Adaptive Real-Time Anomaly Detection with Improved Index and Ability to Forget*

Kalle Burbeck and Simin Nadjm-Tehrani
Department of Computer and Information Science
Linköping university SE–581 83 Linköping, Sweden

E-mail: {kalbu, simin}@ida.liu.se

Abstract

Anomaly detection in IP networks, detection of deviations from what is considered normal, is an important complement to misuse detection based on known attack descriptions. Performing anomaly detection in real-time places hard requirements on the algorithms used. First, to deal with the massive data volumes one needs to have efficient data structures and indexing mechanisms. Secondly, the dynamic nature of today's information networks makes the characterization of normal requests and services difficult. What is considered as normal during some time interval may be classified as abnormal in a new context, and vice versa. These factors make many proposed data mining techniques less suitable for real-time intrusion detection. In this paper we extend ADWICE, Anomaly Detection With fast Incremental Clustering. Accuracy of ADWICE classifications is improved by introducing a new grid-based index, and its ability to build models incrementally is extended by introducing forgetting. We evaluate the technique on the KDD data set as well as on data from a real (telecom) IP test network. The experiments show good detection quality and illustrate the usefulness of adapting to normality.

1. Introduction

Intrusion detection is an important part of computer system defence. Due to increasing complexity of the intrusion detection task, the use of many IDS sensors to increase coverage, and the need for improved usability of intrusion detection, a recent trend is alert or event correlation [3, 6]. Correlation combines information from multiple sources to improve information quality. By correlation the strength of different types of detection schemes may be combined, and weaknesses compensated for.

The main detection scheme of most commercial intrusion detection systems is *misuse detection*, where known bad behaviours (attacks) are encoded into signatures. In *anomaly detection* normal behaviour of users or the protected system is modelled, often using machine learning or data mining techniques rather than given signatures. During detection new data is matched against the normality model, and deviations are marked as anomalies. Since no knowledge of attacks is needed to train the normality model, anomaly detection may detect previously unknown attacks.

Anomaly detection still faces many challenges, where one of the most important is the relatively high rate of false alarms (false positives). The problem of capturing a complex normality makes the high rate of false positives intrinsic to anomaly detection except for simple problems. We argue that the usefulness of anomaly detection is increased if combined with further aggregation, correlation and analysis of alarms, thereby minimizing the number of false alarms propagated to the administrator (or automated response system) that further diagnoses the scenario. The fact that normality changes constantly makes the false alarm problem even worse. When normality changes over time, a model with acceptable rates of false alarms may rapidly deteriorate in quality. To minimize this problem and the resulting additional false alarms, the anomaly detection model needs to be adaptable.

The training of the normality model for anomaly detection may be performed by a variety of different techniques and many approaches have been evaluated. One important technique is *clustering*, where similar data points are grouped together into clusters using a distance function. In this paper we develop a new indexing mechanism for ADWICE (Anomaly Detection With fast Incremental Clustering), an adaptive anomaly detection scheme inspired by the BIRCH clustering algorithm [15]. We improve an earlier

*This work was supported by the European project Safeguard [13] IST–2001–32685. We would like to thank Thomas Dagonnier, Tomas Lingvall and Stefan Burschka at Swisscom for fruitful discussions and their many months of work with the test network. The work is currently supported by a grant from CENIIT, Center for Industrial Information Technology at Linköping university.

version of the algorithm used for anomaly detection [2] in two ways: (1) a novel search index that increases precision detection quality, and (2) new means to handle adaptation of the model (forgetting). This paper does not attempt to justify the quality of clustering within anomaly detection or compare performance with other machine learning work. The interested reader is referred to the original report for that purpose [2].

The paper is organised as follows. In section 2 the motivation of this work is presented together with related work. Section 3 describes the indexing technique, and evaluation is presented in section 4. The final section discusses the results and describes opportunities and challenges for future work.

2 Motivation and Related Work

One fundamental problem of intrusion detection research is the limited availability of good data to be used for evaluation. Producing intrusion detection data is a labour intensive and complex task involving generation of normal system data as well as attacks, and labelling the data to make evaluation possible. If a real network is used, the problem of producing good normal data is reduced, but then the data may be too sensitive to be released to other researchers publicly. Learning-based methods require data not only for testing and comparison but also for training, resulting in even higher data requirements. The data used for training needs to be representative for the network to which the learning-based method will be applied, possibly requiring generation of new data for each deployment.

