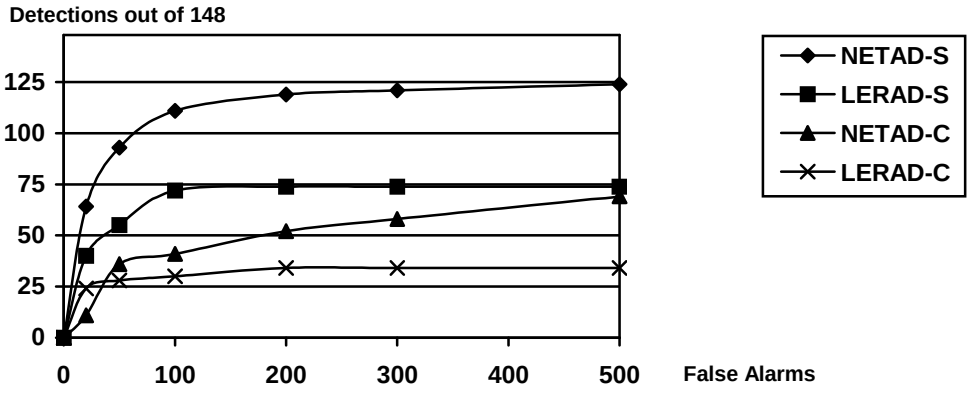

Figure 8.2. Probe, DOS, and R2L attacks detected by NETAD and LERAD on simulated (S) and mixed (C) traffic as 0 to 500 false alarms (0 to 50 per day).

In Table 8.6 we summarize the number of inside sniffer detections by category for LERAD and NETAD on sets S and C. (The totals are more than 177 because there are some R2L-Data and U2R-Data attacks). One striking effect of injecting real network data that we have already seen is the near elimination of detections by source address, which account for about half of all detections in simulation. These detections are, of course, mostly spurious. Another effect, which can be seen in the table, is the elimination of U2R and data detections. Again, this is the behavior we should be seeing. It should be noted that PHAD and ALAD do not detect any U2R or data attacks on the mixed traffic either.

The last column of Table 8.6 shows the number and percentage of attacks detected by the combination of LERAD and NETAD on mixed set C when we take equal numbers of top scoring alarms from each system and consolidate duplicates. The merged system detects 44 attacks, or 30% of the 148 in-spec attacks, better than either system alone. This improvement is possible because the two systems monitor different attributes, and therefore detect different types of attacks. This improvement does not occur when merging LERAD and NETAD on set S because there is

significant overlap due to detections by the client source address artifact. This artifact is shared by both systems.

Category	Total	LERAD-S	LERAD-C	NETAD-S	NETAD-C	Merged-C
Probe	34	11 (32%)	7 (21%)	30 (88%)	12 (35%)	15 (44%)
DOS	60	26 (43%)	5 (8%)	42 (70%)	11 (18%)	9 (15%)
R2L	54	35 (65%)	18 (33%)	39 (72%)	18 (33%)	20 (37%)
U2R	27	15 (56%)	0 (0%)	16 (59%)	0 (0%)	0 (0%)
Data	7	1 (14%)	0 (0%)	2 (28%)	0 (0%)	0 (0%)
Total	177	87 (49%)	30 (17%)	128 (72%)	41 (23%)	44 (25%)
In-Spec	**148**	**72 (49%)**	**30 (20%)**	**111 (75%)**	**41 (28%)**	**44 (30%)**

Table 8.6. Attacks detected by modified LERAD and NETAD at 100 false alarms on simulated set S and mixed real set C. Merged results are LERAD and NETAD combined on set C. Only attacks visible in the inside sniffer traffic are counted. In-spec refers to probes, DOS, and R2L.

8.6. Summary

We tested PHAD, ALAD, LERAD, and NETAD on the IDEVAL data set with real traffic injected, modifying the algorithms as needed to ensure that all of the rules are trained at least partially on real traffic to remove the effects of simulation artifacts in the background traffic. To test whether we were successful, we used a somewhat subjective criteria to test whether the detected attacks were legitimate, i.e. whether the system was detecting a feature of the attack. We found that in every system we tested, that the fraction of legitimate attacks was higher in mixed traffic than in

simulated traffic. However, our criteria is not perfect. On closer examination of some non-legitimate detections, we often find reasonable explanations that do not fit our criteria. Also, some legitimate detections in the simulated set are missed in the mixed set. This could be due to a greater number of false alarms. If nothing else, having twice as much background traffic should generate twice as many alarms.

In table 8.7, we summarize the number of probe, DOS, and R2L attacks (out of 148) detected by the original PHAD, ALAD, LERAD, and NETAD at 100 false alarms on the IDEVAL inside sniffer weeks 3-5, after modification in preparation for injecting real traffic (S), and after injection (sets A, B, and C). All systems are evaluated with EVAL and exclude U2R and data attacks, and attacks for which no evidence is visible in the inside sniffer traffic. Alarms are consolidated with AFIL.

Attacks Detected	**IDEVAL – Orig.**	**Modified – S**	**Mixed – A, B, C**	**Average**
PHAD	39 (26%)	51 (34%)	17, 31, 23	24 (16%)
ALAD	45 (30%)	39 (26%)	11, 15, 12	13 (9%)
LERAD (average)	97 (66%)	72 (49%)	29, 30, 30	30 (20%)
NETAD	111 (75%)	111 (75%)	38, 46, 41	42 (28%)

Table 8.7. Probe, DOS, and R2L attacks detected at 100 false alarms on IDEVAL inside sniffer weeks 3-5, on filtered traffic after modification to accept mixed traffic (S), on mixed sets A, B, and C, and the average over the three mixed sets.

