Two State-based Approaches to Program-based anomaly detection

C. C. Michael, Anup Ghosh *
RST Research Labs
{ccmich,anup}@rstcorp.com

Abstract

This paper describes two recently developed intrusion detection algorithms, and gives experimental results on their performance. The algorithms detect anomalies in execution audit data. One is a simply constructed finite-state machine, and the other monitors statistical deviations from normal program behavior. The performance of these algorithms is evaluated as a function of the amount of available training data, and they are compared to the well-known intrusion detection technique of looking for novel n-grams in computer audit data.

1 Introduction

The goal of intrusion detection is to detect attacks against a computer system. This may be done in a number of ways, such as monitoring network activity (Cf., [13, 15]), monitoring user behavior (Cf., [11, 12]), or monitoring the system state (Cf., [8]). Recently there has also been an interest monitoring program behavior to detect intrusions [3, 5].

Each of these techniques has its own advantages and disadvantages, but the appeal of the last — program-based intrusion detection — lies in the philosophy that normal program behavior can be characterized in an unambiguous way. Unlike the behavior of a human user or the behavior of network traffic, the behavior of a program ultimately stems from a series of machine instructions whose meanings we know. The programs in question are usually system programs, so we also know that their behavior should not change without our knowledge (at least until *after* an attack has taken place and has hopefully been detected). Thus, if intrusions can be detected as deviations from normal program behavior, such an intrusion detection technique would be free from false alarms caused by changes in user behavior patterns, and free as well from missed intrusions caused by attackers that mimic benign users.

The question, however, is how we should characterize normal program behavior. Actually extracting the program's semantics from source code would be difficult — in the general case this is an undecidable problem — so program-based intrusion detection is based on the observed behavior of the program. Indeed, many operating systems provide a mechanism, known as kernel auditing, for recording the behavior of at least some programs in terms of the programs' interaction with the operating system.

In this paper, audit data is condensed into a stream of discrete events that we refer to as *audit events*, and such a stream of events characterizes each execution of each program being monitored. Specifically, our audit data comes from the Sun BSM auditing system, and the events are system calls recorded in the data. This approach is essentially the same one used in other work on program-based intrusion detection, such as [3, 5].

One intrusion detection technique, that of [3], simply characterizes program behavior in terms of audit-data n-grams that are characterized as being either normal or abnormal (depending on whether they were seen previously in training data taken from non-intrusive program executions.) But since program behavior can be complex, it seems natural to look for more expressive mechanisms, such as finite automata, to characterize the normal behavior of a program.

In this paper, we report on completely automated techniques for generating finite automata characterizing a program's normal behavior. After describing our algorithms, we present an empirical comparison to the n-gram-based technique mentioned above.

2 Background and Related Work

In the past [9], finite automata for intrusion detection have been generated with a certain amount of manual intervention. First, the audit data was preprocessed so that commonly occurring sequences of events could be combined into meta-events. Then, the meta-events were used as the alphabet of a finite automaton. The combination of events into meta-events, called *macros* in that paper, was done manually, and though the paper did not say whether the FAs were then also created by hand, it was implied that they were.

We would like to automate the process of inferring fi-

*This work was sponsored under DARPA contract DAAH01-98-C-R145

nite automata. Something along these lines is done in [2], where training data is used to learn hidden Markov models of normal program behavior. This technique proved effective at the task of intrusion detection, but training (using the Baum-Welch algorithm, see [14]) was found to be expensive. This raises the question of whether simpler algorithms might also be effective without requiring as much training.

In Section 3, we will present one algorithm for automatically constructing finite automata from training data. This algorithm builds a finite automaton describing normal program behavior, using audit traces of non-intrusive executions for training data. The finite automaton then treats each sequence of audit events as a string that is either accepted or rejected; a rejection means that the execution generating the audit events is regarded as being anomalous. It should be noted that the inference of finite automata is not intractable in this context, although it *is* intractable in a number of other settings (see [4, 7]). What makes the problem easy in the case of anomaly detection is that the requirements are simple. The finite automaton merely has to accept any training sequence that isn't abnormal. Of course, it should also reject abnormal audit-event sequences, but since there are no abnormal audit-event sequences in the training data this requirement cannot be formalized within the learning algorithm itself. Rather, we will evaluate the performance of the FAs empirically.

Our second algorithm, presented in Section 4, can also be seen as constructing a finite state machine, though the construction is simpler. For this second technique, deviations from normal behavior are measured by looking at *statistics* describing the program's behavior in each state, instead of treating each sequence of audit events as a string to be accepted or rejected by the finite automaton.

