Learning Rules for Anomaly Detection of Hostile Network Traffic

Matthew V. Mahoney and Philip K. Chan
Department of Computer Sciences
Florida Institute of Technology
Melbourne, FL 32901
{mmahoney,pkc}@cs.fit.edu
TR CS-2003-16

Abstract

We introduce an algorithm called LERAD that learns rules for finding rare events in nominal time-series data with long range dependencies. We use LERAD to find anomalies in network packets and TCP sessions to detect novel intrusions. LERAD outperforms the original participants in the 1999 DARPA/Lincoln Laboratory intrusion detection evaluation, and detected most attacks that eluded a firewall in a university departmental server environment.

1. Introduction

An important component of computer security is intrusion detection--knowing whether a system has been compromised or if an attack is occurring. Hostile activity can sometimes be inferred by examining inbound network traffic, operating system events, or changes to the file system, either for patterns signaling known attacks (signature detection), or for unusual events signaling possible novel attacks (anomaly detection). Anomaly detection has the advantage that it can sometimes detect previously unknown attacks, but has the disadvantage that it issues false alarms, because unusual events are not always hostile. Often both approaches are used. For example, a virus detector might scan files for strings signaling known viruses, and might also test for modifications of executable files as indications of possible new viruses.

Network anomaly detection is a particularly difficult problem because higher level (application) protocols are complex and difficult to model, and because data must be processed at high speed. A common approach is to use a firewall with rules programmed by a network administrator to block and/or log packets based on lower level features such as IP addresses and port numbers. This technique can detect or block port scans and unauthorized access to private services (e.g. *ssh*) from untrusted clients. However, detection of attacks on public services such as HTTP (web), SMTP (email), and DNS (host name lookup) currently uses a signature detection system such as SNORT (Roesch, 1999), or Bro (Paxson, 1998) to scan for strings signaling known attacks. The number of rules needed is quite large (SNORT has over 1800) and must be updated frequently. This would not be an effective defense against novel attacks or fast spreading worms such as Sapphire/Slammer, which compromised nearly every vulnerable system worldwide within 30 minutes of its release, its population doubling every 8.5 seconds (Moore et. al., 2003). Network anomaly detection systems such as ADAM (Barbara et. al., 2001a, 2001b), SPADE (Hoagland, 2000), and eBayes (Valdes & Skinner, 2000), use machine learning approaches to model normal network traffic in order to identify unusual events as suspicious, but they do not model application protocols.

We introduce an efficient, randomized algorithm called LERAD (Learning Rules for Anomaly Detection), which can discover syntactic relationships among attributes in order to model application protocols. LERAD differs from association mining approaches such as APRIORI (Agrawal & Srikant, 1994) in that it finds rules with a small set of allowed values in the consequent, rather than a distribution with high confidence or low entropy. We believe this form is more appropriate for "bursty" time series data with long range dependencies, a characteristic of network traffic (Leland, et. al., 1993; Paxson & Floyd, 1995). LERAD outperforms the original participants in the 1999 DARPA/Lincoln Laboratory intrusion detection evaluation, and detected attacks missed by both a firewall and SNORT in a university departmental server environment.

The rest of this paper is organized as follows. In Section 2, we discuss related work in anomaly detection. In Section 3, we describe the LERAD algorithm. In Section 4 we describe experimental results using two attribute sets (packets and TCP sessions) on two data sets (one simulated and one real). In Section 5 we conclude.

2. Related Work

Although an anomaly detection model can be coded by hand (e.g., a firewall), it can be learned from normal (presumably attack-free) data. For example, NIDES compares short term and long term distributions of system performance measures, such as CPU time and number of open files (Anderson et. al., 1999). Anomalies in UNIX system call sequences can signal when a server or operating system component has been compromised. Call sequences have been modeled using n-grams (Forrest et. al., 1996) and neural networks (Ghosh & Schwartzbard, 1999).