Chapter 9

Conclusions

We described several methods of improving network anomaly detection systems and implemented these methods in four systems. The best of these (NETAD) detects 75% of in-spec attacks (probe, DOS, and R2L) at 10 false alarms per day in the 1999 DARPA/Lincoln Laboratory intrusion detection evaluation (IDEVAL), compared to 40% to 55% for the top participants using a combination of methods (signature, host based) in the original blind evaluation. However, an analysis of the IDEVAL data compared to real traffic suggests that the IDEVAL data contains many simulation artifacts that make attacks easy to detect. When we remove these artifacts by injecting real traffic (and verify that most detections are legitimate), we find that the best combination of systems (merging LERAD and NETAD) detects 30% of attacks (median of 3 mixed data sets). We can detect more attacks if we are willing to accept more false alarms. At 50 false alarms per day, modified NETAD detects a median of 69 of 148 attacks (47%) on this data.

9.1. Summary of Contributions

The following is a summary of our contributions.

Time Based Modeling. In anomaly detection, it is common to model the probability of events based on their average frequency in training. Because some events occur in bursts (leading to alarm floods), we use a hybrid time-frequency model. We use a model in which an event is anomalous if it is both rare on average, and has not occurred recently either. For novel events (score = tn/r), the time-based part is t (time since the last anomaly), and the frequency-based part is n/r (low rate of novel values in training). For non-novel events (score = t_i/f_i), the time-based part is t_i,

the time since value *i* was last observed, and the frequency part is f_i, the average frequency of *i* in training.

Protocol Modeling. Some anomaly detection systems monitor only a few attributes, such as IP addresses and port numbers, because they can identify remote users or characterize their behavior. These types of anomalies can detect port scans and some R2L attacks on private services such as telnet or POP3, but cannot detect most DOS or R2L attacks on public services such as IP, TCP, SMTP, or HTTP. An attack can introduce protocol anomalies in one of four ways.

- By exploiting a bug in the target. If the data which invoked the bug were common, the bug would have been detected and fixed. (Example: detecting *pod* and *teardrop* by IP fragmentation).
- Failing to duplicate the target environment. The more attributes we monitor, the harder it is for an attacker to get everything right. (Example: detecting *mailbomb* and *sendmail* because the normal HELO/EHLO handshake was omitted).
- Evasion. Low-level attacks on the IDS used to hide high-level attacks on the target can backfire if we monitor low-level protocols. (Example: detecting *portsweep* in FIN scanning mode).
- Symptoms of a successful attack. The output of a compromised target differs from normal. (Example: detecting DOS attacks by broken TCP connections).

On real traffic, many different attributes detect only a few attacks each. No single attribute dominates.

Rule Learning. In LERAD we introduced a rule learning algorithm that automatically generates good rules for anomaly detection from training data with arbitrary nominal attributes. Like association mining algorithms such as RIPPER or APRIORI, it is off-line, requiring more than one pass through the training data. Unlike these algorithms, its goal is different (to find rules with high *n/r*), and it is randomized. We use matching attributes in pairs of training samples to suggest rule candidates, then remove redundant rules in favor of higher *n/r* estimated on a small sample.

Rule Validation. In LERAD and NETAD we remove, or assign a low weight to, rules that generate false alarms on an attack-free validation set taken from the end of the training data.

Filtering. Most attacks can be detected by examining just a very small fraction of the traffic. We look at only the first few client packets of inbound sessions, less than 2% of the total traffic. All of our implementations can process 3 weeks worth of filtered data in times ranging from a few seconds to one minute on a 750 MHz PC.

Continuous Modeling. A practical anomaly detection system must train and test continuously on the same traffic. Some decrease in accuracy is inevitable as the IDS trains on hostile traffic as if it were normal. This effect is smaller if the system models non-novel values (t_i/f_i). However, rule validation cannot be used without attack-free traffic. Also, off-line (multiple pass) rule learning algorithms such as LERAD cannot be used.

IDEVAL Simulation Artifacts. We found several attributes in the IDEVAL training and background traffic that make it artificially easy to detect attacks by anomaly detection, in spite of great efforts to avoid this problem. These attributes are client IP addresses, TTL, TCP options, TCP window size, SMTP and HTTP keywords, and HTTP and SSH client versions. These rules are "good" in the IDEVAL data (r is small and does not grow) and "bad" in real traffic (r is large and grows steadily).

Removing Artifacts. Some artifacts in the IDEVAL data can be removed by injecting real traffic. Evaluation with mixed traffic requires that IDS rules not affected by the real traffic in training be turned off. On all of our systems plus SPADE, this technique increases the fraction of detections judged legitimate. On real traffic, a greater fraction of LERAD rules are removed by validation.

Merging IDS Outputs. Sometimes a combination of intrusion detection systems can detect more attacks than any of its components. The technique is to take the highest scoring alarms from each system and consolidate duplicates. This technique works best if the components are equally strong but detect different types of attacks.

9.2. Limitations and Future Work

We described several network anomaly detection techniques and implemented them in four experimental algorithms. However, our systems have a number of limitations.

- False alarms are a problem because unusual events are not necessarily hostile.
- Alarm reports are not helpful. The system can report anomalies but cannot help the user decide whether the alarm is hostile or not. Making this decision requires the user to examine the traffic and to be an expert in network protocols and security.
- Rule validation requires attack-free training data, which is hard to obtain.
- Rule learning requires multiple passes through the training data, which prevents the method from being used online.
- Application payload monitoring will become impractical with the increased use of encryption. Eventually this problem will extend to lower level protocols as well (e.g. virtual private networks).

In addition, there are problems with testing our systems.

