Incrementally Learning Rules for Anomaly Detection

by

Denis Petrussenko

A thesis submitted to the College of Engineering at
Florida Institute of Technology
in partial fulfillment to the requirements
for the degree of

Master of Science
in
Computer Science

Melbourne, Florida
May, 2009

CS-2009-02

We the undersigned committee
hereby approve the attached thesis

Incrementally Learning Rules for Anomaly Detection

by
Denis Petrussenko

____
Philip K. Chan, Ph.D., Major Advisor
Associate Professor, Computer Science

____
Marius Silaghi, Ph.D.
Assistant Professor, Computer Science

____
Georgios C. Anagnostopoulos, Ph.D.
Assistant Professor, Electrical and Computer Engineering

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

Abstract

Title: Incrementally Learning Rules for Anomaly Detection

Author: Denis Petrussenko

Committee Chair: Philip K. Chan, Ph.D.

LERAD is an algorithm which learns rules that can be used for anomaly detection. However, because it is an offline algorithm, all training data has to be present before rules can be generated. We desire to create rules incrementally, as training data becomes available. Furthermore, accuracy should not suffer, remaining similar to offline LERAD. We present an algorithm that accomplishes this by carrying a small amount of data (namely, rules and sample sets) between days and pruning rules after the final day. Experimental results show that the difference in accuracy between incremental and offline LERAD is small enough to be statistically insignificant. Additionally, incremental LERAD achieves similar accuracy to offline while generating fewer rules, thereby decreasing overhead during detection.

Table of Contents

List of Figures

List of Tables

1) Introduction

Intrusion detection is usually split into two approaches: signature and anomaly detection. With signature detection, attacks are analyzed and unique descriptions are generated that describe them. This allows for extremely accurate detections of known attacks. However, the drawback is that attacks need to be analyzed and have a signature before they can be detected. This approach does not work well with new or unknown threats. With anomaly detection, a model is built to describe normal behavior and anything that does not fit the model is marked as an anomaly, allowing for the detection of previously unseen threats. The major drawback is that not all anomalous activity is malicious and false alarms become a massive issue. This work is about an intrusion detection algorithm that uses machine learning techniques to detect anomalies.

In machine learning, a set of "normal" (i.e. attack-free) data is processed by the learning algorithm, which results in a model being generated that represents the training data. The model typically takes the form of weights, statistics or rules. In this paper, the model is composed of rules. After a model is generated, it can be used for anomaly detection. Records from a test set, which is data that has not been seen previously, are compared against the model. In cases where a record does not fit the model, an alert is generated. The alerts are then used to flag intrusions. See **Fig. 1** for an illustration of this concept.

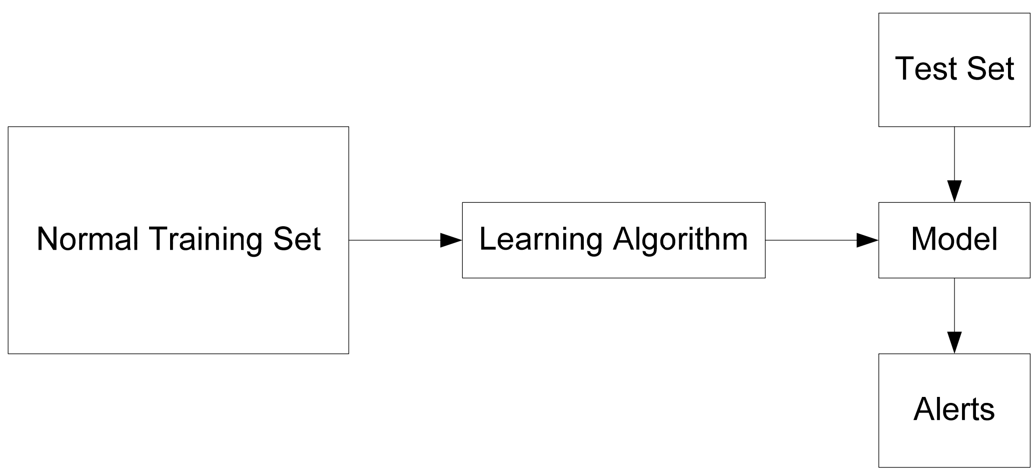

Fig. 1: Machine Learning Applied To Anomaly Detection

An example of anomaly detection is shown in **Fig. 2**. Data records are represented by solid shapes and outlined ovals show the model, with each oval corresponding to a single rule. In this case, the set of all rules (ovals) makes up the model from **Fig. 1**. Each rule describes some data records and ideally, all rules together describe most training data. Squares represent training data, with testing data being shown by diamonds, triangles and circles. Diamonds fit into the model and will not generate alarm scores (i.e. alarm value will be 0). Triangles are outside the model, but still pretty close. This will result in low alarm scores, which may or may not be treated as an alarm, depending on the state of the detector. Finally, circles are far away from the known model and therefore will trigger large alarm scores, marking them as anomalies in virtually all cases. In general, the further a record is away from the known model, the more likely it is to be treated as an anomaly.

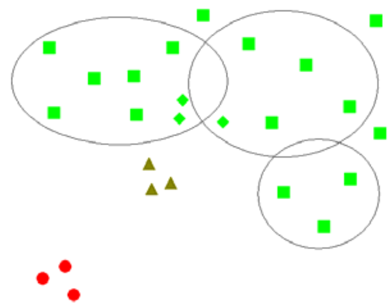

Fig. 2: Anomaly Detection With Rules

LERAD, an algorithm proposed by Mahoney & Chan (2003), generates rules that describe normal behavior. These rules can then be used to detect anomalies. However, LERAD is an offline algorithm that generates rules only after seeing all training data. With intrusion detection, it is beneficial to have an up-to-date system at all times. Time spent waiting for training data is time during which previously unseen normal system activity can trigger false alarms, leading to missed detections.

We want to start creating and updating rules as soon as possible, so that we can detect anomalies with new updates rules frequently. For example, we would like to have updated rules every day. We desire to generate rules incrementally, as data becomes available. Furthermore, performance should be no worse than the offline algorithm. Because the goal is similar accuracy, the difference between incremental and offline algorithms should be statistically insignificant on multiple datasets.

Offline LERAD works by creating a small sample set to represent the large training set (representing, for example, a week of data) and picking records from it at random, using matching attributes from both records to form rules. They are then trained and validated on the rest of the training data. Our incremental algorithm performs similar steps, working with smaller sets of training data (for example, with each day of the week). The sample set is shrunk from the offline version and carried between days. However, it looks similar to offline after processing the same amount of data. Rules are generated in the same fashion as before and also carried between days. We then modify the sample set structure to include some statistics about the training data it was generated from, leading to generated rules having closer anomaly scores to those from offline. Furthermore, we address the issue of online generating too many rules by pruning them after all training data is processed.

Our contributions include:

- an algorithm which creates rules that can be used for anomaly detection incrementally, as data becomes available;
- an incremental version of LERAD with accuracy similar, to the point of their difference not being statistically significant, to the original, offline version of LERAD;
- although incremental LERAD boasts similar accuracy to offline, this is accomplished with fewer rules, leading to less overhead during detection.

The next section contains related work. Following that, section 3 describes details of our approach, with evaluation and analysis provided in section 4. The paper is concluded in section 5, with an overview and some limitations. Note that because LERAD is an offline algorithm, we refer to it as OFF in this paper. Our version is referred to as incremental LERAD, or INCR.

2) Related Work

2.1) Overview

2.2) Rule Learning

LERAD is a rule algorithm that generates association rules. However, because they are created strictly from one class of training data (normal), they are used for characterization. In general, there are two general kinds of rule learning algorithms, those that learn classification rules and those that learn association rules (see **Fig. 3**). Rules that can have any attribute as the consequent are known as association rules. They normally present valuable information about relationships in the dataset. Rules where the attribute of the consequent is fixed are classification rules, meant to separate the data into known classes. When the consequent is present, the value is used as the class label. If the consequent is missing, that can be thought of as an implicit class label. That is, if a rule matches at all, the record belongs to the single default class. Rules that are meant to describe a single class are known as characterization rules, as opposed to rules that split data into several classes, which are known as classification rules. Problems that characterization rules are applied to (such as anomaly detection) are harder to work with than multi-class problems. This is due to the fact that the boundary around a single class is usually far less apparent than when a solution space is split into multiple known classes.

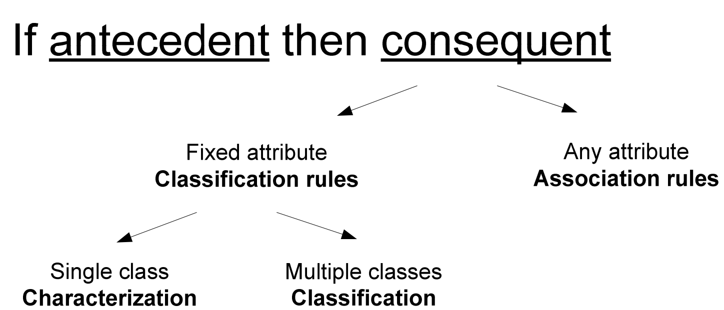

Fig. 3: Rule types

CN2 (Clark and Niblett, 1989) is an induction rule learning algorithm that generates classification rules to label available training data. It uses a beam search approach and evaluates rules based on their classification accuracy over training data. The resulting ruleset is able to classify all training data into known classes.

Another algorithm that generates rules for classification of data is RIPPER (Cohen, 1995). Like CN2, it also generates rules to classify all training data. Both algorithms employ a greedy beam search to find the best rules. However, during rule generation, where CN2 just keeps growing a rule until it stops increasing in performance, RIPPER both grows and prunes the rules that it generates, effectively increasing the beam size.

Marginal Method (Das and Schneider, 2007) looks for anomalous instances in the training set. This is accomplished by building a set of rules that look at attributes which are determined to be statistically dependent. The resulting rules are used to determine which training records are unexpected or anomalous. They can then be used to classify training data as “normal” or “abnormal”.

APRIORI (Agrawal and Srikant, 1994) generates all association rules above certain confidence and support levels that apply to the training data. The resulting rules are not used to classify training data, but rather to predict trends present in it. Because they are valuable in predicting shopping behavior, for example, it is desirable to find as many association rules for a given dataset as possible. The rules are meant to provide information and are not used for classification tasks.

WSARE (Wong et al, 2005) generates a single rule for a target day, along with a probability of it being true. The goal is to describe as many anomalous training records as possible. This is accomplished by establishing a baseline and looking for training records that deviate from it. The output rule is the best (most statistically significant) description of all relationships in the training data. Again, it is meant to provide information and not used for classification. However, since the rules describe the most anomalous occurrence, any record that matches them can be classified as abnormal.

2.3) Anomaly Detection

The goal of anomaly detection is to determine expected or normal behavior of a system and flag instances that do not follow the normal characteristics. This usually involves building a model of the system and comparing new data with the model. A number of anomaly detection algorithms have been developed and applied to host intrusion detection, analyzing data collected from system calls, web logs, general system attributes and network traffic. While most anomaly detection algorithm applications center on analyzing a single source of data, this is not a requirement. For example, LERAD is applied both to network logs and system calls. Furthermore, anomaly detection does not always involve detecting computer intrusions. It can be applied to any process where anomalies exist and are useful to detect, such as manufacturing or epidemiology.

VtPath (Feng et al, 2003) analyzes system calls at the kernel level. Looks at call stack during system calls and builds an execution path between subsequent calls. Such a model allows for detection of a corrupted stack, unexpected return addresses, changing system calls and previously unseen virtual paths between them.

Kruegel et al (2003) look at actual system call parameters, instead of just system calls. A model is built for each system call per application and deviation from all models is analyzed during detection. Models are built from parameter length, character distribution, structure and tokens.

Kruegel and Vigna (2003) examine activity logs for web applications, looking at a specific subset of parameterized web requests. Models are built on request parameters, based on their presence, length, character distribution, structure and tokens.

A slightly different approach is taken by Robertson et al (2006). Again, parameterized web requests are analyzed, but models are now built for each resource, based on parameter length, character distribution, structure and tokens. During detection, parameters outside the model are treated as alerts and a signature is generated that can match same or similar ones. Heuristics are used to classify alerts into attack types, but only after an anomalous event occurs to trigger an alarm. Anomaly signatures are used to group many alerts together, simplifying the system administrator's job.

