ADMIT: Anomaly-based Data Mining for Intrusions

Karlton Sequeira [*]and Mohammed Zaki [†]
Computer Science Department
Rensselaer Polytechnic Institute, Troy, New York 12180
{sequek,zaki}@cs.rpi.edu

ABSTRACT

Security of computer systems is essential to their acceptance and utility. Computer security officers use intrusion detection systems to assist them in maintaining computer system security. This paper deals with the problem of differentiating between masqueraders and the true user of a computer terminal. Prior efficient solutions are less suited to real time application, often requiring all training data to be labeled, and do not inherently provide an intuitive idea of what the data model means. Our system, called ADMIT, relaxes these constraints, by creating user profiles using semi-incremental techniques. It is a real-time intrusion detection system with host-based data collection and processing. Our method also suggests ideas for dealing with concept drift and affords a detection rate as high as 80.3% and a false positive rate as low as 15.3%.

1. INTRODUCTION

Security of computer systems is vital to their utility and acceptance. It is maintained by monitoring audit logs. The amount of information passed over networks and the very size of these networks has been increasing exponentially. The quantity of traffic makes it mandatory for network administrators and security experts to use specific tools, called intrusion detection systems (IDS), to prune down the monitoring activity. According to the 2000 Computer Security Institute/FBI computer crime study, 85% of the 538 companies surveyed, reported an intrusion or exploit of their corporate data, with 64% suffering a loss [13]. Thus, IDS are becoming increasingly important.

For an IDS to be powerful it must run continually, be adaptable to user behavior changes, be fault tolerant (i.e. crashes must not require retraining or relearning of behavior), be impervious to subversion, be scalable, should impose minimal overhead, be configurable, and should show graceful degradation of service[20]. Other challenges to contend with include, determining what audit data to collect and what data model to use to represent it, dealing with noisy, high-dimensional, categorical audit data, and satisfying generic requirements like automation and real-time detection [11].

The specific problem we seek to solve is that of differentiation between masqueraders and the true user of a computer terminal. We do so by augmenting conventional password authentication measures, with a continuously running terminal-resident IDS program, called ADMIT (**A**nomaly-based **D**ata **M**ining for **I**n**T**rusions), which monitors the terminal usage by each user and creates an appropriate profile and verifies user data against it.

1.1 Background

Intrusion detection systems are primarily of two types [10]: *signature-based*, in which audit data is searched for patterns known to be intrusive, or *anomaly-based*, in which aberrations from normal usage patterns are searched for. The former are susceptible to new attacks and hence reliance on these schemes is decreasing. While anomaly-based methods can detect new attacks, they are prone to higher false positive rates, as user behavior is often erratic and hard to model. Some methods [5] use a hybrid of the two to combine their benefits and counter their disadvantages.

Most IDS efforts so far have been targeted at network-level data [21], which does not solve our problem, since a poorly chosen password may be guessed, an authorized user may turn hostile or a terminal may be left unattended, allowing an intruder to gain access without using the network. Other efforts have been carried out using system call-level data [7, 14]. Early methods involved immunocomputing-based approaches by Forrest et al. [22] and recently Cabrera et al. [3], which examined fixed-length contiguous sequences of system calls, pertaining to privileged programs like sendmail, known to have exploits. In a related vein, Lee et al. [14], used a machine learning classifier, RIPPER, to produce rules to classify sequences as “normal” or “abnormal”. However, system-call level data is far too fine grained, which increases overhead.

Instead of network-level data, our method concentrates on *user command-level* data. In doing so, even if the masquerader gains access through whatever means, (s)he still runs the risk of being detected. In contrast, the risk of detection is much lower in IDS based on system-call or network level data. For instance, the intruder does not have to exploit the system calls or the network, having gained higher privileges.

Ryan et al. [18] has also suggested that every user leaves a

[*]This work was supported by Pitney Bowes, Inc.
[†]This work was supported in part by NSF CAREER Award IIS-0092978, and NSF Next Generation Software Program grant EIA-0103708.

print on the terminal, which could be picked up using artificial neural networks (ANNs). User-profile based endeavors include statistical-based methods such as IDES [5], NIDES [8], and EMERALD [16], which create multi-level usage profiles (i.e., at user or group levels). DuMouchel [6], created contiguous command sequence-based probability transition matrices, which serve as user profiles. Schonlau et al [19] test a variety of statistical methods for building user profiles.

Clustering is an unsupervised learning technique, in which data is partitioned into meaningful subgroups called clusters based on some similarity measure. Prior efforts using clustering for intrusion detection include those by Portnoy [17] and Zamboni [23]. Portnoy used non-real-time clustering to group unlabeled network data and labeled them based on the assumption that the proportion of network data that is anomalous is very low. Zamboni observed that the distribution of test points to clusters changes significantly at the time of attacks, which can be used as an indicator of anomalous behavior.

The work most closely related to ours is that by Lane and Brodley [12, 11], who used both instance-based learning [1] (IBL) as well as Hidden Markov Models (HMM) techniques to create user profiles for user command data. Like our method, they too use clustering, however only for model scaling (i.e., limiting the number of sequences representing the user). The IBL approach by Lane [11] provides the advantage that the class of a test point is based on the points most similar to it in the training database, rather than the similarity of all database points, thereby limiting problem complexity. Statistically-based anomaly-detection methods [17] often assume that anomalies form a very small proportion of audit data. But an intruder may use such "anomalous" sequences repeatedly, so that their frequency becomes high enough frequency to be labeled "normal". However, as the IBL approach is not statistically based, even a single labeled point is useful and using this ploy is futile. At the same time, a coincidence may result in contamination and hence expert supervision is essential. Also, since nearly all computation in IBLs takes place at run time, it is a slow alternative. Also, IBLs do not inherently provide a method of model scaling.

Most of the IDS efforts, thus far, require substantial training data, which must all be labeled. Also, the more successful efforts are less suited to real time application; the experiments conducted suggest that a considerably large sequence of anomalous commands is crucial to the performance. Our method suggests an environment in which these constraints may be relaxed. Also, unlike our method, all the models mentioned so far such as HMMs, ANNs, etc., do not intrinsically convey information about the user behavior to the security officer.

1.2 Contributions

It is our belief that current IDS techniques are not sophisticated enough to completely automate the intrusion detection problem, and that expert supervision is ultimately necessary. Hence, we try to minimize the work of the intrusion analyst by providing likely alarms, rather than completely automating the process.

ADMIT is a user-profile dependent, temporal sequence clustering based, real-time intrusion detection system with host-based data collection and processing. Using user profiles makes it hard for intruders who have gained access to higher privileges, to actually use them, since doing so would most probably create a usage pattern different from the true user's profile. The new pattern would be labeled as an anomaly and thus give them away.

There are several reasons why we used clustering to model the user behavior. Firstly, it affords us automatic model scaling. A cluster, with low variance, can be efficiently represented by its center. Hence the run time constraint in IBLs is relaxed depending on the *cluster support*, i.e., the number of sequences assigned to a cluster. Secondly, by putting a constraint on the cluster support, we can reduce noise and retain more relevant clusters. Thirdly, if the *intra-cluster similarity threshold* (i.e., the minimum acceptable similarity between a cluster's center and any other sequence assigned to it) is set high enough, some test sequences may remain unassigned. These sequences are said to be "anomalous" and hence an alarm (of type A) may be raised. In general, different types of alarms can be raised; Type A refers to a single anomalous sequence, while Type B refers to a sequence of type A alarms in near succession. The profile may be updated by clustering the anomalous sequences. Finally only the centers of the anomalous clusters are displayed to the analyst, thereby effecting significant data reduction as compared to the IBL scheme [11] in which all flagged sequences are offered to the security officer.

