View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by Missouri University of Science and Technology (Missouri S&T): Scholars' Mine

Missouri University of Science and Technology

Scholars' Mine

Computer Science Faculty Research & Creative Works

Computer Science

01 Apr 2007

An Automatically Tuning Intrusion Detection System

Zhenwei Yu

Jeffrey J.-P. Tsai

Thomas Weigert

Missouri University of Science and Technology

Follow this and additional works at: https://scholarsmine.mst.edu/comsci_facwork

Part of the Computer Sciences Commons

Recommended Citation

Z. Yu et al., "An Automatically Tuning Intrusion Detection System," *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, Institute of Electrical and Electronics Engineers (IEEE), Apr 2007.
The definitive version is available at https://doi.org/10.1109/TSMCB.2006.885306

This Article - Journal is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Computer Science Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

An Automatically Tuning Intrusion Detection System

Zhenwei Yu, Jeffrey J. P. Tsai, *Fellow, IEEE*, and Thomas Weigert

***Abstract*—An intrusion detection system (IDS) is a security layer used to detect ongoing intrusive activities in information systems. Traditionally, intrusion detection relies on extensive knowledge of security experts, in particular, on their familiarity with the computer system to be protected. To reduce this dependence, various data-mining and machine learning techniques have been deployed for intrusion detection. An IDS is usually working in a dynamically changing environment, which forces continuous tuning of the intrusion detection model, in order to maintain sufficient performance. The manual tuning process required by current systems depends on the system operators in working out the tuning solution and in integrating it into the detection model. In this paper, an automatically tuning IDS (ATIDS) is presented. The proposed system will automatically tune the detection model on-the-fly according to the feedback provided by the system operator when false predictions are encountered. The system is evaluated using the KDDCup'99 intrusion detection dataset. Experimental results show that the system achieves up to 35% improvement in terms of misclassification cost when compared with a system lacking the tuning feature. If only 10% false predictions are used to tune the model, the system still achieves about 30% improvement. Moreover, when tuning is not delayed too long, the system can achieve about 20% improvement, with only 1.3% of the false predictions used to tune the model. The results of the experiments show that a practical system can be built based on ATIDS: system operators can focus on verification of predictions with low confidence, as only those predictions determined to be false will be used to tune the detection model.**

***Index Terms*—Attack detection model, classification, data mining, intrusion detection, learning algorithm, model-tuning algorithm, self-organizing map (SOM).**

I. Introduction

INTRUSION detection is a process used to identify abnormal activities in a computer system. Traditional intrusion detection relies on the extensive knowledge of security experts, in particular, on their familiarity with the computer system to be protected. To reduce this dependence, various data-mining and machine learning techniques have been used in research projects: Audit Data Analysis and Mining (ADAM) [1] combined the mining of association rules and classification to discover attacks from network traffic data. The Information and Systems Assurance Laboratory (ISA) intrusion detection system (IDS) employed multiple statistics-based analysis techniques, including chi-square [2] and exponentially weighted moving averages based on statistical process control [3], multivariate statistical analysis based on Hotelling's T^2 test [4], and clustering [2]. Mining Audit Data for Automated Models for Intrusion Detection (MAMAD ID) [5] applied association rules and a frequent episodes program. The Minnesota INtrusion Detection System (MINDS) [6] included a density-based outlier detection module and an association-pattern analysis module to summarize network connections.

However, Julish pointed out that data-mining-based intrusion detection usually relies on unrealistic assumptions on the availability and quality of training data [7], which causes detection models built on such training data to gradually lose efficiency in detecting intrusions as the real-time environment undergoes continuous change.

The quality of training data has a large effect on the learned model. In intrusion detection, however, it is difficult to collect high-quality training data. New attacks leveraging newly discovered security weaknesses emerge quickly and frequently. It is impossible to collect all related data on those new attacks to train a detection model before those attacks are detected and understood. In addition, due to the new hardware and software deployed in the system, system and user behaviors will keep on changing, which causes degradation in the performance of detection models. As a consequence, a fixed detection model is not suitable for an IDS. Instead, after an IDS is deployed, its detection model has to be tuned continually. For commercial products (mainly signature/misuse-based IDS), the main tuning method has been to filter out signatures to avoid generating noise [8] and add new signatures. In data-mining-based intrusion detection, system parameters are adjusted to balance the detection and false rates. Such tuning is coarse, and the procedure must be performed manually by the system operator. Other methods that have been proposed rely on "plugging in" a special purpose submodel [9] or superseding the current model by dynamically mined new models [10]–[12]. Training a special-purpose model forces the user to collect and construct high-quality training data. Mining a new model in real time from unverified data incurs the risk that the model could be trained by an experienced intruder to accept abnormal data.

In this paper, we present an automatically tuning IDS (ATIDS). Our system takes advantage of the analysis of alarms by the system operators: the detection model is tuned on-the-fly with the verified data, yet the burden on the system operator is minimized. Experimental results show that the system achieves up to 35% improvement in terms of misclassification cost compared with the performance of a system lacking the model-tuning procedure. If only 10% false predictions are used to tune the model, the system still achieves roughly 30% improvement. When tuning is delayed only a short time, the system achieves about 20% improvement with only 1.3% false predictions used to tune the model. Selective verification on predictions with low

Manuscript received April 2, 2005; revised October 14, 2005. This work was supported in part by a grant from Motorola. This paper was recommended by Associate Editor N. Chawla.

Z. Yu and J. J. P. Tsai are with the Department of Computer Science, University of Illinois at Chicago, Chicago, IL 60607 USA (e-mail: zyu@cs.uic.edu; tsai@cs.uic.edu).

T. Weigert is with Motorola, Schaumburg, IL 60196 USA.

Digital Object Identifier 10.1109/TSMCB.2006.885306

1083-4419/$25.00 © 2007 IEEE

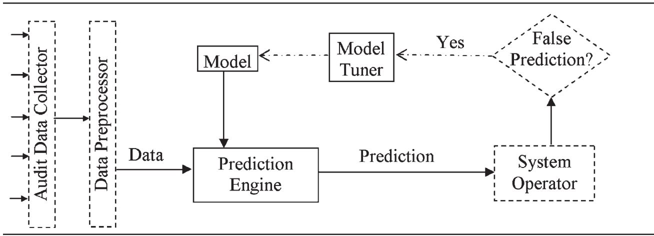

Fig. 1. System architecture of ATIDS.

confidence can further be used to improve efficiency by avoiding verification of high-confidence (typically true) predictions.

We present system architecture and core system components in Section II. To tune a model, two problems must be addressed. The first problem is how to tune the model, which is addressed by the algorithm presented in Section II-C. The second problem is when to tune the model. We present experiments regarding this issue in Section IV. Section III discusses the evaluation of the performance of an IDS and introduces the dataset used in our analysis. Experimental results obtained on this dataset are shown in Section IV. Section V surveys related work.

II. System Architecture

An IDS is a monitoring system which reports alarms to the system operator whenever it infers from its detection model that abnormal conditions exist. The system operator must verify the correctness of the reported alarms and execute defensive actions against an intruder. Our IDS relies on the fact that verified results provide useful information to further improve the IDS. Fig. 1 depicts the system architecture of our proposed ATIDS; we included the system operator to show the interaction between the user and the system. The prediction model is created in the training stage. The prediction engine analyzes and evaluates each obtained data record according to the prediction model. The prediction results are reported to the system operator. The system operator verifies the results and marks false predictions, which are then fed back to the model tuner. The model tuner automatically tunes the model according to the feedback received from the system operator. We discuss these three components in detail, ignoring the data collector and preprocessor familiar from conventional systems.

A. Prediction Model and Learning Algorithm

Different model representations have been used in detection models presented in the literature, among them are rules (signatures) [1], [5], decision trees [13], neural networks [14], statistical models [2], [3], or Petri nets [15]. In order to allow tuning parts of the model easily and precisely without affecting the rest of the model, we choose rules to represent the prediction model. In an earlier study, this model has demonstrated a good performance [16]. Our model consists of a set of binary classifiers learned from the training dataset by the simple learner with iterative pruning to produce error reduction (SLIPPER) [17], a binary learning algorithm. The initial creation of the detection model is shown in the block diagram in Fig. 2. The preprocessor prepares all binary training datasets from the original training dataset. The algorithm capturing this

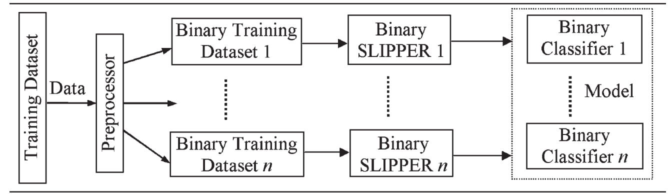

Fig. 2. Creation of initial model for ATIDS.

r2l	1775.6	0	IF service = other, logged_in = 1.
r2l	755.992	0	IF service = imap4, count <= 4.
r2l	428.9	0	IF num_failed_logins >= 1, src_bytes >= 104.
r2l	283.005	0	IF service = login duration >= 67.
....			......
r2l	0.0990266	0.000523598	IF num_access_files >= 1.
r2l	0.338368	0.00415511	IF service = ftp_data, duration <= 5068, count <= 2.
r2l	0.15948	0.00364366	IF service = ftp_data, dst_bytes <= 5.
unknown	0	0.927592	IF.
epsilon 3.21564e-05			

Fig. 3. Example of binary classifier.

preprocessor and the details of creating the prediction model have been described in [16].