Shavlik and Shavlik (2004) propose monitoring various system attributes to determine unauthorized or abnormal system usage. A model is built by learning weights for all attributes during normal operation. During detection, scores are generated based on weights and probabilities of attributes having certain values. This approach examines overall computer usage, not any particular application.

A comparative study of outlier detection algorithms applied to network data is done by Lazarevic et al (2003). Specifically, algorithms analyzed include distance to k nearest neighbors, just nearest neighbor, Mahalanobis distance, density based local outliers and unsupervised support vector machines. Density based local outliers performed far better than others. However performance was not high enough to allow for on-line network data analysis.

PAYL (Wang et al, 2005) examines anomalous packets coming into and then being sent out from the same host. A model generated for each deployment site, using n-grams from network packet datagrams, looking at distribution of single bytes. The algorithms used for input/output correlation also provide signatures that can detect similar activity. This data can be quickly shared between various sites to deal with zero-day threats.

3) Approach

3.1) Original LERAD (Offline)

LEarning Rules for Anomaly Detection, or LERAD (Mahoney & Chan, 2003), is an efficient, randomized algorithm that creates rules in order to model a time series. The rules describe what data should look like and are used to generate alarms when it no longer corresponds to the model. Rules have the format:

$$a_1 = v_{11} \wedge a_2 = v_{23} \wedge \dots => a_c \in \{v_{c1}, v_{c2}, \dots\} \; [r\; n\; w]$$

where a_i is attribute i in the dataset, v_{ij} is the j^{th} value of a_i and r, n and w are statistics used to calculate a score upon violation.

r is the number of unique values in the rule's consequent, which represents how likely the rule is to be violated. For example, a rule with a high r value has already been exposed to a lot of new values, so it is not very surprising to see another new one. On the other hand, rules with a low r value have not seen a lot of variation in the data, so seeing a new value is less expected. The problem is that r values rely on rules to be exposed to a lot of data before they are accurate, which is where n comes in. n is the number of records that matched the rule's antecedent. Each time n is incremented, the r value can potentially be increased as well, if the consequent attribute value in the tuple is not already present in the rule. In effect, n is amount of confidence we have that the r value of a rule is correct. Then rules with high n values and low r values are desired as they reflect properties of data that are known to not change, generating a larger "surprise" when they are violated. This was proposed in Witten & Bell (1991).

w is a measure of how well a rule performs on the validation set. It is calculated by observing the difference in mutual information between instances where rules conform to records and when they are violated (Tandon & Chan, 2007). Let Y be the target attribute of rule r_i and $\underline{X}$ be the antecedent of r_i, then:

$$w_i\left(\frac{Y \in \{y_1, y_2, \dots, y_j\}}{Y \notin \{y_1, y_2, \dots, y_j\}}\middle|\underline{X}\right) = I\big(Y \in \{y_1, y_2, \dots, y_j\}; \underline{X}\big) - I\big(Y \notin \{y_1, y_2, \dots, y_j\}; \underline{X}\big)$$

where $I(a; b)$ is the mutual information of a and b, calculated as:

$$I(a; b) = P(a, b) \log\left(\frac{P(a, b)}{P(a)P(b)}\right)$$

The $\frac{n}{r}$ values of rules can be thought of as the *belief* of a rule about its own quality, e.g statistics that predict how well a rule should perform. The w value is more of a *predictiveness* score, i.e. how well the rule actually performs.

After rules are generated and their statistics are calculated, they are used by LERAD to generate anomaly scores on unseen data. For each record $d \in D$, every rule r_i will either match or not match antecedent values. Only rules that do match are used to compute the anomaly score. Let R be the set of rules whose antecedents match d, then score is calculated by:

$$Score(d) = \sum_{r_i \in R} t_i w_i \frac{n_i}{r_i}$$

where t is the amount of time since this rule was last involved in an alarm. The goal is to look for rule violations that are most surprising, and a rule that has been violated recently is more likely to be violated again, as opposed to a rule that has been matching records for a long time. Scores above a certain threshold are then used to trigger actual alarms.

Input: sample set (D^s), training set (D^t) and validation set (D^v)
Output: LERAD rule set R

1. Generate candidate rules R' from D^s and evaluate them
2. Perform coverage test – select a "minimal" set from R' that covers D^s:
   a. Sort R' in increasing order of probability of being violated
   b. Discard rules from R' that do not cover any attribute values in D^s
3. Train the rest of R' on D^t
4. Validate R' on D^v
   a. Increase weight on rule conformance (increase rule belief)
   b. Decrease weight on rule violation (reduce rule belief)

Fig. 4: Main steps of Offline LERAD algorithm (Adapted from Fig. 1 in Tandon & Chan, 2007)

The LERAD algorithm is composed of four main steps in pseudocode shown in **Fig. 4**, with the corresponding dataflow in LERAD displayed in **Fig. 5**. Rule generation (step 1) involves picking antecedent and consequent attributes based on similarities of tuples randomly picked from the sample set. After a sufficient number of rules are generated, a coverage test is performed to minimize the number of rules (step 2). This is accomplished by only keeping enough rules to describe D^s and removing those that do not cover additional records (e.g. improve the model). Step 3 is to then expose the rules to the training set and update their n and r values based on how they apply to D^t. Finally, the weight of evidence is calculated for each rule (step 4) by applying it to D^v and observing how many times it conforms or violates. The output is a set of rules R, which is used to generate alarms on unseen data.

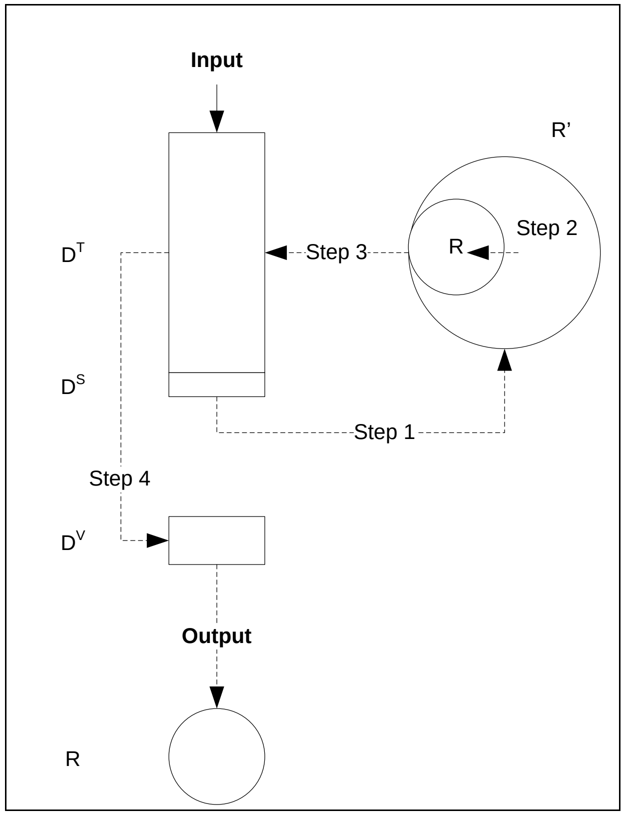

Fig. 5: LERAD data flow

While applying LERAD on data from **Table 1**, rules that are generated in various stages are shown in **Table 2**. Candidate rules in **a)** are a possible outcome of step 1. Rule 1 was generated by picking records 3 and 5 from the **sample set**. The first matching attribute happened to be *destination*, so it became the rule consequent. *port* was the next matching attribute, therefore it was added to the antecedent. Since there were no other matching attributes, the antecedent did not grow further. In a similar fashion, rule 2 was created from records 3 and 4 and rule 3 from records 1 and 2. Note that LERAD will produce more rules on the same data than those in **a)**, the ones shown are just an example. The rules are sorted by their *n/r* values during step 2, as shown in **b)**, to ensure the coverage test works properly. When checking rules in **b)** against the mock sample set, rule 1 will match both antecedent and consequent values on all records that rule 3 would. Because rule 3 is not able to match any records that have not been matched, or covered, by rule 1, it is removed, as shown in **c)**. During step 3, rules are trained on all data, which is not shown, usually resulting in expanded consequent values (and increased *rs)*. For example, records 27 and 33 in the training set provide extra consequent values for rule 1. The *n* values also increase for most rules at this stage, as rules match previously unseen records. Note that rules from the validation set are not used for training, since they are held out for the next step. The validation step applies candidate rules, shown in **d)**, to the validation set. Statistics are updated and the w value is calculated and in original LERAD, rules with $w \leq 0$ are deleted. In this case, rows in the validation set act as negative evidence for rule 2 from **d)**, causing it to be removed. Then final output of LERAD is then presented in **e)**.

Table 1: Mock data sets

Training Set	word1	word2	ip	dest	port
1	HELO	example.com	10.0.0.17	10.0.0.200	25
5	HELO	sample.com	10.0.0.24	10.0.0.201	25
		…			
14	GET	/about.html	10.0.0.8	10.0.0.15	80
18	POST	/about.html	10.0.0.13	10.0.0.15	80
22	POST	/index.html	10.0.0.4	10.0.0.15	80
		…			
27	…	…	…	10.0.0.16	80
33	…	…	…	10.0.0.10	80

Sample Set	word1	word2	ip	dest	port
1	HELO	example.com	10.0.0.17	10.0.0.200	25
2	HELO	sample.com	10.0.0.24	10.0.0.201	25
3	GET	/about.html	10.0.0.8	10.0.0.15	80
4	POST	/about.html	10.0.0.13	10.0.0.15	80
5	POST	/index.html	10.00.4	10.0.0.15	80

Validation Set	word1	word2	ip	dest	port
1	HELO	somehost.com	10.0.0.12	10.0.0.202	26
2	HELO	otherhost.com	10.0.0.12	10.0.0.203	27
		…			

Table 2: Example rules

a) Candidate rules after step 1 (rule generation)		*n/r*
1	$port = 80 => dest \in \{10.0.0.15\}$	3/1
2	$word2 = /about.html \wedge port = 80 => dest \in \{10.0.0.15\}$	2/1
3	$word1 = HELO => port \in \{25\}$	2/1

b) Candidate rules during step 2 (ranking before coverage test)		*n/r*
1	$port = 80 => dest \in \{10.0.0.15\}$	3/1
2	$word1 = HELO => port \in \{25\}$	2/1
3	$word2 = /about.html \wedge port = 80 => dest \in \{10.0.0.15\}$	2/1

c) Candidate rules after step 2 (coverage test)		*n/r*
1	$port = 80 => dest \in \{10.0.0.15\}$	3/1
2	$word1 = HELO => port \in \{25\}$	2/1

d) Candidate rules after step 3 (training on entire training set)		*n/r*
1	$port = 80 => dest \in \{10.0.0.15\ 10.0.0.16\ 10.0.0.10\}$	9/3
2	$word1 = HELO => port \in \{25\}$	7/1

e) Final LERAD rules after step 4 (validation)		*n/r*
1	$port = 80 => dest \in \{10.0.0.15\ 10.0.0.16\ 10.0.0.10\}$	9/3

3.2) Basic Incremental Algorithm

Each dataset used for LERAD is divided into three sets: training, validation and test. For day k, let the sample set be D_k^S, training set be D_k^T and validation set be D_k^V. With OFF, training data consisted of one dataset and testing data of another. With INCR, training data is split into roughly equal-sized sets, or “days”, with testing data remaining in a single set. Regardless of being split up, the training data for INCR is exactly the same as for OFF. The sample set is generated by randomly copying a small amount of records from D_k^T. Lastly, D_k^V consists of a small fraction (e.g. 10%) of all training set records, removed before training data is loaded. Training records that are moved into the validation set are chosen at random, but the exact records used in INCR and OFF are always the same.

