Detecting Intrusions Using System Calls: Alternative Data Models†

Christina Warrender
Stephanie Forrest
Barak Pearlmutter

Dept. of Computer Science
University of New Mexico
Albuquerque, NM 87131-1386
{christy,forrest,bap}@cs.unm.edu

Abstract

Intrusion detection systems rely on a wide variety of observable data to distinguish between legitimate and illegitimate activities. In this paper we study one such observable—sequences of system calls into the kernel of an operating system. Using system-call data sets generated by several different programs, we compare the ability of different data modeling methods to represent normal behavior accurately and to recognize intrusions. We compare the following methods: Simple enumeration of observed sequences, comparison of relative frequencies of different sequences, a rule induction technique, and Hidden Markov Models (HMMs). We discuss the factors affecting the performance of each method, and conclude that for this particular problem, weaker methods than HMMs are likely sufficient.

1. Introduction

In 1996, Forrest and others introduced a simple intrusion detection method based on monitoring the system calls used by active, privileged processes [4]. Each process is represented by its *trace*—the ordered list of system calls used by that process from the beginning of its execution to the end. This work showed that a program's normal behavior could be characterized by local patterns in its traces, and deviations from these patterns could be used to identify security violations of an executing process.

There are two important characteristics of the approach introduced in [4]. First, it identifies a simple observable (short sequences of system calls) that distinguishes between normal and intrusive behavior. This observable is much simpler than earlier proposals, especially those based on standard audit packages, such as SunOS's BSM. Second, the method used to analyze, or model, the sequences is also much simpler than other proposals. It records only the presence or absence of sequences; it does not compute frequencies or distributions, or identify which sequences are most important. The advantage of such a simple approach is computational efficiency, but the question naturally arises of whether more accurate models of the data might be possible.

Over the past several years, many statistically-based learning techniques have been developed. Several such methods have the potential for generating more accurate and/or more compact models of the system-call data, and at least two groups have published results of their own experiments on alternative models applied to system calls [13, 6]. Most of the available methods, however, were designed for specific applications, and each has its own idiosyncrasies. The goal of our paper is to compare these various methods as systematically as possible across a larger and more realistic suite of data sets than has been used in the past.

2. Choosing Applicable Methods

There are many ways in which system call data could be used to characterize normal behavior of programs, each of which involves building or training a model using traces of normal processes.[1] In this section, we discuss several alternative approaches to this task, and select four for more careful investigation. The list of methods discussed here is

† In 1999 IEEE Symposium on Security and Privacy, IEEE Computer Society pp. 133-145 (1999).

[1] The empirical approach taken here ignores the family of methods based on formal specification of a program's legal activities, such as [9].

by no means exhaustive, but it does cover those we believe to be most appropriate for our problem.

2.1. Enumerating Sequences

The methods described in [4, 7] depend only on enumerating sequences that occur empirically in traces of normal behavior and subsequently monitoring for unknown patterns. Two different methods of enumeration were tried, each of which defines a different model, or generalization, of the data. There was no statistical analysis of these patterns in the earlier work.

The original paper used *lookahead pairs* [4]. The database of normal patterns consisted of a list for each system call of the system calls that follow it at a separation of 0, 1, 2, up to k system calls. This method can be implemented efficiently, and it gave good results on the original (synthetic) data sets.

The later paper reported that contiguous sequences of some fixed length gave better discrimination than lookahead pairs [7]. The database of normal behavior remained compact, and computational efficiency was still reasonable. As the earlier method was known as *time-delay embedding* (tide), this method was called *sequence time-delay embedding* (stide). In the comparisons reported below, we use contiguous sequences.

2.2. Frequency-based methods

Frequency-based methods model the frequency distributions of various events. For the system-call application, the events are occurrences of each pattern of system calls in a sequence.

One example of a frequency-based method is the *n-gram vector* used to classify text documents [3]. Each document is represented by a vector that is a histogram of sequence frequencies. Each element corresponds to one sequence of length n (called an *n-gram*), and the value of the element is the normalized frequency with which the n-gram occurs in the document. Each histogram vector then identifies a point in a multidimensional space, and similar documents are expected to have points close to each other. In [3], Damashek used the dot product between two histogram vectors as a measure of their similarity, but he pointed out that other measures are possible. A set of documents can be represented by one or more centroids of the set's individual histograms, and dot products can be taken with the resulting centroid rather than an individual histogram vector to test for membership in the set.

Adapting this method to traces of the system calls used by computer programs is straightforward. One or more centroid vectors could be used as the model for normal, and individual traces whose vectors were too distant from this centroid would be considered anomalous. However, this approach is not suitable for on-line testing because trace vectors cannot be evaluated until the program has terminated. It is also difficult to determine what size vector to use; the space of all possible sequences is much too large, and we cannot guarantee that the subset of sequences observed in traces of normal behavior is complete. Finally, the coarse clustering of documents in [3] does not suggest sufficient precision to discriminate between normal and intrusive traces of the same program.

Other frequency-based methods examine sequences individually, making them suitable for on-line use. Determination of whether a sequence is likely to be anomalous is based on empirically determined frequencies for that sequence, but the approaches taken can be quite different, as the next two examples illustrate.

Helman and Bhangoo propose ranking each sequence by comparing how often the sequence is known to occur in normal traces with how often it is expected to occur in intrusions [5]. Sequences occurring frequently in intrusions and/or infrequently in normal traces are considered to be more suspicious. Unfortunately, frequencies of each sequence in all possible intrusions are not known *a priori*. We must, therefore, choose a frequency distribution for abnormal sequences by assumption. Several possibilities for choosing this distribution are mentioned in [5], the simplest of which is to assume that the abnormal distribution is uniform.

The Helman and Bhangoo method makes several assumptions that are problematic for the system-call application. First, it assumes that the data are independent and stationary. Although a series of complete program traces might well be stationary (no ordered correlations among separate traces) [7], the sequences within the trace are not. Programs often have different distributions of sequences at the beginning of their execution than they do at the end, and there might be many such distinct regions within the trace [10]. Also, sequences of system calls are clearly not independent, especially when the sequences overlap as ours do. A second problem is that of characterizing the frequencies of abnormal sequences accurately.