Network based anomaly detectors generally model low level attributes. SPADE uses a joint probability model based on counting TCP client address/port combinations, assigning high anomaly scores to rare combinations. ADAM uses market basket analysis to learn associations among addresses, subnets, ports, day of the week and hour of the day, then passes low-probability events to a decision tree classifier trained on labeled attacks for classification as normal, known attack or unknown attack. eBayes models short-term event rates (e.g. ICMP error intensity) in addition to ports and addresses, using a naive Bayes classifier with mechanisms to adjust categories and add new categories. Events not easily categorized raise an alarm.

ALAD (Mahoney & Chan, 2002b) models application protocols as an allowed set of header keywords conditioned on server address and port number. For example, the allowed set of HTTP keywords might be the set GET, POST, Accept, Host, User-Agent, etc. A shortcoming of ALAD is that it cannot discover new relations among attributes, for example, the conditions under which the host field would contain a predictable value.

Besides association rules (Agrawal & Srikant, 1994) do not allow a set of values of in the consequent, their algorithms usually generate all rules that are above a user-specified confidence and support. The resulting large rule sets incur unacceptable overhead in our domain where large amounts of data are monitored. Classification rules (Cohen, 1995) algorithms require a certain attribute to represent the class labels, which are not available in our domain.

3. Rule Learning Algorithm

The goal of LERAD is to find conditional rules that identify unexpected events in a time-series of tuples of nominal (unordered) attributes (e.g. packet field values, or words in a TCP session). The time series exhibits long range dependency: given two tuples, the number of matching attribute values decreases as the time interval between the tuples increases. (We have observed this behavior in network packets over time scales from milliseconds to weeks, and it is consistent with fractal, "bursty" models described by Paxson and Floyd (1995) and Leland et. al.

(1993)). Thus, by "unexpected", we mean that an event has not occurred for a long time, independent of its average rate (which cannot be measured reliably).

The LERAD algorithm finds conditional rules of the form "if $A_1 = x_1$ and $A_2 = x_2$ and ... $A_m = x_m$ then $A_{m+1} \in X = \{x_{m+1}, x_{m+2}, ..., x_{m+r}\}$", where the A_i are nominal attributes, x_i are values, and $m \geq 0$. The set X consists of all values of A_{m+1} observed at least once among the n training instances that satisfy the antecedent. At the end of training, we fix X and n. During testing, if an instance satisfies the antecedent but A_{m+1} is not in the set X of allowed values, we generate an anomaly score of tn/r, where t is the time since the rule was last violated, n is the support, and $r = |X|$, the number of allowed values. Otherwise the score is 0. The anomaly score for the test instance is $\Sigma\, tn/r$ where the summation is over all rules. The score is used to rank alarms, with higher values indicating a greater probability of hostility.

The anomaly score $tn/r = (1/t)(r/n)$ is the inverse product of two probabilities, and is also used in ALAD. The factor $1/t$ is the short term average rate, going back only to the last event (which may have been during training). This factor is useful for eliminating bursts of alarms, since only the first alarm in a burst will have a high score. The factor r/n is the average rate of "anomalies" in training, i.e. the fraction of tuples where the observed value was seen for the first time. It is equivalent to the PPMC method of estimating the probability of novel events for data compression models (Bell, Witten, & Cleary, 1989). Thus, a high n/r indicates a rule which is unlikely to be violated in testing.

The number of possible rules is huge. Given m attributes with k values each, there are potentially $m(m-1)^{k+1}$ rules. For example, in one version of LERAD, we use $m = 32$ packet byte pairs with $k = 2^{16}$ values each, for over 10^{97740} possible rules. To cope with this complexity, LERAD uses a randomized sampling approach. First, it samples pairs of tuples with one or more matching attributes to suggest rules that satisfy both tuples. Then, working with a small sample, S, of the training data, it removes "redundant" rules, keeping just enough rules to cover the values in S without duplication (and favoring rules with higher n/r). Next, it trains the rules on the full training set, fixing n and r.