In designing ADMIT we preferred host-based processing and data collection. While centralized IDS architectures [10] offer lower processing overhead on hosts and easier defense against subversion (just one component to protect), they are difficult to dynamically reconfigure and do not scale. Also, the penalty is very high if subverted. ADMIT, like other distributed IDS [20, 15] does not have these problems and by having a host-based architecture, distributed-related co-ordination problems and sniffing-related methods of user spoofing are eliminated. A preview of the ADMIT architecture is presented in the next section.

2. ADMIT ARCHITECTURE

There are two main stages in our approach to mining intrusions. In the training phase the user profiles are created, and in the testing phase the user command stream data is verified against the corresponding profile. The complete architecture of ADMIT is shown in Figure 1.

User data enters the system by the monitoring of UNIX shell command data [11], captured via the (t)csh history file mechanism. A recognizer for the (t)csh command language parses user history data and emits them as a sequence of tokens in an internal format. Also, all commands between logging on and logging off are referred to as a session and corresponding delimiters (*SOF* and *EOF*, respectively) are inserted to indicate the same. We process data within different sessions separately to avoid patterns created by coincidence across sessions from being incorporated into the user profiles. An example session could be: **SOF*; ls -l; vi t1.txt; ps -eaf; vi t2.txt; ls -a /usr/bin/*; rm -i /home/*; vi t3.txt t4.txt; ps -ef; *EOF** (in the real data, instead of a ';', we have a new line character).

During training, the commands entered by a user are stored in that user's audit data table according to the time of entry. During testing, the command data is directly used to detect anomalies via online sequential classification.

We consider each process/command in the history data together with its arguments as a single token (*SOF* and

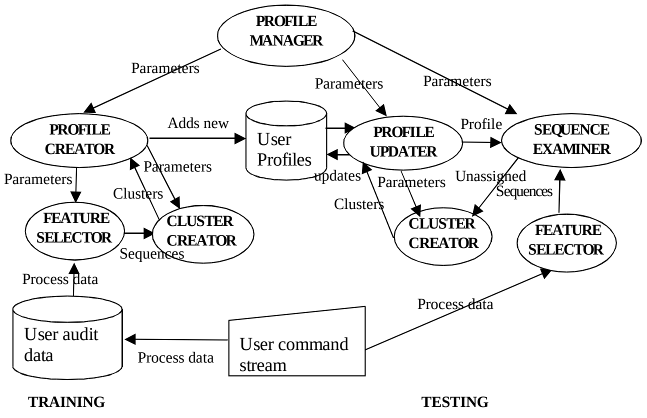

Figure 1: ADMIT's Architecture: Training/Testing

EOF are not considered to be tokens). However, to reduce the alphabet size, we omitted filenames in favor of a file count as in Lane [11]. For example, the user sequence given above is converted to the following set of tokens $T = \{t_i : 0 \leq i < 8\}$, where t_0= ls -l, t_1= vi <1>, t_2= ps -eaf, t_3= vi <1>, t_4= ls -a <1>, t_5= rm -i <1>, t_6= vi <2>, and t_7= ps -ef. The notation <n> gives the number of arguments (n) of a command. For instance, the command *vi t1.txt* is tokenized as *vi <1>*, while *vi t3.txt t4.txt* as *vi <2>*.

ProfileManager is the top-level module in ADMIT; it is responsible for the security of a terminal. It has a number of deputies, whose operations it configures, coordinates and decides. *ProfileCreator*, during training, creates profiles for users authorized to use the terminal. During testing, *ProfileUpdater* updates profiles for those users, while *SequenceExaminer* examines each sequence of tokens of process data and determines if that is characteristic or not, of the user thought to have produced it. Based on the decision, it takes action specified by the ProfileManager.

The ProfileCreator and ProfileUpdater both use two submodules. *FeatureSelector* parses the source command data for a user, cleans it up by replacing argument names by numbers (e.g., *cat test.txt > sort* becomes cat <1> > sort) and converts it into tokens as described above. It then converts the token data from each session, into sequences of tokens, whose length is specified by the ProfileManager. *ClusterCreator* converts the input array of sequences into clusters which form the profile of the user they originate from.

At startup, the ProfileManager initializes the ProfileCreator, ProfileUpdater and SequenceExaminer with various parameters. At training time, the ProfileCreator instructs FeatureSelector, as to what data to parse, and how to clean and tokenize it. The FeatureSelector makes sequences out of the tokens, whose length is determined by the ProfileManager. Thus, *a sequence s, of specified length l, is a list of tokens, occurring contiguously in the same session of audit data*, i.e., $s \in T^l$, where T is the token alphabet.

The ProfileCreator also initializes ClusterCreator and passes on the clustering algorithm and similarity measure, specified by the ProfileManager, so that the ClusterCreator can convert the sequences into clusters which are added to the user profile. Thus, *a cluster c, is a collection of sequences of user-initiated command data, such that all its sequences are very similar to others within itself using some similarity measure Sim(), but different from those in other clusters.*

If the clusters have sufficiently high intra-cluster similarity and low variance, the entire cluster may be represented by the center sequence. We define the cluster *center* (denoted s_c) as the sequence having the maximum aggregate similarity to the other sequences within the cluster. That is, if $c = \{s_0, s_1, \ldots, s_{n-1}\}$ is a cluster with n sequences, then

$$s_c = \max_{s_i \in c}\{\sum_{j=0}^{n-1} Sim(s_i, s_j)\} \qquad (1)$$

The clusters make up the profile for a given user. *A profile p, is the set of clusters of sequences of user-initiated command data whose centers characterize the user behavior.* Thus, for user u,

$$p_u = \{c_i | (Sim(s_{c_i}, s) \geq r, \forall s \in c_i) \wedge (Sim(s_{c_i}, s_{c_j}) \leq r', i \neq j)\}$$

where r and r' are the intra-cluster and inter-cluster similarity thresholds, respectively, s_{c_i} is the center of cluster c_i, and $Sim(s_1, s_2)$ is the similarity between two sequences. More formally, $Sim : T^l \times T^l \to \Re$, where T is the token alphabet and l is the length of the sequences. We expect p_u to include several clusters to adequately capture the usage variations for user u. Having fine grained (but not overly so) clusters also allows us to reduce false alarms while testing. Profiles thus created by the ProfileCreator are added to the pool of user profiles P.

During testing, the SequenceExaminer instructs FeatureSelector to parse, clean and tokenize command stream data of the user u currently logged in and convert them into sequences. These sequences are matched against the corresponding profile p_u, obtained from the pool of user profiles P by the ProfileUpdater. An alarm of type A may be sent to the security officer for each anomalous sequence. In contrast to the ProfileCreator's ClusterCreator, only the anomalous sequences, deemed so by the SequenceExaminer, are clustered by the ProfileUpdater's ClusterCreator. These can be added to user u's profile p_u, at the behest of the analyst.

3. ADMIT ALGORITHMS

In this section, we use a running example to explain the methods used in ADMIT during training and online testing.

To match command sequences against user profiles, it is essential to establish a similarity measure, which we write as $Sim(s_1, s_2)$, between two categorical command sequences, s_1 and s_2. It is an atomic function of the clustering and classification process and hence should be quick. Also, it is important to note that, we consider all commands to be of equal importance. The following similarity metrics have been proposed in the literature [11]: 1.) MCP (Match Count Polynomial Bound) counts the number of slots in the two sequences for which both have identical tokens and the count is the similarity score for the two sequences. For example, if s_1 = { vi <1>, ps -eaf, vi <1>, ls -a <1>}, and s_2 = {vi <1>, ls -a <1>, rm -i <1>, vi <2>}, then MCP for s_1 and s_2 is 1 since they are identical only in slot 1. 2.) MCE ,(Match Count Exponential Bound) is a variant of MCP, in that it doubles its value for each matching position. 3.) MCAP/MCAE (Match Count with Adjacency Reward and Polynomial/Exponential Bound) is a variant of MCP/MCE [11], where adjacent matches are rewarded. Lane [11] reported that MCAP is typically better than the others, so we use that in our study.