- They were not tested in a live environment.
- Our systems were developed with access to the test data, which introduces a bias. For this reason, we cannot claim an improvement over the IDEVAL participants.
- The IDEVAL data appears to contain many simulation artifacts. These are responsible for the majority of detections in our best systems, and an artificially low false alarm rate.

Our solution to the artifact problem was to inject real traffic into the simulation, but that introduces new problems.

- Real data contains unlabeled attacks.
- Experiments with real data cannot be reproduced because privacy and security concerns do not allow the data to be made public.

- There is a 3 year time lag between IDEVAL and the real traffic that we used. Protocols evolve. The problem can only get worse.
- Not all of the protocols seen in IDEVAL are available in the real data (e.g. *telnet*).
- Evaluation with mixed traffic requires a careful analysis and system modifications to make sure that all rules depend at least partially on real traffic. Testing whether this was done correctly is subjective.

Given that attacks are now common, the approach of Newman et al (2002) of using honeypots in a live environment might be a more practical solution to the evaluation problem. Still, this is a labor intensive and error prone process because the attacks are not under control of the experimenter, and must be identified and labeled. Running several independent systems in parallel might be helpful because attacks could be labeled by consensus.

Despite these problems, anomaly detection in its present form might still be useful as a tool for offline forensic analysis after an attack, helping a network administrator pinpoint the attack within gigabytes of traffic, rather than as a first line defense. In this respect, there are many small improvements that could be made to our systems, for example:

- Adding session attributes, for example, packet rate. Currently each packet or TCP stream is evaluated independently.
- Being smarter about parsing the application payload (especially binary protocols like DNS). Tokenizing words using white space does not work for every protocol.
- Adding user feedback – allowing the user to specify whether an anomaly is hostile or not, and thus whether it should be added to the training data.
- Developing an online rule learning algorithm. Currently LERAD requires two passes, but this obstacle should not be insurmountable.

We do not expect that all of the problems with network anomaly detection will be easily solved. Computer security will be a problem for quite some time. We do not pretend to have solved the problem. Instead, we have outlined some principles by which the problem can be approached.

References

Note: Internet references are current as of Feb. 2003.

L. A. Adamic (2002), "Zipf, Power-laws, and Pareto - A Ranking Tutorial", http://ginger.hpl.hp.com/shl/papers/ranking/ranking.html

R. Agrawal & R. Srikant (1994), "Fast Algorithms for Mining Association Rules", *Proc. 20th Intl. Conf. Very Large Data Bases*.

D. Anderson,. et. al. (1995), "Detecting Unusual Program Behavior using the Statistical Component of the Next-generation Intrusion Detection Expert System (NIDES)", Computer Science Laboratory SRI-CSL 95-06.

T. Armstrong (2001), "Netcat - The TCP/IP Swiss Army Knife", http://www.sans.org/rr/audit/netcat.php

S. Axelsson (1999), "Research in Intrusion Detection Systems: A Survey", TR 98-17, Chalmers University of Technology.

D. Barbara, J. Couto, S. Jajodia, L. Popyack, & N. Wu (2001a), "ADAM: Detecting Intrusions by Data Mining", *Proc. IEEE Workshop on Information Assurance and Security*, 11-16.

D. Barbara, N. Wu, & S. Jajodia (2001b), "Detecting Novel Network Attacks using Bayes Estimators", *Proc. SIAM Intl. Data Mining Conference*.

T. Bell, I. H. Witten, J. G. Cleary (1989), "Modeling for Text Compression", *ACM Computing Surveys* 21(4), 557-591, Dec. 1989.

S. M. Bellovin (1993), "Packets Found on an Internet", *Computer Communications Review*, 23(3) 26-31, http://www.research.att.com/~smb/papers/packets.ps

R. Beverly (2003), "MS-SQL Slammer/Sapphire Traffic Analysis", MIT LCS, http://momo.lcs.mit.edu/slammer/

CERT (2003a), "CERT Advisory CA-2003-04 MS-SQL Server Worm", http://www.cert.org/advisories/CA-2003-04.html

CERT (2003b), "CERT/CC Statistics 1988-2002", http://www.cert.org/stats/cert_stats.html

G. W. Cleary, W. J. Teahan (1995), “Experiments on the zero frequency problem”, *Proc. Data Compression Conference*, 480.

W. W. Cohen (1995), "Fast Effective Rule Induction", *Proc. 12th International Conference on Machine Learning*.

R. Deraison (2003), NESSUS, http://nessus.org

D. Farmer & W. Venema (1993), "Improving the Security of Your Site by Breaking Into it", http://www.fish.com/satan/admin-guide-to-cracking.html

S. Floyd & V. Paxson (2001), "Difficulties in Simulating the Internet", *IEEE/ACM Transactions on Networking*.

S. Forrest, S. A. Hofmeyr, A. Somayaji, & T. A. Longstaff (1996), "A Sense of Self for Unix Processes", *Proc. 1996 IEEE Symposium on Computer Security and Privacy*.

S. Forrest (2002), Computer Immune Systems, Data Sets and Software, http://www.cs.unm.edu/~immsec/data-sets.htm

Fyodor (1998), "Remote OS detection via TCP/IP Stack FingerPrinting", http://www.insecure.org/nmap/nmap-fingerprinting-article.html (updated June 11, 2002)

Fyodor (2002), "NMAP Idle Scan Technique (Simplified)", http://www.insecure.org/nmap/idlescan.html

Fyodor (2003), NMAP, http://www.insecure.org/nmap/

W. Gale & G. Sampson (1995), "Good-Turing Frequency Estimation without Tears", *Journal of Quantitative Linguistics* (2) 217-237.