These algorithms were tested in the 1999 Lincoln Laboratories intrusion detection evaluation, and the results were briefly reported in [**?**]. This paper compares the performence of these algorithms to that of n-gram macthing on a larger corpus of training and test data, and also evaluates the three algorithms when they are trained on varying amounts of data.

3 An automatic technique for creating finite automata for BSM data.

Our goal in this section is to examine the automatic creation of finite automata for host-based intrusion detection. Since data representing intrusive behavior is not used during training, the first goal is simply to build a finite automaton that accepts all audit-event sequences in the training data, but without being so generous that it accepts *all* data, or being so rigid that it rejects every novel audit sequence after training.

By way of example, we could create an FA with a single state, where every audit event results in a transition from that state back to itself. We could also create an FA that has no cycles and accepts exactly the audit-event sequences occurring in the training data.

The first approach is too weak because it tends to accept *any* sequence of audit events, and thus fails to notice abnormal audit-event sequences. The second approach is probably too strong, because it rejects any sequence as being abnormal unless exactly the same sequence was seen during training. Our goal is to create a reasonably expressive FA, but one that can still generalize. Of course, this is a qualitative requirement.

The first question is how to define the states of the automaton. The technique reported in this section associates each state with one or more n-grams of audit events, where n is a parameter of the learning algorithm. For example, the FA might have a state corresponding to the event sequence `lstat, open, ioctl`, and enter that state whenever the sequence `lstat, open, ioctl` is seen. The idea, however, is to be parsimonious in the creation of new states, and not simply have one state in the FA for every n-gram of audit events. Instead, we will have more that one n-gram assigned to most of the states.

During training, separate automata are created for the different programs whose audit data are available for training. As with the intrusion detection systems of [3], the training algorithm is presented with a series of n-grams taken from non-intrusive BSM data for a given program.

During training, the audit data is split into sub-sequences of size $n + \ell$ by a sliding window. For example, with $n = 2$ and $\ell = 1$, the first two windows created from the sequence `a, b, c, d, e` would be `a, b, c` and `b, c, d`. The first n elements of the window are used to define a state, and the last ℓ elements are used to label a transition coming out of that state. The construction of the automaton proceeds by deciding where the outgoing transitions will lead.

This decision is made by referring to the first n audit events in the *next* window, which define the next state that the automaton should enter for this particular training sequence. If we define the *current state* to be the one defined by the first n elements of the current window, then there are three possibilities:

1. The current state already has an outgoing edge that corresponds to the last ℓ events in the window, and that edge leads to the correct state (i.e., the state defined by the first n elements of the next window). In this case, no modifications are made to the FA.

2. The current state has outgoing edges that correspond to the last ℓ events in the window, but none of these edges lead to the right state. In this case, the FA may contain the correct state (but no edge from the current

state to the desired state), or else the FA may not even have a state corresponding the next n-gram.

We simply create a state for the new n-gram if one doesn't already exist. In either case, we create a transition from the current state to the new state, and label that transition with the last ℓ events of the current window.

3. The current state has no outgoing edges that correspond to the last ℓ events of the window. If there is already an state assigned to the next n-gram, then we simply create a transition to that state, and label it with the ℓ events as in the previous case.

   However, if the next n-gram doesn't have any state assigned to it, we can assign any one of the already existing states, or create a new state, without introducing any prediction errors. Currently, the algorithm just creates a transition back to the current state, and assigns the new n-gram to the current state (where it joins whatever n-grams were assigned to that state previously).

In all three cases, the FA transitions to the state assigned to the new n-gram, and this becomes the current state when we examine the next window of events from the audit data.

Note that any given state may correspond to more than one n-gram, due to the way merging is done in the third case.

3.1 Some examples of automatically generated finite automata

The finite automata constructed by the preceding algorithm depend on the amount of training data, as well as on the values of n and ℓ. No n-gram has more than one state assigned to it, so there can be no more than k^n states for a program that produces k unique audit events. (In practice, the number of states is usually much smaller: many states are assigned to more than one n-gram, and far fewer than k^n n-grams actually appear in the training data.) Each state has at most k^ℓ outgoing transitions, so the total number of transitions is bounded by $k^{n+\ell}$. Once again, the actual number of edges tends to be much smaller due to the limited number of unique ℓ-grams that actually appear in the training data.

Figure 1 shows the automata created for the `lpr` program after training on data collected at MIT's Licoln Laboratories for an intrusion detection evaluation. The first automaton was created using one week's worth of data, with $n = 7, \ell = 4$. The second was created from the same week of data with $n = 2, \ell = 1$, and the third was trained using eight weeks of data with $n = 2, \ell = 1$.

(To save space, these diagrams don't have the edges labeled.)