The binary SLIPPER learning algorithm proposed by Cohen and Singer [17] is a general-purpose rule-learning system based on confidence-rate boosting [18]. A weak learner is boosted to find a single weak hypothesis (an IF–THEN rule), and then, the training data are reweighted for the next round of boosting. Unlike other conventional rule learners, data covered by learned rules are not removed from the training set. Such data are given lower weights in subsequent boosting rounds. All weak hypotheses from each round of boosting are compressed and simplified, and then combined into a strong hypothesis, constituting a binary classifier. An example of a binary classifier is shown in Fig. 3. This example is part of a binary classifier of the initial model in our system described below. Each rule starts with a predictive label (such as "r2l" or "unknown" in Fig. 3), followed by two parameters used to calculate the confidence in predictions made by this rule. The keyword "IF" introduces the conditions of the rule. These conditions are used to check whether the rule covers a data sample.

In SLIPPER [17], a rule R is forced to abstain on all data records not covered by R and predicts with the same confidence C_R on every data record x covered by R

$$C_R = \begin{cases} \frac{1}{2}\ln\left(\frac{W_+}{W_-}\right), & \text{if } x \in R \\ 0, & \text{if } x \notin R. \end{cases} \quad (1)$$

W_+ and W_- represent the total weights of the positive and negative data records, respectively, covered by rule R in the round of boosting the rule, which was built in. For example, for the first rule in the binary classifier shown in Fig. 3, W_+ is 1775.6 and W_- is zero. In order to avoid "divide by zero" errors when W_- is zero, (1) is transformed to (10) by adding a small value ε. Although the data are reweighted in each round of boosting during the training phase, the confidence value associated with a rule will remain unchanged once the rule has been included in the final rule set.

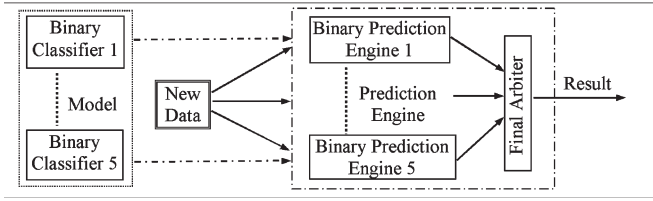

Fig. 4. Prediction on new data in ATIDS.

In SLIPPER, an objective function such as (6) from [17] is used to search for a good rule with positive confidence during each round of boosting. The selected rule with positive confidence is compared with a default rule with negative confidence to determine the result of boosting. A default rule covers all data records and, thus, does not have conditions; all default rules are compressed into a single final default rule. For example, the last rule in Fig. 3 is a final default rule.

SLIPPER is a time-efficient learning algorithm. For example, it took 2 h to learn a model from roughly half million training records on a Pentium IV system with a 512-MB RAM running at 2.6 GHz.

B. Prediction Engine

Binary learning algorithms can only build binary classifiers. For intrusion detection, the minimal requirement is to alarm in case intrusive activity is detected. Beyond alarms, operators expect that the IDS will report more details regarding possible attacks, at least the attack type. We group attacks into categories such as denial-of-service (dos), probing (probe), remote-to-local (r2l), and user-to-root (u2r). Correspondingly, we constructed five binary classifiers from the training dataset. One binary classifier ("BC-Normal") predicts whether the input data record is normal. The other four binary classifiers ("BC-Probe," "BC-Dos," "BC-U2r," and "BC-R2l") predict whether the input data record constitutes a particular attack. For example, the binary classifier "BC-Probe" predicts whether the input data record is a probing attack. The prediction engine in our system consists of five binary prediction engines together with a final arbiter, as shown in Fig. 4. We refer to this multiclassifier version of SLIPPER as MC-SLIPPER. The training procedure used to construct the initial model for MC-SLIPPER is described in detail in [16]. Each binary prediction engine outputs a prediction result on the input data according to its binary classifier, and the final arbiter determines and reports the result to the system operator.

The binary prediction engine is the same as the final hypothesis in SLIPPER [17], which is

$$H(x) = \operatorname{sign}\left(\sum_{R_t : x \in R_t} C_{R_t}\right). \tag{2}$$

In other words, the binary prediction engine sums up the confidence values of all rules that cover the available data. A positive sum represents a positive prediction. The magnitude of the sum represents the confidence in the prediction. For example, if some rules in the binary classifier shown in Fig. 3 cover a data record and the sum of confidence values of these rules is 10.0, then the corresponding binary prediction engine will output the result "r2l, 10.0." This result means that the data are r2l attack and the confidence in the prediction is 10.0. If some other rules in this classifier cover another data record but the sum of the confidence values of those rules is -5.0, then the corresponding binary prediction engine will output the result "unknown, 5.0." This result means that the corresponding binary prediction engine does not think that this data record represents a r2l attack and its confidence in this prediction is 5.0. But, the prediction engine does not know whether this data record is normal or belongs to a different attack type.

Obviously, conflicts may arise between the results of the different binary prediction engines. For instance, there might be more than one positive prediction on a single data record, or there may be no positive prediction. We implemented three arbitration strategies to resolve such conflicts [16]. In the first strategy ("by confidence"), we compare the prediction confidence (PC) obtained from all binary classifiers and choose the prediction with the highest confidence as the final result [see (3)]

$$i = \{j | \mathrm{PC}_j = \mathrm{MAX}\{\mathrm{PC}_1, \mathrm{PC}_2, \ldots, \mathrm{PC}_n\}\}. \tag{3}$$

$\mathrm{PC}_1, \ldots, \mathrm{PC}_n$ stands for the PC values from binary prediction engine 1 through n. The result i indicates the ith class label (normal or attack type) representing the final result.

A comparison is made between PC values from different binary classifiers. Such comparison may not be fair because each binary classifier has its distinct rule set. Our second arbitration strategy ("by confidence ratio") compares the PC ratios (PCR), which are relative values

$$i = \{j | \mathrm{PCR}_j = \mathrm{MAX}\{\mathrm{PCR}_1, \mathrm{PCR}_2, \ldots, \mathrm{PCR}_n\}\}. \tag{4}$$

$\mathrm{PCR}_j = \mathrm{PC}_j / \mathrm{MAX}\{\mathrm{PC}_j^1, \mathrm{PC}_j^2, \ldots, \mathrm{PC}_j^m\}$, where PC_j stands for the PC of prediction engine j on the data record in the test dataset and PC_j^m stands for the PC values of prediction engine j on all data from the mth example in the training dataset.

For the third arbitration strategy, we built a three-layer back-propagation (BP) neural network [19] with five input, seven hidden, and five output nodes, which takes confidence ratios as inputs and selects the class with the highest output as the final class, as shown in (5)

$$i = \{j | \mathrm{NNO}_j = \mathrm{MAX}\{\mathrm{NNO}_1, \mathrm{NNO}_2, \ldots, \mathrm{NNO}_n\}\}. \tag{5}$$

$\mathrm{NNO}_j = \sum_k (\omega_{jk} \cdot \sum(\omega_{kl} \cdot \mathrm{PCR}_l))$ stands for the output of the neural network via the output node j. ω_{jk} and ω_{kl} are weights on the connection between output node j and hidden node k and the connection between hidden node k and input node l, respectively.

C. Model-Tuning Algorithm

Each classifier in MC-SLIPPER is a set of rules. As shown in (2), only rules that cover a data record contribute to the final prediction on this data record. This property ensures that, if we change a rule, the prediction on data records not covered by this

rule will not be affected. Having only minor side effects during tuning is essential to ensure performance improvement from tuning. We do not change rules; new rules can be learned using other methods, such as incremental learning techniques, should such are proven to be necessary. During tuning, we change the associated confidence values to adjust the contribution of each rule to the binary prediction. Consequentially, tuning ensures that, if a data record is covered by a rule in the original model, then, it will be covered by this rule also in the tuned model and vice versa. To limit possible side effects, we only change the associated confidence values of positive rules as a default rule covers every data record.

When a binary classifier is used to predict a new data record, two different types of false predictions may be generated according to the sum of confidence values of all rules that cover this data record. When the sum is positive, the binary classifier predicts the data record to be in the positive class. If this prediction is false, it is treated as a false positive prediction and labeled "P." When the sum is negative, the binary classifier predicts the data record to be in the negative class. If this prediction is false, it is considered a false negative prediction and labeled "N." We use the label "T" to indicate a true prediction. The sequence of prediction results can then be written as: $\ldots\{\mathrm{P}\}^l\{\mathrm{N}\}^i\{\mathrm{T}\}^j\ldots$ where $l > 0$, i and $j \geq 0$, and $i + j > 0$. Obviously, when the classifier makes a false positive prediction, the confidence values of those positive rules involved should be decreased to avoid the false positive prediction made by these rules on the subsequent data. When the classifier makes a false negative prediction, the confidence values of the positive rules involved should be increased to avoid false negative predictions made by these rules on the successive data. Formally

$$C'_R = \begin{cases} p \cdot C_R, & \text{if } R \propto \mathrm{P} \\ q \cdot C_R, & \text{if } R \propto \mathrm{N} \end{cases} \tag{6}$$

where $p < 1$, $q > 1$, and $R \propto \mathrm{P}$ imply that a positive rule R contributes to a false positive prediction. We multiply a pair of relative values (p and q) with the original confidence values to adjust these values, rather than adding or subtracting a constant, for the following two reasons. For one, in SLIPPER, all rules except the default rule have positive confidence values. Using relative values, we can ensure that new confidence values remain positive, although they may be very small when the adjustment procedure is repeated many times in the same direction. Further, confidence values are quite different for each rule. Without additional information, it is difficult to choose a reasonable constant for all rules.