Input: m sample sets (D_1^S to D_m^S), m training sets (D_1^T to D_m^T) and m validation sets (D_1^V to D_m^V)
Output: LERAD rule set R
For each day k:

1. Generate candidate rules R'_k from $D_k^{CS} = \{D_1^S \cup D_2^S \cup \ldots \cup D_k^S\}$ and evaluate them
2. Perform coverage test – select a "minimal" set of candidate rules that covers D_k^{CS}:
    a. Sort R'_k in increasing order of probability of being violated
    b. Discard rules from R'_k that do not cover any attribute values in D_k^{CS}
3. Discard rules from R'_k that are already present in R_{k-1} (e.g. were created during any previous day)
4. Create daily rule set R_k by combining old rules with new: $R_k = R_{k-1} \cup R'_k$
5. Train R_k on D_k^T
6. Validate R_k on D_k^V
    c. Increase weight on rule conformance (increase rule belief)
    d. Decrease weight on rule violation (reduce rule belief)

Fig. 6: Main steps of incremental LERAD algorithm

INCR applies the same basic algorithm to training data as OFF. The main difference is that rules are carried between (**Fig. 6**, step 4), and trained on, later days (**Fig. 6**, step 5 and 6). Sample set data is also carried between days. For sample set generation, INCR mirrors OFF and creates a new sample set D_k^S for each day k of data. However, daily sample set size is shrunk from the OFF size proportional to the number of days in training data, $\left|D_k^S\right| = \frac{|D^S|}{m}$. This prevents INCR from having access to many more sample records to create rules with than OFF, which would result resulting in different rules. Using more records during rule generation is not necessarily detrimental, but it does stray from the OFF algorithm, and so was avoided. Furthermore, instead of using just D_k^S, sample sets from previous days are now carried over and joined together into the "combined" sample set D_k^{CS} (**Fig. 7**), from which R'_k (new rules for day k) are generated (**Fig. 6**, step 1). Because for any day k, D_k^T and D_k^V are massive compared to D_k^S, they are not carried over between days.

As before, a coverage test is performed (**Fig. 6**, step 2) after R'_k is generated to ensure that rules maintain their quality level, only now using D_k^{CS} instead of D_k^S, since that is what the rules were created from. After coverage test, the remaining rules in R'_k are compared with R_{k-1}, which contains rules from all previous days. Any rules present in R'_k that already exist in R_{k-1} are removed from R'_k (**Fig. 6**, step 3). This is not an issue because at this point, rules in R'_k have only been trained on D_k^{CS}, which consists mostly data that R_{k-1} has already been exposed to. The only information lost is what R'_k gained from training on D_k^S (which is part of D_k^{CS}). This is remedied by merging new rules with old, $R_k = R_{k-1} \cup R'_k$ (**Fig. 6**, step 5) and training R_k on D_k^T (which is a superset of D_k^S). Before training, rules from R_{k-1} already have some statistics from previous days and rules from R'_k have statistics from being trained on D_k^{CS}. These are not reset before training, statistics from D_k^T are simply added to it. That way, a rule trained (for example) on days 1 to k has the same information as a rule trained on D^T that consists of all records from days 1 to k. After training, R_k is validated on D_k^V. Again, the previous weight value is not reset but rather combined with the value from day k.

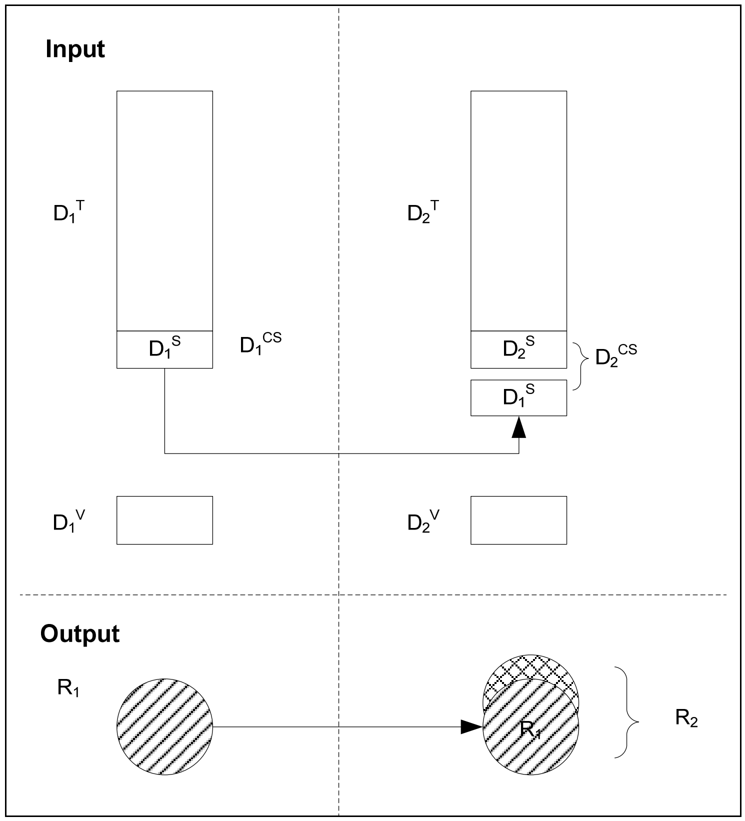

Fig. 7: Incremental LERAD data flow for day 1 and 2

The flow of data from day 1 to day 2 of incremental LERAD is shown in **Fig. 7**. It looks similar to offline LERAD, shown in **Fig. 5**, with the exception that sample sets and rules are passed along. The combined sample set D_k^{CS} is shown growing in size, with previous sample sets appended to the next day. For any day, the candidate rule set R'_k is created and then significantly shrunk by the coverage test. The remaining rules are passed on to the next day (after training and validation).

Fig. 8 shows the transfer of information through all days of incremental LERAD. Each day involves creating a new R'_k from the sample sets of all previous days. Most of R'_k is lost in the coverage test, with some tuples removed due to duplicates from previous days as well. After each day, R_k grows a little bit, after rules from day k are added to all previous ones.

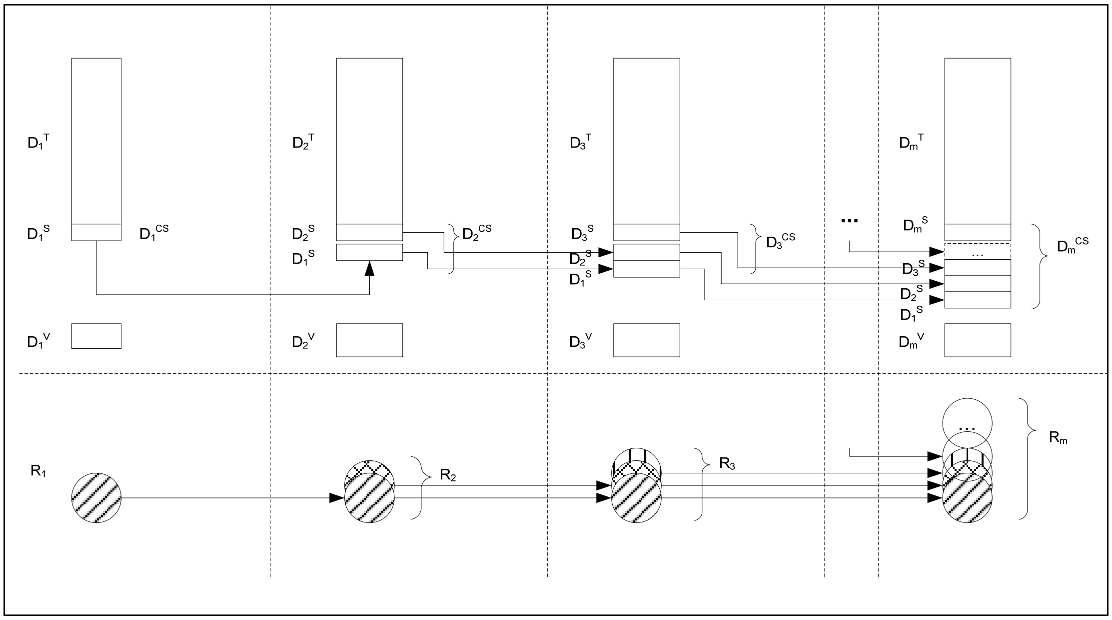

Fig. 8: Incremental LERAD data flow for all days

3.3) Collecting Appropriate Statistics

LERAD generates rules strictly from the sample set, which is a comparatively small collection of records that is meant to be representative of the entire dataset. The first step in LERAD rule generation is to pick antecedent and consequent attributes based on similarities of tuples randomly picked from the sample set. Afterwards, known consequent values for each rule may change in the training phase and rules can be completely dropped in the coverage test, but the antecedent and consequent attributes are never changed after they are initially picked. Thus, the sample set is solely and entirely responsible for the structure of all rules generated by LERAD.

Using D_k^{CS} instead of D^S allowed INCR to generate rules with the same structure as those in OFF. However, even though rules were structurally the same, the statistics they carried were different. Comparing rules present in both INCR and OFF, but only detecting attacks in OFF, a number of deficiencies can be seen in INCR rules (see **Table 3**).

Table 3: Rules that detected attacks in OFF but did not detect in INCR

OFFLINE Rules	w	n	r	INCREMENTAL Rules	w	n	r
if SA3=172 then SA1 = 113 114 112 115	2.92	10857	4	**if SA3=172 then SA1 = 113 114 112**	2.03	1586	3
if DUR=0 F1=.S then DP = 113 25 80 79 22 515	2.11	19139	6	**if DUR=0 F1=.S then DP = 113 25 80 79 22 515**	1.88	11118	6
if W3=. then W2 = .	3.44	7596	1	**if W3=. then W2 = .**	3.38	6408	1
if W1=.^@GET then W3 = .HTTP/1.0^M^ .align=	3.06	12867	2	**if W1=.^@GET then W3 = .HTTP/1.0^M^ .align=**	2.99	11260	2

All statistics carried by INCR rules are lower than those of OFF, resulting in missed detections due to lower alarm scores.

To prevent INCR from missing detections that OFF correctly identifies, rules generated by both algorithms need to contain similar statistics, in addition to their similar structure. For INCR rules generated early on, this is not an issue. For example, the last two rules in **Table 3** were generated on day 2 (of 7) and all their statistics are close to OFF. On the other hand, the first rule was generated on the last day and consequently has a significantly smaller n value, as well as smaller r and w. This is caused by rules not having access to data from days before they were generated.

In order to completely eliminate this problem, data from all days would have to be kept around, so that all rules can be trained on all days. With an incremental algorithm, this is not feasible. Instead, additional statistics are kept that are representative of each D_k^T. INCR is augmented to carry an additional piece of information for each day:

$$ScalingFactor_k = \frac{|D_k^T|}{|D_k^S|},$$

where k refers to a certain day of training data. By using tuples from D_k^S and repeating each one $ScalingFactor_k$ times, a “virtual” training set D_k^{VT} is created (see **Fig. 9**), which is used to represent the training set from that day.

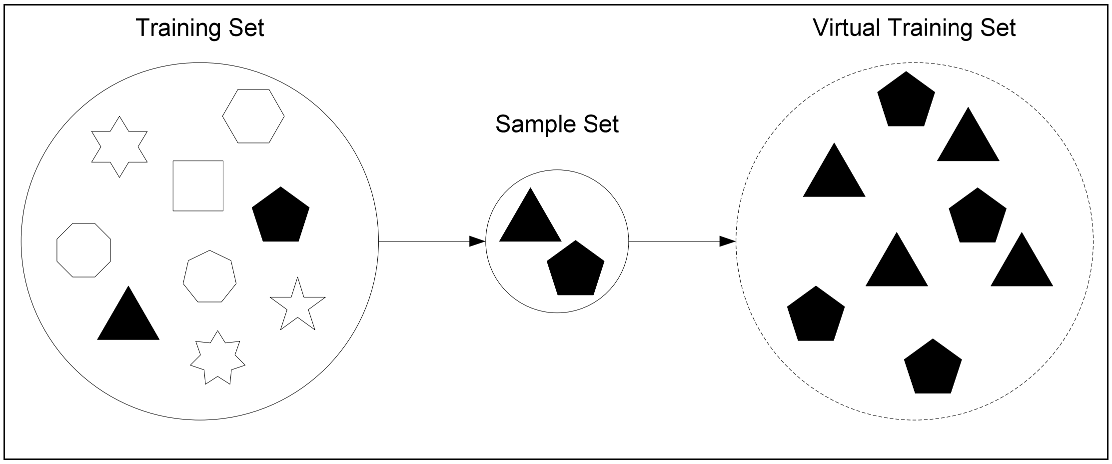

Fig. 9: Sample Set Expansion

Then for each day k, all previous virtual training sets (from D_1^{VT} to D_{k-1}^{VT}) are joined together to create a "combined" virtual training set D_k^{CVT}:

$$D_k^{CVT} = D_1^{VT} \cup D_2^{VT} \cup \ldots \cup D_{k-1}^{VT}.$$

As before, new rules for day k are generated on the combined sample set D_k^{CS}. However, now $D_k^{CVT} \cup D_k^{T}$ is used for training instead of just D_k^{T}, allowing rules to obtain consequent values that may have only been present in previous days and enabling n counts to reflect records present across all days. Note that the virtual set is purely an abstract notion, it is not actually created. Sample sets are simply used in a manner that is consistent with having a virtual training set. During training, when a rule matches a record, its n value is increased by $ScalingFactor_k$ instead of just 1. Consequent values are simply appended, along with r beign incremented, if they don't yet exist in the rule. The weight of evidence calculation for new rules is also modified to collect statistics from D_k^{CS} in addition to D_k^{V}. Naturally, this only applies to new rules. Those from previous days were trained on the actual training sets that D_k^{CVT} attempting to approximate and so are not exposed to D_k^{CVT} or D_k^{CS}. This approach yielded rules with all statistics much closed to OFF, as shown in **Table 4**.