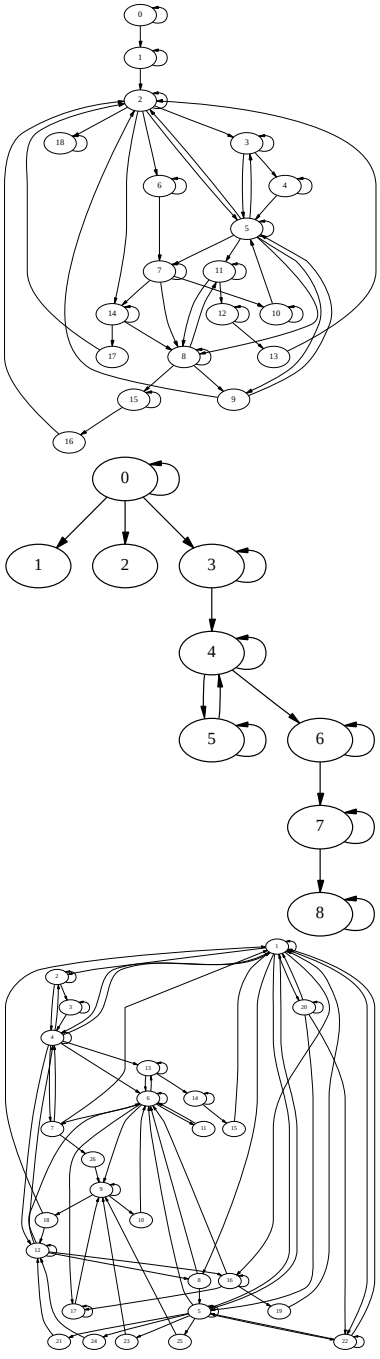

Figure 1. Finite automata constructed for `lpr` with $n = 2, \ell = 1$ on Week 2 of the Lincoln Labs data, constructed with $n = 7, \ell = 4$ on the same data, and constructed with $n = 2, \ell = 1$ for all seven weeks of data.

4 String Transducer

The state-based anomaly detector just described detects things that have never happened before. In this sense, it is like the system of [3], which raises an alarm when it sees n-grams of audit events that were not seen in the training data. The difference between the two systems lies the model of normal behavior they construct from the training data.

However, neither of these systems notices *statistical* deviations from normal behavior, short of patterns that have never been seen before at all. The second detection algorithm we discuss in this paper makes an attempt to detect subtler statistical deviations from normalcy.

A string transducer is an algorithm that associates an sequence of input symbols with a series of output symbols. String transducers are most often used in computational biology and computational linguistics, where they are usually implemented using finite automata whose transitions or states are associated with output symbols. In the current context we use automata as well, but the input sequence is a sequence of audit events, and the output sequence is a series of numbers describing how that sequence deviates from normal behavior

Our use of string transducers as intrusion detectors is based on an examination of the *probabilities* of the output symbols at each state. During training, we estimate the probability distribution of the symbols at each state, and during testing, deviations from this probability distribution are taken as evidence of anomalous behavior. (The details of this are given in Section 4.1).

Our implementation of this idea is relatively simple. We use a finite automaton whose states correspond to n-grams in the BSM data. For each state, we also record information about the *successor ℓ-grams* that are observed in the training data when the system is in that state. (These are the same ℓ-grams that would have been used to label the transitions in the algorithm of Section 3.)

During training, our goal is to gather statistics about these successor ℓ-grams; we estimate the probability of each ℓ-gram by counting. (Note that the construction of this FA is even simpler than that of the one described earlier, due to the $1-1$ correspondence between states and audit-data n-grams.)

During actual intrusion detection, the deviation of the successor ℓ-grams from their expected values are, in some sense, used as anomaly scores. Of course, the anomaly scores are usually non-zero, but if the program is behaving normally these deviations should average out over time.

In the ideal case, it can be shown that the anomaly scores are uncorrelated if the probability distributions have, in fact, been correctly estimated (this is due to the fact that the deviations are then an innovations process; see [1]). That means that if we subtract the mean anomaly score for each state from the actual anomaly scores generated there, the result is zero-mean noise.

If these values are integrated over a sufficiently long period, the result should be close to zero if the program is behaving normally. However, if abnormal program behavior results in a significant deviation of the successor ℓ-grams from their expected values, then the resulting scores will not integrate to zero, and this fact can be used to detect anomalous behavior.

In practice, there are obviously a number of factors preventing the realization of this ideal case.

1. If the probabilities of the successor ℓ-grams have not been accurately estimated (perhaps due to insufficient data), then the deviations may not be uncorrelated. Note that accurate prediction is impossible if the distribution of the ℓ-grams depends on an unpredictable aspect of user behavior.

2. During detection, n-grams may be encountered that do not correspond to any known state because they were not seen during training. This makes it impossible to generate an anomaly score for the successor ℓ-gram.