A. K. Ghosh & A. Schwartzbard (1999), "A Study in Using Neural Networks for Anomaly and Misuse Detection", *Proc. 8'th USENIX Security Symposium*.

J. W. Haines,, R.P. Lippmann, D.J. Fried, M.A. Zissman, E. Tran, & S.B. Boswell (2001), 1999 DARPA Intrusion Detection Evaluation: Design and Procedures", MIT Lincoln Laboratory technical report TR-1062.

J. Hoagland (2000), SPADE, Silicon Defense, http://www.silicondefense.com/software/spice/

Horizon (1998), "Defeating Sniffers and Intrusion Detection Systems", *Phrack* 54(8), http://www.phrack.org

B. A. Huberman & L. A. Adamic (1999), "The Nature of Markets in the World Wide Web", http://ideas.uqam.ca/ideas/data/Papers/scescecf9521.html

"ICSA 1998 Computer Virus Prevalence Survey" (1998), http://www.windowsecurity.com/uplarticle/10/98virsur.doc

K. Kendall (1998), "A Database of Computer Attacks for the Evaluation of Intrusion Detection Systems", Master's Thesis, Massachusetts Institute of Technology.

W. E. Leland, M. S. Taqqu, W. Willinger, & D. W. Wilson (1993), "On the Self-Similar Nature of Ethernet Traffic", *Proc. ACM SIGComm.*

U. Lindquist, & P. Porras, "Detecting Computer and Network Production-Based Expert System Toolset (PBEST)", *Proc. 1999 IEEE Symposium on Security and Privacy*.

R. Lippmann, & J. W. Haines (2000), "Analysis and Results of the 1999 DARPA Off-Line Intrusion Detection Evaluation, in Recent Advances in Intrusion Detection", *Proc. Third International Workshop RAID.*

R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, & K. Das (2000), "The 1999 DARPA Off-Line Intrusion Detection Evaluation", *Computer Networks* 34(4) 579-595. Data is available at http://www.ll.mit.edu/IST/ideval/

M. Mahoney (2003a), "Network Traffic Anomaly Detection Based on Packet Bytes", *Proc. ACM-SAC*.

M. Mahoney (2003b), Source code for PHAD, ALAD, LERAD, NETAD, SAD, EVAL3, EVAL4, EVAL and AFIL.PL is available at http://cs.fit.edu/~mmahoney/dist/

M. Mahoney & P. K. Chan (2001), "PHAD: Packet Header Anomaly Detection for Identifying Hostile Network Traffic", Florida Tech. technical report CS-2001-04

M. Mahoney & P. K. Chan (2002a), "Learning Models of Network Traffic for Detecting Novel Attacks", Florida Tech. technical report CS-2002-08.

M. Mahoney & P. K. Chan (2002b), "Learning Nonstationary Models of Normal Network Traffic for Detecting Novel Attacks ", *Proc. SIGKDD*, 376-385.

M. Mahoney & P. K. Chan (2003), "An Analysis of the 1999 DARPA/Lincoln Laboratories Evaluation Data for Network Anomaly Detection", Florida Tech. technical report CS-2003-02.

J. McHugh (2000), "Testing Intrusion Detection Systems: A Critique of the 1998 and 1999 DARPA Intrusion Detection System Evaluations as Performed by Lincoln Laboratory", *ACM TISSEC* 3(4) 262-294.

T. Mitchell (1997), *Machine Learning*, New York: McGraw Hill.

M. Mitzenmacher (2001), "A Brief History of Generative Models for Power Law and Lognormal Distributions", Harvard Univ. TR-08-01, http://citeseer.nj.nec.com/461916.html

D. Moore, V. Paxson, S. Savage, C. Shannon, S. Staniford, & N. Weaver (2003), "The Spread of the Sapphire/Slammer Worm", CAIDA, ICSI, Silicon Defense, UC Berkeley EECS and UC San Diego CSE, http://www.caida.org/outreach/papers/2003/sapphire/sapphire.html

P. G. Neumann & P. A. Porras (1999), "Experience with EMERALD to DATE", *Proc. 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, 73-80.

D. Newman, J. Snyder, & R. Thayer (2002), "Crying wolf: False alarms hide attacks", *Network World Fusion*, June 24, http://www.nwfusion.com/techinsider/2002/0624security1.html

V. Paxson (1998), "Bro: A System for Detecting Network Intruders in Real-Time", *Proc. 7'th USENIX Security Symposium*.

V. Paxson, S. Floyd (1995), "The Failure of Poisson Modeling", *IEEE/ACM Transactions on Networking* (3) 226-244.

V. Paxson (2002), The Internet Traffic Archive, http://ita.ee.lbl.gov/

T. H. Ptacek & T. N. Newsham (1998), Insertion, Evasion, and Denial of Service: Eluding Network Intrusion Detection, http://www.robertgraham.com/mirror/Ptacek-Newsham-Evasion-98.html

M. Roesch (1999), "Snort - Lightweight Intrusion Detection for Networks", *Proc. USENIX Lisa '99*.

S. Sanfilippo (2003), HPING, http://www.hping.org

A. Schwartzbard & A.K. Ghosh (1999), "A Study in the Feasibility of Performing Host-based Anomaly Detection on Windows NT", *Proc. 2nd Recent Advances in Intrusion Detection (RAID)*.

D. Shkarin (2002), "PPM: One Step to Practicality", *Proc. Data Compression Conference.*

Solar Designer (1998), Designing and Attacking Port Scan Detection Tools, Phrack 53(13), http://www.phrack.org

E. H. Spafford (1988) "The Internet Worm Program: An Analysis", Purdue Technical Report CSD-TR-823.

S. Staniford, V. Paxson, & N. Weaver (2002), "How to 0wn the Internet in your Spare Time", *Proc. 11'th USENIX Security Symposium*.