Table 4: Problem rules after using D_k^{CT} instead of D_k^{T} (6)

OFFLINE Rules	w	n	r	INCREMENTAL Rules	w	n	r
if SA3=172 then SA1 = 113 114 112 115	2.92	10857	4	**if SA3=172 then SA1 = 113 114 112**	2.11	10553	3
if DUR=0 F1=.S then DP = 113 25 80 79 22 515	2.11	19139	6	**if DUR=0 F1=.S then DP = 113 25 80 79 22 515**	1.89	16979	6
if W3=. then W2 = .	3.44	7596	1	**if W3=. then W2 = .**	3.39	7393	1
if W1=.^@GET then W3 = .HTTP/1.0^M^ .align=	3.06	12867	2	**if W1=.^@GET then W3 = .HTTP/1.0^M^ .align=**	3.00	12247	2

This approach brings n values much closer to OFF than it does r or w values. The $ScalingFactor$ value only offers an extra benefit to n, since it is the only value directly affected by it.

Input: m sample sets (D_1^S to D_m^S), m training sets (D_1^T to D_m^T) and m validation sets (D_1^V to D_m^V)
Output: LERAD rule set R
For each day k:

1. Generate candidate rules from D_k^{CS} and evaluate them
2. Perform coverage test – select a "minimal" set of candidate rules that covers D_k^{CS}:
    a. Sort candidate rules in increasing order of probability of being violated
    b. Discard rules that do not cover any attribute values in D_k^{CS}
3. Discard candidate rules that were generated and saved during any previous days
4. Train remaining candidate rules on D_k^{CVT}
5. Validate candidate rules on D_k^{CS}
6. Combine candidate rules with all rules from days 1 to $k-1$
7. Train all rules on D_k^T
8. Validate all rules on D_k^V

Fig. 10: Incremental LERAD with corrected rule statistics

The main changes from general INCR are lines 4 and 5 of **Fig. 10**. New rules are now trained on data carried over from previous days before being joined with older rules. This allows their statistics values to be close to those of OFF rules. They are then merged with older rules and collectively trained on D_k^T, just as in general INCR.

While the current sample set approach yields results that parallel the OFF algorithm, it will have to be changed to obtain optimal results with a purely incremental implementation. Currently, the sample set grows without bound in order to match the OFF sample set. However, in the real world, there will need to be a limiting mechanism on its growth. One such mechanism is to not keep sample sets older than a certain number of days.

3.4) Pruning Rules

INCR generates new rules for each training day, merging them together with rules from previous days to arrive at the final rule set. Because new rules are generated multiple times, when using the same dataset, INCR creates far more rules than OFF. By design, a lot of rules are common to both INCR and OFF and share the same structure. However, there is usually a set of extra rules unique to INCR. Some of these rules are beneficial, causing detections that would otherwise have been missed, and some are harmful, resulting in false alarms that drown out legitimate detections. There are two approaches to dealing with harmful rules: either good rules can be made louder, to stand out above the noise, or bad rules can be removed, lowering the noise level. Changing the loudness of rules involves altering the calculation of their statistics, which would cause INCR rules to be different from OFF. This conflicts with the goal of INCR, leaving just the rule removal option.

Rules have a number of properties that could potentially be used to predict how good or bad they will be. However, analysis yielded the number of generations (or birthdays) as the best predictor of bad rules. Let B be the number of times a rule was generated. Most of the rules that were present only in INCR and were only creating false alarms had low B values, which was not the case for rules that provided detections (see **Table 5**).

Table 5: INCR rules responsible for detections and false alarms

INCREMENTAL Rules	Type	B	w	n	r
if DP=25 F1=.S F2=.AP then W1 = .^@EHLO .^@HELO	DET	5	2.70	13152	2
if DP=25 F3=.AF then W1 = .^@EHLO . .^@HELO	DET	7	2.84	13611	3
if SA2=016 F3=.AF then SA1 = 113 114 112 115	DET	3	2.91	10296	4
if F3=.AF W7=. then W6 = .	**FA**	**1**	2.87	5080	1
if F2=.AP W3=.HTTP/1.0^M^ then W1 = .^@GET	**FA**	**2**	3.01	10478	1
if F1=.S W3=.HTTP/1.0^M^ then W4 =.Referer: .Host: .User-Agent: .Connection:	**FA**	**1**	2.08	10837	4

The B heuristic allowed for removal of unwanted rules. INCR was modified to remove rules with B values below a certain threshold from the final rule set, after all rules have been generated. This resulted in the removal of rules that were causing false alarms, leading to an increase in performance. For example, the LL/tcp dataset went from final INCR rule count of 250 with no rule dropping to just 68 rules when *B* was set to 3, compared to 77 rules in OFF. While the exact value of *B* depends on the dataset, experiments show that $B = 2$ tends to provide closest AUC values to OFF. *B* is currently determined by performing a sensitivity analysis across all possible values. Further work is needed to establish the ideal *B* value during training.

Note that rules are removed in other areas of INCR as well. During the coverage test, redundant rules that do not provide additional information are deleted. Rules that acquire negative or zero weights during the validation are removed as well. This section introduces a third point of rule removal into the algorithm, as the existing ones are not sufficient.

Input: m sample sets (D_1^S to D_m^S), m training sets (D_1^T to D_m^T) and m validation sets (D_1^V to D_m^V)
Output: LERAD rule set R
For each day k:

1. Generate candidate rules from D_k^{CS} and evaluate them
2. Perform coverage test – select a "minimal" set of candidate rules that covers D_k^{CS}:
    a. Sort candidate rules in increasing order of probability of being violated
    b. Discard rules that do not cover any attribute values in D_k^{CS}
3. Delete candidate rules that were generated and saved during any previous days (duplicates)
    a. Increment B values of each old rule for each deletion caused by it
4. Train remaining candidate rules on D_k^{CVT}
5. Validate candidate rules on D_k^{CS}
6. Combine candidate rules with all rules from days 1 to $k-1$
7. Train all rules on D_k^T
8. Validate all rules on D_k^V

Remove all rules with low B values.

Fig. 11: Incremental LERAD with corrected rule statistics and rule pruning

The main change rule pruning brings to the general INCR algorithm is the last line in **Fig. 11**. To determine which rules will be dropped, the B value is calculated on line 3-a. Note that rules that are removed during the coverage test on line 2 do not contribute to the B value, as they would not have been generated by OFF in the first place.

Again, this approach is best suited to matching the performance of OFF. In a true incremental algorithm, it is not acceptable for rules to grow without bound. A possible solution is to modify INCR to drop rules that have not been re-generated in a certain number of days.

4) Empirical Evaluation

In this section, we evaluate the performance of incremental LERAD and compare it to offline LERAD, to determine if they are similar.

4.1) Data

Five different datasets were used for evaluation:

1. The DARPA / Lincoln Laboratory (LL TCP) contained 185 labeled instances of 58 different attacks. The full attack taxonomy is available in Kendell (1999).
2. The UNIV set comprised of over 600 hours of network traffic, collected over 10 weeks from a university departmental server (Mahoney and Chan, 2003). It contained a total of six attacks: a port scan from inside the firewall, an external HTTP proxy scan and DNS version probe, as well as the Nimda, Code Red II and Scalper worms. The port scan consisted of two parts, a cgi-bin/htsearch exploit aimed at retrieving the local passwd file, followed by scan for other vulnerabilities on open ports.
3. The DARPA BSM set was an audit log of system calls from a Solaris host. There were 33 attacks present, spread across 11 different applications. Dataset evaluation performed in Lippmann et al (2000).
4. The Florida Tech and University of Tennessee at Knoxville (FIT/UTK) dataset contained macro execution traces with 2 attacks (Mazeroff et al, 2003). One was a distributed denial of service attack and the other provided behavior similar to the “Love bug” worm, corrupting user files, modifying the registry and executing a program.
5. Finally, the University of New Mexico (UNM) set included system calls from 3 applications (Forrest et al, 1996). *login* and *ps* traces came from Linux machines, *lpr* originated from a SUNOS 4.1.4 host. There were a total of 8 distinct attacks present.

4.2) Experimental Procedures

For all datasets, training data was entirely separate from testing. LL training data consisted of 7 days, ~4700 records each, with almost 180,000 records in testing. With UNIV, week 1 was split into 5 days of training data, ~2700 records each, and weeks 2 through 10 were used for testing (~143,000 records). In BSM, week 3 was separated into 7 days or ~26,000 records each, with weeks 4 and 5 used for testing (~350,000 records). FIT/UTK had 7 days of training data, with ~13,000 records each and ~13,000 records for testing.

Several adjustable parameters were employed in the experiments. The size of D_k^S was set to $\frac{100}{m}$, where 100 was the sample set in offline experiments and m was the number of training days. See section 3.2 for the reasoning behind this. This still results in extremely small sample set sizes when compared to the training set. For example, for the LL dataset, $|D_k^S| = 0.3\%$ of $|D_k^T|$, putting the sample set size at well below 1% of the training set. Validation set D_k^V was set to 10% of training data, candidate rule set R_k' was set to 1000 and the maximum number of attributes per rule was set to 4, all to mirror Tandon & Chan (2007) experiments. The rule-pruning parameter B was set to 2, as experiments showed this produced the closest performance curve to OFF.

On every dataset, both incremental and offline LERAD were ran 10 times each, with varying random seeds. For datasets with multiple applications, a separate model was created for each application and the results averaged together, weighted by the number of training records for that application. As applications have vastly different amounts of training records, their results cannot be simply averaged together. Because LERAD is looking for anomalous activity, applications with more training records, or activity, have a higher number of alarms. Therefore they are more relevant to LERAD's performance on the whole dataset.

For rule comparison, each INCR run is compared to all OFF runs and average counts of rules involved are taken. Then all INCR runs are averaged together for each dataset. For datasets with multiple applications, the results for each application are then averaged together for the whole dataset.

4.3) Criteria

In anomaly detection, the goal is to flag novel events. But because not every novel event is necessarily a threat, false alarms are a core problem of the field. With a sufficiently large dataset, even low alarm rates produce too many attack notifications to be useful. Because of this, we focus on extremely low false alarm rates, specifically 0.1%.

To show performance, the false alarm rate is varied in small increments between 0 and the maximum value (that which results in false alarm rate of 0.1%) and the percentage of valid detections is measure for each one. This data is used to plot an ROC (receiver operating characteristic) curve, where the X axis is false alarm rate and the Y axis is the detection rate (see **Fig. 12**). Calculating the area under this curve results in a value, called area under curve or AUC, which represents absolute performance for the algorithm. Higher AUC values mean the ROC curve climbs faster and/or higher, indicating better performance.