Classification-based methods [4, 9] require training data that contains normal data as well as good representatives of those attacks that should be detected, to be able to separate attacks from normality. Producing a good coverage of the very large attack space (including unknown attacks) is not practical for any network. Also the data needs to be labelled and attacks to be marked. One advantage of *clustering-based methods* [5, 10, 12, 14] is that they require no labelled training data set containing attacks, significantly reducing the data requirement. There exist at least two approaches.

When doing *unsupervised anomaly detection* [5, 12, 14] a model based on clusters of data is trained using unlabelled data, *normal as well as attacks*. If the underlying assumption holds (i.e. attacks are sparse in data) attacks may be detected based on cluster sizes, where small clusters correspond to attack data. Unsupervised anomaly detection is a very attractive idea, but unfortunately the experiences so far indicate that acceptable accuracy is very hard to obtain. Also, the assumption of unsupervised anomaly detection is not always fulfilled making the approach unsuitable for attacks such as denial of service (DoS) and scanning.

In the second approach, which we simply denote *(pure) anomaly detection* in this paper, training data is assumed to consist *only of normal data*. Munson and Wimer [10] used a cluster-based model (Watcher) to protect a real web server, proving anomaly detection based on clustering to be useful in real life. ADWICE uses pure anomaly detection to reduce the training data requirement of classification-based methods and to avoid the attack volume assumption of unsupervised anomaly detection. By including only normal data in the detection model the low accuracy of pure anomaly detection can be significantly improved.

In a real live network with connection to Internet, data can never be assumed to be free of attacks. Pure anomaly detection also works when some attacks are included in the training data, but those attacks will be considered normal during detection and therefore not detected. To increase detection coverage, attacks should be removed from the training data to as large an extent as possible, with a trade-off between coverage and data cleaning effort. Attack data can be filtered away from training data using updated misuse detectors, or multiple anomaly detection models may be combined by voting to reduce costly human effort.

One of the inherent problems of anomaly detection in general is the false positives rate. In most realistic settings normality is hard to capture and even worse, is changing over time. Rather than making use of extensive periodical retraining sessions on stored off-line data to handle this change, ADWICE is fully incremental making very flexible on-line adaptation of the model possible without destroying what is already learnt.

An intrusion detection system in a real-time environment needs to be fast enough to cope with the information flow, have explicit limits on resource usage, and adapt to changes in the protected network in real-time. Many proposed clustering techniques require quadratic time for training [7], making real-time adaptation of a cluster-based model hard. They may also not be scalable, requiring all training data to be kept in main memory during training, limiting the size of the trained model. We argue that it is important to consider scalability and performance in parallel to detection quality when evaluating algorithms for intrusion detection. Most work on applications of data mining to intrusion detection consider those issues to a very limited degree or not at all. ADWICE is scalable since compact cluster summaries are kept in memory rather than individual data. ADWICE also has better performance compared to similar approaches due to the use of local clustering with an integrated improved search index.

3 The Anomaly Detection algorithm

This section describes how ADWICE handles training and detection, more details on the old index may be found in the original paper [2]. The present implementation re-

quires data to be numeric. Data is therefore assumed to be transformed into numeric format by pre-processing.

3.1 Basic concepts

The basic clustering concepts are presented in the original BIRCH paper [15] and the relevant parts are summarized here. An ADWICE model consists of a number of clusters, a number of parameters, and a tree index in which the leaves contain the clusters. A central idea of BIRCH inherited by ADWICE is to store only condensed information (cluster feature) instead of all data points of a cluster. A *cluster feature* is a triple $CF = (n, \boldsymbol{S}, SS)$ where n is the number of data points in the cluster, $\boldsymbol{S}$ is the linear sum of the n data points and SS is the square sum of all data points. From now on we represent clusters by cluster features (CF).

Given n d-dimensional data vectors $\boldsymbol{v}_i | i = 1 \ldots n$ in a cluster CF the *centroid* $\boldsymbol{v}_0$ is defined as $\boldsymbol{v}_0 = \sum_i \boldsymbol{v}_i / n$, which may be computed given only the summary CF. The distance between a data point $\boldsymbol{v}$ and a cluster CF is the Euclidian distance between $\boldsymbol{v}$ and the centroid, denoted $D(\boldsymbol{v}, CF)$ while the distance between two clusters CF_i and CF_j is the Euclidian distance between their centroids, denoted $D(CF_i, CF_j)$. If two clusters $CF_i = (n_i, \boldsymbol{S_i}, SS_i)$ and $CF_j = (n_j, \boldsymbol{S_j}, SS_j)$ are merged, the CF of the resulting cluster may be computed as $(n_i + n_j, \boldsymbol{S_i} + \boldsymbol{S_j}, SS_i + SS_j)$. This also holds if one of the CFs is only based on one data point making incremental update of CFs possible.