In addition, we proposed using the LCS metric (Longest Common Subsequence), which gives the length of the longest subsequence of tokens that the two sequences have in common. It too is polynomially bounded in the length l, i.e. $O(l^2)$ [4]. While LCS is a quadratic algorithm in comparison to the exact match-based linear algorithms, it does represent similarity between phase-shifted sequences. For example, for the s_1 and s_2 from above, LCS is 2, since they both share the subsequence vi <1>, ls -a <1>. The same result may be achieved using edit distance dot-plots with the length of the longest diagonal corresponding to the similarity between the sequences and other sequence alignment algorithms. Note that all of the proposed metrics produce a similarity measure having a whole value, i.e., $Sim : T^l \times T^l \to [0, l]$. This might restrict the granularity of differentiation.

3.1 Dynamic Training

Initial user profiles in ADMIT are mined in the training phase from user commands at their host machine. There are three main steps during training: 1) data pre-processing, 2) clustering user sequences, and 3) cluster refinement. These steps are described in detail below.

3.1.1 Data Pre-processing

ADMIT collects user audit data, by monitoring the command history files of the users. We use the following history as a running example: *SOF*; ls -l; vi t1.txt; ps -eaf; vi t2.txt; ls -a /usr/bin/*; rm -i /home/*; vi t3.txt t4.txt; ps -ef; *EOF*

The FeatureSelector parses, cleans and tokenizes the audit data, within each session specified by the ProfileManager. The above command stream results in the token list, $T = \{t_i : 0 \leq i < 8\}$, where t_0= ls -l, t_1= vi <1>, t_2= ps -eaf, t_3 = vi <1>, t_4= ls -a <1>, t_5= rm -i <1>, t_6= vi <2>, and t_7 = ps -ef.

The FeatureSelector next creates sequences of length l from the tokens based on the order of their creation time, as specified by the ProfileManager. For example, if $l = 4$, the set of user sequences is given as $S = \{s_i : 0 \leq i \leq |T| - l\}$, where:

s_0 = { ls -l, vi <1>, ps -eaf, vi <1> }
s_1 = { vi <1>, ps -eaf, vi <1>, ls -a <1>}
s_2 = { ps -eaf, vi <1>, ls -a <1>, rm -i <1>}
s_3 = {vi <1>, ls -a <1>, rm -i <1>, vi <2>}
s_4 = {ls -a <1>, rm -i <1>, vi <2>, ps -ef }

3.1.2 Clustering User Sequences

Once tokens have been converted into sequences, we next cluster them using a suitable algorithm.

K-Means [9] is an often favored clustering algorithm because it allows reallocation of samples even after assignment and it converges quickly. The problem with basic k-means is that the random allocation of cluster centers reduces its accuracy. Also, k (number of clusters) and t (number of iterations) are hard to set to achieve a good clustering. During each iteration, k-means first assigns each point to the closest cluster center and then recalculates the cluster centers. The first step takes time $O(\delta k N)$, where δ is the cost of computing similarity between any two sequences, and N is the number of sequences to be clustered. Recalculating the cluster centers based on Equation 1 takes time $O(\delta n^2)$, and since there are k clusters, the time for the second step is $O(\delta k n^2)$ (with the simplifying assumption that all clusters have an equal number of points n). The cost of k-means per iteration is then given as $\delta k N + \delta k n^2 = O(\delta k(n^2 + N))$. Since there are t iterations, the total cost of k-means is $O(t\delta k(n^2 + N))$. Note that for the MCP and LCS metrics, $\delta = O(l)$ and $\delta = O(l^2)$, respectively.

We do not want to use any preset value of k for the different users in our system. Thus instead of the basic k-means approach, we use a dynamic clustering approach to group a user's sequences, where clusters are grown when needed, as shown in the pseudo-code below:

```
**DynamicClustering** $(r, S_u, p_u, S_u^c)$:
//$r$ is intra-cluster similarity threshold
//$S_u$ is a set of a user $u$'s sequences to be clustered
//$p_u$ is the user $u$'s profile, i.e., set of clusters
//$S_u^c$ is the set of user $u$'s cluster centers
1.  $S_u^a = S_u$ //set of user $u$'s "anomalous" sequences
2.  while $(S_u^a \neq \emptyset)$ //"anomalous" sequences exist
3.      select random $s_c \in S_u - S_u^c$ as new cluster center
4.      $c_{new} = \{s_c\}$ //i.e., initialize new cluster
5.      $S_u^c = S_u^c \cup s_c$
6.      for all remaining sequences $s_i \in S_u - S_u^c$
7.          if $(Sim(s_i, s_c) \geq r)$
8.              if ( $Sim(s_i, s_c') < Sim(s_i, s_c) \forall s_c' \in S_u^c - s_c$)
9.                  $c_{new} = c_{new} \cup \{s_i\}$
10.                 recalculate cluster center, $s_c$ for $c_{new}$
11.     $p_u = p_u \cup c_{new}$
12.     $S_u^a = S_u^a - c_{new}$
```

Consider how DynamicClustering works on our example sequences (with $r = 3$). Initially $S_u = S_u^a = \{s_0, s_1, s_2, s_3, s_4\}$, $p_u = S_u^c = \emptyset$. Within the while loop we pick a random sequence as the new center, say s_0. For all remaining sequences in $S_u - S_u^c$, where $S_u^c = \{s_0\}$, we compute similarity to the new center s_0. Using LCS as the similarity metric we get $Sim(s_1, s_0) = 3$ since *vi <1>, ps -eaf, vi <1>* is their LCS. For the other sequences we get: $Sim(s_2, s_0) = 2$, $Sim(s_3, s_0) = 1$, and $Sim(s_4, s_0) = 0$. Since s_1 passes the threshold, we add it to the new cluster to get $c_{new} = \{s_0, s_1\}$. Now the new $S_u^a = \{s_2, s_3, s_4\}$ and we repeat the while loop. After a few steps we may find that the profile is given as $p_u = \{c_0 = \{s_0, s_1\}, c_1 = \{s_2\}, c_2 = \{s_3, s_4\}\}$.

Let's look at the time complexity of DynamicClustering. The for loop (line 6) takes at most $O(N)$ time if there are N sequences. Also, if r is well-chosen (i.e., reasonably high), cluster reassignment drops considerably enough to make the assumption that line 10 executes $O(1)$ times for each point during the execution of the entire algorithm. The cost of incrementally recalculating the centers as a cluster grows in size from 1 to n is $\delta \times (1^2 + 2^2 + 3^2 + ... + n^2))$, i.e., $O(\delta n^3)$. Hence for k clusters, it will be $O(\delta k n^3)$. Finally, the while loop repeats $O(k)$ times, where k is the number of clusters. Thus DynamicClustering has complexity $O(\delta k(N + n^3))$. As an optimization we recalculate cluster centers lazily, for example, every time the cluster support doubles since the last recalculation. Lazy update improves the algorithm complexity to $O(\delta k(N + n^2))$, while at the same time ensuring that a center closely represents its cluster. Notice that this time is typically much better than k-means, since DynamicClustering is a one-pass algorithm. It also has the advantage that k need not be pre-set, it is found dynamically.