H. S. Stone (1993), *High-Performance Computer Architecture*, Reading MA: Addison-Wesley.

M. Taylor (2000), RK Software, http://rksoft.virtualave.net/index.html

M. Tyson, P. Berry, N. Williams, D. Moran, & D. Blei (2000), "DERBI: Diagnosis, Explanation and Recovery from computer Break-Ins", http://www.ai.sri.com/~derbi/

A. Valdes & K. Skinner (2000), "Adaptive, Model-based Monitoring for Cyber Attack Detection", *Proc. RAID, LNCS 1907*, 80-92, Springer Verlag. http://www.sdl.sri.com/projects/emerald/adaptbn-paper/adaptbn.html

G. Vigna, S.T. Eckmann, & R.A. Kemmerer (2000), "The STAT Tool Suite", *Proc. DARPA Information Survivability Conference and Exposition (DISCEX)*, IEEE Press

G. Vigna & R. Kemmerer (1999), "NetSTAT: A Network-based Intrusion Detection System", *Journal of Computer Security*, 7(1), IOS Press.

I. H. Witten, T. C. Bell (1991), “The Zero-Frequency Problem: Estimating the Probabilities of Novel Events in Adaptive Text Compression”, *IEEE Trans. on Information Theory*, 37(4): 1085-1094

J. Wolf (2000), "Government Computer Security Gets Low Grade", Reuters, Sept. 11.

G. K. Zipf (1935), *The Psycho-Biology of Language, an Introduction to Dynamic Philology*, M.I.T. Press.

Appendix A

Example LERAD Run

Appendix A lists the rules generated by one run of LERAD, the detected attacks, and the top scoring alarms as described in Section 5.2.2. This run includes UDP and ICMP packets, and detects 111 attacks at 100 false alarms according to EVAL3.

A.1. Rules

The rules are sorted by descending *n/r*, where *n* is the number of inbound TCP streams and UDP and ICMP packets satisfying the antecedent, and *r* is the number of values observed in training. The format is as follows:

Rule-number n/r if $A_1=v_1$ $A_2=v_2$ ... $A_k=v_k$ then $A_{k+1} = v_{k+1}$ v_{k+2} ... v_{k+r}

where the A_i are attributes and v_i are values. Values are represented by strings, and sometimes have a leading dot so that empty strings can be represented. The attributes are as follows:

- DATE, TIME in the form MM/DD/YY, HH:MM:SS.
- SA3, SA2, SA1, SA0: source IP address as 4 decimal bytes (0-255).
- DA1, DA0: lower two bytes of the destination IP address (always 172.16.x.x).
- F1, F2, F3: first, next to last, and last TCP flags, in the form ".10UAPRSF", where a character is present if the corresponding flag bit is set to 1. The flags correspond to the two reserved TCP flags, URG, ACK, PSH, RST, SYN, FIN. For example, ".AP" means that

the ACK and PSH flags are set. A dot by itself indicates that no flags are set. For UDP packets, F1 = "UDP, F2 = F3 = ".". For ICMP packets, F1 = "ICMP", F2 is the type field (0-255), and F3 is the code field (0-255).