A leaf node contains at most LS (*leaf size*) entries, each of the form CF_i where $i \in \{1, \ldots, LS\}$. Each CF_i of the leaf node must satisfy a *threshold requirement* (TR) with respect to the threshold value T to allow the cluster to absorb a new data point $\boldsymbol{v}$. In the current implementation the threshold requirement is $D(\boldsymbol{v}, CF_i) \leq T$.

3.2 Training

The algorithm for training works with a parameter M (maximum size of the model, denoted by total number of clusters), and the parameter LS. The threshold T may be initialized to zero.

Given a model and a new data vector $\boldsymbol{v}$, a search for the closest cluster is performed. If the threshold requirement is fulfilled, the new data point may be merged with the closest cluster, otherwise a new cluster needs to be inserted:

Case 1: If it is possible to add clusters to the model (the size is still below M), we find the leaf where the new cluster should be included and insert the cluster if there is still space in the leaf. Otherwise we need to split the leaf and insert the new cluster in the most suitable of the new leafs.

Case 2: If the size (number of clusters) of the model has reached the maximum M we need to reduce the model size by allowing clusters to grow. The threshold T is increased, relaxing the threshold requirement and the model is rebuilt. When rebuilding the model, the old cluster features are removed and reinserted into the model. Due to the increase of T, previously close (similiar) clusters may be merged, thereby reducing the size of the model. The new data point $\boldsymbol{v}$ is then inserted into the new model.

Rebuilding the model requires much less effort than the initial insertion of data since only cluster features rather than individual data points are inserted. If the increase of T is too small, a new rebuild of the tree may be needed to reduce the size below M again. A heuristic described in the original BIRCH paper [15] may be used for increasing the threshold to minimize the number of rebuilds, but in this work we use a simple constant to increase T conservatively (to avoid influencing the result by the heuristic).

Below is an algorithmic description of the training phase of ADWICE, in which only the main points of the algorithm are presented and some simplifications made to facilitate presentation. Note also that the index is abstracted away at this point.

```
1:  procedure TRAIN(v, model)
2:      closestCF = findClosestCF()
3:      if thresholdRequirementOK(v,closestCF) then
4:          merge(v,closestCF)
5:      else
6:          if size(model) < M then
7:              leaf = getLeaf(v,model)
8:              if spaceInLeaf(leaf) then
9:                  insert(newCF(v), leaf)
10:             else
11:                 splitLeaf(leaf, newCF(v))
12:             end if
13:         else
14:             increaseThreshold()
15:             model = rebuild(model)
16:             TRAIN(v, model)
17:         end if
18:     end if
19: end procedure
```

The first implementation of ADWICE [2] used the original BIRCH index: a tree structure where the non-leaf nodes contained one CF for each child, summarizing all clusters contained in the child below. Unfortunately the original index results in a suboptimal search where the closest cluster is not always found. Although this does not decrease processing performance, accuracy suffers. If a cluster included in the normality model is not found and the test data is normal, an index error results in an erroneous false positive and degrades detection quality. Because of this unwanted property a new grid-based index was developed pre-

serving the adaptability and good performance of ADWICE and designed to eliminate the index errors of BIRCH.

3.3 New indexing mechanism

A *subspace* of d-dimensional space is defined by two vectors, $\boldsymbol{ssMax}$ and $\boldsymbol{ssMin}$, specifying for each dimension an interval or slice. A *grid* is a division of space into subspaces. In two dimensions this results in a space divided into rectangles. The idea of the new grid index is to use a *grid-tree* which is a sparse, possibly unbalanced tree with the parameters threshold (T), leaf size (LS) and maximum number of clusters (M). Each node specifies a subspace and each increase in depth decreases the size of the subspace facilitating searching by gradually *zooming in* on the data space. We explain the index by a simple 2-dimensional example illustrated by Figure 1.

Each dimension d_i has a maximum value Max_i and a minimum value Min_i, in the running example of Figure 1 $Min_1 = Min_2 = 0$ and $Max_1 = Max_2 = 1$. For unbounded domains those extreme values are in practice realized during feature extraction. The user divides each dimension d_i in a number of *slices* denoted $NumSlices_i$, in the example $NumSlices_1 = 3$ and $NumSlices_2 = 4$. The *slice width* may be computed as $SliceWidth_i = (Max_i - Min_i)/NumSlices_i$. In the example this results in $SliceWidth_1 = 0.33$ and $SliceWidth_2 = 0.25$. A function $getSliceDimension(depth)$ mapping tree depth to dimension index is specified, i.e $[0 \rightarrow d_1, 1 \rightarrow d_2]$.