SRI takes a different approach in its Emerald system [8]. Rather than using static distributions to define normal and abnormal behavior, Emerald compares short-term frequency distributions from new, unknown traces with the longer-term historical distribution. Prior knowledge (or estimation) of the abnormal frequencies is not required. The long-term distribution can be continually updated, with more weight being given to recent data, so that stationarity is not required. This does, however, allow the possibility of an intruder maliciously training the system to shift its definition of normal closer to the pattern produced by intrusive behavior.

Central to both methods is the idea that rare sequences are suspicious. We chose to implement a minimal version of a frequency-based method that would allow us to evaluate this central idea.

2.3. Data mining approaches

Data mining approaches are designed to determine what features are most important out of a large collection of data. In the current problem, the idea is to discover a more compact definition of normal than that obtained by simply listing all patterns occurring in normal. Also, by identifying just the main features of such patterns, the method should be able to generalize to include normal patterns that were missed in the training data.

Lee and others used this approach to study a sample of system call data [13, 12]. They used a program called "RIPPER" to characterize sequences occurring in normal data by a smaller set of rules that capture the common elements in those sequences. During monitoring, sequences violating those rules are treated as anomalies. Because the results published in [13] on synthetic data were promising, we chose this method for further testing.

2.4. Finite State Machines

A machine learning approach to this problem would construct a finite state machine to recognize the "language" of the program traces. There are many techniques for building either deterministic or probabilistic automata for this sort of task, for example, [1, 16, 10]. These methods generally determine the frequencies with which individual symbols (system calls in our case) occur, conditioned on some number of previous symbols. Individual states in the automaton represent the recent history of observed symbols, while transitions out of the states indicate both which symbols are likely to be produced next and what the resulting state of the automaton will be. Many, but not all, of the algorithms for building these automata are based on the assumption that the data are stationary.

A particularly powerful finite state machine is the hidden Markov model, used widely in speech recognition and also in DNA sequence modeling [15, 14]. A hidden Markov model (HMM) describes a doubly stochastic process. An HMM's states represent some unobservable condition of the system being modeled. In each state, there is a certain probability of producing any of the observable system outputs and a separate probability indicating the likely next states. By having different output probability distributions in each of the states, and allowing the system to change states over time, the model is capable of representing nonstationary sequences.

HMMs are computationally expensive, but very powerful. There is a great deal of information available on them, and their usefulness has been demonstrated in many areas. For these reasons, we decided to use HMMs as the finite state machine representative for our experiments.

3. Data Sets

The original studies of the system-call approach were conducted primarily on synthetic data sets[2] [4, 13, 7, 6]. Although the earlier studies on synthetic data sets were suggestive, they are not necessarily good predictors of how the methods will perform in fielded systems. Consequently, we have used a wider variety of data sets for our current study. These include "live" normal data (traces of programs collected during normal usage of a production computer system), different kinds of programs (e.g., programs that run as daemons and those that do not), programs that vary widely in their size and complexity, and different kinds of intrusions (buffer overflows, symbolic link attacks, Trojan programs, and denial-of-service). We use programs that run with privilege (with one exception, described below), because misuse of these programs has the greatest potential for harm to the system. Table 1 summarizes the different data sets and the programs from which they were collected. All of these data sets are publicly available and carefully described at http://www.cs.unm.edu/~immsec/data-sets.html. Intrusions were taken from public advisories posted on the Internet.

Each trace is the list of system calls issued by a single process from the beginning of its execution to the end. This is a simple definition, but the meaning of a process, or trace, varies from program to program. For some programs, a process corresponds to a single task; for example, in `lpr` each print job generates a separate trace. In other programs, multiple processes are required to complete a task. In some, such as `named`, a single daemon process runs continuously, monitoring events or awaiting requests, and occasionally spawning subprocesses to handle certain tasks. Even in processes that are not daemons, the number of system calls per trace varies widely, as can be seen by comparing the data for `lpr` and that for `xlock`.

Data for `lpr` were collected at two universities under identical conditions (OS, version of `lpr`, etc.), but with different users and network configurations. The UNM normal data set includes fifteen months of activity, while the MIT data set includes two weeks. Each set includes a large number of normal print jobs and a single `lprcp` symbolic link intrusion that consists of 1001 print jobs. Detection of an

[2] Synthetic traces are collected in production environments by running a prepared script; the program options are chosen solely for the purpose of exercising the program, and not to meet any real user's requests.

Program	Intrusions	Normal data available		Normal data used for training		Normal data used for testing	
	Number of traces	Number of traces	Number of system calls	Number of traces	Number of system calls	Number of traces	Number of system calls
MIT `lpr`	1001	2,703	2,926,304	415	568,733	1,645	1,553,768
UNM `lpr`	1001	4,298	2,027,468	390	329,154	2,823	1,325,670
`named`	2	27	9,230,572	8	677,340	12	7,690,572
`xlock`	2	72	16,937,816	72	778,661	1	16,000,000
`login`	9	12	8,894	12	8,894	–	–
`ps`	26	24	6,144	24	6,144	–	–
`inetd`	31	3	541	3	541	–	–
`stide`	105	13,726	15,618,237	150	246,750	13,526	15,185,927
`sendmail`	–	71,760	44,500,219	4,190	2,309,419	57,775	35,578,249

Table 1. Amount of data available for each program. "Normal data used for training" refers to models built with sequence length six; sequence length ten models used more training data. The same test data were used for both sequence lengths; this includes all normal data not used for training either set of models.

anomaly in any of these 1001 traces is considered successful detection of the intrusion.

The `named` normal data consist of a single daemon trace and traces of its subprocesses, collected for one month. The intrusion against `named` is a buffer overflow; we used two sample traces of this intrusion.

Data for `xlock` include 71 synthetic traces, and a single live trace. The live trace, however, is very long; `xlock` generates a huge number of system calls as it continually updates the user's screen and it was left running for two days to collect these data. The intrusion used here is also a buffer overflow. As with `named`, we used two sample traces of the same intrusion.

The `login` and `ps` normal data sets are relatively small. These are simpler programs, and little variation in normal behavior is expected from additional traces. The small data set, however, means that there is insufficient data for thorough analysis of false positives.