- SP, DP: source and destination port numbers (0-65535).
- LEN: floor($\log_2$(payload length in bytes))
- DUR: floor($\log_2$(duration in seconds))
- W1-W8: first 8 words of the payload, with a leading dot to distinguish empty strings. Words are delimited by white space (spaces, tabs, carriage returns, linefeeds, etc.) and truncated at 8 characters. Nonprintable characters are represented in the form ^C, where C is the character obtained by adding 64 to the ASCII code. For example, ^@ is a NUL byte (ASCII code of 0).

```
1 39406/1 if SA3=172 then SA2 = 016
2 39406/1 if SA2=016 then SA3 = 172
3 28055/1 if F1=.UDP then F3 = .
4 28055/1 if F1=.UDP then F2 = .
5 28055/1 if F3=. then F1 = .UDP
6 28055/1 if F3=. then DUR = 0
7 27757/1 if DA0=100 then DA1 = 112
8 25229/1 if W6=. then W7 = .
9 25221/1 if W5=. then W6 = .
10 25220/1 if W4=. then W8 = .
11 25220/1 if W4=. then W5 = .
12 17573/1 if DA1=118 then W1 = .^B^A^@^@
13 17573/1 if DA1=118 then SA1 = 112
14 17573/1 if SP=520 then DP = 520
```

```
15 17573/1 if SP=520 then W2 = .^P^@^@^@

16 17573/1 if DP=520 then DA1 = 118

17 17573/1 if DA1=118 SA1=112 then LEN = 5

18 28882/2 if F2=.AP then F1 = .S .AS

19 12867/1 if W1=.^@GET then DP = 80

20 68939/6 if then DA1 = 118 112 113 115 114 116

21 68939/6 if then F1 = .UDP .S .AF .ICMP .AS .R

22 9914/1 if W3=.HELO then W1 = .^@EHLO

23 9914/1 if F1=.S W3=.HELO then DP = 25

24 9914/1 if DP=25 W5=.MAIL then W3 = .HELO

25 9898/1 if F1=.S F3=.AF W5=.MAIL then W7 = .RCPT

26 28882/3 if F2=.AP then F3 = .AP .AF .R

27 28055/3 if F1=.UDP then SA1 = 112 115 001

28 34602/4 if F3=.AF then F2 = . .S .AP .AS

29 68939/8 if then SA3 = 172 196 197 194 195 135 192 152

30 68939/8 if then F3 = . .S .AP .AF .3 .0 .AS .R

31 68939/8 if then F2 = . .S .AP .3 .A .0 .AS .8

32 29549/4 if F1=.S then F2 = . .S .AP .A

33 39406/6 if SA2=016 then SA1 = 112 113 115 114 000 116

34 12885/2 if DP=80 then W1 = . .^@GET

35 12867/2 if W1=.^@GET then W3 = .HTTP/1.0 .align=

36 25169/4 if W3=. then LEN = 0 5 4 3

37 30237/5 if SA2=016 DUR=0 W8=. then F2 = . .S .AP .0 .AS

38 28055/5 if F1=.UDP then DP = 520 137 514 138 161

39 68939/13 if then SA2 = 016 037 115 182 168 169 218 027 008 227
073 007 013
```

```
40 20732/4 if SA3=172 W3=. then LEN = 0 5 4 3
41 35132/7 if W8=. then F3 = . .S .AP .AF .0 .AS .R
42 28786/6 if F2=. then LEN = 0 5 7 6 9 8
43 9585/2 if SA2=016 F3=. LEN=7 then SA0 = 234 050
44 12838/3 if DP=25 then W1 = . .^@EHLO .^@HELO
45 25229/6 if W6=. then W4 = . .^Ppd .syslogd: .lupitam@ .randip@z
.^@^F^@^@
46 68939/17 if then DUR = 0 5 11 7 12 1 13 15 10 6 9 4 3 8 2 14 16
47 68939/17 if then LEN = 0 5 11 7 12 13 15 10 6 9 4 3 8 14 16 18
17
48 35132/9 if W8=. then DUR = 0 5 7 12 1 6 4 3 2
49 68939/18 if then SA1 = 112 113 075 115 091 114 218 251 060 000
001 033 151 248 177 216 116 215
50 58458/16 if DUR=0 then DP = 520 0 113 25 137 23 80 20 79 514 515
1023 22 138 1022 161
51 68939/19 if then DP = 520 0 113 25 137 23 80 135 20 79 21 514
515 1023 22 138 139 1022 161
52 68939/19 if then DA0 = 255 020 105 100 234 084 168 148 169 204
194 050 207 149 005 010 087 044 201
53 12838/4 if DP=25 then W3 = . .HELO .MAIL .^@^@^@^@
54 25229/8 if W6=. then W3 = . .^@^A .^@^A^@^D .PASS .6667^M^
.05:02:40 .05:02:41 .^@^A^@^@
55 35132/13 if W8=. then W5 = . .^@^A^@^D .^@ .! .^P^@^A .^Pp^E^@^
.SYST^M^ .^G .^Ppd .st4 .restart .QUIT^M^ .+^F^A^B^
56 2398/1 if SA3=195 SA2=115 then SA1 = 218
```

57 35132/16 if W8=. then DA0 = 255 020 105 100 234 084 168 148 169 204 194 050 207 149 005 010

58 39406/19 if SA2=016 then DA0 = 255 020 105 100 234 084 168 148 169 204 194 050 207 149 005 010 087 044 201

59 27757/14 if DA0=100 then DUR = 0 5 11 7 12 1 10 6 9 4 3 8 2 16

60 14743/8 if DA1=112 W8=. then DA0 = 020 100 194 050 207 149 005 010

61 3521/2 if DA0=100 F2=.AP W6=.User-Age then W4 = .Referer: .Connecti

62 34887/20 if DUR=0 W8=. then W7 = . .^Ps .O^@^@^@^ .^@^@^@^@ .^@^H^T^@ .esp0:^ ./sbus@1f .Zr2r2r2^ .UUUUUU^@ .Zq2q2q2^ .iv^@^@^@ .Z^@^A^@^ .ZqTqTqT^ .uiciPy^@ .Zp^Tp^Tp .pleaP}^@ .Zqiqiqi^ .^H^@Py^@ .+^F^A^B^ .youPy^@^

63 25365/15 if W7=. then DP = 520 0 113 25 137 23 80 135 20 79 21 514 1023 22 1022

64 15051/9 if LEN=7 then DUR = 0 5 7 10 6 9 4 8 2

65 14826/9 if DA1=112 LEN=7 then DUR = 0 5 7 10 6 9 4 8 2

66 35132/30 if W8=. then W6 = . .^@^F`^@ .^Pp^E^@^ .L^@^H .O^@^@^@^ .N^@^H .W .^@^H .^Ppd .^PqT^@^@ .P^@^H .M^@^H .^@^@`^B^ .^@^H^T^@ .at .is .QUIT^M^ .y^@^H^I .UUUUUU^@ .I^@^H .;^@^H .Z^@^A^@^ .@^@^H .^@^H^W .+^@^H .^A^@^H .-^@^H .WP^X^P^@ .0P^P^P^@ .+^F^A^B^

67 28055/25 if F1=.UDP then W8 = . .^Ps .waiting .135.13.2 .reverse .authenti .for .768 .generati .196.227. .to .by .196.37.7 .197.182. .ON .8mm .0^ .driver .SMB%^@^@ .+^F^A^B^ .135.8.60 .194.7.24 .195.115. .197.218. .195.73.1

```
68 28055/34 if F1=.UDP then W6 = . .FAEDDACA .^@^A^@^D .^@^F`^@
.Connecti .SYSERR(l .SYSERR(g .SYSERR(m .SYSERR(r .^Ppd .Could
.Password .SYSERR(c .Generati .RSA .Timeout .SYSERR(e .SYSERR(w
.SYSERR(s .Closing .to .SYSERR(y .root' .SYSERR(b .4 .LOGIN
.SYSERR(j .SYSERR(h .<Exabyte .at .is .SYSERR(f .EIFFENEF .+^F^A^B^
69 34887/297 if DUR=0 W8=. then W3 = . .^@^A .^@^A^@^D .^AvL
.6667^M^ .^AvK .W .^AvJ .^@^@ .^AvI .^AvG .^AvF .^AvD .^AvC .^AvB
.^AvA .^Av@ .^Av? .^Av= .^Av< .^Av; .^Av: 23:00:01 .S^@^@ .U^@^@
.W^@^@ .Y^@^@ .\^@^@ ._^@^@
```