Each entity of a non-leaf node is mapping from the slice number to a child node (see Figure 1), realized by a hash table to facilitate quick search and reducing space by not representing empty children. The leaf nodes contain the cluster features similarly to the original BIRCH index.

During training the general training algorithm of section 3.2 is followed. Assume we want to train the model of Figure 1 with the new data point $\boldsymbol{v} = \langle 0.28, 0.2 \rangle$ by inserting $\boldsymbol{v}$ in the closest cluster. Merging of a data point with a cluster CF_i may only be performed if the $D(\boldsymbol{v}, CF_i) < T$ implying those are the only clusters we need to consider when searching for the closest one.

To find the closest cluster we perform a top-down search of the grid-tree. At the root, i.e. $depth = 0$, we obtain $getSliceDimension(0) = d_1$. Given the value of $\boldsymbol{v}$ in dimension i ($\boldsymbol{v}[i]$) and $SliceWidth_i$ we compute the slice number into which $\boldsymbol{v}$ fits, in the example slice $\underline{1}$. However, since $\boldsymbol{v}$ is close to the border between slices $\underline{1}$ and $\underline{2}$ it may be the case that the closest cluster is located in slice $\underline{2}$. We compute a search width $[v[i] - T, v[i] + T]$ and search in all slices overlapping the search width interval, in this case $\underline{1}$ and $\underline{2}$. No child exists for slice $\underline{2}$ due to lack of clusters in the corresponding subspace, eliminating the need for search in $\underline{2}$. In slice $\underline{1}$ we descend and find a leaf-cluster and return b as the closest cluster since $D(\boldsymbol{v}, b) < D(\boldsymbol{v}, a)$. Our intuition and experience tells us that in our domain the grid is sparsely populated decreasing the need for searching multiple slices at each depth.

During learning/adapting the normality model there are three cases in which the nodes of the grid tree need to be updated:

- If no cluster is close enough to absorb the data point, $\boldsymbol{v}$ is inserted into the model as a new cluster. If there does not exist a leaf subspace in which the new cluster fits, a new leaf is created. However, there is no need for any additional updates of the tree, since nodes higher up do not contain any summary of data below.
- When the closest cluster absorbs $\boldsymbol{v}$, its centroid is updated accordingly. This may cause the cluster to move in space. A cluster may potentially move outside its current subspace. In this case, the cluster is removed from its current leaf and inserted anew in the tree from the root, since the path all the way up to the root may have changed. If the cluster was the only one in the original leaf, the leaf itself is removed to keep unused subspaces without leaf representations.
- If a cluster is removed/forgotten (section 3.5) the index is only changed if the leaf is now empty in which case the leaf of the removed cluster is also removed.

Compared to the continuously updated original BIRCH index, the need for grid index updates are very small, since the first case above requires only insertion of one new leaf, and the second case occurs infrequently. The third case requires most often no index update at all.

In case of a split of a leaf, the original leaf is transformed to a non leaf node. The new node computes its split dimension according to its depth in the tree, and the clusters of the original leaf are inserted in the new node resulting in creation of leaves as children to the node.

The $getSliceDimension(depth)$ function should be defined manually or automatically according to statistics of the input data to minimize the depth of the tree, thereby maximizing performance. For example, if all data have the same or almost the same value in a certain dimension, this dimension is not useful for slicing, since the depth of the tree will increase without distributing the clusters into multiple children.

In most cases not all dimensions need to be used for slicing (thus limiting the height of the tree), assuming that the function $getSliceDimension(depth)$ is selected appropriately. However, if the situation arises that all useful dimensions have been used for slicing, and still the number of clusters to be inserted does not fit in one leaf due to the limited leaf size (LS), then the LS parameter for that leaf can be increased, affecting only that leaf locally. This is the