If the updating is performed n times, the sum of the confidence values of the positive rules will be

$$\sum C'_R = \begin{cases} p^n \cdot \sum C_R, & \text{if } R \propto \mathrm{P} \\ q^n \cdot \sum C_R, & \text{if } R \propto \mathrm{N}. \end{cases} \tag{7}$$

Because $p < 1$, $q > 1$, and the confidence value of the default rule is unchanged, trivially, there exists a number n, such that, after updating the confidence values n times, the sign of the sum of the confidence values of all rules (both positive rules and the default rule) will be changed. That means the tuned classifier could make a true prediction on the data where the original classifier made a false prediction. Formally, $\exists n, (\{\mathrm{P}\}^n)_o \Rightarrow (\{\mathrm{P}\}^{n-1}\mathrm{T})_t$, where $()_o$ represents the sequence of prediction results based on the original classifier and $()_t$ stands for the sequence of prediction results from the tuned classifier.

Because confidence values are updated after the user identifies a false prediction and the tuned classifier cannot be used to predict the data again when the original classifier made a false prediction, the benefit of such tuning depends on the subsequent data. Note that $(\mathrm{PN})_o \Rightarrow (\mathrm{PN})_t$ as the classifier decreases the confidence values after it makes a false positive prediction. We only discuss the tuning after a false positive prediction; tuning after a false negative prediction is similar. Assume that tuning is aggressive by setting appropriate values for p and q in (6). After tuning on a false positive prediction, the classifier would make a true prediction on the data over which the original classifier made a false positive prediction, but it might still make a false negative prediction on the data over which the original classifier made a true prediction. This assumption can be written as

$$\begin{aligned} (\mathrm{PP})_o &\Rightarrow (\mathrm{PT})_t \\ (\mathrm{PT})_o &\Rightarrow (\mathrm{PN})_t. \end{aligned} \tag{8}$$

Tuning improves performance when the misclassification cost decreases. We consider the benefit of tuning B_t to be the change in misclassification cost. If tuning corrects a false prediction, then we shall say that it gains benefit (+1); if tuning incorrectly changes a formerly true prediction, it loses benefit (−1). For any general sequence of prediction results $\ldots\{\mathrm{P}\}^l\{\mathrm{N}\}^i\{\mathrm{T}\}^j\ldots$, where $l > 0$, i and $j \geq 0$, and $i + j > 0$, the following are observed.

1) If $l = 1$ and $i \neq 0$, then $(\mathrm{PN})_o \Rightarrow (\mathrm{PN})_t$, tuning will be performed on a false negative prediction, and tuning neither gains nor loses benefit.
2) If $l = 1$ and $i = 0$, but $j \neq 0$, then $(\mathrm{PT})_o \Rightarrow (\mathrm{PN})_t$, tuning loses benefit (−1), and further tuning will be performed on a false negative prediction.
3) If $l = 2$ and $i \neq 0$, then $(\mathrm{PPN})_o \Rightarrow (\mathrm{PTN})_t$, tuning gains benefit (+1), and further tuning will be performed on a false negative prediction.
4) If $l = 2$ and $i = 0$, but $j \neq 0$, then $(\mathrm{PPT})_o \Rightarrow (\mathrm{PTN})_t$, tuning neither gains nor loses benefit, and further tuning will be performed on a false negative prediction.
5) If $l = 3$ and $i \neq 0$, then $(\mathrm{PPPN})_o \Rightarrow (\mathrm{PTTN})_t$, tuning gains benefit (+2), and further tuning will be performed on a false negative prediction.
6) If $l = 3$ and $i = 0$, but $j \neq 0$, then $(\mathrm{PPPT})_o \Rightarrow (\mathrm{PTTN})_t$, tuning gains benefit (+2), and further tuning will be performed on a false negative prediction.
7) When $l > 3$, $(\{\mathrm{P}\}^l)_o \Rightarrow (\mathrm{P}\{\mathrm{T}\}^{l-1})_t$, tuning gains benefit $(+(l-1))$.

The total benefit of tuning on all false predictions is calculated by

$$B_t = -\sum(\mathrm{PT})_o + \sum(\mathrm{PPN})_o + 2 \cdot \sum(\mathrm{PPP})_o + \cdots + (l-1) \cdot \sum\left(\{\mathrm{P}\}^l\right)_o + \cdots \tag{9}$$

where $l > 3$ and $\sum(\ldots)_o$ is the count of pattern $(\ldots)$ in the sequence of prediction results of the original classifier. The

total benefit of tuning depends on how many continuous false positive predictions the original classifier made on the dataset.

In SLIPPER, each rule has two parameters, W_+ and W_-. They are computed from the training dataset in the round of boosting when the rule is constructed. In practice, the confidence value is "smoothed" by adding ε to avoid extreme confidence values

$$C_R = \frac{1}{2}\ln\left(\frac{W_+ + \varepsilon}{W_- + \varepsilon}\right). \tag{10}$$

The number of choices for the pair of values p and q is large. For example, we might choose 0.75 and 1.5 to adjust the confidence values conservatively, or we might choose 0.25 and 4 to adjust the confidence values more aggressively. Confidence values could also be adjusted by relying on dynamic relative values. In this paper, we set $p = 0.5$ and $q = 2$. According to (6), p can be any value greater than zero but less than one. We choose the middle value for p to avoid aggressive updating yet keep tuning efficient. We set q to be $1/p$ to ensure that the original confidence value is restored when it turns out that tuning had led to an incorrect prediction and the rule is subsequently tuned again to yield the original prediction.

When a false positive prediction is made on a data record, from the point of view of training, this is a negative data record for those rules that cover the data but misclassified it as a positive data record. Therefore, we can move some weights from W_+ to W_- while keeping the sum of weights unchanged as follows:

$$\begin{aligned} C'_R &= \frac{1}{2}\ln\left(\frac{W_+ + \varepsilon}{W_- + \varepsilon}\right) \\ &= p \cdot C_R = \frac{1}{2}\cdot\frac{1}{2}\ln\left(\frac{W_+ + \varepsilon}{W_- + \varepsilon}\right) \\ W'_+ + W'_- &= W_+ + W_-. \end{aligned} \tag{11}$$

By solving (11), we get new values for W'_+ and W'_-

$$\begin{aligned} W'_+ &= \frac{\sqrt{W_+ + \varepsilon}}{\sqrt{W_+ + \varepsilon} + \sqrt{W_- + \varepsilon}}(W_+ + \varepsilon + W_- + \varepsilon) - \varepsilon \\ W'_- &= \frac{\sqrt{W_- + \varepsilon}}{\sqrt{W_+ + \varepsilon} + \sqrt{W_- + \varepsilon}}(W_+ + \varepsilon + W_- + \varepsilon) - \varepsilon. \end{aligned} \tag{12}$$

Similarly, if a false negative prediction is made on a data record, the data can be considered a positive data record for those rules that cover it but misclassified it as a negative data record. Again, we can move some weights from W_- to W_+ while keeping the sum of the weights unchanged

$$\begin{aligned} C'_R &= \frac{1}{2}\ln\left(\frac{W'_+ + \varepsilon}{W'_- + \varepsilon}\right) \\ &= q \cdot C_R = 2\cdot\frac{1}{2}\ln\left(\frac{W_+ + \varepsilon}{W_- + \varepsilon}\right) \\ W'_+ + W'_- &= W_+ + W_-. \end{aligned} \tag{13}$$

Solving (13) yields new values for W'_+ and W'_-

$$\begin{aligned} W'_+ &= \frac{(W_+ + \varepsilon)^2}{(W_+ + \varepsilon)^2 + (W_- + \varepsilon)^2}(W_+ + \varepsilon + W_- + \varepsilon) - \varepsilon \\ W'_- &= \frac{(W_- + \varepsilon)^2}{(W_+ + \varepsilon)^2 + (W_- + \varepsilon)^2}(W_+ + \varepsilon + W_- + \varepsilon) - \varepsilon. \end{aligned} \tag{14}$$

III. Performance Evaluation

In this section, we will examine how to assess the performance of an IDS and how to improve the system based on the experimental data. We will rely on the KDDCup'99 dataset provided by Defense Advanced Research Projects Agency (DARPA) as this dataset contains several weeks of attack data and has been used to assess the performance of a number of IDS. While this dataset contained labeled data, in order to mitigate the burden of manually labeling training data in real-life situations, we developed a supporting tool. We will use the total misclassification cost (TMC) as the primary indicator of system performance. In order to be able to improve our system based on the experimental results, we also develop a methodology of studying the performance of individual rules.