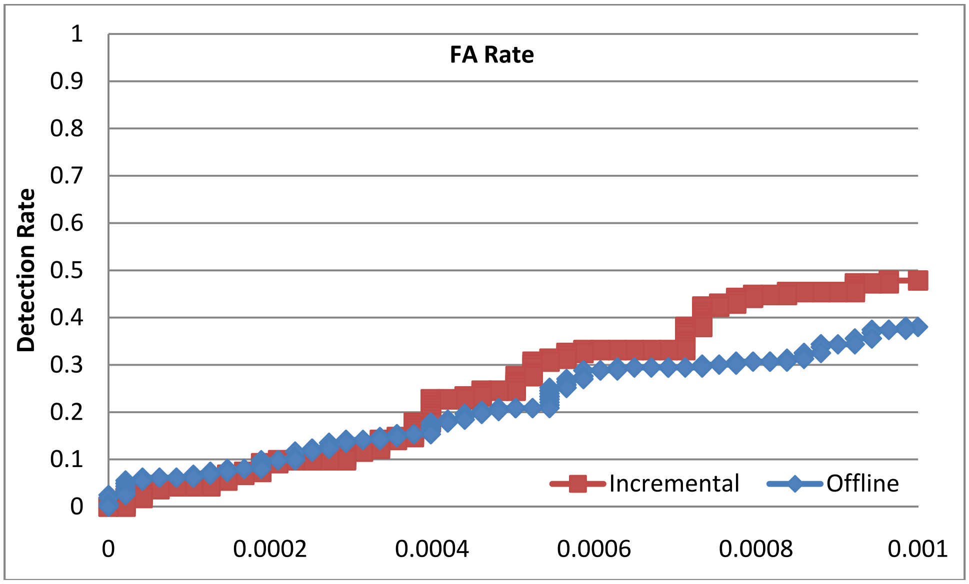

Fig. 12: Example ROC Curves

We chose to set the maximum AUC value to 1.0, which occurs when there are no false alarms and every single attack is detected at all times. In our tests, we only look at false alarm rates up to 0.1% (from a maximum of 100%, naturally), thereby concentrating on the first $\frac{1}{1000}$th of the ROC curve. This brings the maximum AUC value possibly encountered to 0.001, which is the highest performance that is possible in our tests.

To properly average together multiple applications from a single dataset, let $CT_{app} = |D_{app}^{t}| + |D_{app}^{v}|$ be the count of training records for an application. Then the average AUC for the whole dataset is then computed as:

$$AUC^{avg} = \sum_{app \in Apps} \left(AUC_{app} \frac{CT_{app}}{\sum_{app \in Apps} CT_{app}} \right)$$

Another approach would be to also include the size of the testing set when calculating CT_{app}. However, because the testing set does not actually affect the rules generated by LERAD, it is not included in the AUC^{avg} calculation.

For example, UNM contains 3 applications: *login* (5923 training records), *lpr* (1351870 training records) and *ps* (3130 training records). Each one was treated as a separate dataset for rule generation, with separate rules and ROC curves. After rules were generated and evaluated, the AUCs were averaged together across 10 runs, for both INCR and OFF, and then weighted-averaged together to form

$AUC_{UNM}^{avg} = \left(AUC_{login}^{avg} 0.0043 + AUC_{lpr}^{avg} 0.9934 + AUC_{ps}^{avg} 0.0023\right)$, which was used as UNM's AUC. Because *lpr* has so many more records than *login* or *ps*, it heavily influences the final dataset AUC.

4.4) Establishing the Pruning Parameter

One of the best ways to measure performance of INCR and OFF algorithms is to compare the areas under their ROC curves, or AUC values. Let ΔAUC be the difference in AUC values between INCR and OFF:

$$\Delta AUC = AUC_{INCR} - AUC_{OFF}$$

Then a positive ΔAUC value indicates that INCR is performing better than OFF. Conversely, when INCR performs worse than OFF, ΔAUC is negative, with lower values indicating worse performance. The goal of INCR is to be as close as possible to OFF, therefore ΔAUC values closest to 0 are desired.

In incremental LERAD, the pruning threshold B has the largest effect on AUC. As described in section 3.4, B is the number of times a rule was generated (or "born") during training. After training, rules are removed from INCR that have B values less than a certain threshold. Since there is no apparent way of picking a B value, all are compared. Most datasets did not produce any rules for values of $B > 5$, either because they were only split into 5 days or because no rules were generated more than 5 times in a row. Thus the maximum B value tested was set to 5. Because no rule can exist without being generated at least once, any values of $B < 1$ would produce the same result as $B = 1$, namely keeping all rules that were generated during all days.

The B values and resulting ΔAUCs across all datasets are shown in **Fig. 13**. With BSM, LL TCP and UNIV datasets, INCR performance is similar to OFF, resulting in ΔAUC values fairly close to zero. With UNM, there were no attacks detected by OFF below 0.1% false alarm rate, while INCR did detect some attacks, resulting large ΔAUC differences. The opposite situation occurred with FIT/UTK, with INCR detecting no attacks.

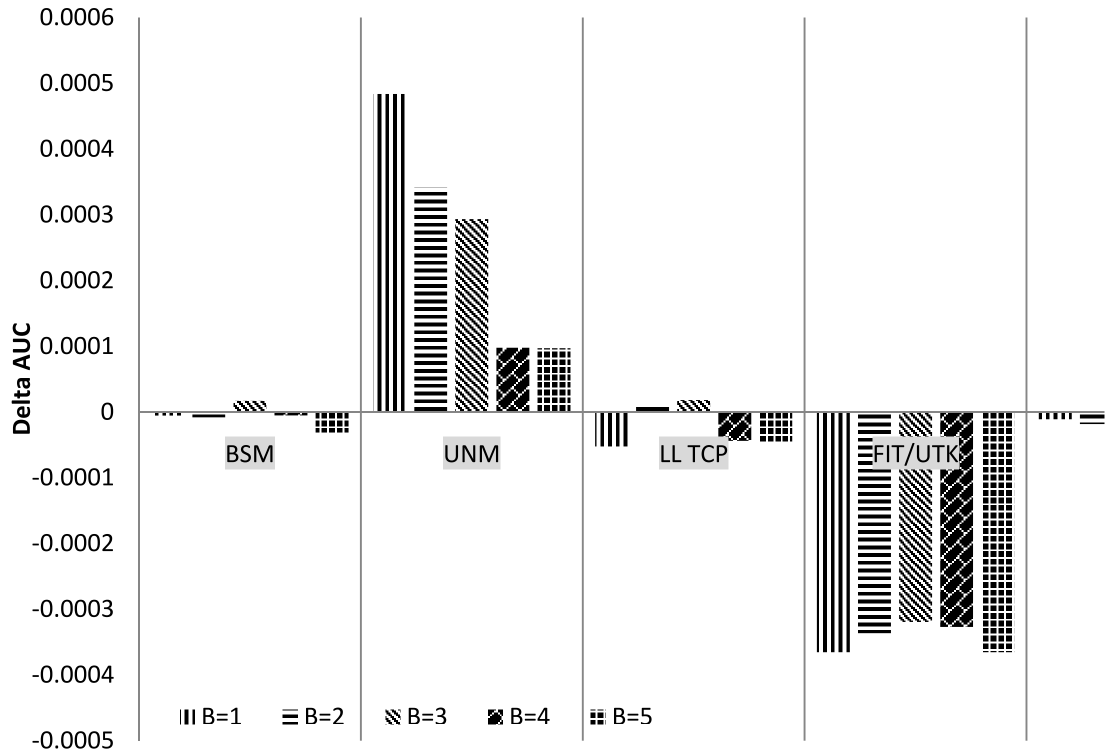

Fig. 13: ΔAUCs versus Bs

Overall, there is no obvious relationship between B and ΔAUC values across all datasets.

To determine whether ΔAUC differences are statistically significant, and under which B values, we perform the two-sample T-test on data presented in **Fig. 13**. Specifically, the AUC values from 10 INCR runs from each dataset are compared against AUCs from 10 OFF runs. For multi-application datasets, AUC values are averaged together, weighted by number of training records (as discussed in 4.2). $B = 4$ leads to statistically insignificant AUC differences between INCR and OFF across most datasets (BSM, UNM, LL TCP, and UNIV, see **Table 6**). For FIT/UTK, there was no statistically insignificant AUC difference for any B, due to very poor performance of INCR.

Table 6: P(T<=t) two-tail for two-sample T-test

Dataset	B=1	B=2	B=3	B=4	B=5
BSM	0.27	0.18	0.02	0.66	0.01
UNM	0.00	0.00	0.01	0.17	0.17
LL TCP	0.01	0.57	0.37	0.08	0.10
FIT/UTK	0.00	0.00	0.00	0.00	0.00
UNIV	0.40	0.17	0.08	0.06	0.04

Table 6 shows the absolute two-tail probabilities in Student's t distribution (with degrees of freedom = 9), computed from experimental data. For each dataset and B value, **Table 6** contains the probability that INCR and OFF are not significantly different. Because the goal is to have similar performance, we concentrate instead on results where INCR and OFF do not have a statistically significant performance difference. Therefore, cells with probability over 0.05 are highlighted, to show which instances are not significantly different. That is, they do not have a probability < 0.05 of being significantly different, which is required in order to be at least 95% confident in their difference. Note that some probabilities are 0.0 – this happens when ΔAUC values are extremely large. In UNM, OFF had extremely small AUC values, which were very different from the AUCs that INCR produced with low B values. However, as Bs increased and performance fell (see **Fig. 13**), INCR performance became almost as poor as OFF, low enough to no longer be statistically significant. In the FIT/UTK dataset, the situation was reversed. INCR produced low scores across the board, never coming close to OFF.

To conclusively determine which B values result in statistically insignificant ΔAUCs across all datasets, we apply the paired T-test. For each dataset, the average AUC value for each B was paired with the average OFF AUC for that dataset. Again, as with **Table 6**, the number in **Table 7** represent the probabilities associated with Student's T-test. In order to be at least 95% confident that incremental and offline AUCs are statistically different, the probability values in **Table 7** need to be less than 0.05. Since that is not the case, none of the Bs yielded statistically different performance between INCR and OFF. This is due mostly to the similar variations of AUCs between all datasets, which were exhibited by both OFF and INCR. The two least different B values were 2 and 3, and since $B = 2$ had insignificant performance difference on 3 of 5 datasets (see **Table 6**), it was chosen as the B value for all other experiments.

Table 7: P(T<=t) two-tail for paired two sample T-test

B=1	B=2	B=3	B=4	B=5
0.95	0.98	0.98	0.44	0.39

4.5) Ruleset Sizes

After the B value is picked, rules from INCR and OFF are analyzed to determine the causes of AUC differences. There are four different cases that are possible when comparing an INCR rule to an OFF one. Let α be the set of rules that have the same antecedent attributes, same consequent attribute and the same consequent values. They are then structurally exactly alike, such as rules 1 and 2 in **Table 8**. The set β consists of rules that have the same antecedent attributes and the same consequent attribute, but different consequent values. How different they are is not important, as long as they are not exactly alike. For example, rules 1 and 3 in **Table 8**. Then γ is the set of rules that have the same antecedent attributes but a different consequent attribute. The values in the consequent are irrelevant, since rules with consequent attributes cannot be compared, such as rules 1 and 4 in **Table 8**. Finally, the set λ contains rules that have different antecedent attributes (see rules 1 and 5 in **Table 8**). Again, the degree of difference is not important, as long as at least one consequent attribute is different. When comparing INCR rules against OFF rules, each INCR rule will belong to either α, β, γ or λ, which are mutually exclusive and together describe all possible outcomes (see **Fig. 14**).