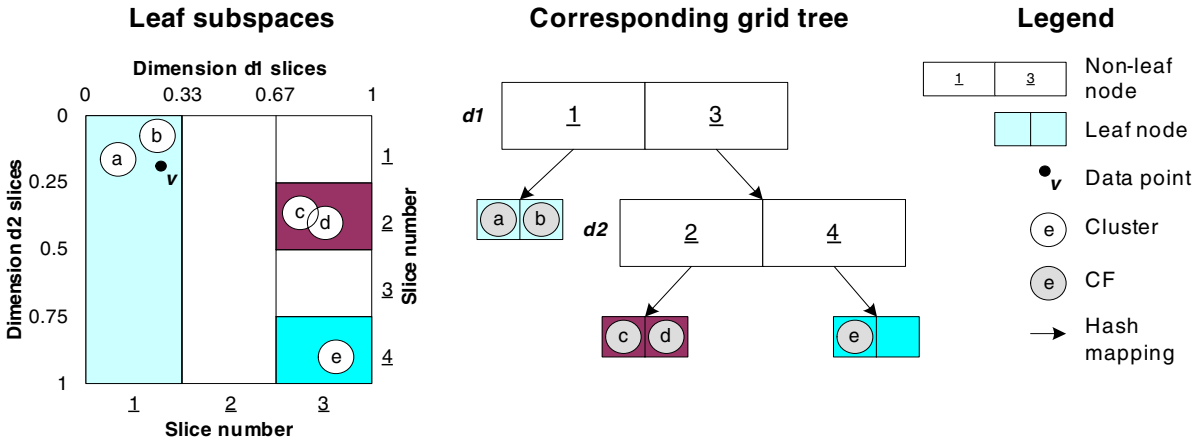

Figure 1. Basic notions of the grid and grid tree

approach that our current implementation adopts. An alternative or complementary approach to handle this situation is to rebuild the tree using a smaller width of the intervals for each dimension.

3.4 Detection

The basic detection procedure is the same for any choice of index and thus similar to the original index [2]. During the detection the model is searched for the closest cluster CF_i (using the index). Then the distance $D(\boldsymbol{v}, CF_i)$ from the centroid of the cluster to the new data point $\boldsymbol{v}$ is computed. Informally, if D is small, i.e. lower than a limit, $\boldsymbol{v}$ is similar to data included in the normality model and $\boldsymbol{v}$ should therefore be considered normal. If D is large, $\boldsymbol{v}$ is an anomaly.

Let the threshold T be the limit (L) used for detection. Using two parameters E_1 and E_2, $MaxL = E_1 * L$ and $MinL = E_2 * L$ may be computed. Then we compute the belief b that $\boldsymbol{v}$ is anomalous using the formula below:

$$b = \begin{cases} 0 & \text{if } D \leq MinL \\ 1 & \text{if } D \geq MaxL \\ \frac{D-MinL}{MaxL-MinL} & \text{if } MinL < D < MaxL \end{cases} \quad (1)$$

A belief threshold (BT) is then used to make the final decision. If $\boldsymbol{v}$ is considered anomalous an alarm is raised. The belief threshold may be used by the administrator to change the sensitivity of the anomaly detection. For the rest of the paper, to simplify the evaluation, we set $E1 = E2 = E$ so that $\boldsymbol{v}$ is anomalous if and only if $D > MaxL$.

For the grid-tree the search for the closest cluster during detection differs slightly from the search during training. When deciding search width now $MaxL$ rather T needs to be used.

3.5 Adaptation of the normality model

Below we describe two scenarios in which it is very useful to have an incremental algorithm in order to adapt to changing normality.

Scenario 1 New cases of normality require the model to adapt incrementally. In some settings, it may be useful to let the normality model relearn autonomously. If normality drifts slowly, an incremental clustering algorithm may handle this in real-time during detection by incorporating every test data classified as normal with a certain confidence into the normality model. If slower drift of normality is required, a subset of those data based on sampling could be incorporated into the normality model. Even if autonomous relearning is not allowed in a specific network there is need for model adaptation. Imagine that the ADWICE normality model has been trained, and is producing good detection results for a specific network during some time interval. However, in an extension of the interval the administrator recognizes that normality has changed and a new class of data needs to be included as normal. Otherwise this new normality class produces false positives. Due to the incremental property, the administrator can incorporate this new class without relearning the working fragment of the existing normality model. That is, there is no need for retraining the complete model. The administrator may interleave incremental training with detection, completely eliminating the need for downtime required by non-incremental approaches.

Scenario 2 The opposite scenario, is when the model of normality needs to shrink. That is, something that was considered normal earlier is now considered as undesirable. In our approach this situation is adapted to by for-

getting the segment of normality that is no longer considered as normal. This too can be performed autonomously or manually. We have experimented with one autonomous forgetting process which checks the model periodically ($CheckPeriod$) to decide what clusters can be forgotten. If the difference between current time and time of last use is larger than a forgetting threshold ($RememberPeriod$), the cluster is removed from the model. The rest of the model is not influenced.