3. An intrusion may not result in a systematic deviation from the expected ℓ-gram values; in other words, the intrusion may look normal. Although this seems unlikely, we cannot prove that all intrusions really cause the necessary deviations.

4. The window of integration needed to get sufficiently low anomaly scores during normal behavior may be large. This delays the detection of anomalies (though if it prevented them from being detected we would arguably be in case 3).

The fourth is an intrinsic problem of change detection [10]; there is an inevitable tradeoff between the time to detection and the susceptibility to false positives. The third problem is also, in some sense, unavoidable; it seems unlikely that we could guarantee the detection of all intrusions without assuming something about the nature of those intrusions, but we cannot validate such assumptions without knowing the future behavior of attackers. (We may, of course, be able to make guarantees for certain classes of intrusions).

The second problem cited above is more directly related to our our specific application. It results from having too little training data to characterize all states. It dictates that states should not be too highly specialized, since such specialization makes it less likely for all states to be seen during training.

The first problem dictates a wise choice of states. For example, it has been observed that programs go through different phases of behavior [2], so the probability of a given ℓ-gram may depend on how far along the program is in its

execution. Thus, states should reflect the state of the program itself. Even if the distribution of ℓ-grams varies over time, the distribution *from a given state* should be constant. Unfortunately, this condition can best be achieved by using highly specialized states, to avoid having two or more states of the underlying program represented by a single state of the automaton. Thus, the solutions to the first and second problems are in some sense at odds. This tradeoff between expressiveness and ease of training is also well-known in machine learning [16].

4.1 Implementation details

As we have said, the probability densities of the successor ℓ-grams in a given state are estimated by counting (that is, we simply count the number of occurrences of each ℓ-gram in the training data). This is a feasible approach with BSM data because such data tends to be fairly regular; the number of BSM ℓ-grams is much smaller than, say, the number of possible BSM events raised to the ℓth power.

We measure deviations from expected behavior by treating the estimated probability distribution as a vector, which we first normalize with respect to the L_k metric

$$\left(\sum_i |x_i^k|\right)^{1/k}$$

for some k. When a given ℓ-gram occurs during detection, we treat it as a vector with a 1 in the position corresponding to the actual ℓ-gram that was seen, and a 0 in the other positions. The deviation is the L_k distance between this vector and the normalized density vector. In other words, if $\hat{p}_i$ is the estimated probability of the ith ℓ-gram, according to some arbitrary ordering, then the elements of the normalized probability vector are given by

$$h_i = \frac{\hat{p}_i}{\left(\sum_j |\hat{p}_j^k|\right)^{1/k}},$$

and the deviation d_i, reported when the ith ℓ-gram is seen during detection, is given by

$$d_i = 2^{-1/k}\left(\sum_j c_{i,j}^k\right)^{1/k}$$

where

$$c_{i,j} = \begin{cases} 1 - h_j, & \text{if } i = j; \\ h_j, & \text{otherwise.} \end{cases}$$

We treat these as summations over all possible ℓ-grams, though the actual implementation only has to sum over those that were seen during training since p_j is zero for the others. But if a novel ℓ-gram is seen during training, this convention assures that d_i is still defined, and, in fact, its value is just 1.

If we encounter an n-gram that does not correspond to any state, we ignore it. We could alternatively flag it as an anomaly, like n-gram-based detection does, but that would mix the results of two fundamentally different intrusion detection schemes into a single anomaly score. This creates problems, because we found that the anomaly scores generated by the missing n-grams washed out the scores created by measuring the deviations. This happens as follows: first, the missing n-grams generate high anomaly scores for benign executions. As is common practice, we have a threshold that determines how large an anomaly score must be before we signal that an intrusion is taking place. When missing n-grams generate anomaly scores, this threshold has to be raised in order to avoid false positives, but that means that only the larger anomaly scores created by missing n-grams raise an alarm for intrusive sessions. Thus, the missing n-grams dominate the performance of the intrusion detector, and the whole method more or less reduces to n-gram matching as implemented by [3]. Our decision to ignore missing n-grams in the string transducer ensures that we were not simply reimplementing n-gram comparison in a different guise.

5 Experimental results

We performed experiments to compare the intrusion detection systems of Sections 3 and 4 to one another and to the n-gram-based technique of [3]. The latter technique is quite simple: one simply records all n-grams of audit events that occur in the training data, and during the detection phase, the appearance of a novel n-gram is taken as evidence of an intrusion. This technique is generally combined with postprocessing that averages the last k anomaly scores (for some k), leading to a filtered anomaly score that takes on a value between 0 and 1, depending on how many alarms were raised by the last k audit events. This allows us to set an *alarm threshold*; we signal a possible intrusion only when the filtered anomaly score exceeds this threshold. In our experiments, we applied the same type of postprocessing to the string transducer and the state-based intrusion detector.