A. Dataset

A proper dataset must be obtained to facilitate experimentation. In our experimental environment, it was difficult to obtain real-life datasets due to limitations of network size and limited external access. Unfortunately, usable datasets are rarely published as these involve sensitive information such as the network architecture, security mechanisms, and so on. Thus, in this paper, we rely on the publicly available KDDCup'99 intrusion detection dataset. This dataset was collected from a network simulating a typical U.S. Air Force LAN and also reflects dynamic change within the network.

The KDDCup'99 intrusion detection dataset was developed based on the 1998 DARPA intrusion detection evaluation program, prepared and managed by the MIT Lincoln Laboratories. The objective of this program was to survey and evaluate intrusion detection research. Lincoln Laboratories set up an environment to acquire nine weeks of raw TCP data for a local-area network (LAN) simulating a typical U.S. Air Force LAN. This LAN was operated as if it is a true Air Force environment, and it was subjected to multiple attacks. The raw training data dump was about 4 GB of compressed binary TCP data from the first seven weeks of network traffic alone. The data dump was processed into roughly five million connection records. The test data were constructed from the network traffic in the last two weeks, which yielded around two million connection records.

In the KDDCup'99 dataset, each record represents a TCP/IP network connection with a total of 41 features. Domain experts derived some of the features related to content [5]. Statistical features were generated using a 2-s time window. Five classes of connections were identified, including normal network connections. The four classes of abnormal connections (attacks) are dos, probing (probe), r2l, and u2r. Each attack class is further divided into subclasses. For example, class dos includes subclass smurf, neptune, back, teardrop, and so on, representing

TABLE I
DATA DISTRIBUTION IN THE KDDCUP'99 DATASET

	normal	probe	dos	u2r	r2l
Training Data	19.69%	0.83%	79.24%	0.01%	0.23%
Test Data	19.48%	1.34%	73.90%	0.07%	5.21%

popular types of dos attacks. Two training datasets from the first seven weeks of network traffic are available. The full dataset includes about five million records and a smaller subset containing only 10% of the data records but with the same distribution as the full dataset. We used the 10% subset to train our binary SLIPPER classifiers.

The labeled test dataset includes 311 029 records with a different distribution of attacks, then the training dataset (see Table I). Only 22 out of the 39 attack subclasses in the test data were present in the training data. The different distributions between the training and test datasets and the new types of attacks in the test dataset reflect the dynamic change of the nature of attacks common in real-life systems. Sabhnani and Serpen analyzed the dissimilarity between the training dataset and the test dataset [20].

B. Labeling Tool

As a supervised learning system, the binary SLIPPER [17] requires labeled training data. The KDDCup'99 dataset contains both labeled training and test data, and we can easily learn a model by running SLIPPER on the labeled training data and verifying the model on the labeled test data. However, to build a detection model for a particular network, the training data should be collected from that network and labeled before it is used to learn a model. We developed a tool to automatically label the data and thus reduce the effort required to manually verify the training data, while keeping the mislabeling rate low. We use the labeled 10% subset of training data in the KDDCup'99 dataset to demonstrate the performance of our labeling tool.

The labeling tool attempts to cluster data using self-organizing map (SOM) unsupervised learning [21], which does not require labels on the input data. A SOM consists of an array of nodes. Each node contains a vector of weights of the same dimension as the input data. Given a set of data, training a SOM is a procedure to adjust the weights of the nodes. For any input data, the best matching node is selected according to its similarity to all other nodes. Then, the radius of the neighborhood of the best matching node is calculated, which will decrease as the training procedure proceeds. Only the weights of the neighboring nodes are adjusted to make them more similar to the input data. All training data will be grouped into multiple clusters after the training procedure has been repeated many times.

Usually, similarity is measured as the Euclidean distance between the input data and the nodes in the SOM, which works only on numerical features. However, nine out of the 41 features in the KDDCup'99 dataset are symbolic. Six of the symbolic features are binary and can be coded with "1" or "0." We adopt a perfect match policy on the remaining symbolic features ("protocol type," "service," and "flag") and treat them as numeric also. All training data are split into 210 sets based on the values of these three symbolic features; only data with identical values for these three features will be in the same set. The set size varied from 1 to 281 400. The values of the numeric features were normalized to a range between zero and one. The data in every set of size greater than ten were trained using a SOM (a total of 139 sets). The size of each SOM (in terms of the number of nodes) depends on the amount of data in the set. The number of nodes in a SOM is around 10% of the number of data elements in a set. To limit training time, we restrict the largest SOM to 400 nodes. Table II shows the initial settings of SOM size in this paper.

TABLE II
INITIAL SETTING OF SOM SIZE

dataset size	281,400-4,534	1,642-687	538-380	260-154	113-73	51-10
SOM size	20x20	10x10	7x7	4x4	3x3	2x2

After training a SOM, we calculate the following parameters for every input data element: the best matching node, the distance between the input data and the best matching node, the lowest distance to each node, and the average distance of all input data in the set. In this paper, 82 out of the 139 average distances are less than 0.01, 107 out of the 139 average distances are less than 0.02, and 127 out of the 139 average distances are less than 0.1. To improve the performance of the SOM in assigning labels to the input data, the average distance of all input data in a set is used to determine whether a new SOM should be trained for that dataset. To build a new SOM, we can increase its size or/and iteration number or just use different initial random weights.

Once we obtain a satisfying SOM, all the input data are clustered around some nodes. The input data with the lowest distance to its node must be verified and labeled manually. This label is then assigned to all other input data in the same cluster. We assume that, when input data are verified manually, the label is correct. In order to reduce error, additional input data could be verified and labeled manually. In this paper, we also manually labeled all data in a cluster whose distance is over ten times of its lowest distance when the lowest distance for that cluster to its node is over two times the average distance of all data in that set. We verified and manually labeled all data in the datasets of size less than 10. In this paper, 5205 out of the 494 021 input data in the KDDCup'99 10% training dataset (about 1.05%) was verified and labeled manually. Only 462 (about 0.094%) input data elements were mislabeled.

C. System Performance

We evaluated our system on the KDDCup'99 dataset and compared it with other systems, including a system built from PNrules [22], the KDDCup'99 winner [23] and runner-up [24], a multiple-classifier system [25], and a system based on repeated incremental pruning to produce error reduction (RIPPER) [26]. More details on these systems can be found in Section V.

An IDS generates alarms whenever it detects an attack while ignoring normal behavior. That is, any classification of network connection into an attack class will generate an alarm, while a classification as normal will not. Security officers will pay

TABLE III
PERFORMANCE COMPARISON OF VARIOUS IDS

System Name	TMC	Overall Accuracy
MC-SLIPPER by BP Network [16]	68,490	92.70%
MC-SLIPPER by Confidence Ratio [16]	70,177	92.59%
Multiple Classifier System [25]	71,096	93.04%
MC-SLIPPER by Confidence [16]	72,494	92.06%
KDDCup'99 Winner [23]	72,500	92.71%
KDDCup'99 Runner-up [24]	73,287	92.92%
Simple RIPPER [26]	73,622	92.66%
PNrule [22]	74,058	92.59%

TABLE IV
STATISTICAL DATA FOR MC-SLIPPER RULES ON TEST DATASET

Rule	P#	$l = 1$	$l = 2$	$l \geq 3$	$l \geq 100$	$l \geq 1000$	N#	T#	FPR
BC-Normal_29	17,335	21.80%	10.55%	67.66%	31.22%	22.43%	8	53,862	24.36%
BC-Normal_43	16,457	25.91%	15.59%	58.50%	3.01%	0.00%	41	45,999	26.40%
BC-Probe_17	1,471	6.19%	3.54%	90.28%	58.74%	0.00%	6	3,762	28.19%
BC-Probe_3	1,221	4.59%	4.10%	91.32%	70.84%	0.00%	22	2,845	30.41%
BC-Normal_50	2,384	13.76%	7.30%	78.94%	22.73%	0.00%	46	4143	36.97%
BC-Normal_22	10,141	17.00%	14.48%	68.52%	1.38%	0.00%	0	15,868	38.99%
BC-Normal_48	15,497	16.64%	11.74%	71.61%	17.22%	0.00%	13	23,965	39.29%
BC-Normal_49	13,805	12.85%	9.71%	77.44%	42.99%	38.22%	20	16,738	45.23%
BC-Probe_4	10,909	10.63%	8.78%	80.58%	34.11%	0.00%	0	12,218	47.17%
BC-Normal_28	9,179	7.31%	6.69%	86.00%	6.58%	0.00%	2	10,072	47.69%
BC-Normal_33	12,280	10.04%	9.90%	80.06%	2.96%	0.00%	1	12,223	50.12%
BC-Normal_45	5,018	2.21%	1.95%	95.83%	38.36%	0.00%	6	4,313	53.81%
BC-Normal_32	4,824	0.15%	0.00%	99.85%	99.85%	99.85%	3	3,164	60.41%
BC-Normal_11	4,875	0.04%	0.00%	99.96%	99.96%	99.96%	0	1,894	72.02%
BC-Probe_12	11,109	1.52%	1.73%	96.75%	59.84%	0.00%	0	2,787	79.94%
BC-Probe_16	40,917	0.08%	0.04%	99.88%	99.19%	87.17%	108	3,032	93.12%
BC-Probe_5	40,519	0.06%	0.03%	99.91%	99.53%	94.86%	0	2,368	94.48%
BC-Probe_7	40,705	0.02%	0.00%	99.97%	99.90%	99.19%	0	2,117	95.06%
BC-Normal_27	3,647	0.05%	0.11%	99.84%	96.22%	69.07%	0	22	99.40%