(remaining list of 297 values truncated)

A.2. Detected Attacks

This section lists the attacks detected at 100 false alarms by the rules listed in Section A.1. The format is as follows:

TP attack-name rule-number (percent contribution) A=v A?=V

TP indicates a true positive. The rule number (001-169) is the number listed in the first column of Appendix A.1. When more than one rule contributes to the anomaly, the rule that contributes the greatest fraction is shown. The percent contribution is the fraction of the anomaly score contributed by the rule shown. The remainder of the score is from other rules. The conditions of the form "A=v" are the rule antecedents. The consequent is shown by "A?=v" (with a question mark) where *v* is the anomalous value. For example, for the first detection below (of *syslogd*), rule 1 contributes

65.56% of the anomaly score. The rule is "if SA3=172 then SA2 = 016". The anomaly is "SA2 = 005".

```
TP syslogd       001 (65.56) SA3=172 SA2?=005
TP syslogd       001 (93.04) SA3=172 SA2?=005
TP syslogd       001 (96.79) SA3=172 SA2?=003
TP syslogd       001 (98.07) SA3=172 SA2?=005
TP portsweep     007 (99.98) DA1?=118 DA0=100
TP ncftp         012 (40.58) DA1=118 W1?=.
TP ncftp         012 (40.83) DA1=118 W1?=.
TP guesstelnet   012 (45.3) DA1=118 W1?=.^@
TP ncftp         012 (45.66) DA1=118 W1?=.
TP ncftp         012 (50) DA1=118 W1?=.
TP mscan         015 (100) SP=520 W2?=.
TP neptune       016 (100) DA1?=112 DP=520
TP neptune       016 (100) DA1?=114 DP=520
TP eject         018 (55.67) F1?=.AP F2=.AP
TP apache2       018 (57.18) F1?=.AP F2=.AP
TP tcpreset      021 (66.73) F1?=.AP
TP netbus        026 (100) F2=.AP F3?=.S
TP satan         027 (27.65) SA1?=070 F1=.UDP
TP portsweep     027 (59.97) SA1?=124 F1=.UDP
TP portsweep     027 (73.62) SA1?=008 F1=.UDP
TP portsweep     029 (25.52) SA3?=204
TP yaga          029 (30.69) SA3?=206
TP portsweep     029 (30.91) SA3?=202
```

```
TP portsweep     029 (34.04) SA3?=209
TP guest         029 (36.29) SA3?=153
TP guessftp      029 (36.45) SA3?=208
TP ipsweep       029 (40.7) SA3?=204
TP smurf         029 (42.56) SA3?=023
TP neptune       029 (48.42) SA3?=011
TP portsweep     029 (48.55) SA3?=153
TP anypw         029 (48.55) SA3?=204
TP casesen       029 (48.55) SA3?=204
TP ffbconfig     029 (48.55) SA3?=206
TP netcat_setup  029 (48.55) SA3?=207
TP perl          029 (48.55) SA3?=209
TP guesstelnet   029 (48.55) SA3?=209
TP smurf         029 (51.32) SA3?=001
TP ps            029 (53.16) SA3?=209
TP tcpreset      029 (58.16) SA3?=202
TP xterm         029 (95.59) SA3?=202
TP ntinfoscan    030 (37.21) F3?=.AR
TP back          030 (45.85) F3?=.A
TP portsweep     030 (46.38) F3?=.F
TP portsweep     030 (51.68) F3?=.F
TP queso         030 (58.63) F3?=.F
TP portsweep     030 (62.84) F3?=.F
TP dosnuke       031 (32.53) F2?=.UAP
TP queso         031 (99.97) F2?=.10S
TP dosnuke       032 (45.31) F1=.S F2?=.UAP
```

```
TP dosnuke       032 (52.32) F1=.S F2?=.UAP
TP sshprocesstab 033 (45.36) SA2=016 SA1?=118
TP dict          033 (71.54) SA2=016 SA1?=118
TP crashiis      033 (83.71) SA2=016 SA1?=117
TP resetscan     033 (88.58) SA2=016 SA1?=117
TP imap          033 (94.75) SA2=016 SA1?=117
TP processtable  033 (99.09) SA2=016 SA1?=118
TP processtable  033 (99.77) SA2=016 SA1?=117
TP crashiis      035 (100) W1=.^@GET W3?=.
TP crashiis      035 (100) W1=.^@GET W3?=.
TP crashiis      035 (47.82) W1=.^@GET W3?=.
TP phf           035 (50.79) W1=.^@GET W3?=.
TP phf           035 (88.39) W1=.^@GET W3?=.
TP crashiis      035 (96.14) W1=.^@GET W3?=.
TP back          035 (99.82) W1=.^@GET W3?=.
TP phf           036 (58.33) LEN?=6 W3=.
TP back          036 (95.98) LEN?=14 W3=.
TP dosnuke       037 (35.13) SA2=016 DUR=0 F2?=.UAP W8=.
TP queso         037 (99.21) SA2=016 DUR=0 F2?=.10S W8=.
TP udpstorm      038 (43.81) DP?=7 F1=.UDP
TP udpstorm      038 (65.85) DP?=7 F1=.UDP
TP guesstelnet   039 (58.06) SA2?=005
TP apache2       042 (100) F2=. LEN?=11
TP satan         044 (100) DP=25 W1?=.^@
TP mailbomb      044 (43.53) DP=25 W1?=.^@mail
TP mailbomb      044 (53.06) DP=25 W1?=.^@mail
```

```
TP sendmail      044 (84.99) DP=25 W1?=.^@MAIL
TP netcat_breaki 045 (25.35) W4?=.ver^ W6=.
TP guesspop      045 (32.01) W4?=.alie0^M^ W6=.
TP ftpwrite      045 (39.51) W4?=./etc/hos W6=.
TP apache2       045 (51.37) W4?=.User-Age W6=.
TP guessftp      045 (52.97) W4?=.rexn0^M^ W6=.
TP netbus        047 (100) LEN?=19
TP warez         047 (82.74) LEN?=19
TP netbus        047 (99.98) LEN?=19
TP ntfsdos       048 (100) DUR?=8 W8=.
TP ntfsdos       048 (100) DUR?=8 W8=.
TP teardrop      048 (100) DUR?=8 W8=.
TP arppoison     048 (100) DUR?=9 W8=.
TP ntinfoscan    048 (44.76) DUR?=10 W8=.
TP back          048 (87.37) DUR?=9 W8=.
TP crashiis      048 (99.99) DUR?=15 W8=.
TP named         050 (29.27) DP?=53 DUR=0
TP portsweep     050 (40.72) DP?=143 DUR=0
TP named         050 (43.44) DP?=53 DUR=0
TP ls_domain     050 (45.38) DP?=53 DUR=0
TP named         050 (52.19) DP?=53 DUR=0
TP portsweep     051 (30.6) DP?=19
TP ls_domain     051 (57.85) DP?=53
TP ftpwrite      051 (63.05) DP?=513
TP mailbomb      053 (85.55) DP=25 W3?=.rcpt
TP sendmail      053 (88.41) DP=25 W3?=.root@cal
```

```
TP insidesniffer 055 (59.26) W5?=.RCPT W8=.
TP smurf         055 (28.83) W5?=.^F W8=.
TP smurf         055 (38.18) W5?=.^H^@E^@^ W8=.
TP ipsweep       055 (65.77) W5?=.V W8=.
TP ipsweep       055 (68.62) W5?=.^Pp2^H^@ W8=.
TP casesen       059 (100) DA0=100 DUR?=13
TP guest         064 (44.49) DUR?=3 LEN=7
TP secret        064 (50.38) DUR?=11 LEN=7
TP guest         065 (90) DA1=112 DUR?=3 LEN=7
```

