Adaptive Real-Time Anomaly Detection with Fast Indexing 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.

Keywords: Anomaly detection, Adaptive, Real-time, Clustering, Forgetting

1. Introduction
Intrusion detection is an important part of computer system defense. 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 [4, 5, 9]. 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 Cluster-
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. Many different techniques have been applied to pure anomaly detection, including decision trees [11], hidden markov models [3], artificial neural networks [7], self organizing maps [17], finite automata [7] and computer immune systems [22]. Munson and Winer [14] 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 the 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 [10], 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
3 The Safeguard Context

Safeguard [18] was a European research project aiming to enhance survivability of critical infrastructures by using agent technology. The Safeguard agent architecture is presented in Figure 1. This architecture was evaluated in the context of telecommunication and energy distribution networks. Agents were used to improve survivability of those large complex critical infrastructures (LCCIs), by detecting and handling intrusions as well as faults in the protected systems. The key to a generic solution applicable in many infrastructures is in the definition of roles for various agents. There may be several instances of each agent in each LCCI. They run on a platform (middleware) that provides generic services such as discovery and messaging services. The agents are believed to be common for the defence of many infrastructures, but should be instantiated to more specific roles in each domain. The generic roles can be described as follows:

- **Wrapper agents** wrap standard INFOSEC devices and existing LCCI diagnosis mechanisms, and provide their outputs after some filtering and normalisation for use by other agents.
- **Topology agents** gather dynamic network topology information, e.g. host types, operating system types, services provided, known vulnerabilities.
- **Hybrid detector agents** utilise domain knowledge for a given infrastructure, but combine it with behavioural detection mechanisms (e.g. anomaly detection with white lists).
- **Correlation agents** identify problems that are difficult to diagnose with one source of information in the network, by using several sources of information from wrapper, topology, or hybrid detector agents. They use the data sources to order, filter and focus on certain alarms, or predict reduced availability of network critical services. One type of correlation agent performs adaptive filtering and aggregation to further reduce the alarm rates.
- **Action agents** enable automatic and semi automatic responses when evaluation of a problem is finished.
- **Negotiation agents** communicate with agents in other LCCIs to request services and pass on information about major security alarms.
- **HMI (Human-Machine Interface) agents** provide an appropriate interface, including overview, for one or many system operators.
- **Actuator agents** are wrappers for interacting with lower layer software and hardware (e.g. changing firewall rules).

In the context of a management network for telecom service providers we have identified the following needs:

- Reducing information overload ([4, 5] and [2] on aggregation)
- Increasing coverage by providing new sources of information (this paper and [2])
- Increasing information quality by reducing false positives [4, 5]
- Collating information, such as correlating alarms [4, 5] and combining with topology information
- Presenting a global view of a network (in Safeguard Demonstrator)

In this paper we revisit the anomaly detection engine for an instance of the Hybrid detection agent [2]. The agent combines a clustering based anomaly detection engine (ADWICE) with a white-list engine. The white-list engine implements simple specification based intrusion detection [19] where data known to be normal are described by manually constructed signatures. Data considered normal by the white-list engine are not fed into ADWICE. This reduces the size of the normality model without decreasing detection coverage. In our case hosts and services known to produce abnormal behaviour (e.g. DNS server port 53) are filtered away, but rules for arbitrary features can be used.

The Hybrid detection agent including ADWICE is designed to be a generic solution. Given implementation of a small data collection and pre-processing module the agent can be applied to different types of data (e.g. network or host data). Of course the selection of data source and data features influences what types of attacks can be detected.

4 The Anomaly Detection algorithm

This section describes how ADWICE handles training and detection using the new index\(^1\). The present implementation requires data to be numeric. Data is therefore assumed to be transformed into numeric format by pre-processing.

4.1 Basic concepts

The basic clustering concepts are presented in the original BIRCH paper [23] and the relevant parts are summarized here. An ADWICE model consists of a number of

\(^1\)More details on the old index may be found in the original paper [2]
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 $CF = (n, S, SS)$ where $n$ is the number of data points in the cluster, $S$ is the linear sum of the $n$ data points and $SS$ is the square sum of the $n$ data points. From now on we represent clusters by cluster features ($CF$).

Given $n$ $d$-dimensional data vectors $v_i | i = 1 \ldots n$ in a cluster $CF$ the centroid $v_0$ is defined as $v_0 = \sum_i v_i / n$, which may be computed given only the summary $CF$. The distance between a data point $v$ and a cluster $CF$ is the Euclidian distance between $v$ and the centroid, denoted $D(v, CF)$ 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 $CF_i = (n_i, S_i, SS_i)$ and $CF_j = (n_j, S_j, SS_j)$ are merged, the CF of the resulting cluster may be computed as $(n_i + n_j, S_i + S_j, SS_i + SS_j)$. 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 $v$. In the current implementation the threshold requirement is $D(v, CF_i) \leq T$.