different levels of attention to different types of alarms. For example, probe alarms will typically be ignored, but an r2l alarm will be investigated and will result in counter measures, should it be determined to be true. To reflect the different levels of seriousness of alarms, a misclassification cost matrix defining the cost of each type of misclassification is used to evaluate the results of the KDDCup'99 competition. The TMC, which is the sum of the misclassification costs for all test data records, is the key measurement differentiating the performance of each system. Table III compares the performance of each evaluated system. We examine three systems based on MC-SLIPPER, considering the three arbitration strategies [16]. The TMC of our systems is 72 494, 70 177, and 68 490, respectively. All three are better than the KDDCup'99 contest winner, whose TMC is 72 500 [27]. MC-SLIPPER using BP neural network arbitration shows the best performance among the compared systems.

As shown in Table III, the MC-SLIPPER systems achieved excellent TMC. However, TMC does not tell us how well each rule performs on the test dataset. To assess system performance along this dimension, we first extract the sequence of prediction results for each rule from the experimental data. Table IV shows the statistical data for individual rules from different binary classifiers. For space reasons, we only show data for those 19 rules whose false prediction rates are greater than 20% and which cover more than 1% of all test data. These 19 rules contribute to 262 193 out of the 299 471 false positive predictions (87.55%) and have the biggest negative impact on the overall performance of our MC-SLIPPER system when tested on the KDDCup'99 test dataset. In Table IV, the name of rule "xxxx_29" stands for the rule 29 in binary classifier "xxxx." The column titled "P#" shows the false positive predictions to which a rule contributes. Similarly, columns "N#" and "T#" show the false negative predictions and true predictions for each rule, respectively. As can be seen from this table, the number of false negative predictions is small compared to the number of false positive predictions. The last column titled "FPR" is the overall false prediction rate, computed by $(\mathrm{P\#} + \mathrm{N\#})/(\mathrm{P\#} + \mathrm{N\#} + \mathrm{T\#})$. The l in the column titles refers to the number of successive false positive predictions. When l is equal to one, this false prediction is an isolated false prediction. We particularly examine the situations where long sequences of false positive predictions occur.

Table IV exhibits the following two properties of the performance of rules in MC-SLIPPER.

Property 1: Isolated false predictions exist for most rules and can amount to about 25% of all false predictions (see column titled "$l = 1$").

Property 2: Often, false predictions come in long successive prediction sequences (see the columns titled "$l \geq 3$," "$l \geq 100$," and "$l \geq 1000$"). Long successive false prediction sequences provide an opportunity for our system to benefit from model tuning.

Table IV does not show the data for every pattern where the length of successive false positive prediction sequences l is greater than or equal to three. Therefore, we cannot apply (9) to evaluate the benefit of tuning. Compared to the large number of false positive predictions shown in column "P#," the small number of false negative predictions shown in column "N#" can safely be ignored. To estimate the benefit of tuning from the data in Table IV, we first transform (9) to

$$B_t = -\sum(\mathrm{PT})_o - \sum(\mathrm{PN})_o + 2 \cdot \sum(\mathrm{PPP})_o + \cdots + (l-1) \cdot \sum\left(\{\mathrm{P}\}^l\right)_o + \cdots \tag{15}$$

where "$+\sum(\mathrm{PPN})_o$" is removed from (9) and "$-\sum(\mathrm{PN})_o$" is added. Then, combine $(\mathrm{PT})_o$ and $(\mathrm{PN})_o$ to $(\mathrm{P})_o$

$$B_t = -\sum(\mathrm{P})_o + 2 \cdot \sum(\mathrm{PPP})_o + \cdots + (l-1) \cdot \sum\left(\{\mathrm{P}\}^l\right)_o + \cdots. \tag{16}$$

Equation (16) can be rewritten to

$$B_t = -n_1 + \frac{2}{3}n_3 + \cdots + \frac{l-1}{l}n_l + \cdots \tag{17}$$

where $l > 3$ and n_l is the number of false positive predictions which are in the pattern $(\{\mathrm{P}\}^l)_o$. Obviously

$$B_t > -n_1 + \frac{2}{3}(n_3 + \cdots + n_l + \cdots). \tag{18}$$

The column titled "$l = 1$" shows the percentage of n_1, which is the number of false positive predictions in the pattern $(\mathrm{P})_o$, among all false positive predictions; the column titled "$l \geq 3$" shows the percentage of $n_3 + \cdots + n_l + \cdots$ among all false positive predictions.

To analyze the performance for an individual rule, we logged the data coverage and false rates of each rule on every 500 data records. The entire test dataset was divided into 623 sequential

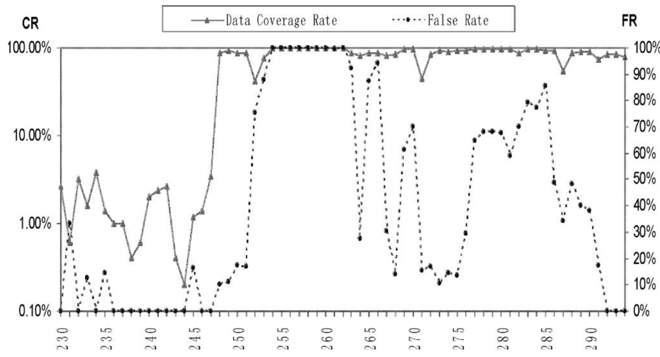

Fig. 5. Test performance of rule 29 of binary classifier "BC-Normal."

segments of 500 data records. We analyze the test performance of each rule and observe another property.

Property 3: The false rate of a rule might change dramatically even in adjoining segments.

We will explain these three properties by an example. The test performance of rule 29 in binary classifier "BC-Normal" on a subset of the data is shown in Fig. 5. The left vertical coordinates showing data coverage uses logarithmic scale. The test data start from segment 230 and run to segment 294. Rule 29 covers 71 205 out of the 311 029 test data records (22.89%), which are distributed among 267 segments out of all the 623 segments (42.86%). In this paper, 17 343 out of the 71 205 (24.36%) final predictions on the corresponding data records covered by rule 29 are false predictions. Rule 29 covers 20 987 out of the 32 500 (64.58%) test data records in the 65 segments shown in Fig. 5. In this paper, 11 750 out of the 20 987 (55.98%) final predictions on the corresponding data records covered by rule 29 in these 65 segments are false predictions.

We first examine segments 230 through 232. The false rates on these three successive segments are 0%, 33%, and 0%, respectively. Only three data records are covered by rule 29 in segment 231. The false prediction in this segment is an isolated false prediction. Note that isolated false predictions exist in many segments but cannot easily be shown in Fig. 5.

Property 2 states that the false predictions made on the data records covered by most of the rules are not distributed evenly on the entire test dataset; instead, they come in clusters. For rule 29, 11 750 out of the 17 343 (67.75%) false predictions are made on the data records in these 65 segments. The fact that 67.75% false predictions occur within 65 successive segments out of the 267 (24.34%) segments also demonstrates Property 1 and is explicit in Fig. 5 by the extreme high false rate (99%–100%) in the segments 254 through 262. Property 2 is helpful for model tuning to reduce false predictions. Using our tuning algorithm, the model is able make true predictions on the data records in segment 254 after a few rounds of tuning rule 29. Once the updated model makes true predictions, the model will keep making true predictions on all subsequent data records until it makes a false prediction on a data record that the original model made a true prediction on.

Fig. 5 also exhibits Property 3. Although the overall false prediction rate is 55.98% in these 65 segments, the false rate on a segment varies from 0% to 100%. For instance, the false rate drops from 92.12% for segment 263 to 27.60% for segment 264 and increases to 87.33% for segment 265.

```
LOOP
  INPUT Testdata (InputData):
    FinalResult = Predict (InputData);
    Feedback = Verify_Predition (InputData, FinalResult);
    IF ( is_false_ prediction (Feedback) Tune_Model (Feedback, FinalResult);
END
```

Fig. 6. Pseudocode for ATIDS with full and instant tuning.