Table 8: Rule comparison example

ID	Rule	Alg.	Cmp.	w	n	r
1	**if DestIP=128.1.2.3 DP=25 then W1 = EHLO HELO**	OFF		2.40	12540	2
2	**if DestIP=128.1.2.3 DP=25 then W1 = EHLO HELO**	INCR	α	2.06	11680	2
3	**if DestIP=128.1.2.3 DP=25 then W1 = HELO**	INCR	β	1.82	10520	1
4	**if DestIP=128.1.2.3 DP=25 then W3 = MAIL**	INCR	γ	2.30	10840	1
5	**If DP=80 W1=GET then W3 = HTTP/1.0 HTTP/1.1**	INCR	λ	2.47	12432	2

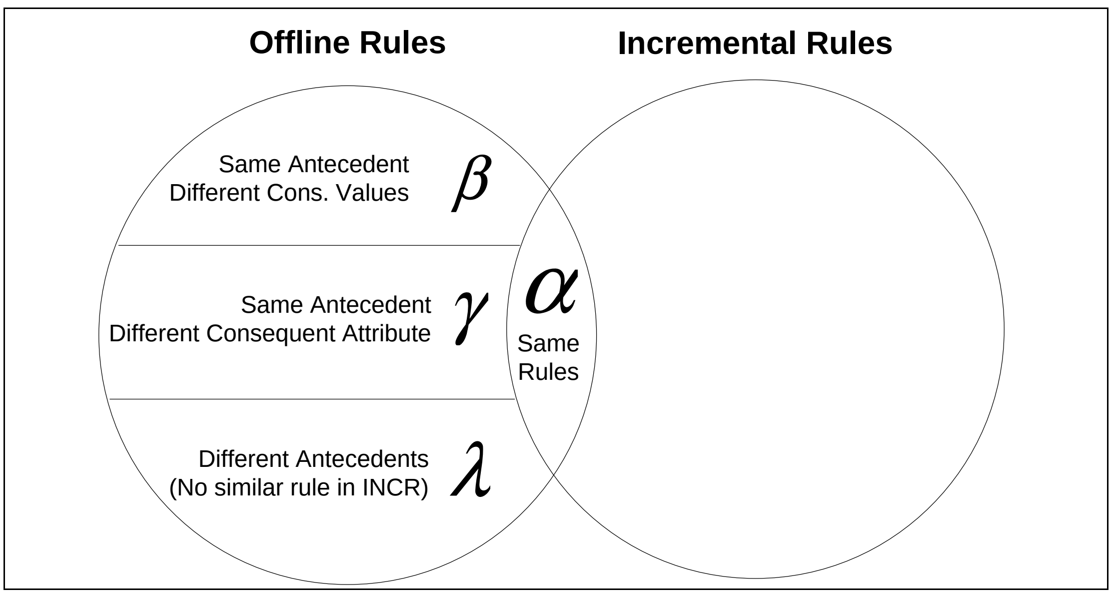

Fig. 14: Possible outcomes of rule comparison

After setting $B = 2$ (discussed in section 4.4), the performance difference between INCR and OFF is statistically insignificant but still present. To understand the reason for the (insignificant) difference, we analyze the sizes of α, β, γ and λ, which are listed in **Table 9**. This can be thought of as looking at the general structure of rules.

Table 9: Average rule set sizes (as percent of total number of OFF rules)

Dataset	α	β	γ	λ	ΔAUC
UNIV	17%	5%	19%	60%	-0.00002
FIT/UTK	2%	1%	3%	94%	-0.00034
LL TCP	24%	5%	22%	49%	0.00001
UNM	27%	14%	21%	38%	0.00034
BSM	17%	5%	11%	66%	-0.00001

As expected, datasets with high α, β and γ values are the ones with smallest ΔAUC values. A large number of similar rules should naturally lead to similar performance, which is indeed the case with UNIV, LL TCP and BSM. UNM also has this property, but the ΔAUC value is large due to poor performance of OFF on that dataset. For FIT/UTK, the number of similar rules is small, due largely to a small amount of INCR rules generated overall. This results in a large difference in AUC values.

Table 10: Average size of final rule sets

Dataset	OFF Total	INCR Total
UNIV	62.9	33.7
FIT/UTK	21.1	5.8
LL TCP	70.7	69.3
UNM	46.8	44.2
BSM	22.0	14.7

Table 10 shows the total rules set size for INCR and OFF, averaged across all 10 experiment runs. Note that the total size of INCR rule set is consistently smaller than the OFF rule set. However, as shown in the previous section, the difference in accuracy is not statistically significant. INCR therefore allows for less overhead during detection, since less rules have to be applied to data when looking for anomalies. While it may seem counterintuitive that an incremental algorithm is performing better (in a sense) than an offline one at the same task, it is a perfectly legitimate state of affairs. This happens because INCR is doing more work and is processing better data than OFF. Information carried over from day to day by INCR, specifically sample sets and rules, is essentially a compressed version of what OFF is working with. Rules created from this data are better at describing the training set than those created by OFF. Since INCR rules are better at modeling the training set, less are needed to achieve the same detection accuracy.

4.6) How Rule Statistics Affect Performance

Having analyzed the general structure of rules, we next look at the statistics they carry. To determine just how close INCR rules are to OFF, their n, r and w values are compared. However, for the comparison to be accurate, rules being compared have to be of the same type. Since n values are dependent only on the antecedent attributes, rules from α, β and γ are used. Comparing r values only makes sense when looking at rules with the same consequent attribute, hence only those from α and β are compared. Finally, because w values describe the effectiveness of a rule as a whole, they can only be compared on rules from α. Note that the λ value is not relevant in comparing rule statistics and therefore ignored.

Because we are only analyzing subsets of rules created by INCR and OFF, a new metric is needed to measure their performance. The AUC is dependent on the activity of all rules, so in cases where λ is large (such as UNIV and BSM datasets), the rules that we are interested in have little influence over the AUC. To gather better performance figures for the rules in α, β and γ, we look at the performance of individual rules.

During detection, $Score(d)$ is calculated for each record d (see section 3.1):

$$Score(d) = \sum_{r_i \in R} t_i w_i \frac{n_i}{r_i}$$

If the score is over a certain threshold, an alarm is triggered. Alarms triggered during attacks are known as detections, or DETs, while those triggered during normal activity are false alarms, or FAs. For each DET or FA triggered on record d, the contribution of each rule r_i is measured by:

$$Contribution(d, r_i) = \frac{t_i w_i \frac{n_i}{r_i}}{Score(d)}$$

Then for each rule r_i, all contributions to detections across the whole dataset are added into DET_{TOTAL}, with all false alarm contributions similarly added into FA_{TOTAL}. The performance of a rule is then gauged by the number of net detections, or ND:

$$ND = DET_{TOTAL} - FA_{TOTAL}$$

Having an exact number that represents how well a single rule is behaving allows us to directly examine the effect that n, r and w have on performance. Since we are interested in the difference in performance between INCR and OFF, the discrepancy between values is used:

$$Discrepancy_X = \frac{X_{INCR} - X_{OFF}}{X_{OFF}}$$

where X is either ND, n, r or w. Discrepancy is normalized in order to bring the performance of all datasets onto a level playing field. Furthermore, discrepancy of ND values is absolute, as any values away from zero are "bad", since we want to show error.

Fig. 15 shows the discrepancies between ND (Y axis) and n (X axis) values of comparable rules, for each dataset. For datasets with multiple applications (UNM and BSM), the largest one was used as a representative.

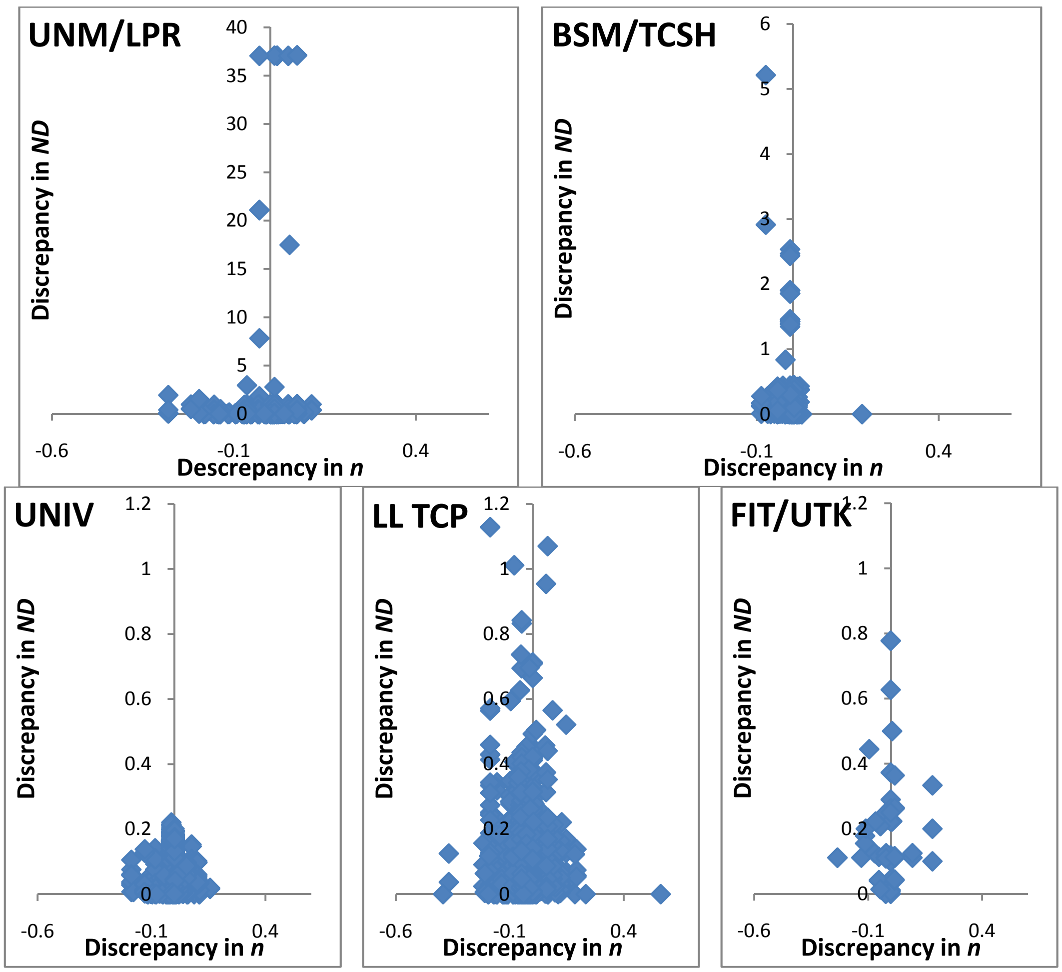

Fig. 15: B=2, discrepancy in n versus discrepancy in ND

The UNM dataset has a large discrepancy due to the fact that OFF did not produce any detections with 0.01% false alarm rate. There were still a number of similar rules that could be compared (see **Table 9**), but only INCR ones had detection scores, resulting in very high performance differences. While the opposite was true with the FIT/UTK dataset, there were far less rules generated this time. Because there were not even a handful of similar rules between runs (see **Table 9**), the scatter plot is far less dense than the other four.

In general, values that are on the negative X axis indicate under-estimation, while those on the positive X axis show over-estimation. The closer X values are to zero, the more similar INCR is to OFF. **Fig. 15** shows that n is equally under and over estimated, which is expected, since n is the value most helped by introducing $ScalingFactor$ (see section 3.3).

Fig. 16 presents discrepancies between r (X axis) and ND (Y axis) values of comparable rules, for each dataset. As before, one application was picked from UNM and BSM to represent the entire dataset.

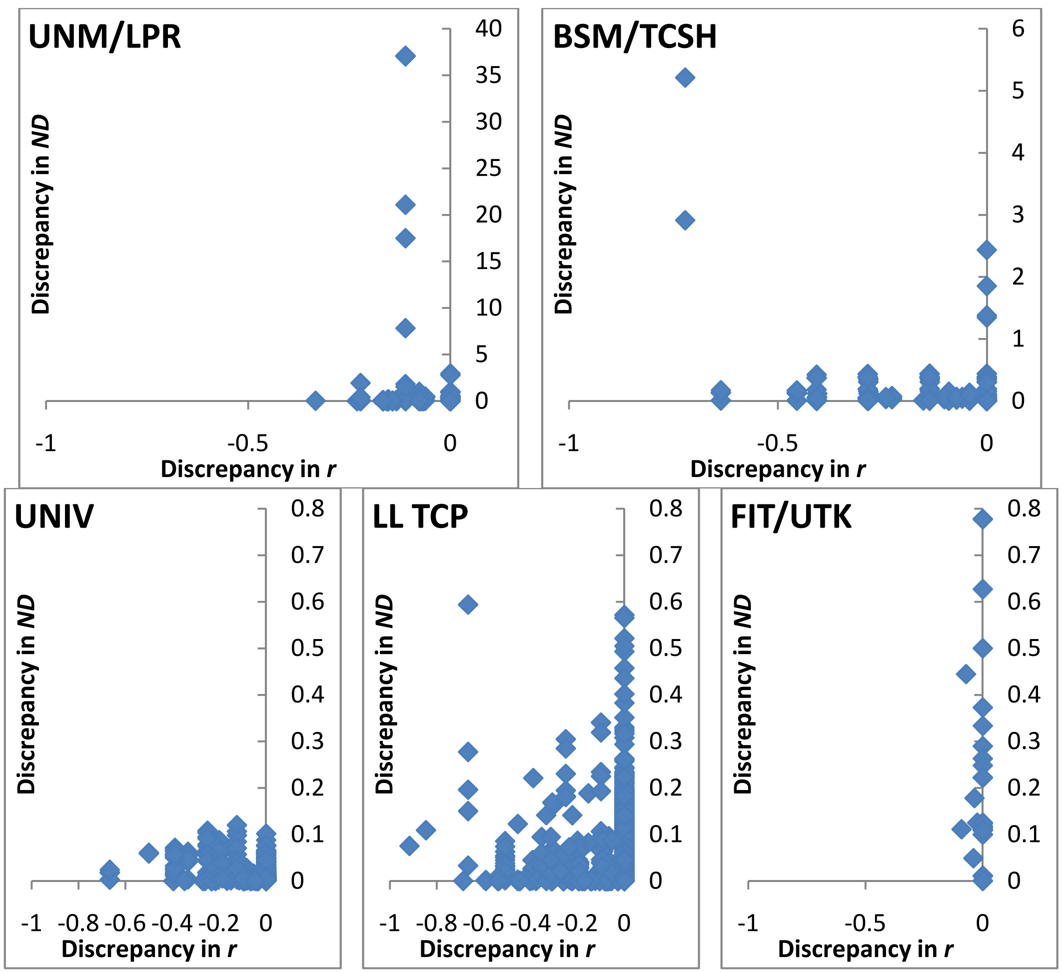

Fig. 16: B=2, discrepancy in r versus discrepancy in ND

This time, there are no points on the positive X axis, indicating that r values are never overestimated. This makes sense, since the r values are not “helped” in any way in INCR. Being exposed to the same (or less) training data, they cannot be overestimated.