While clustering, we aim to produce clusters of high similarity and low variance to deter spoofing attempts and to allow a single sequence to represent the entire cluster. This calls for a large number of clusters, i.e., the number of clusters, $k = O(N)$. Thus, in general n is small and can be neglected. In this case the algorithm has complexity $O(l^2N^2)$ and $O(lN^2)$ for the LCS and MCAP similarity measures, respectively.

3.1.3 Cluster Refinement

The last step in training phase is refinement of the clusters found above. Although DynamicClustering counters all the basic k-means disadvantages, setting the intra-cluster similarity r may require experimentation. Also, a cluster may have a lot in common with another, i.e., sequences assigned to it are as close to it as they are to another cluster. There may also be denser sub-clusters within the larger ones. To tackle these problems, we improve the clustering by merging and splitting clusters as follows:

```
MergeClusters($r', p_u, S^c_u$):
//$r'$ is the inter-cluster similarity threshold
1. For each pair of clusters, $c_i, c_j$ in profile $p_u$, $i \neq j$
2.     if ($Sim(c_i, c_j) \geq r'$)
3.         $c_i = c_i \cup c_j$ //merge clusters
4.         Recalculate center for $c_i$
5.         $p_u = p_u - c_j$ //remove $c_j$ from profile
6.         $S^c_u = S^c_u - s_{c_j}$
```

```
SplitClusters($r, t_s, p_u, S^c_u$):
//$t_s$ is a splitting threshold support
1. For each cluster $c_i$ in profile $p_u$
2.     if ($cluster\_support(c_i) > t_s$)
3.         DynamicClustering($r + 1, c_i, p_u, S^c_u$)
4.         $p_u = p_u - c_i$ //remove $c_i$ from profile
5.         $S^c_u = S^c_u - s_{c_i}$
```

Assume that $p_u = \{c_0, c_1, c_2\}$ and $r' = 2$ from above. Let's see how MergeClusters works. For instance, using LCS, $Sim(c_0, c_1) = Sim(s_0, s_2) = 2$. In this case, the two clusters should be merged to get $c_0 = \{s_0, s_1, s_2\}$ and c_1 will be removed from the profile. Also, the center for c_0 becomes s_1. For clusters that have high support SplitClusters calls DynamicClustering to recluster them into smaller, higher density clusters.

In terms of time complexity MergeClusters takes $O(\delta k^2)$ time, while SplitClusters takes $O(\delta knk')$ time, where k' is the number of resultant clusters after splitting. The splitting algorithm splits only very large clusters; while it may produce many sparse clusters, we found empirically, that it still increases the probability of finding better clusters. The main advantage of these two methods are that they are faster than most other splitting and merging algorithms.

3.2 Online Testing

Once ADMIT creates user profiles it can be used to test for masqueraders. Unlike training, the testing must happen in an online manner as user sequences are produced. Testing consists of four main steps: 1) real-time pre-processing, 2) similarity search within the profile, 3) sequence rating, and 4) sequence classification (normal vs. anomalous). These steps are detailed below.

3.2.1 Real-time Data Pre-processing

Capture user-based process data in real time. We use the following user data as an example: *SOF*; vi t4.txt; vi t4.txt; vi t4.txt; ls -a /home/*; rm -i /home/turbo/tmp/*; ls -a /home/*; vi t2.txt t4.txt; ps -ef;*EOF*.

The FeatureSelector parses, cleans and tokenizes the audit data specified to get the token set $T' = \{t'_i : 0 \leq i < 8\}$, t'_0 = vi <1>, t'_1 = vi <1>, t'_2 = vi <1>, t'_3 = ls -a <1>, t'_4 = rm -i <1>, t'_5 = ls -a <1>, t'_6 = vi <2>, t'_7 = ps -ef.

Next the FeatureSelector creates sequences from tokens, using $l = 4$ as in training to get $S' = \{s'_i : 0 \leq i \leq |T'| - l\}$
s'_0 = {vi <1>,vi <1>,vi <1>,ls -a <1>}
s'_1 = {vi <1>,vi <1>,ls -a <1>,rm -i <1>}
s'_2 = {vi <1>,ls -a <1>,rm -i <1>,ls -a <1>}
s'_3 = {ls -a <1>,rm -i <1>,ls -a <1>,vi <2>}
s'_4 = {rm -i <1>,ls -a <1>,vi <2>,ps -ef}

3.2.2 Profile Search

For each newly created sequence, we compute the highest similarity value within u's profile (assuming for the moment, that these new sequences come from user u), i.e., for each sequence, we find the most similar cluster in p_u. Define the similarity between a sequence s'_i and a profile p_u as follows: $Sim(s'_i, p_u) = \max_{c_j \in p_u}\{Sim(s'_i, s_{c_j}\}$. For example, assume $p_u = \{c_0 = \{s_0, s^*_1, s_2\}, c_1 = \{s^*_3, s_4\}\}$ from section 3.1.3 (cluster centers are indicated with '*'). Then $Sim(s'_0, p_u)$ =-$\max(Sim(s'_0, s_{c_0}), Sim(s'_0, s_{c_1})) = \max(Sim(s'_0, s_1), Sim(s'_0, s_3))$ = $max(3, 2) = 3$. Similarly $Sim(s'_1, p_u) = 3$, $Sim(s'_2, p_u) = 3$, $Sim(s'_3, p_u) = 3$, and $Sim(s'_4, p_u) = 2$.

Although the search for the closest cluster takes time $O(\delta k)$, since we expect a user to have many clusters, one may use more efficient methods like those suggested in [2], where cluster centers may be clustered using K-means using an efficient data structure like a k-d tree to store the clusters in the profile and hence speed up profile search.

3.2.3 Sequence Rating

Simply using the similarity measure between the current sequence s'_i being evaluated and the profile p_u to determine the user authenticity is not advisable. The data is too noisy and a high false positive rate results in the absence of a filter. It is a good idea to approximate the user authenticity, based on the sequences seen so far. In other words, we use the past sequences to determine, if the current sequence is just noise or if it is a true change from profile. We call this process *sequence rating* and we use a number of possible rating metrics to reduce noise in our prediction, namely `LAST_n`, `WEIGHTED`, and `DECAYED_WEIGHTS`.

`LAST_n`: The arithmetic mean of the similarities of the last n sequences. It has finite memory and captures temporal locality present in user command stream. The rating R_j for the jth sequence is calculated as

$$R_j = \begin{cases} \frac{1}{n}\sum_{t=j-n+1}^{t=j} Sim(s'_t, p_u) & \text{if } j \geq n \\ \frac{1}{j+1}\sum_{t=0}^{t=j} Sim(s'_t, p_u) & \text{if } j < n \end{cases}$$

For the five new sequences, using this rating metric with $n = 3$, we would get the following ratings: $R_0 = R_1 = R_2 = R_3 = 3$, and $R_4 = 8/3 = 2.67$. The drawbacks of LAST_n are that it is hard to choose n and all the last n sequences are treated equally. As n increases, performance approaches that of the arithmetic mean of all sequences.

`WEIGHTED`: The weighted mean of the last rating and the current sequence's similarity. The the rating R_j for the jth sequence is calculated as

$$R_j = \alpha * Sim(s'_j, p_u) + (1-\alpha) * R_{j-1}$$

where $R_0 = Sim(s'_0, p_u)$. For example, if $\alpha = 0.33$, then $R_0 = R_1 = R_2 = R_3 = 3$, and $R_4 = 2.66$. In general, it is more sensitive than LAST_n, and one doesn't have to fix n. However, it is hard to choose an optimal weight ratio.