For both `login` and `ps`, we used Trojan intrusions, which allow unauthorized access to the system through a built in "back-door." A number of traces have been collected from each Trojan program. Some of the Trojan program traces were collected from actual Trojan programs installed during a live intrusion. These traces are easy to detect because the Trojan program was a different version from the program it replaced. Other traces are for Trojan programs we created directly from the installed (normal) version of the program. Some of the traces correspond to use of the back door to break into the system, while others are from ordinary users logging in to the Trojaned program normally (without using the back door). Ideally, we would like to detect the presence of Trojan code whether or not it is currently being used for unauthorized access, so each trace is treated as a separate example of an intrusion. However, this is a stringent test, as the foreign code is not being executed.

The `inetd` program is typically started as a foreground process, which initiates a daemon process to run in background and then exits. The daemon process in turn, initiates child processes which perform a fixed set of initialization steps and then execute some other program. Child processes are, therefore, very nearly identical. The normal data for `inetd` include a trace of the startup process, a daemon process, and a representative child process. The intrusion used against `inetd` is a denial-of-service attack that ties up network connection resources. As the attack progresses, more of the system calls requesting resources return abnormally and are re-issued. The intrusion data collected include a startup process, a daemon process, and several child processes, but only the daemon process is expected to show any deviation from normal behavior.

A second denial-of-service attack we tested ties up all the memory available on a system. This affects any running program that requests memory during the denial-of-service attack. In this one case, we departed from our policy of monitoring only privileged processes, and instead traced the analysis program `stide` (which was processing the `sendmail` data). The normal and intrusion data were collected while `stide` was processing the same data, but the latter was interrupted by the denial-of-service attack.

The final data set, `sendmail`, consists only of normal data because this version of `sendmail` running on a production mail server was not vulnerable to any known `sendmail` intrusions. However, we were able to collect a very large set of live normal data, and use this for false positive analysis. Note that these data were collected from a different version of `sendmail` than that used in our earlier papers.

4. Experimental Design

Our objective is to compare several different data-modeling methods using data from several different programs, thereby getting a better overall picture of their relative merits. For accurate intrusion detection, we must correctly classify both intrusions and normal data. Errors in the first category, where intrusions are not identified, are called *false negatives*. Errors in the second category, where normal data are identified as anomalous, are called *false positives*. We wish to minimize both kinds of errors, or equivalently, maximize true positives and minimize false positives. We do not attempt to measure performance in terms of system usage, although we do make some general observations about computational effort.

For most of our data sets, we have only a single intrusion script, and each method has a single threshold above which that intrusion is detected and below which it is missed. To get a better picture of the gradual trade-off between false positives and false negatives that often occurs with multiple intrusions, we combine results across all available programs. By using the composite results, we also can see which methods can be used on multiple data sets with a single set of parameters and which require hand-tuning.

However, using several programs also complicates the design of the experiments. First, we would like to use comparable amounts of data for each program in building our models of normal. Since the programs vary in complexity and a trace does not have a similar meaning in each program, simply choosing a fixed number of traces or system calls to include would not be a good approach. Second, we need to define a consistent measure for comparing false positives.

Figure 1 shows the number of unique sequences as a function of the total number of sequences seen for one of our data sets. The x-axis represents the sequences seen in chronological order, from traces added in the order in which they were collected. At first, almost every sequence is new, but gradually the number of new sequences drops off. One way of establishing a consistent measure of how much training data to use across several programs is to set a target for the slope of this growth curve. Once the rate of encountering new sequences drops below some preset value, we say we have enough data with which to build our model of normal.

Unfortunately, the growth curves for our data sets are not very smooth. Several traces might pass with no new sequences, and then several new sequences are encountered close together. This is not surprising, as a change in system call order affecting one sequence generally affects the nearby sequences as well. We considered several methods for smoothing this curve so as to get a better estimate of the slope, and eventually settled on the following approach. Figure 2 shows several different versions of the growth curve for the same data. The pool of normal data traces is treated as a loop, where the first trace follows the last trace. For each curve shown in figure 2, a different starting point on this loop was chosen randomly, and then the traces were read in order from that point. This allows us to examine variations in the growth curve without reducing the amount of data used or disrupting the chronological ordering of traces.

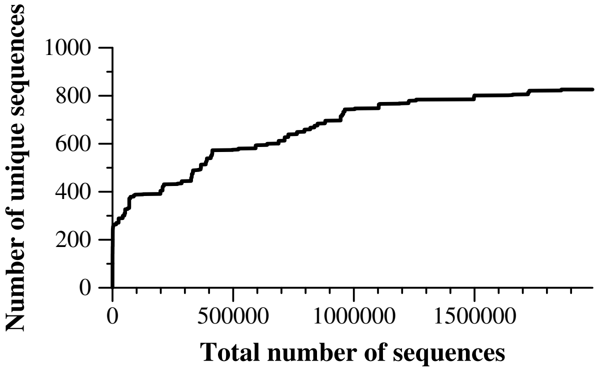

Figure 1. Typical database growth curve. The graph shows how the size of the normal database grows as traces are added chronologically.

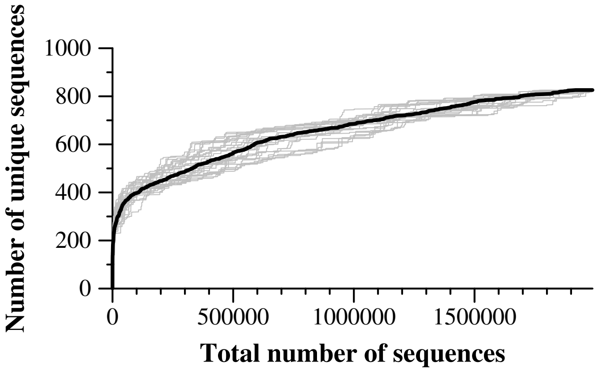

Figure 2. Alternate database growth curves for the same data used in Figure 1. Light lines show standard growth curves for different starting points in the training data; the dark line shows the mean.

The average of these individual growth curves, shown as the darker line in figure 2, gives a smoother estimate of the rate at which the database grows. This is not a precise measure, but a rough way to estimate how much training data should be used. For our experiments, we selected the first point on the average curve at which the local slope was less than one new sequence per 10 traces, and remained that way for at least 10 traces. We used each program's average trace length for this estimate, except in those cases where a single long trace skewed the average. For `xlock`, we averaged the lengths of only the synthetic traces, and for `named`, we used the median trace length. We chose two groups of training traces in this fashion, one for sequences of length six, and another for sequences of length ten. All data not included in the second, larger training set were used for testing. Table 1 shows how much training data we used for each program for sequence length six.