Each time a cluster is used, either during training or detection, forgetting of the cluster is postponed for an amount of time specified by the user. This is similar to the positive influence of repetition in case of human learning.

4 Evaluation

In all following experiments ADWICE with a grid index is used unless otherwise stated.

We choose to start evaluation of detection quality using the KDDCUP99 intrusion detection data set [11] to be able to compare the results with earlier evaluations [2]. Despite the shortcomings of the DARPA related data sets [8] they have been used in at least twenty research papers and are unfortunately currently the only openly available network data set commonly used for comparison purposes. The purpose of this first evaluation is a proof of concept for anomaly detection with ADWICE using a large feature set in real-time, testing the grid index.

Since ADWICE needs only normal data for training attacks in the KDD training data set are removed resulting in 972 781 session records (41 fields summarizing a TCP session) used to train the ADWICE model. This model is evaluated on the KDD testing set consisting of 311 029 session records of normal data and many instances from 37 different attack types.

With a model size of 12 000 clusters, mean processing time per testing instance was 0.5-0.6 ms in off-line experiments on 1.8 GHz P3 computer with 512 MB RAM illustrating the good performance of the old ADWICE model. Figure 2 shows that the primitive search operation of the new grid-index (hashing) is 6-10 times faster than the primitive operation of the old index (Euclidian distance, D) depending on the number of dimensions (60-20 in Figure 2). Full performance evaluation of the new index depends on the parameters and the function $getSplitDimension()$ and is left as future work. The use of 972 781 elements of 41 dimensional data in the normality model illustrates the scalability of ADWICE.

To illustrate forgetting and incremental training in a more realistic setting, a data set generated by the telecom management test network in the European Safeguard project [13] was used, at the time of evaluation consisting of 50 real machines (now more than 100). A period of three

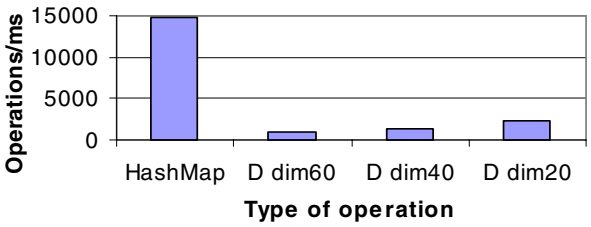

Figure 2. Relative performance of primitive index operations

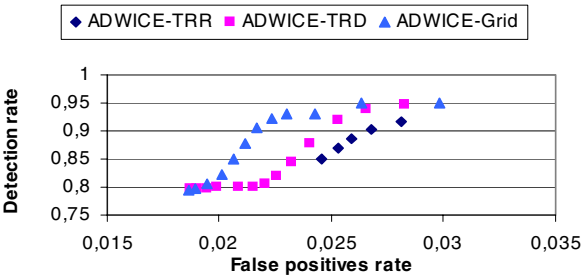

Figure 3. Detection quality of ADWICE on KDD data

days data is used for training an initial model, while the following seven days of data are used for testing. The features include source and destination IP addresses and port numbers, time of day, connection length, bytes transferred and a flag indicating a parsing error.

4.1 Detection quality of ADWICE

In all we have evaluated three different versions of ADWICE as shown in Figure 3. The trade-off between detection rate and false positives rate are realised by changing the detection parameter E.

ADWICE-TRR [2] is base-line with a training phase very similar to the original clustering algorithm BIRCH. ADWICE-TRD [2] improves the first implementation while still using the original index. ADWICE-Grid uses the new grid index rather than the original BIRCH index, eliminating the index-misses and thereby further improving the result. A comparison with a classification based method and unsupervised anomaly detection is present in earlier work [2].