4.2 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 $v$, 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 leaves.

**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 (similar) clusters may be merged, thereby reducing the size of the model. The new data point $v$ 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 [23] 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 are 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] was based on 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. Using cluster summaries as index in the context of local clustering resulted in a suboptimal search where the closest cluster is not always found. Although index errors do 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. The original BIRCH paper used a heuristic in the first phase and global clustering in a later phase to improve the quality of the clusters. Global clustering is not incremental, and the proposed heuristic may reduce index errors but not completely remove them. We therefore choose to implement a new grid-based index preserving the adaptability and good performance of ADWICE while aiming to completely eliminate the index errors of BIRCH.

4.3 New indexing mechanism

A subspace of d-dimensional space is defined by two vectors, \(ss_{Max}\) and \(ss_{Min}\), 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 2.

Each dimension \(d_i\) has a maximum value \(Max\) and a minimum value \(Min\), in the running example of Figure 2 \(Min_1 = Min_2 = 0\) 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 2), 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 4.2 is followed. Assume we want to train the model of Figure 2 with the new data point \(v = (0.28, 0.2)\) by inserting \(v\) in the closest cluster. Merging of a data point with a cluster \(CF_i\) may only be performed if the \(D(v, CF_i) < T\) 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 \(v\) in dimension \(i\) \(v[i]\) and \(SliceWidth_i\), we compute the slice number into which \(v\) fits, in the example slice 1. However, since \(v\) is close to the border between slices 1 and 2 it may be the case that the closest cluster is located in slice 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 1 and 2. No child exists for slice 2 due to lack of clusters in the corresponding subspace, eliminating the need for search in 2. In slice 1 we descend and find a leaf-cluster and return \(b\) as the closest cluster since \(D(v, b) < D(v, a)\). 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, \(v\) 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 \(v\), 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 4.5) 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 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.

### 4.4 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 $D(v, CF_i)$ from the centroid of the cluster to the new data point $v$ is computed. Informally, if $D$ is small, i.e. lower than a limit, $v$ is similar to data included in the normality model and $v$ should therefore be considered normal. If $D$ is large, $v$ is an anomaly.

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

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

A belief threshold (BT) is then used to make the final decision. If $v$ 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 $E_1 = E_2 = E$ so that $v$ 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.
4.5 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 re-training 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 forgetting 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.

5 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 [15] to be able to compare the results with earlier evaluations [2]. Despite the shortcomings of the DARPA related data sets [12] 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 connection 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 3 shows that the primitive search operation of the new grid-index (HashMap, the Java hash table used in implementation) 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 3). 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 telecommunication test network in the European Safeguard project [18] was used, at the time of evaluation consisting of 50 real machines (now more than 100). A period of three 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.

5.1 Detection quality of ADWICE

In all we have evaluated three different versions of ADWICE as shown in Figure 4. 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].

5.2 Incremental training

An important property of ADWICE is that the original model, known to truly reflect recent normality, does not 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 5 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 5. 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 5 shows how almost all alarms now disappear after the second incremental training period.

The need for the two-step incremental learning arose 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).

5.3 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 6 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.
6 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. \( M, \text{numSlices} \)) 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.

The attacker may try to avoid detection by crafting the attack data so that the intrusion detection system considers it to be normal. For rule-base system, if the rules are very specific to avoid false positives, a small change in the attack data may be sufficient to evade detection. When considering learning-based anomaly detection systems, the model needs to be a generalization of normal data to avoid an abundance of false positives due to slight variations in normal data. This generalization may give the attacker opportunities to make her attack look similar to normal data in the context of the normality model, thereby evading detection. The mimicry attack [21] is one such example. When introducing adaptation, another opportunity is presented to the attacker. The attacker could possibly fool the intrusion detection system by introducing data that is considered normal at the “border” of the normality model and thereby slowly over time train the model to include attacks. We call this an adaptation attack, illustrated by Figure 7.

One detection engine should not be used in isolation. If multiple detectors are deployed, depending on what correlation scheme is used, the attacker needs to evade all detectors at the same time to completely avoid detection. This should make evasion attacks much harder. Further study is needed of how fast adaptation should be to avoid a long period of false positives during adaptation and what implications this rate of adaptation will have for adaptation attacks.

Full evaluation of adaptation 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 main 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 ADWISE is a general and flexible anomaly detection scheme.

References