On the testing side, false positives were measured differently from true positives. To detect an intrusion, we require only that the anomaly signal (described below) exceed a preset threshold at some point during the intrusion. However, making a single determination as to whether a normal trace appears anomalous or not is insufficient, especially for very long traces. If a program is running for several days or more, each time that it is flagged as anomalous must be counted separately. The simplest way to measure this is to count individual decisions. The false-positive rate then is the percentage of decisions in which normal data were flagged as anomalous. Note that the same approach cannot be used for measuring true positives. Intrusion traces generally resemble normal traces in large part, and each individual sequence within an intrusive trace is more likely to be normal than not.

5. Building models of normal behavior

We modeled the normal behavior of each of the data sets described in Section 3 using each of the four methods chosen earlier. This process took much longer for HMMs than for the other methods. On our largest data set, HMM training took approximately two months, while the other methods took a few hours each. For all but the smallest data sets, HMM training times were measured in days, as compared to minutes for the other methods. The subsections below explain the details behind each method.

5.1. sequence time-delay embedding (stide)

In sequence time-delay embedding (stide), a profile of normal behavior is built by enumerating all unique, contiguous sequences of a predetermined, fixed length k that occur in the training data. We ran experiments with sequence lengths of six and ten. For a sequence length of six, we slide a window of length six across each trace, one system call at a time, adding each unique sequence to the normal database. The sequences are stored as trees to save space and to speed up comparisons. Building such a database requires only a single pass through the data, unlike some of the methods described below.

At testing time, sequences from the test trace are compared to those in the normal database. Any sequence not found in the database is called a *mismatch*. Any individual mismatch could indicate anomalous behavior, or it could be a sequence that was not included in the normal training data.

To date, all of the real intrusions we have studied produce anomalous sequences in temporally local clusters. This is convenient for defining an on-line measure of anomalous activity. We derive our on-line measure, or anomaly signal, from the number of mismatches occurring in a temporally local region, called a *locality frame*. The data reported below used a locality frame of 20 system calls. At each point in our test trace, we check whether the current sequence is a mismatch, and keep track of how many of the last 20 sequences were mismatches. This *Locality Frame Count* (LFC) gives us our anomaly signal. (A somewhat different approach was taken in [7], where the measure of anomalous behavior was based on Hamming distances between unknown sequences and their closest match in the normal database.)

We then set a threshold on the LFC, below which traces are still considered to be normal. Any time the LFC reaches or exceeds the threshold, an anomaly is recorded. This LFC threshold is the primary sensitivity parameter used in the experiments described below; it ranges from 1 to 20. Lower LFCs tend to catch more intrusions and also give more false positives, higher LFCs tend to produce fewer true and false positives.

5.2. stide with frequency threshold (t-stide)

A simple addition to stide allows us to test the premise that rare sequences are suspicious. For each sequence in the database, we keep track of how often it has been seen in the training data. Once all the training data have been processed, we then determine each sequence's overall frequency. For our experiments, "rare" was defined as any sequence accounting for less than 0.001% of the normal training data. The "t" in "t-stide" represents the addition of this threshold on sequence frequencies.

Sequences from test traces are compared to those in the database, as for stide. Rare sequences, as well as those not included in the database, are counted as mismatches. These mismatches are aggregated into locality frame counts as described earlier. Again, the threshold on locality frame counts is the primary sensitivity parameter.

5.3. RIPPER

RIPPER—Repeated Incremental Pruning to Produce Error Reduction—is a rule learning system developed by William Cohen [2]. It, like other rule learning systems, is typically used for classification problems. Training samples consist of a set of attributes describing the object to be classified, and a target class to which the object belongs. Given enough such examples, RIPPER extracts rules of the form:

classA:- attrib1 = x, attrib5 = y.

classB:- attrib2 = z.

classC:- true.

In this example, class A is chosen if attributes 1 and 5 are x and y, respectively; class B is chosen if attribute 2 is z; and class C is the default class. Conditions can also specify that an attribute not equal a certain value. (For other types of data, more conditions are possible.) Multiple conditions are always taken to mean that all conditions must hold.

For the intrusion detection problem, such classification is useful only if one has a complete set of examples of the abnormal class(es) with which to train the system. We are primarily interested in the application to anomaly detection, where we do not have both positive and negative instances. Lee and others [13, 12] adapted RIPPER to anomaly detection by using it to learn rules to predict system calls within short sequences of program traces.

For each program, we used a list of all unique sequences occurring in that program to create the RIPPER training samples. Each sequence was turned into a RIPPER sample by treating all system calls except the last in a sequence as attributes, and the last as the target class. (This requires renaming the last system call, as RIPPER will not accept classes that look like attributes.) Similar attribute/target pairs were created for test traces, but in that case all sequences were used, not just a sample of each unique sequence.

RIPPER has a difficult time learning rules for classes about which there is not enough information, such as a system call that only occurs at the end of a sequence once [11]. Because the frequencies of each sequence are not being recorded, simple duplication of each sequence y times is effective. We replicated each training sample twelve times to create the training file, as did Lee and Stolfo in [12].

RIPPER takes these training samples and forms a hypothesis—a list of rules to describe normal sequences. For each rule, a violation score is established from the percentage of times that the rule was correctly applied in the training data. For a rule whose conditions were met M times in the training data and whose prediction was correct for T of those times, the penalty for violating that rule is $100M/T$. Lee and others used the average of these violation scores to rank a trace [12], but such a measure is inappropriate for on-line testing. We first used a moving average of these violation scores over the locality frame, but found that gave excessive false positives. Instead, we call each sequence that violates a *high-confidence* rule a mismatch, equivalent to the stide mismatches described earlier. These mismatches then can be aggregated into locality frame counts, also described earlier. We chose *high-confidence* to mean those rules with violation scores greater than 80.