5.1 The training data

Our training data came from several sources, the most important of which was a series simulations of a computer network conducted by Lincoln Laboratories in 1998 and 1999. We have twelve weeks worth of this data, but at the time these experiments were conducted we had not yet separated intrusions from benign behavior in one week of

the data. Some further data was supplied by Johns Hopkins University. Finally we discovered that we needed additional data describing the normal behavior of three programs, `eject`, `fdformat`, and `xterm`, so we collected data for these programs on our own system.

The data contains evidence of several kinds of attacks, which can be divided into two broad classes: (1) probes and denial-of-service attacks, and (2) unauthorized accesses and unauthorized privilege elevations. Our systems are meant to detect the second class of attacks, so we did not evaluate them on attacks in the first class.

Furthermore, some attacks do not leave identifiable traces in the audit data. For example, some attacks consist of moving files to a location where they should not be. Identifying such attacks means knowing what locations are disallowed; in other words, the intrusion detector must know the details of the system's local security policy. Our prototypes currently have no knowledge of local security policies, so they do not detect such attacks. Another attack involves setting up a malicious http client to transfer information off of the system using cookies. This attack does not involve misuse of any existing programs, so (strictly speaking) our systems cannot detect such an attack either. In reality, such attacks are sometimes detected because of statistical variations between the behavior of benign users and malicious users, but we do not know how easily an intruder could avoid this sort of detection. Moreover, these systems are meant to detect program misbehavior, not profile computer users. Therefore we decided that these particular attacks are out of the intended scope of the system. As a result, just under 94% of the access and elevation-of-privilege attacks are in the scope of our system.

There are 183 types of system calls recorded in the audit data, and these events form the inputs to our intrusion detection system. The events are collated into sequences of audit events generated by individual programs, and a different intrusion detector is trained for each program. The anomaly score for a session is the maximum of the anomaly scores for the programs in that session

In many papers that describe n-gram matching (such as [9]), the input to the intrusion detection system consists of higher-level features extracted from the raw stream of audit data. However, the features seem to have been specified by hand, and since our intrusion detection systems must learn normal behavior for many programs, that approach seemed infeasible. Therefore, the algorithms in these experiments are trained on features read directly from the raw sequence of audit data.

The experimental data is divided into user sessions, some of which contain intrusions. Our intrusion detectors examine all program executions that occur in a session, and the highest anomaly score for any of these programs is used as the anomaly score for the session.

5.2 Training the Intrusion Detectors on All Data

To quantify the performance of a given system on a given set of data, we measure how many benign sessions are falsely marked as being intrusive, and measure how many intrusive sessions are overlooked. All three systems output a number between 0 and 1 describing how anomalous a given session is, so the performance of any given system depends on the threshold at which we raise the alarm. By varying the threshold and plotting the percentage of false alarms against the percentage of missed intrusions, we obtain a plot similar to a receiver operating curve, which is a convenient way to visualize the performance of the intrusion detectors.

We tested our systems using seven-fold cross validation. In a series of seven experiments, the system was trained using all of the data except for one of first the seven weeks of data we had from Lincoln Laboratories. The system was then tested using the remaining week of data, in order to test immunity to false alarms. None of the audit data reflecting intrusive activity was used during training, so the systems were tested on all of the intrusive data during each of the seven cross-validation phases.

We tested each system with a number of different choices of parameters. For example, the n-gram matcher was tested with different choices of n, and the string transducer and state tester were also tested with n-grams of different sizes. This gives us less variability for the n-gram matcher than for the string transducer or the state tester, since both of these use two sets of n-grams whose sizes can be varied more or less independently. However, for the state tester, we found that the size of the n-gram being predicted made very little difference in performance. For the string transducer, the best performance for a given conditioning n-gram was always obtained when the n-gram being predicted was of length 1. Thus, we effectively only varied one parameter for each program when searching for the intrusion detector with the best performance. Although our system builds an intrusion detector for each of many programs, the parameters for each detector are the same on a given run. The results we report below were obtained using the choice of parameters that yielded the best results for each of the state tester, the n-gram matcher, and the string transducer. For the n-gram matcher, the best performance was obtained with $n = 4$. For the state tester, the best performance was seen with $n = 8$, and we used $\ell = 1$. For the string transducer, the best performance was obtained with $n = 10$ ($\ell = 1$).

Figure 2 plots the false positives generated by each system against its detection rate. Each point in the plot represents an alarm threshold. The horizontal coordinate is the percentage of benign sessions that were incorrectly classified as intrusive, while the vertical coordinate represents the percentage of intrusive sessions that were detected. Note

that no technique detects more than 94% of the intrusions; this is because some intrusions leave no evidence in the BSM data, as mentioned above.