Finally, LERAD applies a validation step, removing rules that generate anomalies on a separate validation set, V (for example, this could be the last 10% of the training data). Validation favors "well behaved" rules, where the set of allowed values is learned quickly and then does not change, over "poorly behaved" rules, where r grows steadily over time, indicating that future anomalies are likely (Figure 1). For example, we have observed that the set of client IP addresses exhibit "poor" behavior, such that r depends on the length of the training period and the complete set is never learned. If we did not remove this rule, then we would continue to observe new values in testing and generate (probably false) alarms.

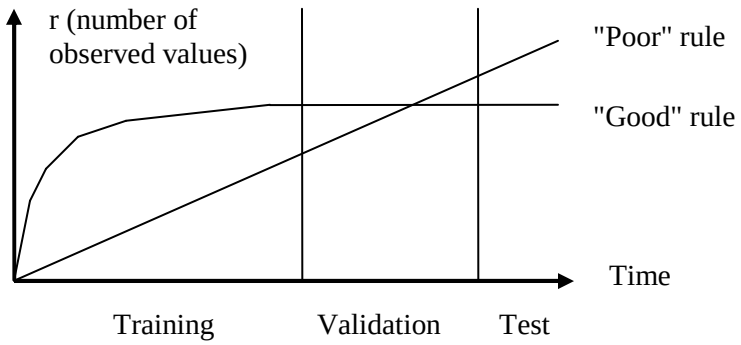

Figure 1. Growth of r for "good" and "poor" rules. The poor rule would generate anomalies during validation and be removed.

LERAD requires two passes, one to sample *S* from the training data to generate rules, and the second for training, validation, and testing. We cannot just draw *S* from the beginning of the training data because we have assumed long range dependencies, so the beginning would not be representative of the rest of the data. The steps are summarized here and explained in detail below.

1. (Rule generation). Randomly sample *L* pairs of training instances from *S*, and generate up to *M* rules per pair that satisfy both instances with n/r = 2/1, generating rule set *R*.
2. (Coverage test). Discard rules from *R* to find a minimal (but not optimal) subset of rules that cover all instance-values in *S*, favoring rules with higher n/r.
3. (Second pass). Train on the full training set, expanding the allowed sets *X* (and recomputing *n* and *r*) for each rule.
4 (Validation test). Discard rules that generate anomalies (values not in *X*) on a validation set, *V*.

3.1. Rule Generation

If the training data were random and we sampled two tuples, it is unlikely that we would find any matching attributes between them. Any matches suggest a regularity to the data. In the rule generation step, we randomly sample pairs of instances from the training set and generate rules that satisfy both instances based on matching attribute values. We randomly choose one of the matching attributes to be the consequent, and a subset of the remaining matching attributes to be conditions in the antecedent. For example, suppose that our sampled pair is shown in Table 1.

Port	Word1	Word2	Word3
80	GET	/	HTTP/1.0
80	GET	/index.html	HTTP/1.0

Table 1. Two example training instances

In Table 1, we see that port, word1, and word3 are matching attributes. Some rules suggested by this pair might be as follows:

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

In general, if there are *m* matching attributes, then there are $m2^{m-1}$ possible rules. However, in our tests, we find that rules with small numbers of attributes (0 to 2) seem to work well for anomaly detection. In our implementation, we generate *m* rules with 0, 1, 2, ... *m* – 1 terms in the antecedent, as in this example with *m* = 3. Furthermore, we place a small upper bound on *m* of *M* = 4 to avoid rules with large numbers of terms.