5.4. Hidden Markov Model

Standard HMMs have a fixed number of states, so one must decide on the size of the model before training. Preliminary experiments showed us that a good choice for our application was to choose a number of states roughly corresponding to the number of unique system calls used by the program. Most of our test programs use an alphabet of about 40 system calls, hence 40-state HMMs were used in most cases. We used a 20-state HMM for `ps` and `stide`, a 35-state HMM for `inetd`, and a 60-state HMM for `sendmail`. The states are fully connected; transitions are allowed from any state to any other state. For each state then, we need to store the probabilities associated with transitions to each other state, and the probabilities associated with producing each system call. For a program using S system calls, and hence a model of S states, this means roughly $2S^2$ values.

In most cases, transition and symbol probabilities were initialized randomly, and then trained using the Baum-Welch algorithm as described in [14]. Occasionally, however, prior knowledge is useful in performing the initialization. This was the case with the `lpr` data sets. A primary difference between `lpr` traces is in the length of the document being printed. This is reflected in the traces as the number of `read-write` pairs. We found that randomly initialized HMMs devoted most of the states and a great deal of training time to modeling the different frequency distributions of this particular subsequence. As a result, these HMMs were less likely to recognize the intrusion. However, when the model was initialized with a predetermined `read` state and `write` state arranged in a loop, the rest of the model states were available to represent other parts of the traces and accuracy improved. We assigned large probabilities to the desired transitions and output system calls for the `read` and `write` states, and low probabilities for the alternatives. Transition and output probabilities for the other states were randomized.

During training, the probabilities were iteratively adjusted to increase the likelihood that the automaton would produce the traces in the training set. Several passes through the training data were required. To avoid over-fitting the training data, the likelihood of the model producing a second set of normal traces (not used in training) was periodically measured. When this second likelihood stopped improving, training was terminated.

As mentioned earlier, training an HMM is expensive. Calculations for each trace in each pass through the training data take $O(TS^2)$, where T is the length of the trace in system calls (see Table 1), and S is the number of states (and symbols). Also, storage requirements are high. The "trellis" of intermediate values that must be kept while performing the calculations for a particular trace requires $T(2S+1)$ floating point values. For our longer traces, these values were written to a memory mapped binary file.

Fortunately, testing is more efficient. A standard way to test an HMM is to compute the likelihood that it will produce data not in the original training set. We, however, used a simpler measure that (unlike the standard method) is not sensitive to trace length and is better suited to on-line use. We use the graph underlying the HMM as a nondeterministic finite automaton. We "read" a trace one system call at a time, tracking what state transitions and outputs would be required of the HMM to produce that system call. If the HMM is a good model of the program, then normal traces should require only likely transitions and outputs, while intrusive traces should have one or more system calls that require unusual state transitions and/or symbol outputs.

At a given time t, there is a list of current possible states. Choosing only the most likely state for any single system call might not be consistent with the best path through the HMM for a sequence of system calls, so we keep track of all possible paths. Thresholds are set for "normal" state transition and output probabilities. Then, if we encounter a system call in the trace which could only have been produced using below-threshold transitions or outputs, it is flagged as a mismatch. Note that we could have used the LFC to aggregate these mismatches, but HMM anomalies are usually not temporally clumped, so we thought it more fair to count individual mismatches. For our experiments, the same threshold was used for both state transitions and outputs. This parameter was the primary sensitivity parameter, with thresholds varying from 0.01 to 0.0000001. Note that HMMs are making anomaly decisions at each system call, rather than on sequences as in the other three methods.

The time to check each system call depends on the model size and the size s of the current list of valid states. The latter tends to stay very small with normal traces, but can include up to all S states after an anomaly has been identified. For each current valid state, our implementation of the program takes O(S) to decide whether there is an anomaly or not. If $s = S$, this means O(S^2) to process one system call. These times could be improved by converting the model to a better representation of the automaton once the testing probability thresholds are known.

6. Results

We tested each of the four data modeling methods on each of the data sets (traces of Unix programs) at several different sensitivity thresholds. False positives are reported for normal data not used during training, and true positives are reported for traces of anomalous behavior. We first present the overall results, and then discuss accuracy on individual data sets.

To get a picture of how well the detection methods perform on a variety of data, we first averaged the results across all the data sets. Figure 3 shows these average results for each combination of data modeling method and sensitivity threshold. A different symbol is used to denote each method, and each point shows performance at a particular threshold. For HMMs, we distinguish between results with randomly-initialized HMMs and those using HMMs initialized to include human knowledge of the modeled program.

In Figure 3, the y-axis represents overall ability to detect anomalies. As mentioned earlier, any above-threshold signal anywhere in the intrusive trace(s) counts as correct detection of the intrusion. The x-axis represents false positives, measured on an individual decision basis rather than by traces. False positives are shown as a fraction of the total number of sequences (or system calls) in a trace of normal behavior, and therefore can range from 0 to 1. The figure, however, shows only the region from 0 to 0.001 which is of primary interest. As a very rough estimate, traces are often on the order of a thousand system calls long. Identifying one in a thousand sequences (or system calls, for HMMs) as anomalous is roughly equivalent to identifying each trace as anomalous. Of course, this does not hold everywhere, because of the vast differences in traces mentioned earlier. However, it does suggest that for practicality, false-positive rates should be well below 0.001. Perfect performance would be correct detection of all intrusions and no false positives, represented by points in the upper left corner of the figure.

For many of the data sets, the individual true-positive rate was either one, if the intrusion(s) was successfully detected, or zero, because there are only two data sets for which some intrusion traces are harder to recognize than others. This makes the true-positive average a simple representation of how many intrusions are detected. The normal data, however, are more varied. With differences between false positives that span several orders of magnitude, the average is heavily influenced by the worst results. Thus, we also show the median scores in the inset of figure 3. Note that the scale for false positives is much smaller on this figure, as the median is significantly lower than the average for all methods and thresholds. (t-stide results do not show up