DECAYED_WEIGHTS: A variant of the weighted mean. Instead of using a constant α weight ratio, we vary it according to the sequence number. We thought of diminishing the sensitivity of the system as time passes. Doing this counters the effects of concept drift (i.e., shift in user profiles), which increases as time passes by, giving lesser sensitivity as the sequence *id* increases. The rating R_j for jth sequence is calculated as

$$R_j = \alpha_j * Sim(s'_j, p_u) + (1-\alpha_j) * R_{j-1}$$

Here weight varies with sequence id, and is given as

$$\alpha_j = \frac{\alpha_{j-1}}{\alpha_{j-1} + 1 - log(\frac{z}{y+j})}, \alpha_0 = 1$$

Thus α_j is a decaying weight as long as $1 - log(\frac{z}{y+j}) > 0$ As an example, if $y = 4100$ and $z = 7500$, then $R_0 = R_1 = R_2 = R_3 = 3$, and $R_4 = 2.66$.

3.2.4 Prediction: Normal vs. Anomalous

Once sequences have been rated, we need to classify them as either "normal", i.e., true user, or as "anomalous", i.e., a possible masquerader. This classification is based on the rating R_j for a given sequence s'_j.

Normal Sequences. We use a threshold value on the rating of a sequence to determine if it is normal or not. The *lower accept* threshold, T_{ACCEPT}, is the threshold rating for a sequence, which, if exceeded by the test sequence's rating, causes the system to label that sequence as having originated from the true user. It is generally an empirically selected value. A normal sequence is added to the profile p_u to the cluster it is the closest to. For example, with $T_{ACCEPT} = 2.7$, for WEIGHTED rating metric ($\alpha = 0.33$) no alarm will be raised for s'_0, since $R_0 = 3 > 2.7$. Similarly, s'_1, s'_2, s'_3 are all deemed to be normal; they are assigned to the nearest profile cluster, e.g., $c_0 = \{s_0, s^*_1, s_2, s'_0, s'_1\}, c_2 = \{s^*_3, s_4, s'_2, s'_3\}$, and cluster centers are recalculated lazily.

Anomalous Sequences. An anomalous sequence is one that doesn't pass the T_{ACCEPT} threshold (e.g., s'_4 is anomalous, since $R_4 = 2.66 < 2.7$). This may occur as the result of any one of three phenomena: 1) noise, e.g., from typing errors, randomness, etc., 2) concept drift, e.g., working on a different project, etc., and 3) masquerader, i.e., the one we want to detect. A lone anomalous sequence is most likely noise. A number of sequences which do not get assigned in near succession suggest a change in the behavior, and are more likely to be an intrusion or concept drift, as compared to evenly distributed anomalous sequences, which are more likely to be noise. The larger the number of anomalous sequences in near succession, the more suspicious the identity of the user. However, these sequences do not have to be contiguous, otherwise IDS spoofing, in which harmful commands are inserted between normal commands to confuse the IDS, would be possible. To get a better estimate of the type of the behavioral change (i.e., noise or otherwise), we use clustering of anomalous sequences on the basis of their sequence ids. Also, we would like to put off clustering anomalous sequences as far as possible, to better estimate the size of behavioral change. However as the size increases beyond a certain threshold $T_{cluster}$, we raise a different type of alarm, called type B alarm. We borrow from Zamboni [23], the idea of monitoring the rate of change of cluster production. A sharp increase in the rate indicates an intrusion.

Thus, incremental clustering of anomalous sequences is basically temporal locality mining. Informally, an anomalous cluster is a chain of anomalous sequences, such that the mean difference in the sequence ids of consecutive pairs is within r_i called the *incremental intra-cluster proximity threshold* and the maximum difference in the sequence ids of consecutive pairs is within r'_i called the *incremental inter-cluster proximity threshold.* The incremental clustering algorithm works as follows:

```
IncrementalClustering($s'_i, S''_a, r, r_i, r'_i, p_u, S^c_u$)
// $s'_i$ is an anomalous sequence
// $S''_a$ is the list of anomalous sequences
// $r_i$ and $r'_i$ are the intra and inter incremental cluster proximity thresholds
// $p_u$ is the profile we are updating incrementally
// $S^c_u$ is the set of cluster centers for $p_u$
// $r$ is intra-cluster threshold used in DynamicClustering
1. if the maximum difference in adjacent sequence ids of $S''_a < r'_i$ and the mean difference in list of sequences ids $< r_i$
2.     $S''_a = S''_a \cup s'_i$
3.     If $(|S''_a| > T_{cluster})$// the cluster threshold
4.         Raise an alarm of type B
5. else
6.     DynamicClustering(r,$S''_a, p_u, S^c_u$)
7.     $S''_a = \{s'_i\}$
```

In line 1, the if conditions maintain nearness between members of an anomalous sequence cluster. In line 2, anomalous sequences conforming to the constraints are added to the anomalous cluster. In line 4 we raise a type B alarm, if the cluster has grown beyond a threshold size. All such clusters are interpreted to be a significant change from profile. In line 6, since the cluster does not satisfy the constraints, the sequences within it are mined using DynamicClustering on the basis of the *Sim* function and added to the profile, and the s'_i is added to a new cluster. Consider how IncrementalClustering works on our example. Initially, $p_u = \{c_0, c_1\}, S''_a = \emptyset, r = 3, S^c_u = \{s_1, s_3\}$. Since $R_4 = 2.66 < (T_{ACCEPT} = 2.7)$, hence $s'_i = s'_4$. In line 2, s'_4 is assigned to S''_a. In line 6, $p_u = p_u \cup (c_3 = \{s'_4\})$ Note that in general, a sequence may get a different label depending on the rating metric used (for the same value of T_{ACCEPT}).

Thus, after testing the sequence stream S', the profile will become $p_u = \{c_0 = \{s_0, s_1, s_2, s'_0, s'_1\}, c_2 = \{s_3, s_4, s'_2, s'_3\}, c_3 = \{s'_4\}$

4. APPLICATION STUDY

In the discussion below SELF refers to the true user and OTHER to the masquerader. The system classifies a command stream as ACCEPT if it considers it to be from the true user (SELF), otherwise it classifies the stream as REJECT.

IDS are evaluated on the basis of accuracy, efficiency and usability [15] according to the following metrics: 1) *De-*

tection Rate gives the percentage of attacks that a system detects, i.e., the percentage of OTHER sequences that receives a rating below T_{ACCEPT} (or OTHER REJECT). 2) *False Positive Rate* is the percentage of SELF sequences that the system incorrectly determines to be intrusive, i.e., the percentage of SELF sequences that receive a rating below T_{ACCEPT} (or 100 - SELF ACCEPT). 3) *Time-To-Alarm (TTA)* [11] is the mean time to alarm generation, i.e., the mean number of sequences between two sequences that receive a rating below T_{ACCEPT}. It is a measure of the system's sensitivity and efficacy in detecting intrusions in real time. 4) *Data reduction* is a measure of the system's usability in terms of reducing the data the network security officer has to browse through.

As such high SELF ACCEPT and OTHER REJECT are desirable, as they indicate a low false positive rate and a high detection rate, i.e., high accuracy. A high TTA indicates that there is considerable time between alarms, which is desirable for SELF since SELF should not raise alarms, but is undesirable for OTHER.