3.2. Coverage Test

The rule generation step creates a large number of rules, many of which are not needed. For example, we do not need both the rules "if port = 80 then word3 = HTTP/1.0" and "if word1 = GET then word3 = HTTP/1.0" since either one will generate an alarm if *word3* is different (and nothing else). The extra rule causes two problems. First, it takes twice as long to test the tuple. Second, the anomaly score would be twice as high for this value, not because it is twice as unlikely, but because there were two rules for it. Thus, we remove the extra rule. We keep the "better" rule, the one with higher n/r (as estimated on *S*). The algorithm is as follows:

Train the rule set R (from step 1) on S.
Sort R by decreasing n/r.
For each rule R_i in R
    If R_i does not cover any unmarked values, then remove it from R.
    Mark the unmarked values in S covered by R_i

A rule is said to *cover* a value of an instance if that instance satisfies the antecedent and the value satisfies the consequent. As an example, suppose that S is as shown in Table 2 and we have the following rule set R sorted by n/r on S. The notation (R_i) means that rule R_i is the first rule to mark the value.

R_1: if port = 80 then word1 = GET (n/r = 2/1) (marks 2 values)
R_2: word1 = GET or HELO (n/r = 3/2) (marks 1 new value)
R_3: if word2 = /index.html then word1 = GET (n/r = 1/1) (marks 0 new values, removed)
R_4: word2 = /, /index.html, or pascal (n/r = 3/3) (marks 3 values)

Port	**Word1**	**Word2**	**Word3**
80	GET (R_1)	/ (R_4)	HTTP/1.0
80	GET (R_1)	/index.html (R_4)	HTTP/1.0
25	HELO (R_2)	pascal (R_4)	

Table 2. Sample *S*

In this example, we remove rule R_3 because no new marks could be added. The only value it covers (the second GET) was already marked by R_1 (and would have been marked by R_2 as well).

3.3. Second Pass and Validation Step

After we generate the rule set R, we "reset" the rules, then read the training, validation, and test data. During training, we learn the set X for each rule. During validation, we continue training, but discard any rule for which $|X| = r$ grows. During testing, we first fix X, n and r, then if we observe any value not in X we generate an anomaly score of tn/r. For example, suppose our data is as shown in Table 3 and we had the following rules in R from pass 1:

- R_1: if port = 80 then word1 = ...
- R_2: if port = 25 then word1 = ...

Phase	**Time**	**Port**	**Word 1**	**Rule R_1**	**Rule R_2**
Training	1	80	GET	n/r = 1/1	
	2	80	GET	n/r = 2/1	
	3	25	HELO		n/r = 1/1
Validation	4	25	MAIL		Remove rule
Test	5	80	GET		
	6	80	POST	tn/r = 5(2/1)	
	7	80	GET		
	8	80	POST	tn/r = 2(2/1)	
	9	80	HEAD	tn/r = 1(2/1)	

Table 3. Example training, validation, and test sets

During training, we fill in the consequent values:

- R_1: if port = 80 then word1 = GET (*n/r* = 2/1)
- R_2: if port = 25 then word1 = HELO (*n/r* = 1/1)

During validation, rule R_2 is violated, so we remove it. During testing, R_1 is violated at times 6, 8, and 9, generating anomaly scores of *tn/r*. The value of *n/r* is fixed at 2/1 before the start of testing. For the first value of *t*, we use the last "anomaly" in training, which occurred at time 1 when the value "GET" was added to the allowed set. Thus, the values of *t* are 5, 2, and 1.

4. Experimental Evaluation

4.1. Evaluation Data

We tested LERAD on two data sets, one synthetic and one real. The first is the 1999 DARPA/Lincoln Laboratory intrusion detection evaluation (IDEVAL) data set, a widely used benchmark using synthetic network traffic (Lippmann et. al., 2000a, 2000b). The IDEVAL data set simulates a typical Air Force network with four "victim" hosts (SunOS, Solaris, Linux, and Windows NT) under attack by 201 instances of 58 attacks over a two week test period mostly drawn from published sources such as the Bugtraq mailing list. In addition, three weeks of training data were provided, both attack-free and with labeled attacks. The data includes sniffed traffic from inside and outside the simulated Internet gateway, Solaris system call logs (BSM), and audit logs and file system dumps from all hosts. We used only the inside sniffer traffic, training LERAD on week 3 (7 days x 22 hours, 2.9 GB of tcpdump files), which contains no attacks, and testing on weeks 4 and 5 (9 days x 22 hours, 6.0 GB).