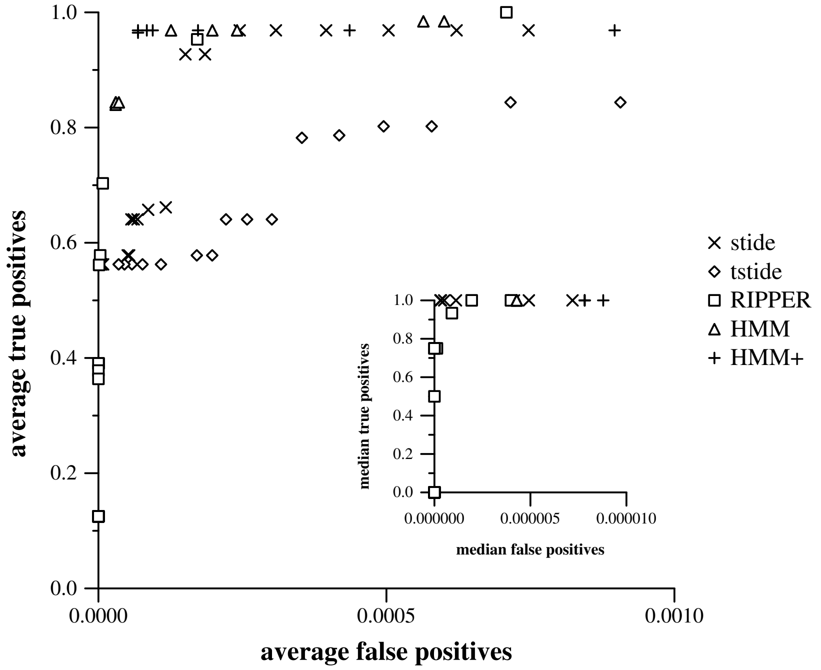

Figure 3. Composite results for each method on all data sets, sequence length 6. Each point represents performance at a particular threshold. True-positive values are the fraction of intrusions identified. For the sequence-based methods, false positives are the fraction of sequences giving mismatches at or above the specified locality frame count threshold. For HMMs, false positives are the fraction of system calls corresponding to state transitions or outputs below the specified probability threshold. Points labeled "HMM" are for only randomly-initialized HMMs, while those for "HMM+" use the specially-initialized HMMs designed to handle lpr data. No t-stide points appear in the median plot because the false positives are off the scale. Results for four HMM thresholds all map to the single median point shown.

on this new scale; although t-stide median results are lower than their corresponding means, they are not as low as those for the other methods). Many of plotted points on the y-axis of the inset (median true positives) are 1.0, because stide and both HMM methods correctly detect a majority of the intrusion traces at all thresholds. RIPPER's true positive rates, however, drop off gradually with increasing threshold, while its false positives drop rapidly; this accounts for the points running down the y-axis.

The composite picture shown in figure 3 gives only a rough outline of the data. Figure 4 shows the relationship between thresholds and true or false positives in more detail. As the sensitivity threshold is relaxed, fewer sequences (or system calls) are identified as anomalous in all traces, affecting both true and false positives. "Relaxed" for the sequence-based methods means an increase in the LFC needed to flag an anomaly, while it means a decrease in the minimum probability for an HMM to generate a normal system call. The RIPPER curves are steeper because RIPPER rarely generates high LFCs. Because RIPPER's rules depend only on a few of the system calls in a sequence, not all sequences in an anomalous part of the trace are classified as anomalous.

We can use the results shown in figure 4 to choose good thresholds for each method. The definition of "good," however, is not fixed. On one system it may be more important to maximize true positives, while on another, minimizing false positives may be key. For the moment, we choose "good" to mean an average true-positive rate above 95%. HMMs, stide and t-stide all have at least one threshold at which the average true-positive rate is 96.9%, missing only two of the `login` intrusion traces. RIPPER's closest match is a true-positive rate of 95.3%, missing three of the `login` intrusion traces. Using these thresholds, we now turn to a comparison of the corresponding false-positive rates.

Figure 5 shows the false-positive rate for each method on each of the six normal test sets, using thresholds chosen as described above. Values for one data set vary over

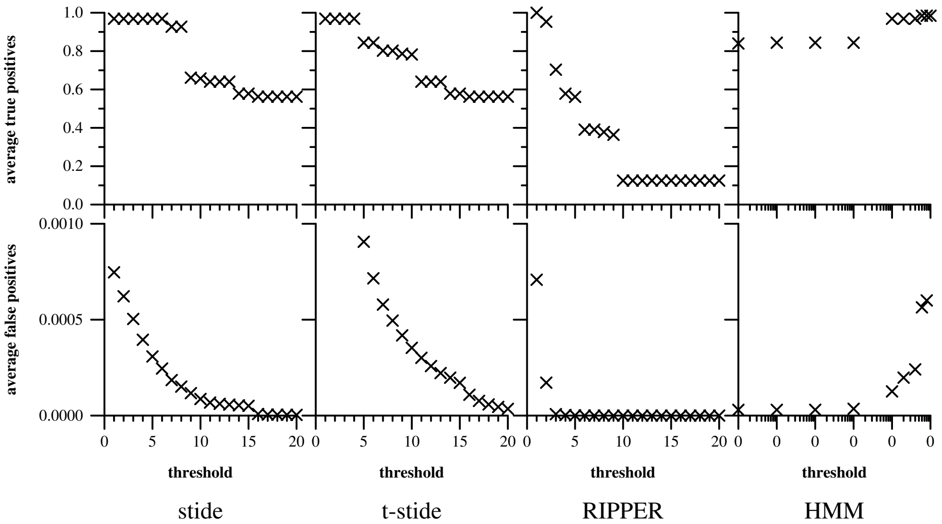

Figure 4. Average true and false positives versus threshold for each method, sequence length 6. HMM results are for randomly-initialized HMMs only.

orders of magnitude (note the logarithmic scale on the y-axis). There is some correlation between long traces and low false-positive rates (ignoring the `named` daemon for the moment), but the number of misclassified normal traces still varies widely between programs. More important than trace length are the complexity of a program and the variability in its usage. The `sendmail` program is larger and more complicated than the other programs, producing traces with a larger variety of system calls and sequence patterns, so it is not surprising that it is more difficult to model. By contrast, `xlock` and `stide` are much simpler programs, which do not interact with a network. They also gave our least realistic data sets—`xlock` because the test data come from just one abnormally long trace, and `stide` because it is an application program. The two `lpr` data sets were produced by the same program, but the UNM false positives are higher for all methods. We speculate that because the UNM data were collected over a much longer period of time, they reflect more changes in network configurations and user behavior than the MIT data.