The performance of the n-gram matcher is slightly better than that of the other two detectors. That is, we can choose an alarm threshold that makes the n-gram matcher give a better tradeoff between detection and false positives than either of the other two techniques.

5.3 Intrusion detector performance as a function of the quanitity of training data

Though the n-gram matcher performed better than the other two intrusion detectors, we found that the amount of training data had a significant affect on the relative performance of the three techniques. To evaluate it, we performed a series of tests where only subsets of the available data were used during training.

In this setup, we specify what percentage of the training data is to be used. For the sake of concreteness, assume we want to use five percent of the data. Recall that all of our intrusion detection schemes build a different classifier for each program; for example, BSM data generated by the program `ps` will be handled by an anomaly detector specifically trained to identify abnormal `ps` traces. Therefore, using five percent of the data means using five percent of the program traces available for each classifier (recall that each trace represents one complete execution of the program in question; we always use complete traces for training). The BSM audit sequences generated by `ps` are collected, and five percent of them are selected at random for use during training (the actual number of audit-event sequences we use is actually rounded up from five percent, which means that there is always at least one sequence available for training).

This procedure was repeated (for all programs) using the same cross-validation scheme as above. In each run, "five percent of the training data" means five percent of the data that would normally be available during that phase of cross-validation. We repeated the procedure using ten percent of the training data, then twenty percent of the training data, and so on, for a total of twenty cross-validation runs having seven train-and-test phases apiece.

Since training data is selected at random, we repeated this entire procedure ten times with different random number seeds. Each time, the data for each of the three classifiers was selected with the same seed, so that, for example, all three classifiers see the same five percent of the training data during each cross-validation phase. In addition, the random number seed is the same regardless of what *percentage* of training data is used during a given run. That means that when we select five percent of the training data for a given seed, and later select ten percent of the training data from the using the same seed, the second set of data is a superset of the first. (We shuffle the training sequences using the random number seed, and then select the first n sequences for training, where n depends on the requested percentage of data.) Thus, each of the ten experiments simulates the situation that we would see if the detectors were trained incrementally from a growing corpus of data.

In Figure 3, we plot our results with the percentage data used for training along the horizontal axis. The vertical axis shows the percentage of benign sessions that were classified as intrusive, with the threshold set for "maximum" detection (that is, detection of all the attacks that leave evidence in the BSM traces).

Figure 3 shows the pointwise average, over ten runs, obtained for the string transducer, the state-based technique, and the n-gram matcher. We see that the results in Figure 2 were somewhat deceptive, because the n-gram matcher just barely catches up to the string transducer when it avails itself of all our training data. For smaller amounts of training data, the string transducer considerably outperforms n-gram matching; after training on five percent of the data, it raises a false alarms for less than three percent of the benign sessions, while the n-gram matcher does so for over thirty percent.

The state machine, like the string transducer, has an initially low false-alarm rate, though it raises more false alarms than the string transducer at all stages of training.

An interesting feature of learning curve for the n-gram matcher is that it has bumps. There is a visible bump just below the point where sixty percent of the training data is used. There is also a bump close to the left edge of the plot which is more visible in the ten individual plots, its position varies somewhat so it is less visible when all the results are averaged.

For a given alarm threshold, the false positive rate cannot go up as more training examples are added (this is because alarms are triggered by features that do not appear in the database of previously observed n-grams, and that database never shrinks). Therefore, these bumps must result from a decrease in the alarm threshold, needed to obtain detection of all the intrusive sessions. In other words, certain n-grams that are useful for detecting intrusions eventually show up in the normal data, and necessitate a decrease of the alarm threshold in order to maintain perfect detection.

Individual runs for the state tester show even greater irregularity and variability. This is largely due to the fact that the training algorithm for the state tester depends on the *order* in which the audit-event sequences are presented, and the sequences were shuffled in our experiments. This reordering had a considerable impact on performance at all stages of training, and in this regard it might have been preferable to use slower but more deterministic algorithms such as Baum-Welch (as in [2]) or state-merging [6]. We are currently investigating the second option.