Because of questions about the accuracy of the IDEVAL simulation (McHugh, 2000; Mahoney & Chan, 2003), we also tested LERAD on 623 hours of traffic sniffed from a university departmental server containing 19 instances of six attacks which we previously identified and labeled. The host is a Sun Ultra-AX i2 running Solaris 5.9 as a web server (over 20,000 pages), hosting several faculty shell accounts and supporting *ssh* (but not *telnet*), FTP, SMTP, IMAP, NFS, RPC, and a printer. Unlike IDEVAL, the host is switched, so only traffic to and from it is visible. Also, the local university network is protected by a gateway firewall. Over 24,000 different client IP addresses were observed, compared to only 29 in the IDEVAL training data.

We collected traffic over 10 weeks (Monday through Friday) from Sept. 30 through Oct. 25 and Nov. 4 through Dec. 13, 2002. Each of the 50 daily traces started at 12:01 AM local time until 2 million packets are collected, usually after about 10 to 15 hours. To reduce the volume of data, packets were truncated to 200 bytes (including Ethernet headers) and filtered with TF, resulting in ten one-week traces totaling 1,663,603 packets (98.4% reduction), in 183 MB of tcpdump files. This required a slight modification to the TCP reassembly algorithm: we stopped after the first TCP packet with a payload (to eliminate gaps caused by truncation) and did not include the payload or TCP flag attributes from subsequent packets.

We used SNORT 1.9.1 (with rules postdating the traffic) and manual inspection to identify attacks: sorting the packets by port and payload, and entering outliers into a search engine, a surprisingly effective technique. SNORT detected a port/security scan from inside the firewall, an external HTTP proxy scan, an external DNS version probe, and one of five instance of the Nimda HTTP worm (CIAC, 2001). By manual inspection we identified the other Nimda instances, ten instances of the Code Red II HTTP worm (Moore, Shannon, & Brown, 2002), and the Scalper worm (CERT, 2002). The Nimda and Code Red II probes were both internal and external. One internal PC was apparently infected by both worms for three days and probed the target about once per hour. The port/security scan is probably the most malicious. It has two parts; first an attempt to retrieve the password file by a *cgi-bin/htsearch* exploit, followed 9 minutes later by a 29 second port scan, with open ports probed further to test for vulnerabilities.

After labeling the attacks, we tested LERAD on weeks 2 through 10, in each case using the previous week as training. By chance, there are no known attacks in week 1. However, there

are generally attacks in the training data which could mask detections in the test data. Code Red and Nimda attacks are spread over several weeks.

4.2. LERAD Configuration

We used two sets of attributes for LERAD: IP packets (LERAD-PKT) and TCP streams (LERAD-TCP). In both cases, we restrict our data set to the first few unsolicited (i.e. client to server) inbound packets in each session. Also, we used the same parameters, $|S| = 100$ samples, $L = 1000$ training pairs to generate candidate rules, a maximum of $M = 4$ matching attributes per sample pair, and a validation set V consisting of the last 10% of the training data, values which work well in our experience. We used tuple count to compute t rather than real time in order to avoid the effects of gaps in the data. Source code for LERAD-PKT and LERAD-TCP are available at (Mahoney, 2003b).

LERAD-PKT uses as attributes the first 32 pairs of bytes in each IP packet--we divide the Ethernet payload into 32 segments of 16 bits each, with no further parsing of the input. Each attribute has a nominal value of 0 to $2^{16} - 1 = 65{,}535$, or *undefined* if the byte pair extends beyond the end of the Ethernet payload. Undefined values are allowed in the consequent of a rule but not the antecedent, e.g. "pair23 = 0, 1, or *undefined*", but not "if pair17 = *undefined*...".