Fig. 17 displays discrepancies between w (X axis) and ND (Y axis) values of comparable rules, for each dataset. Again, UNM and BSM are represented by a single application each.

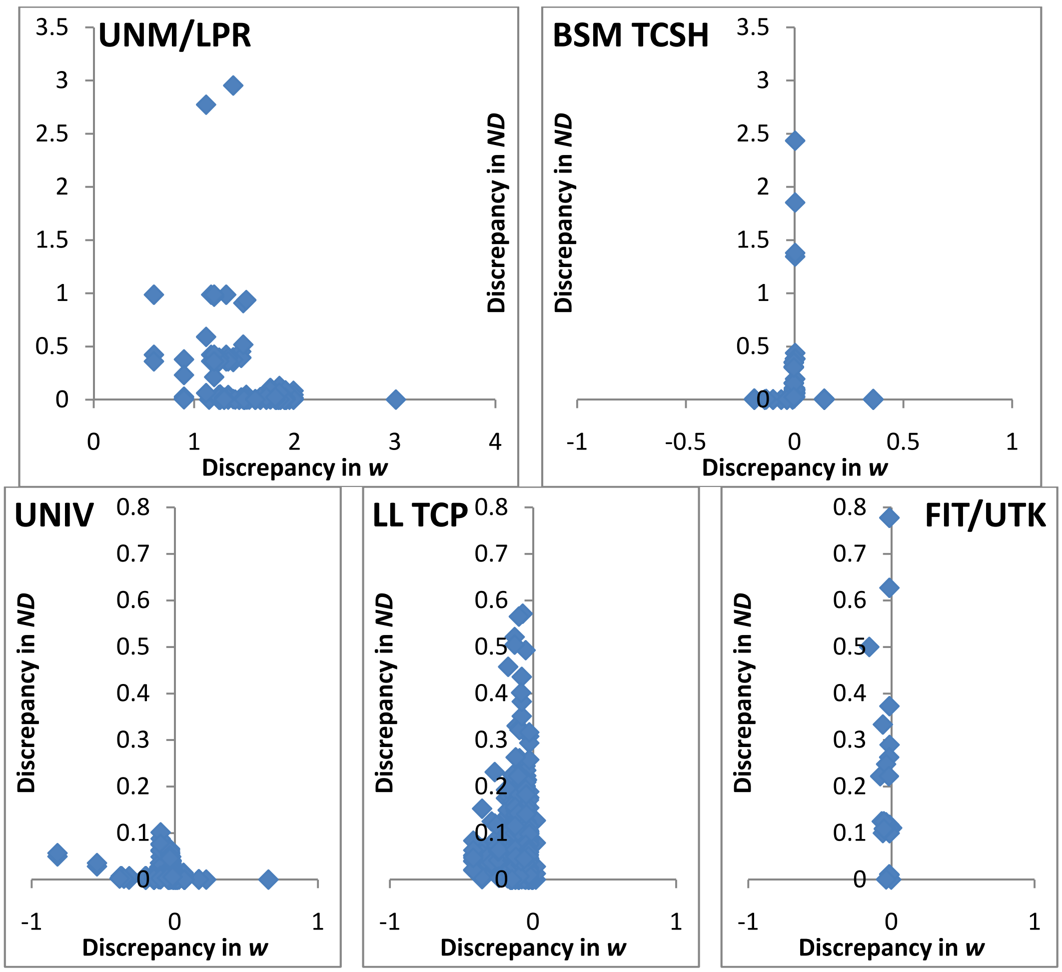

Fig. 17: B=2, discrepancy in r versus discrepancy in ND

In this case, points are on the negative X axis most of the time (UNIV, LL TCP, FIT/UTK), but sometimes on the positive side (UNM, BSM). This indicates that w is mostly underestimated, but can be overestimated at times. Or, put another way, our approach is not the most effective at keeping w statistics. This is not surprising, since $ScalingFactor$ in section 3.3 helps mostly n values, and only affects w values somewhat.

The problem with scatter plots is that they do not good at showing distributions. When many points are clustered over one another, the blob on the chart looks the same whether there are 100 or 1000 points in it. To help visualize the distribution of error, the points from previous figures were averaged together in “buckets”. There are 10 buckets between -0.38 and -0.04, one bucket between -0.04 and 0.04, and 10 buckets between 0.04 and 0.38. For each chart, all points that fall into a certain bucket are averaged together and that value is plotted in bar form on the next three charts. Then each bar represents the average value of all points in that area.

In **Fig. 18**, the average values of n versus ND are presented.

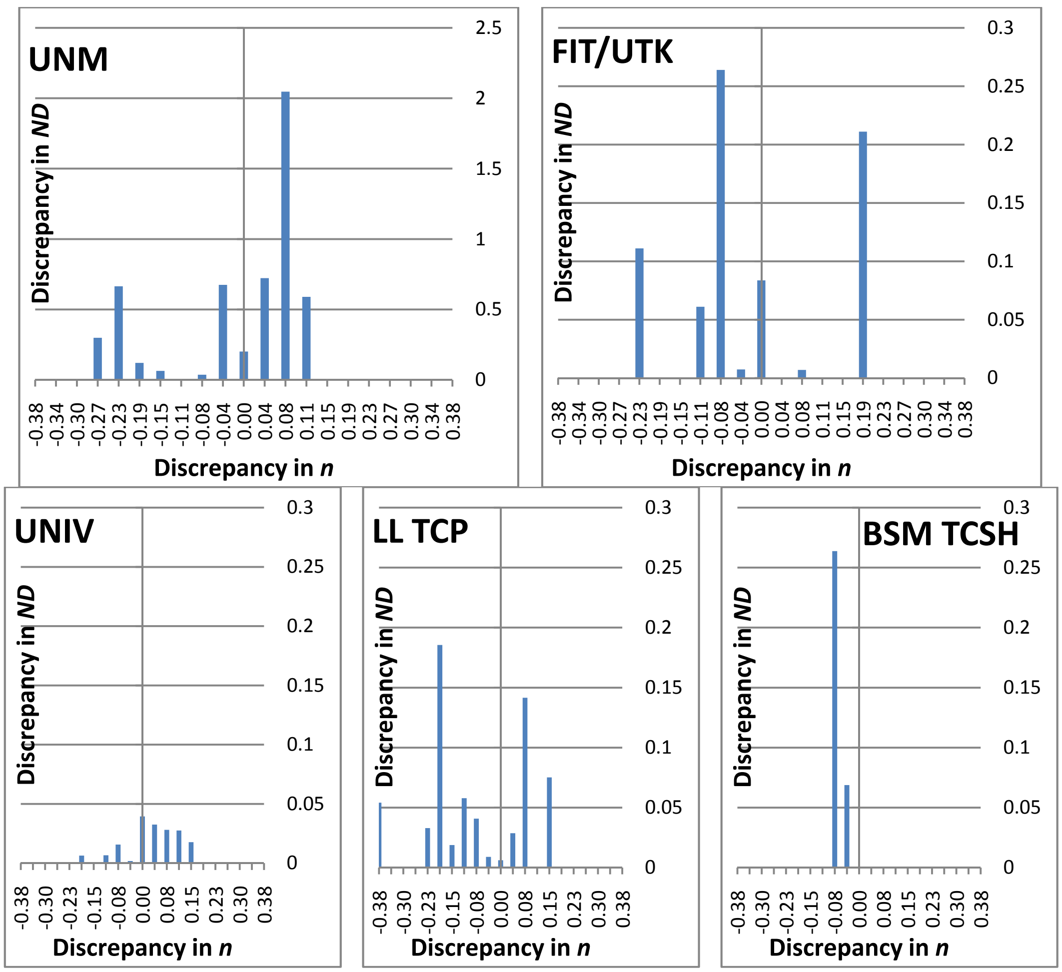

Fig. 18: B=2, average discrepancy in n versus average discrepancy in ND

While a few datasets have fairly well-defined trends (LL TCP, UNM), the trends are weak for the most part. However, the height of bars can also be used to determine relationship between statistics and ND. The higher a bar is, the more error is concentrated in that area. Overall, n produces error bars that are generally under 0.3 for most datasets.

Fig. 19 contains average values of r versus ND.

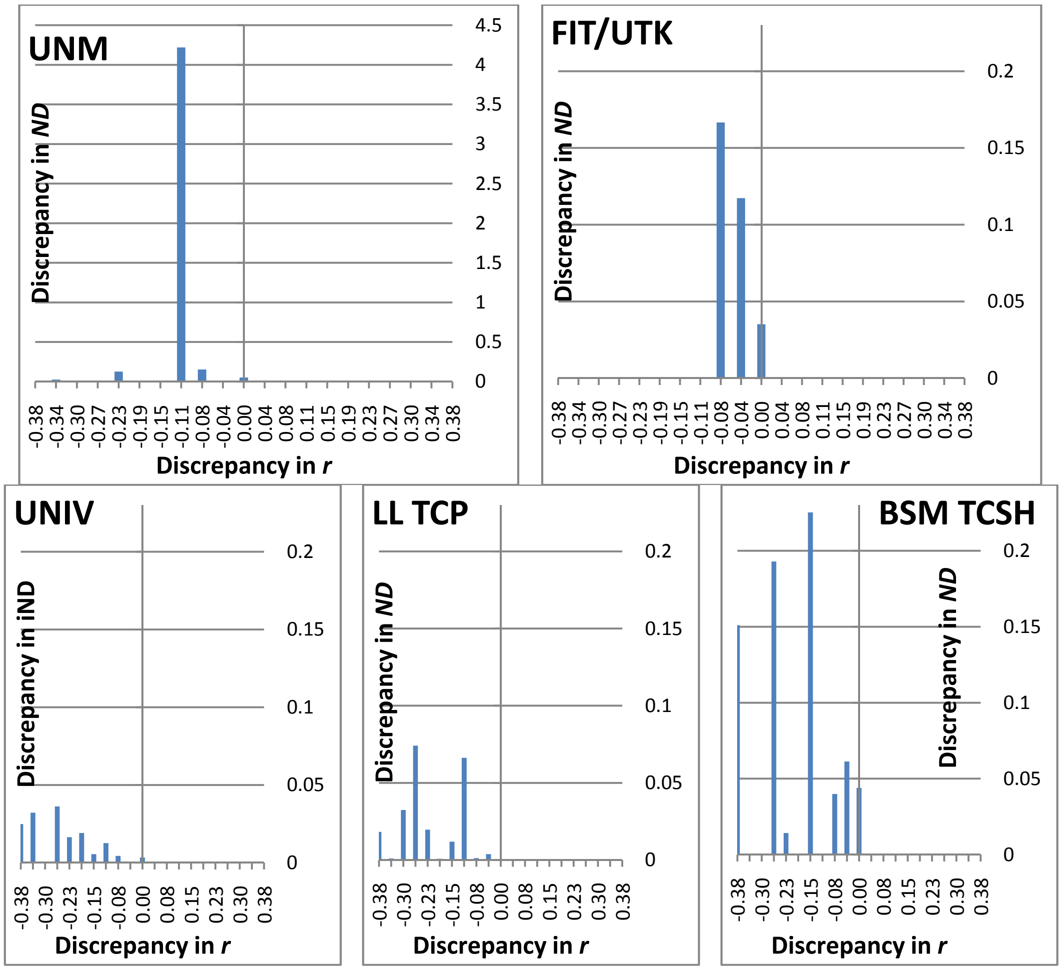

Fig. 19: B=2, average discrepancy in r versus average discrepancy in ND

In this case, UNIV and FIT/UTK show an upward trend from zero that would be expected if r values directly influenced ND values. In general, the error bars are usually below 0.2, which is lower than in **Fig. 18**. Lower bars indicate that error is less concentrated in any particular area, meaning r values are less responsible for NDs. Note that because the buckets are fixed, some points lay outside of the graph and are not included.