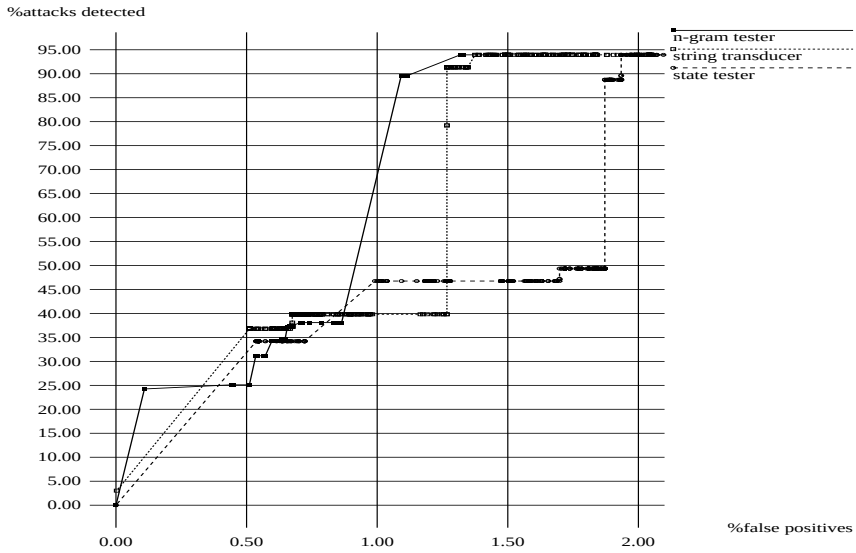

Figure 2. The performance of the n-gram detector, the string transducer, and the state tester, presented as a ROC-style curve. Various alarm thresholds are plotted with the horizontal axis giving the percentage of benign sessions that created false alarms, and the vertical axis representing percentage of intrusive sessions detected. The lines connecting the points are only for visibility.

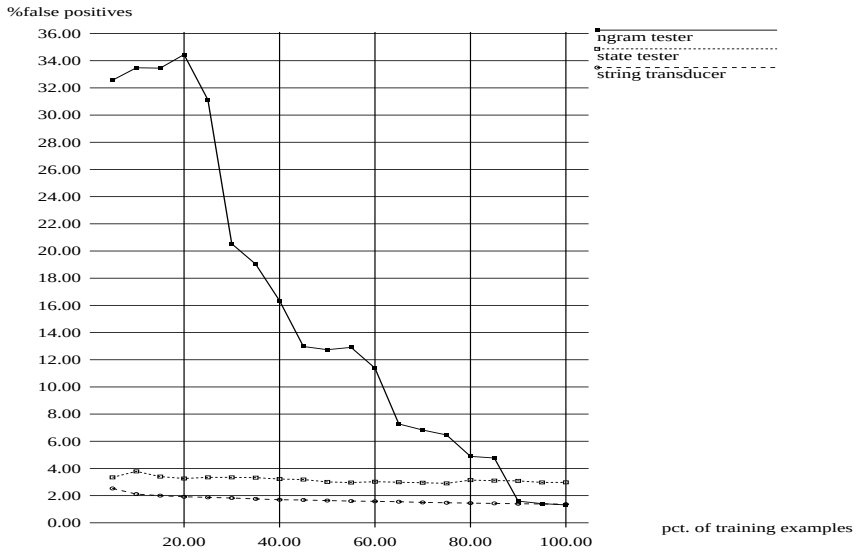

Figure 3. The performance of the n-gram detector, the string transducer, and the state tester, with varying amounts of training data. The horizontal axis represents the percentage of available data actually used during training, and the vertical axis represents the false-alarm rate for the lowest alarm threshold that allowed detection of all intrusions that left evidence in the audit data.

The performance of the state transducer was not only the best of the three, but it was also much more consistent between runs. The program behavior statistics that determine the performance of the string transducer vary only gradually as new data is added. This makes it less susceptible to sudden changes in performance. In fact, all experimental runs of the string transducer revealed monotonic improvement as training data was added.

One final issue that should be mentioned is that, in these experiments, we used the same parameter settings for the learning algorithms, regardless of how much data was available (they were still the settings that led to the best performance in the previous experiment, where all the training data was used). Often, however, it is useful to adjust the parameters of a learning algorithm based on the richness of the available data. For example, maybe it is always possible to find some value of n for which n-gram matching performs better than the other techniques, and it just happens that this optimal value is not 4 unless all of the training data is used.

To briefly investigate this possibility, we performed two more sets of experiments. First, we omitted all but the first seven weeks of Lincoln Labs data during training. In this experiment, the n-gram matcher did, indeed, have better performance with $n = 3$, and it continued to outperform the string transducer even with the smaller amount of data. However, by keeping only seven weeks of the Lincoln data, we also eliminated much of the diversity from the original training set. (That is, the reduced data all comes from the environment of the Lincoln Labs simulation, and none of it comes from our own environment or that of Johns Hopkins.) It may be that this simplification of the environment aided the n-gram matcher more than the string transducer.