The packets input to LERAD are filtered using the TF filter stage of NETAD (Mahoney, 2003a, 2003b). This filter restricts the input to inbound client to server IP packets (determined by the SYN flag for TCP or port number < 1024 for UDP). Furthermore, packets are rate limited to the first 100 payload bytes per session for TCP (plus the remainder of the last packet), and also to 16 packets per minute per session (determined by source and destination addresses and ports for TCP and UDP). ICMP is similarly rate limited. The filter has the effect of removing 98-99% of traffic while still passing evidence for most attacks. To further speed processing, we discard rules with $r > 32$, because rules with high r do not contribute much to the anomaly score.

LERAD-TCP reads attributes of the inbound side of unsolicited (client to server) reassembled TCP sessions. There are 23 attributes: date, time (in integer seconds), source IP address (4 bytes as separate attributes), destination IP address (last 2 bytes; first two are fixed), source and destination port numbers, payload length (in bytes), duration (in integer seconds), TCP flags of the first, next to last, and last packets (as three attributes), and first eight words of the payload. Words are delimited by spaces or linefeeds, and truncated to eight bytes. If there are fewer than eight words, then the remainder are set to empty strings (not undefined). Although the date and time are probably not useful for anomaly detection, we include them for convenience of implementation and to test the robustness of the algorithm.

We used AFIL.PL (Mahoney, 2003b) to consolidate alarms that identify the same target within a 60 second window by keeping only the highest scoring alarm. This postprocessing nearly always reduces the number of false alarms in any system without sacrificing detection.

4.3. Evaluation Criteria

We evaluated LERAD on IDEVAL according to the rules of the 1999 evaluation as implemented by EVAL (Mahoney, 2003b). The system must provide a list of alarms with the time, target IP address and a numeric score for ranking alarms. A system includes a specification of the attacks it is designed to detect, based on its category and the data examined. An attack is counted as detected if it is "in-spec" and one or more alarms identifies the target address within 60 seconds of any portion of the attack. Out of spec detections are ignored. Any other alarm is a false alarm. For LERAD, we count as in-spec the 148 probe, DOS (denial of service), and R2L (remote to local) attacks for which there is evidence in the inside sniffer traffic according to the truth labels. We exclude U2R (user to root) and data attacks, in which the attacker already has local access, because such attacks are more appropriately detected by a host based system. Such

attacks are difficult to detect in network shell sessions or file uploads, and impossible if the traffic is encrypted.

One can trade off between a high detection rate and a low false alarm rate by discarding alarms with scores below a threshold. In 1999, the top four of the 18 original participants detected 40% to 55% of in-spec attacks at a threshold allowing 100 false alarms, or 10 per day (Table 4).

System	In-Spec Attacks	Detected
Expert 1	169	85 (50%)
Expert 2	173	81 (47%)
Dmine	102	41 (40%)
Forensics	27	15 (55%)

Table 4. Attacks detected by the top four systems in the 1999 IDEVAL evaluation at 10 false alarms per day (Lippmann et. al., 2000b)

For the university traffic, we use the stricter criteria that LERAD must exactly identify at least one of the packets or TCP sessions involved in the attack. However we do not distinguish between instances of an attack, because an anomaly detection system is most likely to be used in combination with signature detection, which is more reliable for known attacks. Therefore we consider it sufficient to identify one instance (most likely the first one), after which we would presumably add an appropriate rule to detect future instances. Thus, we count 6 attacks.

We evaluated run time performance on a 750 MHz Duron under Windows Me. Run times do not including packet filtering or TCP reassembly, which are limited solely by disk speed (7 minutes on IDEVAL). LERAD-PKT and LERAD-TCP are about 500 lines of C++ each. Open source code (including filtering and TCP reassembly) is available at (Mahoney, 2003b).