4.2 Incremental training

An important property of ADWICE is that the original model, known to truly reflect recent normality, does not

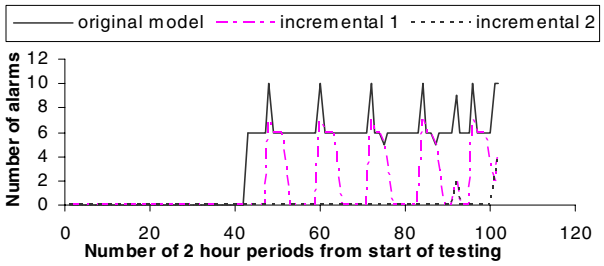

Figure 4. Adapting to changing normality with incremental training

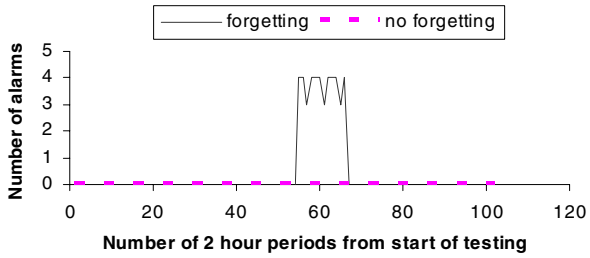

Figure 5. Adapting to changing normality using forgetting

need to be retrained as soon as new cases of normality are encountered. We evaluate this on data generated in the Safeguard test network. The three days of training data is used to train an initial model which is then used for detection on the seven days of testing data. When certain types of traffic (new cases of normality) start producing false alarms the administrator may tell ADWICE to incrementally learn the data causing those alarms to avoid similar false alarms in the future. Figure 4 shows alarms for host x.x.202.183 in three scenarios. Period number 1 starts at time 2004-03-11 00:00 (start of testing data) and each two-hour period presents the sum of alarms related to host x.x.202.183 during the corresponding time interval. At 2004-03-14 12:00, corresponding to period number 43, the host is connected to the network.

In the first scenario, no incremental training is used, and the testing is performed on the original model. This corresponds to the first curve of Figure 4. We see that when the host connects, ADWICE starts to produce alarms and this continues until the testing ends at period 102.

In the second scenario the administrator recognizes the new class of alarms as false positives. She tells ADWICE to learn the data resulting in those false alarms at time 2004-03-14 17:55 (end of period 45).The second curve shows that many of the original false alarms are no longer produced. However, at regular intervals there are still many alarms. Those intervals correspond to non-working hours.

In the third scenario incremental training is done in two steps. After the first incremental training 2004-03-14 a second incremental training is initiated at 2004-03-15 07:55 (end of period 52) when the administrator notices that false alarms related to host x.x.202.183 are still produced. Figure 4 shows how almost all alarms now disappear after the second incremental training period.

The need for the two-step incremental learning arouse since the model differs between working hours and non-working hours. The alarms the administrator used for initial incremental training were all produced during working hours (2004-03-14 12:00 - 2004-03-14 17:55).

4.3 Forgetting

In this section we illustrate the use of forgetting. A model is trained on data from three days of data and is then used for detection with and without forgetting on the following seven days of data. Figure 5 shows alarms for one instance of traffic (host x.x.202.73, port 137) that ceases to be present in the (normal) testing data, making that kind of traffic anomalous. With forgetting this fact is reflected in the normality model. In this experiment a CheckPeriod of 12 hours and RememberPeriod of 3 days (72 hours) are used.

When traffic from host x.x.202.73 on port 137 is again visible in data (periods 55-66) the traffic is detected as anomalous. Without forgetting these anomalies would go undetected.

5 Discussion and future work

We have developed and preliminary evaluated an adaptive network anomaly detector. In related work, real-time detection and indexes for fast matching against the normality model are seldom considered together with the basic detection approach. We consider the index in the detection scheme from the start, since the index may influence not only performance, but also other properties such as adaptiveness.

Preliminary evaluations show that there may be a need for the administrator to tell the anomaly detector to learn not only the data producing the alarm, but also a generalisation of data. For example, the two step incremental training in section 4.2 would not have been necessary if the administrator could have told the system to learn that the data producing alarms was normal both during working and non-working hours. Those experiences call for more intelligent tools, such as sophisticated model and data visualization mechanisms, to help the administrator use anomaly detection in a real environment. Also systematic methods to decide on suitable parameter settings (e.g. $M, numSlices$)

is needed. Full evaluation of how sensitive ADWICE is to minor changes of parameters remains to be done.

Extensions of the work will consider cluster size and frequency of usage as criteria to decide whether a cluster ought to be forgotten. Large clusters, corresponding to very common normality in the past, should be very resistant against forgetting. With this approach also outliers resulting in small clusters in the model could be handled by forgetting them. Another improvement would be to gradually decrease the influence of clusters over time, rather then forgetting them completely.

Full evaluation of forgetting and incremental training requires a long period of real data collection. We see the current work as a proof of concept. It is improbable that parts of normality should be forgotten already after a few days in a real network and suitable configuration of forgetting is network and policy specific. This is a trade-off between (1) performance (decreasing model size), (2) detection of (very) old normality as anomalies and (3) false positives if the normality is forgotten too soon.