We also tried both intrusion detection algorithms with various parameter choices using 50% of the entire training corpus, with the random selection of training data accomplished in the same way as above. In this experiment, we found that changing the parameters of the n-gram matcher had very little effect on its performance (at least for $3 \leq n \leq 17$), which suggests that this algorithm cannot, in fact, be easily fine-tuned to adjust for variations in the amount of training data. For larger values of n, adjusting the parameter of the string transducer also made little difference. The string transducer performed considerably better than the n-gram matcher, achieving perfect detection with just over 1.5% of the benign sessions being labelled as intrusive, as opposed to roughly 9% for the n-gram matcher.

6 Conclusion

In this paper we described two recently-developed algorithms for intrusion detection using program behavior traces. We empirically compared the performance of these algorithms to another well-known intrusion detection method that uses behavior traces as well. Although the latter method, n-gram matching, ultimately achieved slightly better performance that our two techniques, it was much slower to learn, which is to say that it required a great deal more training data to achieve false positive rates comparable to those of our algorithms.

One open question raised by these experiments stems from the somewhat ad-hoc nature of our first algorithm for synthesizing finite-state machines. Could better results be achieved with a better learning algorithm? This question is significant, because we would like to know whether it is always useful to model long-term dependencies in program behavior data. A finite-state machine has the capability of modeling such long-term dependencies, while n-gram matching technique does not. Experimental results that were not yet available when this paper was being prepared indicate that there is, in fact, a benefit to be obtained by using more refined state-machine inference algorithms; these results will be the subject of a future publication.

References

[1] M. Basseville and I. V. Nikiforov. *Detection of Abrupt Changes - Theory and Application*. Prentice-Hall, Inc., Englewood Cliffs, NJ, 1993.

[2] B. P. C. Warrender, S. Forrest. Detecting intrusions using system calls: Alternative data models. In *1999 IEEE Symposium on Security and Privacy*, pages 133–145, 1999.

[3] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128. IEEE Computer Society, IEEE Computer Society Press, May 1996.

[4] Y. Freund, M. Kearns, D. Ron, R. Rubinfeld, R. E. Schapire, and L. Sellie. Efficient learning of typical finite automata from random walks. *Information and Computation*, 138(1):23–48, 10 Oct. 1997.

[5] A. Ghosh, A. Schwartzbard, and M. Schatz. Using program behavior profiles for intrusion detection. In *Proceedings of the SANS Intrusion Detection Workshop*, February 1999.

[6] R. K. Lang, B. Pearlmutter. Results of the abbadingo one dfa learning competition and a new evidence driven state merging algorithm. In *Proceedings of the International Colloquium on Grammatical Inference (ICGA-98)*, volume 1433 of *Lecture Notes in Artificial Intelligence*, pages 1–12. Springer-Verlag, 1998.

[7] M. Kearns and L. Valiant. Cryptographic limitations on learning boolean formulae and finite automata. In *Proceedings of the Twenty First Annual ACM Symposium on Theory of Computing*, pages 433–444, New York, NY, 1989. ACM.

[8] G. H. Kim and E. H. Spafford. The design and implementation of tripwire: A file system integrity checker. In J. Stern, editor, *2nd ACM Conference on Computer and Communications Security*, pages 18–29, Fairfax, Virginia, Nov. 1994. ACM Press.

[9] A. P. Kosoresow and S. A. Hofmeyr. Intrusion detection via system call traces. *IEEE Software*, 14(5):24–42, Sept./Oct. 1997.

[10] T. L. Lai. Information bounds and quick detection of parameter changes in stochastic systems. *IEEE Transactions on Information Theory*, 44(7):2917–2929, 1998.

[11] T. Lane and C. Brodley. An application of machine learning to anomaly detection. In *Proceedings of the 20th National Information Systems Security Conf erence*, pages 366–377, October 1997.

[12] T. Lunt, A. Tamaru, F. Gilham, R. Jagannthan, C. Jalali, H. Javitz, A. Valdos, P. Neumann, and T. Garvey. A real-time intrusion-detection expert system (ides). Technical Report, Computer Science Laboratory, SRI Internationnal, February 1992.

[13] P. Porras and P. Neumann. Emerald: Event monitoring enabling responses to anomalous live disturbances. In *Proceedings of the 20th National Information Systems Security Conference*, pages 353–365, October 1997.

[14] L. Rabiner and B.-H. Juang. *Fundamentals of Speech Recognition*. Prentice Hall (Signal Processing Series), Englewood Cliffs, NJ, 1993.

[15] S. Staniford-Chen, S. Cheung, R. Crawford, M. Dilger, J. Frank, J. Hoagland, K. Levitt, C. Wee, R. Yip, and D. Zerkle. GrIDS – A Graph Based Intrusion Detection System for Large Networks. In *Proceedings of the 19th National Information Systems Security Conference*, 1996.

[16] V. N. Vapnik. *The Nature of Statistical Learning Theory*. Springer, New York, 1995.