TABLE V
PERFORMANCE OF ATIDS WITH FULL AND INSTANT TUNING

Final Arbiter	System Name	Overall Accuracy	TMC	% TMC
BP Network	ATIDS	95.06%	45,002	65.70%
	MC-SLIPPER	92.70%	68,490	
Confidence Ratio	ATIDS	95.52%	46,096	65.68%
	MC-SLIPPER	92.59%	70,177	
Confidence	ATIDS	95.53%	47,306	65.25%
	MC-SLIPPER	92.06%	72,494	

IV. EXPERIMENTS AND RESULTS

In ATIDS, the model is tuned with feedback from the last false prediction, and the updated model will be used to make predictions on the new data. Whether the tuning will yield more accurate predictions, it depends on the next covered data. If the next covered data are the same as or similar to the last covered data the original model made the false prediction on, the updated model must yield at least the same or a more accurate prediction (Property 2). But in the case of isolated false predictions, the next covered data will be sufficiently different from the last covered data where the original model made a true prediction that the updated model might make false predictions on the new covered data (Property 1). In this case, it would be better to stop tuning the model to maintain prediction accuracy. Unfortunately, the system cannot know whether the original model will predict correctly on the next covered data. In the following, we examine the performance of ATIDS with respect to how soon tuning is performed to avoid situations where tuning may lead to deterioration of performance.

A. Full and Instant Tuning

In the first experiment, we assume that the user has enough time to verify the result of every prediction, and every false prediction is identified and fed back to the model tuner. Before the system makes a prediction on the next data record, the model will be tuned instantly. The behavior of the system and its operator are summarized in a pseudocode notation in Fig. 6. In practice, prediction and verification can occur simultaneously as long as new data are not covered by a rule being verified. As long as tuning due to a false prediction by a rule is performed before new data covered by that rule are considered, tuning behaves as if instantaneously, although there is some delay.

We evaluated ATIDS with full and instant tuning on the KDDCup'99 test dataset. The results of ATIDS with full and instant tuning when compared with the original MC-SLIPPER system are shown in Table V.

Compared with the results of MC-SLIPPER, the TMCs of ATIDS drop roughly 35%, and the overall accuracy increases between 2.3% and 3.5%. When comparing the accuracy of each

TABLE VI
PREDICTION PERFORMANCE OF BINARY CLASSIFIERS

Binary Classifier	total number of rules	number of rules with false rate > 50%		number of rules with false rate > 20%	
		ATIDS	MC-SLIPPER	ATIDS	MC-SLIPPER
BC-Normal	52	1	9	4	24
BC-Probe	19	0	6	0	14
BC-Dos	19	0	0	1	1
BC-U2r	10	1	1	3	3
BC-R2l	18	0	6	0	13

binary classifier, almost all binary classifiers benefit from model tuning. Table VI summarizes the performance of each rule for each binary classifier both in ATIDS and MC-SLIPPER.

The model-tuning procedures in ATIDS are effective by taking advantage of Property 1. Especially for binary classifiers "BC-Probe" and "BC-R2l," almost one third of the rules of MC-SLIPPER have over 50% false rates, and more than two thirds of the rules have over 20% false rates. But in ATIDS, none of the rules have false rates over 20%. For binary classifier "BC-Normal," only one rule has over 50% false rate. Three other rules have over 20% false rates. The false rate for rule 29 in binary classifier "BC-Normal" drops from 24.34% to 6.64%. However, we notice that four rules exhibit worse performance after the tuning procedure. Rule 16 in binary classifier "BC-Normal" covers 22 data records, and the false rate increases from 4.55% to 9.09%. Rule 25 in binary classifier "BC-Normal" covers 123 data records, and the false rate increases from 5.69% to 12.20%. Rule 36 in binary classifier "BC-Normal" covers three data records, and the false rate increases from 33.33% to 66.67%. Rule 9 in binary classifier "BC-U2r" covers 62 data records, and the false rate increases from 24.19% to 29.03%. We analyzed the predictions on the data records covered by those four rules and found that Property 3 could be used to explain the degraded performance of rule 16 and rule 25 in binary classifier "BC-Normal." In the predictions made by MC-SLIPPER, the sole false prediction on the data covered by rule 16 is an isolated false prediction, and all the seven false predictions on the data covered by rule 25 are isolated false predictions. After tuning on rules 16 and 25, the tuned model makes a new false prediction on the next data record covered by rules 16 and 25, respectively, while MC-SLIPPER predicted correctly. For rule 36 in binary classifier "BC-Normal," ATIDS makes a false prediction on the first data record covered by rule 36 and the other three rules discussed, while MC-SLIPPER made a true prediction on that data record. The reason here is that the other three rules were tuned in ATIDS before they encountered this data record.

Model tuning using full and instant tuning is effective in improving system performance, but certainly not perfect. In any case, in practice, it is almost impossible for the user to verify every prediction due to the huge amount of data the system processes.

B. Partial But Instant Tuning

In the next experiment, ATIDS will again be tuned instantaneously, but only some false predictions are fed back to tune the model. For example, if only 20% of the false predictions can be caught in time and their corrections fed back into the detection

```
LOOP
  INPUT Testdata (InputData):
    FinalResult = Predict (InputData);
    Feedback = Verify_Predition (InputData, FinalResult);
    IF ( is_false_ prediction (Feedback) AND should_tune_model () )
      Tune_Model (Feedback, FinalResult);
END
```

Fig. 7. Pseudocode for ATIDS with partial and instant tuning.

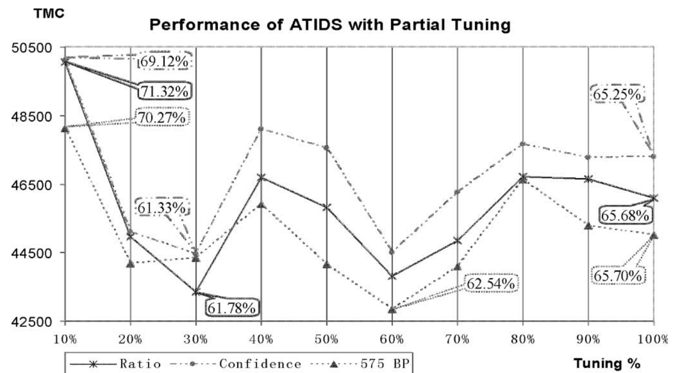

Fig. 8. Performance of ATIDS with partial tuning.

model, will model tuning provide any benefit? The pseudocode for ATIDS with partial tuning is shown in Fig. 7. All predictions are verified, but only some of the false predictions are used to tune the model. All other false predictions are ignored, simulating false predictions which are not caught.

In this paper, we control the percentage of false predictions used to tune the model. We performed a set of experiments, letting the portion of false predictions used to tune the model range from 10% to 100%. The results are shown in Fig. 8. The labels show the ratio of TMC, comparing ATIDS with MC-SLIPPER. The TMCs of ATIDS with 10% partial tuning drop about 30% compared to MC-SLIPPER. The detection model is tuned with feedback from the last false prediction. Whether the tuning will yield more accurate predictions, it depends on the subsequent covered data. If the model is tuned according to a false prediction that is at the beginning of a false prediction cluster (made by the original detection model), then the tuning could reduce many false predictions. But if the model is tuned on an isolated false prediction, the tuned model might make a new false prediction on the next covered data where the original model made a true prediction. The extreme examples are the four rules discussed at the end of the previous section, which have worse performance after tuning. With partial tuning, some isolated false predictions might be skipped to tune the model, which could reduce the new false predictions made by the tuned model. Therefore, a higher tuning percentage does not guarantee a lower TMC, as clearly visible in Fig. 8.

C. Delayed Tuning

In practice, the system operators will take some time to verify a false prediction, yet the positive rules that contributed to the false prediction might cover subsequent data records. Consequentially, tuning will be delayed. In a third experiment, ATIDS with delayed tuning is examined.

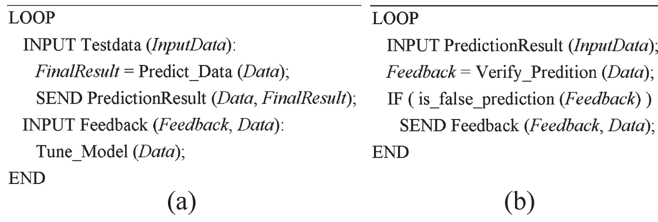

Fig. 9. Pseudocode for ATIDS with delayed tuning.

TABLE VII
PERFORMANCE OF ATIDS WITH DELAYED TUNING

System Name	Delayed Seconds	Overall Accuracy	TMC	% TMC
MC-SLIPPER		92.59%	70,177	
ATIDS	1–3	94.61%	55,982	79.77%
ATIDS	4–10	94.30%	54,841	78.15%
ATIDS	10–20	90.30%	80,186	114.26%
ATIDS	4–20	94.49%	56,087	79.97%

Fig. 9 shows the pseudocode for the ATIDS enhanced with delayed tuning. Prediction and verification are now independent of each other (as represented by the two separate threads in Fig. 9), and delay may be incurred between prediction and the consideration of the feedback from verification. In this paper, the verification thread is simulating the system operator verifying the predictions, and the delay can be controlled. For example, if the verification thread waits 1 s after every prediction, only the first 19 800 data records (about 6.37% of all records) are verified.