Producing good public data for intrusion detection evaluation including anomaly detection and correlation is still an important task for the intrusion detection community and we think collaboration on this task is very important. Two approaches exist:

- A test network or simulator can be used for generation of data, thereby realising a fully controlled environment. Generation of more extensive data sets in the Safeguard test network is ongoing work but requires more resources, primarily to generate good normal data, but also to perform attacks.
- Data can be collected from real live networks. Here, normality is no longer a problem, but privacy is, and so is the issue of attack coverage. Emerging tools [1] can be used to sanitize data with some limitations. Sanitizing data while keeping relevant properties of data so that intrusion detection analysis is still valid is not easy, especially for unstructured information, for example, network packet content. Attacks may have to be inserted into data off-line if the real attack present in data is not enough, since performing attacks in a real network is typically not a viable option.

We would like to explore both paths for evaluation of future work and possibly include not only network data, but also explore other types of data. This is possible since ADWICE is a general and flexible anomaly detection scheme.

References

[1] M. Bishop, B. Bhumiratana, R. Crawford, and K. N. Levitt. How to sanitize data. In *Proceedings of 13th IEEE International Workshops on Enabling Technologies, Infrastructure for Collaborative Enterprises (WETICE 2004)*, pages 217–222, Modena, Italy, 2004. IEEE Computer Society.

[2] K. Burbeck and S. Nadjm-Tehrani. Adwice - anomaly detection with real-time incremental clustering. In *Proceedings of the 7th International Conference on Information Security and Cryptology*, Seoul, Korea, 2004. Springer Verlag.

[3] T. Chyssler, S. Burschak, M. Semling, T. Lingvall, and K. Burbeck. Alarm reduction and correlation in intrusion detection systems. In *Proceedings of Detection of Intrusions and Malware & Vulnerability Assessment, GI SIG SIDAR Workshop (DIMVA 2004)*, volume 46 of *LNI*, pages 9–24, Dortmund, Germany, July 2004. GI.

[4] C. Elkan. Results of the kdd'99 classifier learning. *ACM SIGKDD Explorations*, 1(2):63 – 64, 2000.

[5] Y. Guan, A. A. Ghorbani, and N. Belacel. Y-means a clustering method for intrusion detection. In *Canadian Conference on AI*, volume 2671 of *Lecture Notes in Computer Science*, pages 616–617, Montreal, Canada, 2003. Springer.

[6] J. Haines, D. Kewley Ryder, L. Tinnel, and S. Taylor. Validation of sensor alert correlators. *IEEE Security and Privacy*, 1(1):46–56, 2003.

[7] J. Han and M. Kamber. *Data Mining - Concepts and Techniques*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2001.

[8] M. V. Mahoney and P. K. Chan. An analysis of the 1999 darpa/lincoln laboratory evaluation data for network anomaly detection. In *Recent Advances in Intrusion Detection*, volume 2820 of *Lecture Notes in Computer Science*, pages 220–237, Pittsburgh, PA, USA, 2003. Springer.

[9] S. Mukkamala, G. Janoski, and A. Sung. Intrusion detection using neural networks and support vector machines. In *Proceedings of the 2002 International Joint Conference on Neural Networks (IJCNN '02)*, pages 1702–1707, Honolulu, HI, 2002. Institute of Electrical and Electronics Engineers Inc.

[10] J. Munson and S. Wimer. Watcher the missing piece of the security puzzle. In *Proceedings of the 17th Annual Computer Security Applications Conference*, pages 230–9, New Orleans, LA, USA, 2001. IEEE Comput. Soc.

[11] U. of California Irvine. The uci kdd archive, 2003. http//kdd.ics.uci.edu Acc. February 2004.

[12] L. Portnoy, E. Eskin, and S. Stolfo. Intrusion detection with unlabeled data using clustering. In *ACM Workshop on Data Mining Applied to Security*, 2001.

[13] Safeguard. The safeguard project, 2003. http//www.ist-safeguard.org/ Acc. May 2004.

[14] K. Sequeira and M. Zaki. Admit anomaly-based data mining for intrusions. In *Proceedings of the 8th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 386–395, Edmonton, Alberta, Canada, 2002. ACM Press.

[15] T. Zhang, R. Ramakrishnan, and M. Livny. Birch an efficient data clustering method for very large databases. *SIGMOD Record 1996 ACM SIGMOD International Conference on Management of Data*, 25(2):103–14, 1996.