4.4. Results on IDEVAL

We ran LERAD-PKT and LERAD-TCP five times each with different random number seeds and averaged the results. After filtering with TF, our implementation of LERAD-PKT processes 362,934 IDEVAL training packets and 738,719 test packets in 110 seconds, about 10,000 packets per second. In a typical run it generates 1500-1800 candidate rules, reduces this to 200-210 after the coverage test and 80-90 after validation. At 100 false alarms it detects an average of 48.2 (range 40 to 52) of 148 in-spec attacks, or 33%.

After TCP reassembly, LERAD-TCP processes 35,455 training sessions and 178,099 test sessions in 60 seconds (3500 sessions per second). In a typical run it generates 1100-1200 candidate rules, reduced to 80-100 after the coverage test, and 55-70 after validation. At 100 false alarms it detects an average of 95.2 (range 92 to 100) of 148 in-spec attacks, or 64%. Although this compares favorably with the original 1999 evaluation, even without signature detection, we caution that a comparison would be biased because we had access to the labeled test data during development.

In about half of the detections, LERAD detects anomalies in the server port number (e.g. a scan) or in the exploited protocol, often at the application layer. For example, LERAD-TCP detects *sendmail*, an SMTP buffer overflow, because the session starts with "MAIL" rather than the usual "HELO" or "EHLO". Although this is legal, it differs from normal client behavior. LERAD-PKT detects *teardrop* and *ping of death*, two DOS attacks that exploit IP fragmentation reassembly bugs, by the presence of IP fragments, a legal but seldom used feature of the IP protocol. We believe that vulnerabilities are found mostly in rarely used features because they receive the least field testing.

The other half of the detections appear to be due to simulation artifacts. Because of the small number of client IP addresses in training (29), LERAD can detect many attacks on public services (HTTP, SMTP, and DNS) by source address anomalies. Many unrelated attacks are also detected by TTL, TCP option, and TCP window size anomalies, which we believe are due to idiosyncrasies in the machines used to simulate the attacks.

4.5. Results on University Server Traffic

We tested LERAD-PKT and LERAD-TCP five times each with different random number seeds on weeks 2 through 10, using the previous week as training for the current week. The results are shown as a detection-false alarm graph in Figure 2. At 10 false alarms per day per detector, LERAD-PKT detects an average of 1.4 attacks and LERAD-TCP detects 2.4. Combining the output of both systems (and therefore doubling the false alarms) would detect 3 of 6 attacks (50%) because of overlapping detection of *scan*. The number of detections can be increased to 3.8 by allowing 20 false alarms per day per detector.

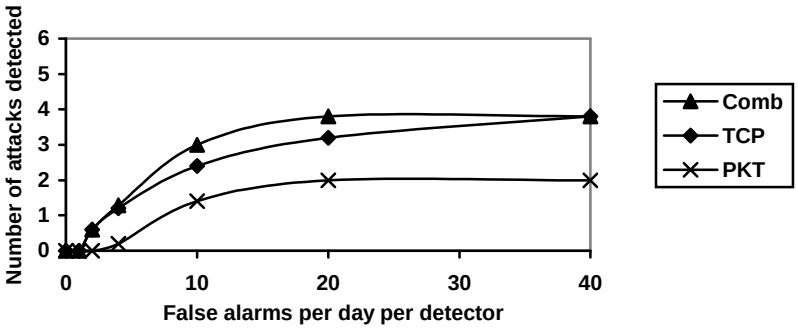

Figure 2. Average number of attacks detected by LERAD-TCP, LERAD-PKT, or a combined system (Comb) at 0 to 40 false alarms per detector per 24 hours.

Table 5 breaks down the detection probability by attack. We rank the attacks (somewhat subjectively) by decreasing maliciousness. For example, we ranked Code Red II higher than the other worms because it drops a backdoor allowing remote access via a web browser (and by Nimda). We consider the port/security scan to be the most dangerous because it tests for a large number of vulnerabilities and is likely to result in a compromise. As the table shows, the more malicious attacks tend to have a higher probability of detection.