The results of the ATIDS with delayed tuning are shown in Table VII. The experimental results show that the performance of ATIDS with delayed tuning depends on the amount of the delay. When tuning is not delayed too long, the impact of tuning is positive, and the system gains performance improvement from tuning. In delayed tuning, the percentage of false predictions used to tune the model is low. For example, in the experiment with 4–10-s delay, only 230 false predictions are used to tune the model, which is only 1.3% of all false predictions. But when tuning is delayed too long, the overall impact of tuning is negative, as demonstrated by the experiment with 10–20-s delay. Property 3 stated that the false rate of a rule might change dramatically even in adjoining segments. In this paper, it will take 20–40 s to predict the 500 records in one segment. If tuning is delayed 20 s in a segment with high false prediction rate, after tuning is complete, the system might predict on the data record in the next segment. If the original false prediction rate in the second segment is low, the updated model could then make many false predictions. A small delay in tuning, however, has positive impact on system performance. In the experiment with 4–20-s delay, 129 false predictions are used to tune the model, and in 71 instances, feedback is delayed more than 10 s, yet ATIDS still has a lower TMC. Some negative tuning could be avoided. If it takes too much time to identify a false prediction, the tuning on this false prediction could be skipped as long as the prediction result is not fed back to the model tuner.

TABLE VIII
STATISTICS FOR SELECTIVE VERIFICATION

Condition	TVP		Percentage of False Predictions					
	#	%	normal	probe	dos	u2r	r2l	overall
PC < 0.0	37,430	12.03%	79.91%	81.50%	80.96%	92.00%	39.95%	79.45%
PC < 1.0	49,409	15.89%	86.37%	83.11%	93.11%	93.50%	58.88%	87.01%
PC < 2.0	54,486	17.52%	90.52%	85.79%	94.00%	95.50%	72.90%	90.53%
PC < 3.0	61,657	19.82%	94.06%	87.33%	94.43%	97.00%	75.93%	93.17%
PC < 4.0	68,856	22.14%	96.09%	96.85%	94.89%	98.00%	82.71%	95.58%

The experimental results show that the performance of ATIDS with delayed tuning depends on the amount of the delay. When tuning is not delayed too long, the impact of tuning is positive, and the system gains performance improvement from tuning. In delayed tuning, the percentage of false predictions used to tune the model is low. For example, in the experiment with 4–10-s delay, only 230 false predictions are used to tune the model, which is only 1.3% of all false predictions. But when tuning is delayed too long, the overall impact of tuning is negative, as demonstrated by the experiment with 10–20-s delay. Property 3 stated that the false rate of a rule might change dramatically even in adjoining segments. In this paper, it will take 20–40 s to predict the 500 records in one segment. If tuning is delayed 20 s in a segment with high false prediction rate, after tuning is complete, the system might predict on the data record in the next segment. If the original false prediction rate in the second segment is low, the updated model could then make many false predictions. A small delay in tuning, however, has a positive impact on system performance. In the experiment with 4–20-s delay, 129 false predictions are used to tune the model, and in 71 instances, feedback are delayed more than 10 s, yet ATIDS still has a lower TMC. Some negative tuning could be avoided. If it takes too much time to identify a false prediction, the tuning on this false prediction could be skipped as long as the prediction result is not fed back to the model tuner.

The 1–3-, 4–10-, 10–20-, and 4–20-s delays reported in Table VII refer to the actual time taken in our experiment. Remember that the test data were constructed from the network traffic of two weeks, which yielded around two million connection records. The 4–20-s delay in our experiments is equivalent to a 0.5–3-min delay in real time.

Although the system operator could catch all false predictions if every prediction is verified, this is inefficient because most predictions should be correct. Note that, in real life, the system operator will develop trust in the predictions of the IDS and verify only suspect predictions. Selective verification on the predictions could improve the efficiency of catching false predictions. We can trust that most false predictions have low confidence values. Thus, PC can be used to block those predictions that have high confidence values from being submitted for verification, that is, the system operator verifies only predictions with low confidence values and ignores all others.

Table VIII compares selective verification with different conditions on the results obtained from delayed tuning with a random 4–10-s delay.

In Table VIII, PC stands for prediction confidence, and TVP indicates the total number of verified predictions. Looking at the first row, the condition "PC < 0.0" indicates that only predictions with negative confidence values were verified. In this paper, 37 430 out of the 311 029 predictions (12.03%) are verified, and 79.45% (14 072 out of 17 712) of all types of false predictions were caught. The efficiency of verification has improved greatly, and 37.60% (14 072 out of 37 430) of the verified predictions are false predictions. In the experiment with

4–10-s delay, about 90.78% false predictions that were fed back to tune the model have PC values less than 4.0. In our experiment with 4–20-s delay, about 84.50% false predictions that were fed back to tune the model have PC values less than 4.0.

Note that, usually, one might want to rely on the confidence ratio introduced in Section II-B rather than the confidence values as these are scaled into the range between -1 and 1 based on the training data and thus easier comparable. However, since the data are scaled based on the training data, it could be possible that when encountering new data with extreme values, the confidence ratio may still fall outside of this range.

V. Related Work

Sabhnani and Serpen [25] built a multiclassifier system using multilayer perceptons, K-means clustering, and a Gaussian classifier after evaluating the performance of a comprehensive set of pattern recognition and machine learning algorithms on the KDDCup'99 dataset. The TMC of this multiclassifier system is 71 096, and the cost per example is 0.2285. However, the significant drawback of their system is that the multiclassifier model was built based on the performance of different subclassifiers on the test dataset. Giacinto *et al.* [28] proposed a multiclassifier system for intrusion detection based on distinct feature representations: content, intrinsic, and traffic features were used to train three different classifiers, and a decision fusion function was used to generate the final prediction. The cost per example is 0.2254. No confusion matrix of the prediction is reported in their study. Kumar [26] applied RIPPER to the KDDCup'99 dataset. RIPPER is an optimized version of incremental reduced error pruning (IREP), which is a rule-learning algorithm optimized to reduce errors on large datasets. The TMC is 73 622, and the cost per example is 0.2367. Agarwal and Joshi [22] proposed an improved two stage general-to-specific framework (PNrule) for learning a rule-based model. PNrule balances support and accuracy when inferring rules from its training dataset to overcome the problem of small disjuncts. For multiclass classification, a cost-sensitive scoring algorithm was developed to resolve conflicts between multiple classifiers using a misclassification cost matrix, and the final prediction was determined according to Bayes optimality rule. The TMC is 74 058, and the cost per example is 0.2381 when tested on KDDCup'99 dataset. Pfahringer constructed an ensemble of 50×10 C5 decision trees as a final predictor using a cost-sensitive bagged boosting algorithm [23]. The final prediction was made according to minimal conditional risk, which is a sum of error cost by class probabilities. This predictor won the KDDCup'99 contest. The TMC is 72 500, and the cost per example is 0.2331. Levin's kernel miner [24] is based on building the optimal decision forest. A global optimization criterion was used to minimize the value of the multiple estimators, including the TMCs. The tool placed second in the KDDCup'99 contest. The TMC is 73 287, and the cost per example is 0.2356.

There are two approaches in updating the detection model: Add a submodel or supersede the current model. Lee *et al.* [9] proposed a "plug-in" method as a temporary solution. When new intrusion emerges, a simple special-purpose classifier is trained to detect only the new attack. The new classifier is plugged into the existing IDS to enable detection of the new attack. The main existing detection models remain unchanged. When a better model or a single model that can detect the new intrusion as well as the old intrusions is available later, the temporal model can be replaced. This method takes advantage of the fact that training a new specific classifier is significantly faster than retraining a monolithic model from all data, and thus, it enables detection of new attacks as soon as possible. However, before the new classifier can be trained, high-quality training data should be collected. For a very new attack, it is not an easy task to collect the appropriate training data. Training then becomes the job of the system operators who usually lack the knowledge to train a model. Having the newly mined model supersede the current detection model was presented in various systems [10]–[12]. The study in [10] and [11] proposed an architecture to implement adaptive model generation. In this architecture, different detection model generation algorithms have been developed to mine the new model on real-time data. The new model can supersede the current model on-the-fly. The study in [12] deployed incremental mining to develop new models on real-time data and update the detection profile in an adaptive IDS. The profile (model) for the activity during an overlapping sliding window is incrementally mined, and the similarity between the recent and base profiles is evaluated. If the similarity stays above a threshold level, the base profile is taken to be a correct reflection of the current activities. When the similarity falls below the threshold, the rate of change is examined. If the change is abrupt, it is interpreted as an intrusion. The base profile will not be updated. Otherwise, it is treated as a normal change, and the base profile will be updated. However, this system cannot deal with situations where both intrusive and normal behavior changes occur within the sliding window. Because those models are mined on real-time data, an experienced attacker could train the model gradually to accept intrusive activity as normal.

VI. Conclusion

Because computer networks are continuously changing, it is difficult to collect high-quality training data to build intrusion detection models. In this paper, rather than focusing on building a highly effective initial detection model, we propose to improve a detection model dynamically after the model is deployed when it is exposed to new data. In our approach, the detection performance is fed back into the detection model, and the model is adaptively tuned. To simplify the tuning procedure, we represent the detection model in the form of rule sets, which are easily understood and controlled; tuning amounts to adjusting confidence values associated with each rule. This approach is simple yet effective. Our experimental results show that the TMC of ATIDS with full and instant tuning drops about 35% from the cost of the MC-SLIPPER system with a fixed detection model. If only 10% false predictions are used to tune the model, the system still achieves about 30% performance improvement. When tuning is delayed by only a short time, the system achieves 20% improvement when only 1.3% false predictions are used to tune the model. ATIDS imposes a relatively small burden on the system operator: operators need to mark the false alarms after they identify them.