Finally, **Fig. 20** shows the average discrepancy in w compared with the average discrepancy in ND.

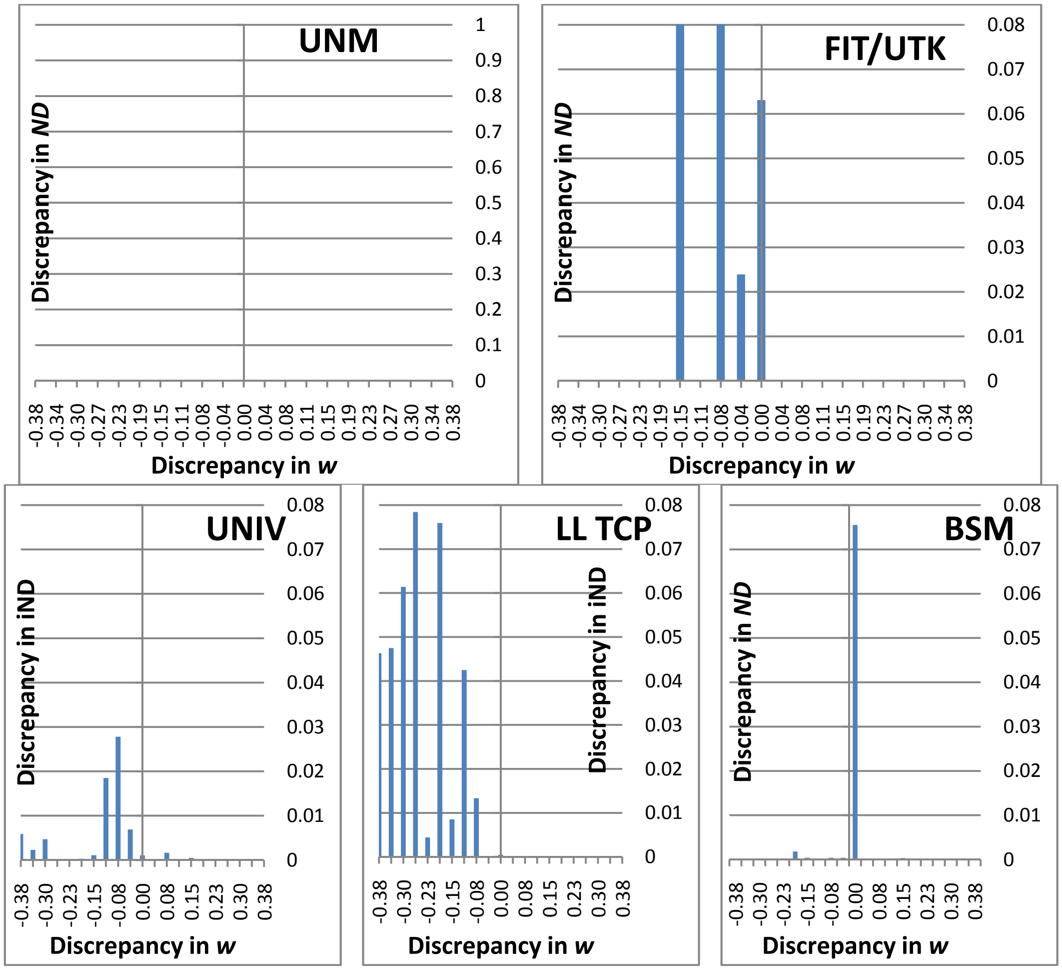

Fig. 20: B=2, average discrepancy in w versus average discrepancy in ND

Because all points in UNM lay outside of thes fixed buckets, they do not show up in **Fig. 20** at all. With the other datasets, most points are located between -0.4 and 0.4, and therefore are displayed. Once again, trends that would suggest a correlation between w and ND are not clearly present. However, since most bars are below 0.08, we can conclude that w has the least effect on ND.

The previous three charts showed that n is most responsible for the error between INCR and OFF, followed by r and finally w. This supports our choice of $ScalingFactor$ to help statistics in section 3.3, since it mostly helped the n values.

5) Conclusions

5.1) Summary of Findings

We introduced an incremental version of the LERAD algorithm, which generates rules before all of training data is available, improving them as more data is analyzed. Because rules are generated and updated incrementally, we can perform anomaly detection with an updated model more often, for example every day.

The incremental nature of our algorithm does not affect performance. Experiments show that after processing the same amount of data, the difference between the accuracy of incremental and offline algorithms is statistically insignificant.

Furthermore, although the incremental algorithm has similar accuracy to offline, it generates fewer rules (see **Table 10**). This leads to lower detection overhead, since fewer rules have to be applied to the same data to produce the same accuracy.

Experiments showed that the initial version had some rules similar to OFF, but their statistics were different and performance overall was worse than OFF. The total number of rules was also much larger, leading to a higher level of noise, which drowned out some legitimate detections. Improvements were made to INCR to reduce the differences in generated rules and performance.

To enable the incremental version to have the same performance as offline, several improvements had to be implemented. Because rules generated on later days are only exposed to the sample sets from previous days, their statistics about previous data were skewed. To combat this, the structure of sample sets was changed to include how many training records each sample set tuple represents. Then after they are first generated, rules are trained on virtual sets that roughly represent the training sets of previous days. This approach improved the quality of rule statistics and brought them much closer to the offline version, with experiments showing an increase in accuracy. The performance boost is expected, since improved statistics decreased Δn and Δw values, both of which were shown to have an effect on performance difference (see Section 4.6).

Due to the fact that INCR has m chances to generate rules when processing m training days, it consistently generated far more rules than OFF. This was a problem because a larger number of rules was creating much more noise during scoring, drowning out legitimate detections. To fix it, we introduced a method to prune rules after all training data is processed, based on the number of times said rules were generated (or "born"). Since there was not obvious way of picking the pruning threshold B, experiments had to be performed on all values, to pick the one that resulted in the lowest ΔAUC.

Experimental results showed this approach yielded performance differences from offline that were small enough to be statistically insignificant.

5.2) Limitations and Possible Improvements

The current approach to calculating statistics from previous days is beneficial to n, somewhat beneficial to w and not at all relevant to the r value. The distribution of consequent values would have to be modeled somehow to help r values, and to a lesser degree, ws. Furthermore, rare records that make it into the sample set have the same weight on rule statistics are those that occur frequently. This can be remedied by recording the probability of encountering each row in the training set prior to adding it to the sample set. Finally, the pruning threshold value B is currently picked by performing a sensitivity analysis. Further analysis is needed to deduce the proper value directly from training data.

Because the focus was on accuracy, incremental LERAD is geared towards producing the same results as offline LERAD. Some changes need to be made before it can be used in an online fashion. First, sample sets are currently carried over for all days, regardless of the actual day count. This approach is well suited for producing the same results as offline after processing the same amount of data, but it will be problematic when used in the real world. The growth of the sample set needs to be constrained at some point, before it grows larger than the available memory. One approach to resolving this issue is to only keep a certain number of sample sets around, removing (or "forgetting") those that get too old. For example, after processing data from chunk *k*, all sample sets gathered from chunks 1 to *k-7* can be removed, resulting in the last 7 sample sets being used at any given time. This would fix unconstrained sample set growth. However, this issue affects more than just the sample sets. Rules themselves also have this problem, since they are only pruned after the last day. A more useful approach would be to apply rule pruning after some set number of days. For example, pruning can be evoked each time 7 chunks of data are processed. Naturally, these suggestions are not ideal and further work is needed to determine the best solution.

References

Agrawal, R., & Srikant, R. (1994). Fast algorithms for mining association rules. In *Proc. 20th Int. Conf. very Large Data Bases, (VLDB),* 487-499.

Chandola, V., Banerjee, A., & Kumar, V. (2008, to appear). Anomaly detection: A survey. *ACM Computing Surveys.*

Clark, P., & Niblett, T. (1989). The CN2 induction algorithm. *Machine Learning, 3*(4), 261-283.

Cohen, W. W. (1995). Fast effective rule induction. In *Proceedings of the Twelfth International Conference on Machine Learning,* 115-123.

Das, K., & Schneider, J. (2007). Detecting anomalous records in categorical datasets. In *Proceedings of the 13th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 220-229.

Feng, H., Kolesnikov, O., Fogla, P., Lee, W., & Gong, W. (2003). Anomaly detection using call stack information. In *Proceedings of Symposium on Security and Privacy.* 62-75.

Forrest, S., Hofmeyr, S., Somayaji, A., & Longstaff, T. (1996). A sense of self for unix processes. In Proceedings of *1996 IEEE Symposium on Security and Privacy.* 120-128.

Gates, C., Taylor, C. (2006). Challenging the Anomaly Detection Paradigm - A provocative discussion. In *Proceedings of the 2006 workshop on New security paradigms*, 2006. 21-29.

Kruegel, C., Mutz, D., Valeur, F., & Vigna, G. (2003). On the detection of anomalous system call arguments. *Proceedings of RAID, Lecture Notes in Computer Science, 2808*, 326-344.

Kruegel, C., & Vigna, G. (2003). Anomaly detection of web-based attacks. In *Proceedings of the 10th ACM Conference on Computer and Communications Security,* 251-261.

Lazarevic, A., Ertoz, L., Kumar, V., Ozgur, A., & Srivastava, J. (2003). A comparative study of anomaly detection schemes in network intrusion detection. In *Proceedings of the Third SIAM International Conference on Data Mining*, 25-36.

Lippmann, R., Haines, J. W., Fried, D. J., Korba, J., & Das, K. (2000). The 1999 DARPA off-line intrusion detection evaluation. Computer Networks, 34(4), 579-595.

Mahoney, M. V., & Chan, P. K. (2003). Learning rules for anomaly detection of hostile network traffic. In *Proc. of International Conference on Data Mining (ICDM),* 601-604.

Mazeroff, G., De Cerqueira, V., Gregor, J., & Thomason, M. G. (2003). Probabilistic trees and automata for application behavior modeling. Paper presented at the *41st ACM Southeast Regional Conference Proceedings,* 435-440.

Robertson, W., Vigna, G., Kruegel, C., & Kemmerer, R. A. (2006). Using generalization and characterization techniques in the anomaly-based detection of web attacks. In *Proceedings of the 13th Symposium on Network and Distributed System Security (NDSS)*

Shavlik, J. & Shavlik, M. (2004). Selection, combination, and evaluation of effective software sensors for detecting abnormal computer usage. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining,* 276-285.

Tandon, G. & Chan, P. (2007). Weighting versus Pruning in Rule Validation for Detecting Network and Host Anomalies. In *Proc. ACM Intl. Conf. on Knowledge Discovery and Data Mining (KDD)*. 697-706.

Wang, K., Cretu, G., & Stolfo, S. J. (2005). Anomalous payload-based worm detection and signature generation. In *Proceedings of RAID*, *Lecture Notes in Computer Science, 3858,* 227-246.

Wong, W. K., Moore, A., Cooper, G., & Wagner, M. (2005). What's strange about recent events (WSARE): An algorithm for the early detection of disease outbreaks. *Journal of Machine Learning Research, Vol 6*, 1961-1998.