Attack	**PKT**	**TCP**	**Combined**
Inside port/security scan	0.8	1.0	1.0
Code Red II worm	0.6		0.6
Nimda worm		0.4	0.4
Scalper worm		0.8	0.8
HTTP proxy probe		0.2	0.2
DNS version probe			
Average number of detections (out of 6)	**1.4**	**2.4**	**3.0**

Table 5. Estimated probability of detection by LERAD-PKT, LERAD-TCP, and their combination in university server traffic at 10 false alarms per day per detector

Most of the attacks are detected by anomalies in application layer protocols. Three attacks (*Nimda*, *Scalper*, and *proxy*) are detected by anomalies in the HTTP *host* field. LERAD-

TCP learns a rule similar to "if the destination is the server and word4 = *Host*: then word5 = (server name or IP address)". *Nimda* and *Scalper* fill in generic values (*www* or *Unknown*), and *proxy* fills in *www.yahoo.com*.

LERAD-PKT detects Code Red II by an unusual TCP segmentation of the HTTP request "*GET /default.ida?NNNNNN"* (with executable code following the long string of N's). Normally this would appear in a single TCP packet but the worm segments this into two packets following "GET ". One of the anomalies in the port/security scan is the unusual HTTP probe, "*GET / HTTP\1.0*" (with a backslash).

5. Concluding Remarks

The traditional approach to network anomaly detection has been to learn a "user" model, similar to a firewall, in which each server is restricted to a set of trusted clients (possibly none) based on past usage. However, this approach does not detect attacks on unrestricted services (such as IP or HTTP) that exploit flaws in the server's implementation of the protocol. We can sometimes detect these attacks because the attacking client's implementation of the protocol is somehow unusual. The difference may be that it uses a rare (and therefore poorly tested) feature (e.g. IP fragmentation), or the difference is idiosyncratic (e.g. the unusual but legal TCP segmentation in Code Red II), or that it is too much work (and unnecessary) to get it right (e.g. the *Host* anomalies), or simply a bug (e.g. "*HTTP\1.0*"). Although many of these attacks could easily be modified to elude detection, anomalies due to carelessness are nevertheless common.

Network protocols are complex, so we use association mining to discover syntactic relationships between attributes. We are interested in rare events, so instead of searching for conditional rules with high support and confidence, we search for rules with high support and a small and stable set of allowed values in the consequent without regard to their distribution. Our proposed LERAD is efficient for three reasons. First, we examine only a small fraction of the traffic. Second, we generate rules using only a small sample of the training data. Third, we use a coverage test to build a small set of rules that sufficiently covers the data.

Anomaly detection of any kind suffers from the problem of false alarms because unusual events are not necessarily hostile. In LERAD, there is no obvious way to distinguish true and false alarms. Those attributes and rules that detect the most attacks are the same ones that generate most of the false alarms. Often an anomaly is unrelated to the mode of attack and sheds little light on the nature of the vulnerability, even to an expert in network protocols. While these problems are difficult, we believe that the false alarm rate can be reduced through better modeling, for example, adding attributes to represent statistical properties such as packet rate. We are currently researching better tokenization algorithms for the application payload (as opposed to using white space) in order to extend LERAD to binary protocols. We are also investigating single-pass versions of LERAD in which rules are added, updated, applied to testing, and removed continuously.

Acknowledgments

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

References

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.

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.

CERT (2002), "MA-044.072002 : Apache Worm", http://www.mycert.org.my/advisory/MA-044.072002.html

CIAC (2001), "The W32.nimda Worm", http://www.ciac.org/ciac/bulletins/l-144.shtml

W. Cohen (1995), "Fast Effective Rule Induction", ICML.

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.

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

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

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

R. Lippmann, & J. W. Haines (2000a), "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 (2000b), "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, 346-350.

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 (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 Laboratory 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.

D. Moore, C. Shannon, J. Brown (2002), "Code-Red: a case study on the spread and victims of an Internet worm", IMW, http://www.caida.org/outreach/papers/2002/codered/codered.pdf

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

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.

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

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