These results are encouraging. We plan to extend this system by tuning each rule independently. Another direction is to adopt more flexible rule adjustments beyond the constant factors relied on in these experiments. We have further noticed that if system behavior changes drastically or if the tuning is delayed too long, the benefit of model tuning might be diminished or even negative. In the former case, new rules could be trained and added to the detection model. If it takes too much time to identify a false prediction, tuning on this particular false prediction is easily prevented as long as the prediction result is not fed back to the model tuner.

ACKNOWLEDGMENT

The authors would like to thank the reviewers for their comments, which have helped in improving the presentation of this paper.

REFERENCES

[1] D. Barbara, J. Couto, S. Jajodia, L. Popyack, and N. Wu, "ADAM: Detecting intrusions by data mining," in *Proc. IEEE Workshop Inf. Assurance and Security*, Jun. 2001, pp. 11–16.

[2] N. Ye, S. Emran, X. Li, and Q. Chen, "Statistical process control for computer intrusion detection," in *Proc. DISCEX II*, Jun. 2001, vol. 1, pp. 3–14.

[3] N. Ye, S. Vilbert, and Q. Chen, "Computer intrusion detection through EWMA for auto correlated and uncorrelated data," *IEEE Trans. Rel.*, vol. 52, no. 1, pp. 75–82, Mar. 2003.

[4] N. Ye, S. Emran, Q. Chen, and S. Vilbert, "Multivariate statistical analysis of audit trails for host-based intrusion detection," *IEEE Trans. Comput.*, vol. 51, no. 7, pp. 810–820, Jul. 2002.

[5] W. Lee and S. Stolfo, "A framework for constructing features and models for intrusion detection systems," *ACM Trans. Inf. Syst. Secur.*, vol. 3, no. 4, pp. 227–261, Nov. 2000.

[6] L. Ertoz, E. Eilertson, A. Lazarevic, P. Tan, J. Srivastava, V. Kumar, and P. Dokas, *The MINDS—Minnesota Intrusion Detection System: Next Generation Data Mining*. Cambridge, MA: MIT Press, 2004.

[7] K. Julish, "Data mining for intrusion detection: A critical review," IBM, Kluwer, Boston, MA, Res. Rep. RZ 3398, Feb. 2002. No. 93450.

[8] I. Dubrawsky and R. Saville, *SAFE: IDS Deployment, Tuning, and Logging in Depth*, CISCO SAFE White Paper. [Online]. Available: http://www.cisco.com/go/safe

[9] W. Lee, S. Stolfo, and P. Chan, "Real time data mining-based intrusion detection," in *Proc. DISCEX II*, Jun. 2001, pp. 89–100.

[10] E. Eskin, M. Miller, Z. Zhong, G. Yi, W. Lee, and S. Stolfo, "Adaptive model generation for intrusion detection systems," in *Proc. 7th ACM Conf. Comput. Security Workshop Intrusion Detection and Prevention*, Nov. 2000. [Online]. Available: http://www1.cs.columbia.edu/ids/publications/adaptive-ccsids00.pdf

[11] A. Honig, A. Howard, E. Eskin, and S. Stolfo, "Adaptive model generation: An architecture for the deployment of data mining-based intrusion detection systems," in *Data Mining for Security Applications*. Norwell, MA: Kluwer, 2002.

[12] M. Hossian and S. Bridges, "A framework for an adaptive intrusion detection system with data mining," in *Proc. 13th Annu. CITSS*, Jun. 2001. [Online]. Available: http://www.cs.msstate.edu/~bridges/papers/citss-2001.pdf

[13] X. Li and N. Ye, "Decision tree classifiers for computer intrusion detection," *J. Parallel Distrib. Comput. Prac.*, vol. 4, no. 2, pp. 179–180, 2003.

[14] J. Ryan, M. Lin, and R. Miikkulainen, "Intrusion detection with neural networks," in *Proc. Advances NIPS 10*, Denver, CO, 1997, pp. 943–949.

[15] S. Kumar and E. Spafford, "A pattern matching model for misuse intrusion detection," in *Proc. 17th Nat. Comput. Security Conf.*, 1994, pp. 11–21.

[16] Z. Yu and J. Tsai, "A multi-class SLIPPER system for intrusion detection," in *Proc. 28th IEEE Annu. Int. COMPSAC*, Sep. 2004, pp. 212–217.

[17] W. Cohen and Y. Singer, "A simple, fast, and effective rule learner," in *Proc. Annu. Conf. Amer. Assoc. Artif. Intell.*, 1999, pp. 335–342.

[18] S. Robert and S. Yoram, "Improved boosting algorithms using confidence-rated predictions," *Mach. Learn.*, vol. 37, no. 3, pp. 297–336, Dec. 1999.

[19] L. Faussett, *Fundamentals of Neural Networks: Architectures, Algorithms, and Applications*. Englewood Cliffs, NJ: Prentice-Hall, 1994.

[20] M. Sabhnani and G. Serpen, "Why machine learning algorithms fail in misuse detection on KDD intrusion detection data set," *Intell. Data Anal.*, vol. 8, no. 4, pp. 403–415, 2004.

[21] T. Kohonen, *Self-Organizing Maps*. New York: Springer Verlag, 1997.

[22] R. Agarwal and M. Joshi, "PNrule: A new framework for learning classifier models in data mining (a case-study in network intrusion detection)," in *Proc. 1st SIAM Conf. Data Mining*, Apr. 2001. [Online]. Available: http://www.siam.org/meetings/sdm01/pdf/sdm01_30.pdf

[23] B. Pfahringer, "Winning the KDD99 classification cup: Bagged boosting," *ACM SIGKDD Explor.*, vol. 1, no. 2, pp. 65–66, 1999.

[24] I. Levin, "KDD-99 classifier learning contest LLSoft's results overview," *ACM SIGKDD Explor.*, vol. 1, no. 2, pp. 67–75, 1999.

[25] M. Sabhnani and G. Serpen, "Application of machine learning algorithms to KDD intrusion detection dataset within misuse detection context," in *Proc. Int. Conf. Mach. Learn.: Models, Technol. and Appl.*, Jun. 2003, pp. 209–215.

[26] V. Kumar, "Data mining for network intrusion detection: Experience with KDDCup'99 data set," presented at the Presentation Workshop Netw. Intrusion Detection, Aberdeen, MD, Mar. 2002.

[27] C. Elkan, "Results of the KDD'99 classifier learning," *SIGKDD Explor., ACM SIGKDD*, vol. 1, no. 2, pp. 63–64, Jan. 2000.

[28] G. Giacinto, F. Roli, and L. Didaci, "A modular multiple classifier system for the detection of intrusions in computer networks," in *Proc. 4th Int. Workshop MCS*, Jun. 2003, pp. 346–355.

Zhenwei Yu received the B.S. and M.S. degrees in computer science from Northern (Beijing) Jiaotong University, Beijing, China, in 1996 and 1999, respectively. He is currently working toward the Ph.D. degree at the University of Illinois at Chicago, Chicago.

His research interests are software/network security, intrusion detection, application of data-mining technology on network security, adaptive learning algorithms, and neural network.

Jeffrey J. P. Tsai (S'83–M'85–SM'91–F'96) received the Ph.D. degree in computer science from Northwestern University, Evanston, IL.

He was a Visiting Computer Scientist with the U.S. Air Force Rome Laboratory in 1994. He is currently a Professor with the Department of Computer Science, University of Illinois at Chicago, where he is also the Director of the Distributed Real-Time Intelligent Systems Laboratory. He is an author or coauthor of ten books and over 200 publications in the areas of knowledge-based software engineering, software architecture, formal modeling and verification, distributed real-time systems, sensor networks, ubiquitous computing, trustworthy computing, intrusion detection, software reliability and adaptation, multimedia systems, and intelligent agents.

Dr. Tsai has chaired or cochaired over 20 international conferences and serves as an editor for nine international journals. He received a University Scholar Award from the University of Illinois Foundation in 1994, an IEEE Meritorious Service Award from the IEEE Computer Society in 1994, and an IEEE Technical Achievement Award from the IEEE Computer Society in 1997. He is a Fellow of the American Association for the Advancement of Science, Society for Design and Process Science, and the Institute of Innovation, Creativity, Capital (IC2) at the University of Texas at Austin.

Thomas Weigert received the M.B.A. degree from Northwestern University, Evanston, IL, and the Ph.D. degree in philosophy from the University of Illinois, Chicago.

He is the Sr. Director of Software Engineering with Motorola, Schaumburg, IL. He is the author of a textbook as well as over 40 articles on the application of artificial intelligence techniques to the development of product software, in particular, for real-time distributed systems.

Dr. Weigert serves on standard bodies for software design languages with the International Telecommunications Union and the Object Management Group.