A.3. Top Scoring Alarms

This section shows the top scoring alarms in the same format as Appendix A.2, sorted by decreasing anomaly score. The first column indicates a true positive (TP), a false positive (FP), or a duplicate detection of an attack detected by a higher scoring alarm (--). An IDS is evaluated by the number of TP up to the first 100 FP, ignoring duplicates. The highest scoring alarm is a detection of *portsweep* in which 99.98% of the anomaly score is from rule 7 in Appendix A.1. Only alarms through the first 10 false alarms are shown.

```
TP portsweep     007 (99.98) DA1?=118 DA0=100
TP syslogd       001 (98.07) SA3=172 SA2?=005
-- portsweep     012 (50) DA1=118 W1?=.
TP syslogd       001 (96.79) SA3=172 SA2?=003
TP ncftp         012 (40.58) DA1=118 W1?=.
FP               017 (77.55) DA1=118 SA1=112 LEN?=6
-- portsweep     020 (100) DA1?=117
```

```
TP neptune       016 (100) DA1?=112 DP=520
TP mscan         015 (100) SP=520 W2?=.
TP syslogd       001 (93.04) SA3=172 SA2?=005
TP apache2       018 (57.18) F1?=.AP F2=.AP
FP               026 (99.83) F2=.AP F3?=.AR
TP neptune       016 (100) DA1?=114 DP=520
FP               017 (75.74) DA1=118 SA1=112 LEN?=6
TP tcpreset      021 (66.73) F1?=.AP
FP               029 (48.55) SA3?=206
-- tcpreset      018 (100) F1?=.AP F2=.AP
TP ncftp         012 (45.66) DA1=118 W1?=.
FP               030 (63.08) F3?=.F
TP phf           035 (50.79) W1=.^@GET W3?=.
TP queso         030 (58.63) F3?=.F
TP dosnuke       032 (52.32) F1=.S F2?=.UAP
-- neptune       028 (100) F2?=.A F3=.AF
FP               029 (61.83) SA3?=206
-- apache2       034 (54.65) DP=80 W1?=.^@^@^@^@
TP queso         031 (99.97) F2?=.10S
TP satan         044 (100) DP=25 W1?=.^@
TP satan         027 (27.65) SA1?=070 F1=.UDP
TP sendmail      053 (88.41) DP=25 W3?=.root@cal
TP netbus        047 (100) LEN?=19
FP               017 (76.76) DA1=118 SA1=112 LEN?=6
FP               015 (100) SP=520 W2?=.
-- ncftp         033 (63.26) SA2=016 SA1?=118
```

```
TP processtable  033 (99.77) SA2=016 SA1?=117
-- queso         031 (100) F2?=.10S
-- portsweep     052 (62.3) DA0?=030
TP queso         037 (99.21) SA2=016 DUR=0 F2?=.10S W8=.
TP dosnuke       037 (35.13) SA2=016 DUR=0 F2?=.UAP W8=.
FP               020 (81.53) DA1?=117
FP               021 (45.72) F1?=.AR
```