For our experimental study, we use command stream data collected from eight UNIX users from Purdue University [11] over varying periods of time (USER0 and USER1 are the same person working on different projects). The time over which the data for each user was collected is not known, so we use the number of sessions as an indicator of time. Since there were 500 sessions for the user with the fewest sessions, we use the first 500 sessions from each user as our dataset. We further split the data for each user into five overlapping folds (i.e., blocks) of 225 sessions each (i.e., sequence numbers 0-224, 69-293, 138-362, 207-431, 275-499). Each of the folds is used independently of the others for testing and the results reported are the average over the five folds.

For training and testing, each fold of 225 sessions is further split into two parts, the first 125 sessions are used for training and the latter 100 for testing. In each fold, for each user, the system creates a profile of SELF by training on 125 sessions of SELF data. It then independently tests the first 250 sequences of the last 100 sessions of the corresponding fold, for all users, including SELF, against this profile. Unless otherwise indicated, we perform experiments using LCS similarity measure and using the `DECAYED_WEIGHTS` rating metric with y=6750 and z=7500. The intra-cluster threshold similarity $r = 3$, the sequence length l=5, the cluster support $t_s = 15$, and inter-cluster threshold $r' = 2$.

All the experiments assume that training data is labeled. However, this is not a hard requirement. After clustering the training data it can be labeled easily, with substantial decrease in the work of the security officer. This relaxes the requirements imposed by other methods, at no additional cost. In the following graphs, we plot the variation in ADMIT's performance as a function of its configuration parameters, generally varied one at a time.

4.1 Effect of Sequence Length

The accuracies resulting from different sequence lengths varied from user to user. Hence we report the mean of the readings of all users. We tested the performance for sequence length $l = \{2, 3, 4, 5, 6, 7, 8, 9, 10, 12, 15, 20\}$. A different T_{ACCEPT} was used for each value of l [1]. Also, the intra-cluster threshold similarity was set as $r = l - 2$ and the training data set sizes were 150 sessions long. Otherwise the experimental setup was identical to that mentioned above.

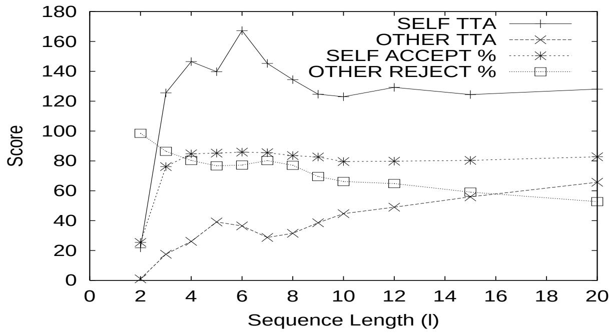

Figure 2: Effect of Sequence Length

As l decreases, the mean accuracy, i.e., the mean of SELF ACCEPT and OTHER REJECT, increases and OTHER TTA decreases as visible in Figure 2. However, for very small sequences l=2, SELF ACCEPT becomes very high, while OTHER REJECT drops below an acceptable level. This is because for all sequence lengths, $(l - r)$ is a constant viz. 2. Hence, as sequence length decreases, the ratio, $\frac{l-r}{l}$ increases. This ratio is more closely related to the amount of variance tolerable in the cluster than $(l - r)$. Ideally, we would like the ratio to be identical for all values of l, but r is a whole number. Thus, as the ratio increases, the variance increases and hence the model gets more generalized. For l=2, OTHER REJECT drops, as the model becomes too general.

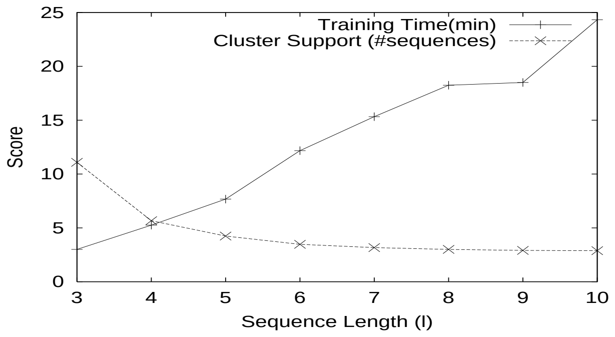

Figure 3: Sequence Length: Time, Cluster Support

As shown in Figure 3, mean cluster support also increases with smaller length, since the more general the model, the more the sequences that can be assigned to a user. Cluster support is crucial in determining the extent to which classification time improves over the IBL method, thereby increasing applicability to real-time use. Clustering time drops as fewer clusters are produced. Although mean accuracy for small values of l is very high, the difference between the SELF ACCEPT and OTHER REJECT percentages is cause for worry. Also, a cluster having a low value of $\frac{l-r}{l}$ implies unacceptably high variance within the cluster, thereby diminishing the capacity of the cluster center to represent the entire cluster. This variance was particularly disturbing as it increases the vulnerability of IDS spoofing. It was not possible to make the r vary sufficiently to minimize its effects on the performance, as it must be a whole value and variations in either direction introduced a considerable, rather than gradual, change in performance (see Section 4.5). Hence,

[1]Empirically we determined that $T_{ACCEPT} = 0.55 * l + 0.1$ gave us high accuracy for each length.

a suitable choice for sequence length with reasonably high mean accuracy and low difference in SELF ACCEPT and OTHER REJECT percentages to counter spoofing is 5. As we used LCS, for the similarity measure, the time for training does not increase linearly with sequence length, as evident from the graph.

4.2 Effect of Training Data Size

The training data set size is an indicator of the amount of concept drift in the user data. Learning from too much historical data may incorporate irrelevant concepts in the user profile. It also indicates the amount of training data required to create a satisfactory profile. T_{ACCEPT} was fixed at 2.75 for these experiments, and the training data set sizes used were {50, 75, 100, 125, 150, 200, 250, 275} sessions. As evident from Figure 4, as training data size decreases, SELF ACCEPT and SELF TTA decrease, since fewer concepts are being learnt. OTHER REJECT increases steadily for the same reason. The average of SELF ACCEPT and OTHER REJECT, i.e., mean accuracy tends to peak at about 125 sessions of training data and dip on either side.

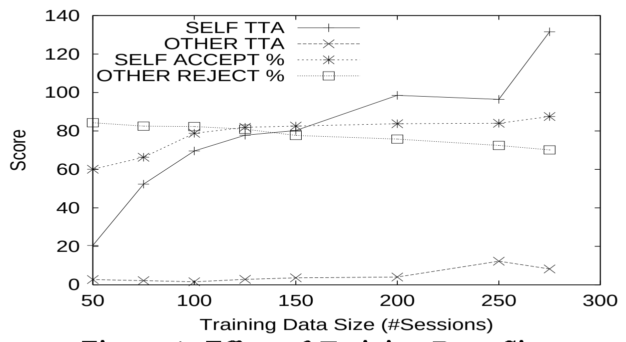

Figure 4: Effect of Training Data Size

For training sets of size greater than 125 sessions, there is a tendency to learn too many concepts (i.e., sequence clusters), all of which may not be relevant to the user currently due to the principle of temporal locality. The reverse happens for training sets of size less than 125 sessions. Again, a large difference between the SELF ACCEPT and OTHER REJECT is not good. Hence, 125 sessions is a suitable training dataset size for our data.

4.3 Effect of Sequence Rating Metric

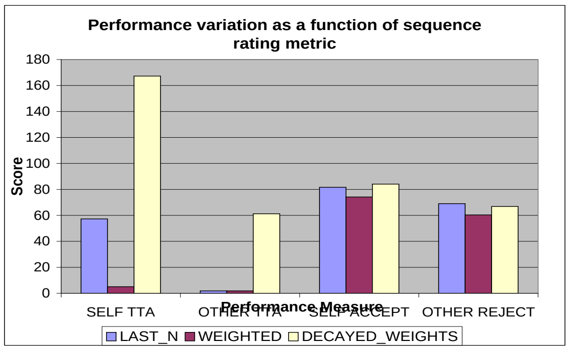

Figure 5: Effect of Rating Metric

Performance of different rating metrics depends on different T_{ACCEPT} values, so we use different values determined empirically. We tried out three of the methods described earlier viz. `LAST_n, WEIGHTED` and `DECAYED_WEIGHTS`.