It is possible that we simply did not use enough training data to characterize the more complex data sets adequately, even though our training set sizes were determined by the variability of system call patterns (see Section 4). The `sendmail` training set was larger than any other program's training set, and the `lpr` training sets used a larger percentage of the available data. Yet `sendmail` and UNM `lpr` had the worst false-positive rates.

Results across modeling methods for a particular data set are more similar than results for the same method across different data sets. Although it is clear that t-stide consistently performs below the other methods, none of the other methods is a clear winner. We would need to better understand why false-positive rates vary so much between data sets before we could conclusively identify one method as best. That is, our data suggest that there is no single best choice for all the data sets. However, none of the methods (other than t-stide) would be a bad choice.

All of the results presented in figures 3, 4, and 5 were computed using a sequence length (window size) of six, and the corresponding choice of training data. Results for sequence length ten are qualitatively similar, but there are a few points worth noting. We expected the results for the sequence-based methods to be similar for the two sequence lengths because we chose the size of the training set based on the sequence length. The training set chosen for sequence length ten was much larger than that for sequence length six, reflecting the fact that there are many more possible sequences of length ten than of length six. RIPPER and stide average results appear to be slightly better for sequence length six, despite training on less data, possibly because the smaller sequences allow better generalization. However, HMMs do not depend on the sequence length; HMM accuracy was better for most data sets on the larger training set.

However, these trends in the average behavior do not hold for all programs. Each method has some programs for which sequence length six false positive rates are lower and

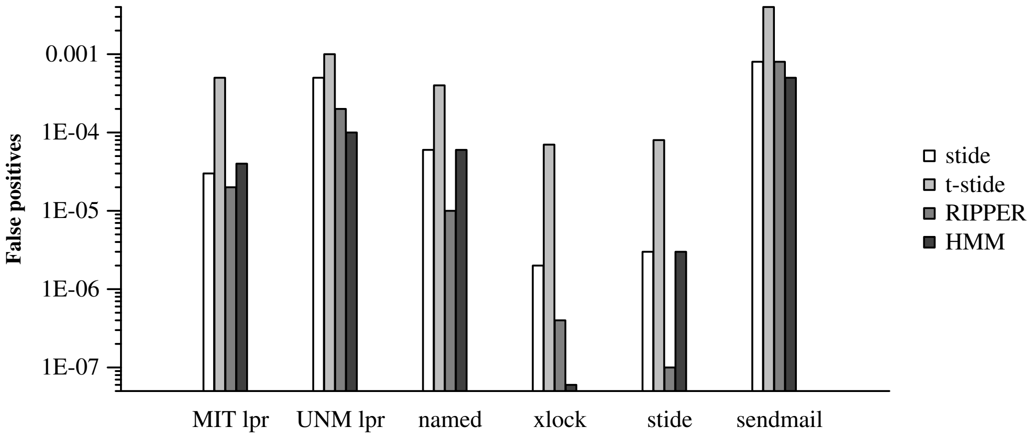

Figure 5. False-positive rates for each of six data sets, sequence length six. stide threshold: 6, t-stide threshold: 4, RIPPER threshold 2, HMM threshold = 0.001. Note that the RIPPER true-positive rate at this threshold is is slightly lower than those of the other methods. False-positive rates are shown on a logarithmic scale.

others for which they are higher. This is likely due to the fact that our results depend somewhat on which traces or sequences are included in the training set. A different choice of training data, even if the amount of data were the same, would yield different test results in some cases. We have not attempted to measure this variation or obtain an error estimate for the false positives. The comparisons made here are suggestive, rather than tests of statistical significance.

7. Discussion

Intrusion detection is an important and active area of research. Various research groups have suggested methods that look promising on at least one set of data. But in order to choose from among these different methods, we need good comparisons between them on a variety of data. Such comparisons are not easy. Differences in how the methods work and large variations in the amount and types of data both complicated our study.

One such difference is in the way anomaly signals are generated. The sequence-based methods tend to produce multiple mismatches even for a single misplaced system call, because that system call affects multiple sequences. Because HMMs, as used here, check only a single system call at a time, they automatically produce fewer mismatches. This biases the results in favor of HMMs. An alternative that seems more fair at first is to compare *peaks*. A trace region where locality frame counts are all above 0 shows up as a peak in a graph of the anomaly signal over time. Such a peak might be equivalent to a single HMM anomaly. But the size of the locality frame might cause multiple anomalies to be lumped together in one such peak. Perhaps contiguous mismatches would be a better definition of *peak*. However, in either case, it is impossible to calculate a percentage for false positives, because there is no notion of how many such peaks are possible in a given trace.

Regardless of how the false positives are calculated, more test data would improve our confidence in the results. Although we have collected data for a spectrum of different kinds of programs and intrusions, we still have a relatively small number of data sets. Each individual data set carries too much weight in the final outcome, and adding results for one more data set might change the composite results enough to favor other methods.

Studies such as ours can always be conducted more thoroughly. As with collecting data, there are no predetermined stopping criteria. Each modeling method has a number of parameters that affect anomaly signals, but only a few representative variations were investigated here. Also, there is a random element to both RIPPER and HMMs, so results for these methods should ideally be averaged over multiple trials.

For these reasons, we cannot definitively determine which method is best. However, we can make some general statements about which properties of the data were helpful or harmful for each of the methods.

We purposely chose methods that could handle discrete data, but the large number of distinct system calls used is a problem for some methods. In training an HMM, the time for each pass is roughly proportional to the square of the alphabet size (number of different system calls). More complex programs using more system calls require significantly longer training times. On the testing side, each decision in the current implementation requires a number of tests directly proportional to the number of distinct system calls. By contrast, in stide and t-stide, the number of system calls is only an indirect factor in training and test times, because of the way the data are stored. The search time for a sequence in the database depends on the number of branches in the sequence trees. Although the number of possible branches at each level is equal to the number of system calls, the number encountered in practice is significantly fewer. These methods scale dramatically better with the number of system calls used.

Scaling with the length of the traces is another factor. All of the methods have training and test times that are linear in the length of the trace. However, the training algorithms for HMMs and RIPPER make multiple passes through the training data, whereas stide and t-stide require only a single pass to build their normal databases. Also, as mentioned earlier, HMMs must store intermediate data while training, with the number of floating point values proportional to the trace length multiplied by the number of states. For long traces, this is very expensive.