For `LAST_n` we use $n = 20$, for `WEIGHTED` we set $\alpha = 0.05$ and for `DECAYED_WEIGHTS`, we used $z = 7500$ and $y = 6500$. From Figure 5 it is evident that the performance of `DECAYED_WEIGHTS` is the most satisfactory because although `LAST_n` has a lower OTHER TTA as compared to `DECAYED_WEIGHTS`, its SELF TTA is significantly smaller as well. However, the accuracy measures for `LAST_n` have a smaller difference than those for `DECAYED_WEIGHTS`. However, choice of the metric depends upon the security policy in place. For example, in a policy where security is premium and having a relatively high false alarm rate is tolerable, `LAST_n` is a good choice. In most other cases, `DECAYED_WEIGHTS` would be preferred.

4.4 Sensitivity Variation

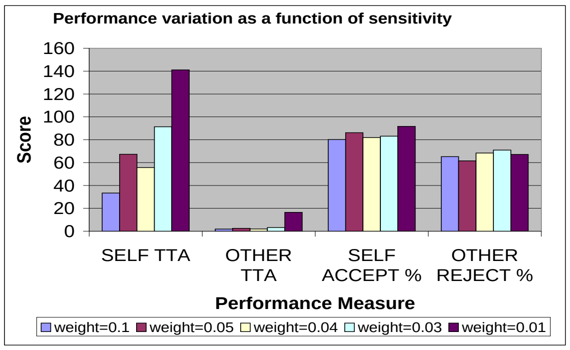

Figure 6: Effect of Sensitivity

Sensitivity is a critical feature of an IDS. It is an indicator of the response time of the IDS. Overly sensitive IDS responds to noise and hence has a high false positive rate. For different rating metrics, sensitivity varies dramatically. We prefer to define sensitivity in terms of the weight of current sequence's similarity to the profile in the current sequence's rating, i.e., for `LAST_n, WEIGHTED` and `DECAYED_WEIGHTS`, the sensitivity is $\frac{1}{n}$, α and α_j respectively. For example, for `LAST_n, WEIGHTED, DECAYED_WEIGHTS`, Weight = 0.01 would imply n=100, $\alpha = 0.01$, $\alpha_j = 0.01$ respectively. For these experiments, we fixed T_{ACCEPT} and use `LAST_n` rating metric.

Figure 6 shows that as the weight of the current sequence's similarity to the profile increases, SELF TTA and OTHER TTA decreases due to noise being logged by the IDS. A less sensitive IDS has a slow response time.

4.5 Effect of Intra-cluster Threshold

The intra-cluster similarity threshold r controls the amount of variance permitted within a cluster and hence it decides how tightly the profile fits the test data. We tested the performance for all possible values of r, i.e., 1, 2, 3, and 4 (since $l = 5$). Figure 7 shows that as the value of r increases, SELF ACCEPT increases and OTHER REJECT decreases. This is because the model progresses from over-fitting to becoming too generalized. Also important, is the steep rate at which it switches in performance at one value of r to the next, due to the whole values of the similarity metric chosen.

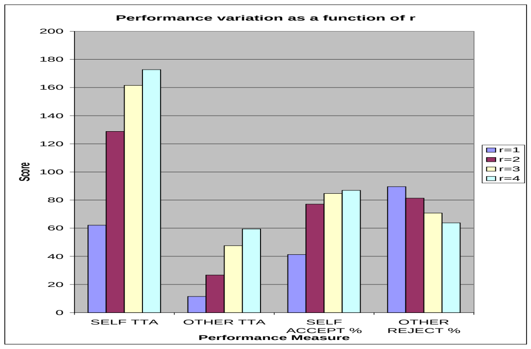

Figure 7: Effect of Intra-cluster Threshold

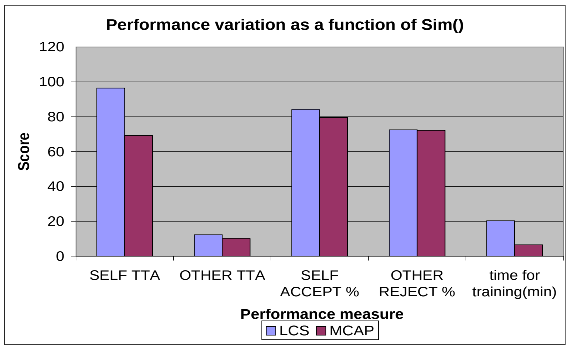

Figure 9: Effect of Similarity Metric

4.6 Effect of Test Data Size

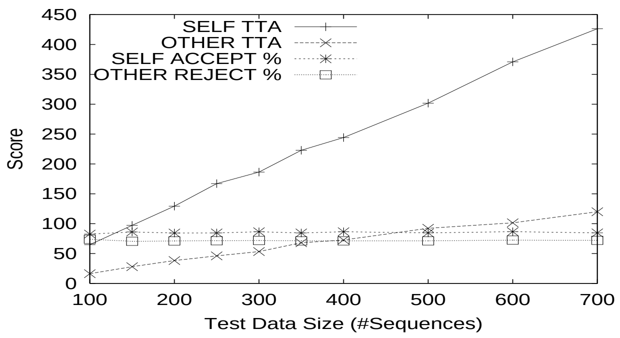

Figure 8: Effect of Test Data Size

The test data size is crucial as human behavior is constantly changing and the performance degrades as a result of being tested on concepts not learnt during the training phase. Also, it emphasizes the rate at which user behavior changes, i.e., concept drift. We tested the model developed for test data sizes of {100, 150, 200, 250, 300, 400, 500, 600, 700} sequences. Figure 8 shows that as test data size increases, performance hardly varies. This is probably due to the lack of sufficient concept drift in the user behavior to register. The variation of TTA is a result of the fact that if there are no anomalous sequences in a test set, the TTA is assumed to be the size of the set.

4.7 Effect of Similarity Metric

We tried out two similarity measures, viz. MCAP and LCS. According to results from Lane [11], MCAP performs better than MCP, MCE and MCEP, hence we did not test them. Note that the empirically selected T_{ACCEPT} corresponding to each metric is different.

As seen in Figure 9, LCS is slower than MCAP in terms of performance as it is an $O(l^2)$ algorithm (see rightmost set of bars). However, since small l yields good accuracy, we chose a small sequence length (l=5), and the cost overhead is tolerable. On the other hand LCS outperforms MCAP in all categories other than OTHER TTA. Thus choosing the similarity measure involves a tradeoff between two of our desirable characteristics viz. minimal overhead versus accuracy. We opt for accuracy, as we believe that the overhead becomes truly intolerable at training time, which is done at initialization. The overhead incurred during the testing phase, we believe is acceptable in return for the resultant accuracy.

4.8 Real-time Learning

In the next set of experiments, we allowed real-time learning as well, i.e., concepts learnt during testing are added to the profile. We use l_sup to denote the minimum number of anomalous sequences seen so far, for online clustering to be applied to group them. In the extreme bars of Figure 10 for each performance measure, we see that real-time learning improves SELF ACCEPT marginally, while OTHER REJECT decreases substantially. This suggests that IDS perform better when they do not learn during real-time (for $l\_sup = \infty$, i.e., never invoke online clustering). This is because, in the absence of expert supervision, the IDS is fooled easily. In other words, although new concepts are learnt, due to which SELF ACCEPT increases, new concepts are learnt from the masquerader as well and hence OTHER REJECT decreases.

This problem can be remedied in a number of ways: 1) Before admitting new clusters to the profile, we do send a warning to the security officer, who can then guide the system. 2) We could create user classes by clustering across user profiles [11], i.e., such classes could possibly differentiate users on the basis of their skill, or the types of applications they were using. Thus each user would belong to a class. We could admit new clusters to the profile if they existed in profiles of other users belonging to the same user class. 3) We can monitor the rate at which the profile is changing, i.e., the rate at which new clusters are being added to the profile. If that changes dramatically, we could stop admitting clusters to the profile[23]. 4) To avoid creating clusters for noise, cluster a list of anomalous sequences only if they meet certain support requirements, i.e., l_sup.

In Figure 10, where we have tested option 4) from above, SELF ACCEPT increases and so does OTHER REJECT as l_sup increases, due to elimination of noise from the concepts learnt.

It was observed that when we tested option 3) from above, by monitoring the number of incrementally mined clusters during real-time learning, the mean ratio of clusters produced when tested with OTHER data as compared to SELF data is 1.85. Thus, nearly twice the number of clusters are produced during times of attacks as compared to during times of normal usage. Also it was observed that during times of normal usage, the number of type B alarms raised is 2.41 times that produced during testing against OTHER

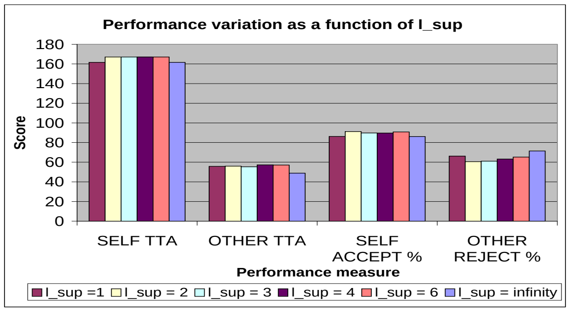

Figure 10: Real-time Learning

data, i.e., larger clusters of anomalous sequences are produced during times of attacks.

5. DISCUSSION

Our system achieves approximately 80% detection rate and 15% false positive rate. While these numbers are lower than those reported by Ryan et al [18] (96% and 7% respectively) and DuMouchel et al [6] (78% and 6.6% respectively), their dataset is different, and it is thus not meaningful to directly compare these numbers. On the other hand, our results are better than those reported by Lane and Brodley [12] (74% and 28% respectively), who used the same data as in this paper. The advantages of ADMIT over the other implementations are that it requires a much shorter training time, summarizes the data and achieves model scaling simultaneously. ADMIT is better suited to real-time application than the methods of DuMouchel and Lane, as it can use shorter window sequences. Keep in mind that raw accuracy numbers only give a partial picture of the complex process of detecting intrusions. For instance in setting parameters to maximize accuracy in ADMIT, we have traded off time and sensitivity by using the LCS algorithm and rating metrics respectively. Also, our work does not advocate the use of LCS, `DECAYED_WEIGHTS`, etc. It rather represents the advantages of using them in comparison to others. The actual selection of parameters depends on the security policy requirements. Also, as the training data can be clustered, only the centers require to be labeled by the security officer, thereby reducing the requirement of labeled data.

Future Work: . Open problems that we plan to address include reducing amount of training data required by establishing user classes and using sequences from user class as initial clusters for user believed to belong to that class (as in section 4.8). Other improvements would be using different parameters for different users and parameter selection using cross-validation. Integration with profiles based on biometric data, e.g., keystroke monitoring are future directions of research.

6. REFERENCES

[1] D. Aha, D. Kibler, M.Albert. Instance-based learning algorithms. Machine learning, 6(1):37-66, 1991.

[2] K. Alsabti, S. Ranka, V. Singh. An efficient K-means Clustering Algorithm. In 11th International Parallel Processing Symposium, 1998.

[3] J.B.D. Cabrera, L. Lewis, R.K. Mehra. Detection and Classification of Intrusions and Faults using Sequences of System Calls. SIGMOD Record, 30(4), pp 25-34. December 2001.

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest. Introduction to Algorithms. McGraw-Hill. 1990.

[5] D. E. Denning. An Intrusion-Detection Model. IEEE Transactions on Software Engineering, 13(2):222-232, February 1987.

[6] W. DuMouchel. Computer Intrusion Detection Based on Bayes Factors for Comparing Command Transition Probabilities. In National Institute of Statistical Sciences Tech. Report 91, February 1999.

[7] S.A. Hofmeyr, S. Forrest, A. Somayaji. Intrusion Detection using sequences of system calls. In Journal of Computer Security, 6:151-180, 1998.

[8] H. S. Javitz, A. Valdez. The NIDES Statistical Component: Description and Justification. In Technical Report A010, Computer Science Lab, SRI International, March 1993.

[9] L. Kaufmann, P.J. Rousseeuw. Finding Groups in Data: An Introduction to Cluster Analysis. John Wiley and Sons. March 1990.

[10] S. Kumar, E. H. Spafford. A pattern matching model for misuse intrusion detection. In 17th National Computer Security Conference, pp. 11-21, 1994.

[11] T. Lane. Machine Learning Techniques for the Computer Security Domain of Anomaly Detection. Ph. D. Thesis, CERIAS TR 2000-12, Purdue University, August 2000.

[12] T. Lane, C. E. Brodley. Temporal Sequence Learning and Data Reduction for Anomaly Detection. ACM Transactions on Information and System Security, 2:295-331, 1999.

[13] D. J. Langin. Out of the NOC(a) and Into the Boardroom: Director and Officer Responsibility for Information Security. July 30, 2001. URL: http://www.recourse.com/news/press/releases/r073001.html

[14] W. Lee, S. J. Stolfo. Data Mining Approaches for Intrusion Detection. In Proceedings of the 7th USENIX Security Symposium, January 1998.

[15] W. Lee, S. Stolfo, P. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, J. Zhang. Real Time Data Mining-based Intrusion Detection. In DARPA Information Survivability Conference and Exposition II. June 2001.

[16] P. A. Porras, P. G. Neumann. EMERALD: Event monitoring enabling responses to anomalous live disturbances. In 20th National Information Systems Security Conference, October 1997.

[17] L. Portnoy, E. Eskin, S. Stolfo. Intrusion detection with unlabeled data using clustering. In ACM Workshop on Data Mining Applied to Security (DMSA 2001), November 2001.

[18] J. Ryan, M.J. Lin, R.Miikkulainen. Advances In Neural Information Processing Systems 10, Cambridge, MA: MIT Press 1998.

[19] M. Schonlau, W. DuMouchel, W. Ju, A. Karr, M. Theus, Y. Vardi. Computer Intrusion: Detecting Masquerades. Statistical Science, 16:1-17. February 2001.

[20] J. S. Subramaniyan, J. O. Garcia-Fernandez, D. Isacoff, E. Spafford, D. Zamboni. An Architecture for Intrusion Detection Using Autonomous Agents. In 14th Annual Computer Security Applications Conf, December 1998.

[21] A. Valdes, K. Skinner. Adaptive, Model-based Monitoring for Cyber Attack Detection, Lecture Notes in CS, No. 1907, Springer-Verlag, pp. 80-92, October 2000.

[22] C. Warrender, S. Forrest, B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In IEEE Symposium on Security and Privacy, 1999.

[23] D.Zamboni. Using clustering to detect abnormal behavior in a distributed intrusion detection system. Unreleased Technical Report, Purdue University. August, 2001.