The number of unique sequences in a data set is not directly related to the trace length. In fact, longer traces often repeat a small number of sequences many times. As mentioned earlier, the primary difference between `lpr` traces is the number of `read`-`write` pairs. Also, in the long `xlock` live trace, the bulk of the data consists of the same five system calls repeated over and over. This is one reason why RIPPER is trained on only the unique sequences, and not on the raw data. Otherwise, those few very common sequences would dominate, and few or no rules would be extracted for the other sequences. It also suggests a problem for the frequency-based methods. With a few sequences accounting for a large percentage of the data, frequencies of other normal sequences tend to look insignificant, and can be flagged as anomalous. However, these common sequences do not dominate every trace; in the shorter sequences frequencies are more evenly distributed. Because of this, simple methods for comparing rare and common sequences are insufficient, although more sophisticated approaches could perhaps make better use of the frequency information.

Each method we used was designed to take advantage of the locality of intrusions. The sequence based methods, using locality frame aggregates of the mismatch counts, all focus on the local history of system calls. Although HMMs have the potential to capture some long-term history as well, the way we used them also concentrated on local events. This is partly because of our choice of model size; more states would be required to give the HMMs a longer memory.

In these relatively small HMMs, each state might be used to characterize multiple parts of the traces. A single state producing primarily `read` system calls, for example, might represent several different program states in which reading data is required. Transitions out of that state might reflect the different possibilities for going on to `write` or to `close` or to another `read`. There is a potential here for missing anomalies, because such state transitions might make it possible to mix prefixes and suffixes that do not go together. However, there is also a potential for better generalization than that offered by the sequence-based methods. As an example, if the training data include examples of a system call being used one, three, or four times in a row, an HMM will likely accept a trace using that system call twice in a row. The sequence based methods (with the possible exception of RIPPER) would identify at least some mismatches. In the data we have studied, such sequences are always false positives, and do not contribute to identifying anomalies.

An additional factor in evaluating methods is the degree to which training can be automated. The ability to add human knowledge to the model might be helpful, but such knowledge should not be required.

8. Conclusions

We compared four methods for characterizing normal behavior and detecting intrusions based on system calls in privileged processes. Each method was tested on the same suite of data sets, consisting of different types of programs and different intrusion techniques. On this test suite, three of the four methods performed adequately. Hidden Markov models, generally recognized as one of the most powerful data modeling methods in existence, gave the best accuracy on average, although at high computational costs. Surprisingly, the much simpler sequence time-delay embedding method compared favorably with HMMs. We conclude that for this problem, the system call data are regular enough for even simple modeling methods to work well. The average results indicate that it might be possible to achieve increased accuracy with HMMs, provided significant computational resources are available to train and run them.

However, no one method consistently gave the best results on all programs, and results between programs varied more than results between methods. Variations in false positives were due more to the complexity of the traced programs and their environments than to differences in the analysis methods. Although there are multitudes of alternative methods that were not tested, our results demonstrate that for this problem, several methods perform well. We believe that the choice of data stream (short sequences of system calls) is a more important decision than the particular method of analysis.

Historically, many computationally sophisticated methods have been applied to the intrusion-detection problem, yet there are few well-accepted solutions in widespread use. One lesson from this paper is that perhaps a disproportionate amount of attention has been directed to the data modeling problem, and that equal attention should be paid to considering what are the most effective data streams to monitor.

9. Acknowledgments

The authors thank Mark Crosbie, Patrik D'haeseleer, Paul Helman, Geoff Hunsicker, Anil Somayaji, Derek Smith, Al Valdes, Wenke Lee, Carla Marceau and Matt Stillerman for helpful discussions and suggestions. We also thank the MIT AI Lab and the UNM CSSupport group for allowing us to collect data on their production systems. This work was supported by grants from the National Science Foundation (IRI-9711199), DARPA (N00014-96-1-0680), Intel Corporation, and IBM.

References

[1] R. C. Carrasco and J. Oncina. Learning stochastic regular grammars by means of a state merging method. In *Proceedings of the Second International ICGI Colloquium on Grammatical Inteference and Applications*, pages 139–152, Alicante, Spain, 1994.

[2] W. W. Cohen. Fast effective rule induction. In *Machine Learning: the 12th International Conference*. Morgan Kaufmann, 1995.

[3] M. Damashek. Gauging similarity with n-grams: Language-independent categorization of text. *Science*, 267:843–848, Feb. 1995.

[4] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, pages 120–128, Los Alamitos, CA, 1996. IEEE Computer Society Press.

[5] P. Helman and J. Bhangoo. A statistically based system for prioritizing information exploration under uncertainty. *IEEE Transactions on Systems, Man and Cybernetics, Part A: Systems and Humans*, 27(4):449–466, July 1997.

[6] G. G. Helmer, J. S. K. Wong, V. Honavar, and L. Miller. Intelligent agents for intrusion detection. In *Proceedings, IEEE Information Technology Conference*, pages 121–124, Syracuse, NY, September 1998.

[7] S. A. Hofmeyr, S. Forrest, and A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6:151–180, 1998.

[8] H. S. Javitz and A. Valdes. The NIDES statistical component: Description and justification. Technical report, Computer Science Laboratory, SRI International, Menlo Park, CA, March 1993.

[9] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in priviledged programs by execution monitoring. In *Proceedings of the 10th annual Computer Security Applications Conference*, pages 134–144, December 1994.

[10] A. P. Kosoresow and S. A. Hofmeyr. A shape of self for UNIX processes. *IEEE Software*, 14(5):35–42, 1997.

[11] W. Lee, 1998. personal communication.

[12] W. Lee and S. J. Stolfo. Data mining approaches for intrusion detection. In *Proceedings of the 7th USENIX Security Symposium*, 1998.

[13] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from UNIX process execution traces for intrusion detection. In *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. AAAI Press, July 1997.

[14] L. R. Rabiner. A tutorial on Hidden Markov Models and selected applications in speech recognition. *Proceedings of the IEEE*, 77(2):257–286, 1989.

[15] L. R. Rabiner and B. H. Juang. An introduction to Hidden Markov Models. *IEEE ASSP Magazine*, pages 4–16, January 1986.

[16] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25, 1996.