Modeling System Calls for Intrusion Detection with Dynamic Window Sizes

Eleazar Eskin
Computer Science Department
Columbia University
500 West 120th Street, New York, NY 10027
eeskin@cs.columbia.edu

Wenke Lee
Computer Science Department
North Carolina State University
Raleigh, NC 27695-7550
wenke@csc.ncsu.edu

Salvatore J. Stolfo
Computer Science Department
Columbia University
500 West 120th Street, New York, NY 10027
sal@cs.columbia.edu

Abstract

We extend prior research on system call anomaly detection modeling methods for intrusion detection by incorporating dynamic window sizes. The window size is the length of the subsequence of a system call trace which is used as the basic unit for modeling program or process behavior. In this work we incorporate dynamic window sizes and show marked improvements in anomaly detection. We present two methods for estimating the optimal window size based on the available training data. The first method is an entropy modeling method which determines the optimal single window size for the data. The second method is a probability modeling method that takes into account context dependent window sizes. A context dependent window size model is motivated by the way that system calls are generated by processes. Sparse Markov transducers (SMTs) are used to compute the context dependent window size model. We show over actual system call traces that the entropy modeling methods lead to the optimal single window size. We also show that context dependent window sizes outperform traditional system call modeling methods.

1 Introduction

Intrusion Detection Systems (IDS) are becoming an important part of computer security systems. A major advantage of IDS is the ability of the IDS to detect new and unknown attacks by examining audit data collected from a system. Typically this detection is performed through a data mining technique called anomaly detection [1]. Anomaly detection builds models of “normal” audit data (or data containing no intrusions) and detects intrusions based on detecting deviations from this normal model. The performance of these models depends greatly on the robustness of the modeling method and the quantity and quality of the available training data [2]. Much of this data is sequential in nature. The basic units of the modeling technique are short contiguous subsequences obtained with a sliding window. In this paper we present robust methods for choosing the sliding window size and apply them to modeling system call traces, a common type of audit data. We present two methods for setting the window size. The first method is information theory based and estimates the optimal single window size for the data. The second method is probability based and uses a dynamic window size set by the *context*. We compare these methods to traditional system call modeling methods and show how our methods outperform previous approaches.

System call traces are a common type of audit data collected for performing intrusion detection. A system call trace is the ordered sequence of system calls that a process performs during its execution. The trace for a given process can be collected using system utilities such as *strace*. System call traces are useful for detecting a user to root (U2R) exploit or attack. In this type of exploit, a user exploits a bug in a privileged process (a process running as root) using a buffer overflow to create a root shell. Typically, the system call trace for a program process which is being exploited is drastically different from the program process under normal conditions. This is because the buffer overflow and the execution of a root shell typically call a very different set of system calls than the normal execution of the program. Because of these differences, we can detect when a process is being exploited by examining the system calls.

Traditionally, these methods typically build models over short contiguous subsequences of the system call trace. These short continuous subsequences are extracted with a sliding window. Traditionally, system call modeling methods employ a fixed window size. There have been many different methods proposed for building models over these short contiguous subsequences. A survey and comparison of anomaly detection techniques is given in [17]. Stephanie Forrest presents an approach for modeling normal sequences using look ahead pairs [4] and contiguous sequences [7]. Helman and Bhangoo [6] present a statistical method to determine sequences which occur more frequently in intrusion data as opposed to normal data. Lee et al. [9, 8] uses a prediction model trained by a decision tree applied over the normal data. Ghosh and Schwartzbard [5] use neural networks to model normal data. Ye uses a Markov chain-based method to model the normal data [18].

Each of these methods attempt to predict whether a subsequence is more likely to have been generated by a normal process or an exploit process. Typically, the only data that is available is normal data, so this corresponds to predicting how likely an observed sequence is normal or is consistent with the normal data. One way to do this is to use a "prediction" model. For a sequence of length n, we compute how likely the first $n-1$ system calls predict the nth system call. The more consistent the subsequence is with the normal data, then the more accurate the prediction.

All of the above methods use a fixed window size for building the system call trace models. The size of the window is picked a priori presumably based on some intuitive notion of what size works best for the modeling. There is a tradeoff between using shorter or longer sequences. Let Σ be the set of all system calls. Assuming all sequences occur with equal probability and there are $|\Sigma|$ different system calls, we can assume that we see a specific n length sequence with probability $\frac{1}{|\Sigma|^n}$. In general, if we use longer sequences, we have a lot fewer instances of each subsequence in our data. However, intuitively, these instances are more accurate than short sequences. Shorter sequences occur much more frequently, but often are not as accurate as longer sequences. Motivated by this tradeoff there is some optimal sequence length for the models. In related work, Marceau points out the problems of determining a fixed window size and avoids the problem by presenting a model of using multiple sequence lengths for building these kinds of models [11].

As illustrated in this paper, we can determine the optimal window size using the data. An information theoretic framework for selecting optimal window size using *entropy modeling* is described below. Intuitively, we would like to pick the single window size based on what best fits the data and use that window size for the modeling. We present an analysis of this method and show empirically that the method picks the optimal single window size.

However, we can do better than picking a single window size. This is because the best size of the window depends on the *context*. That is the optimal window size depends specifically on the actual system calls in the subsequence being modeled. For some system call subsequences a longer window is optimal while for others a shorter window is optimal. This context dependency stems from the way a trace is produced by a process. At different points in the process, a different window size is optimal for modeling of system calls. We motivate this assertion using "call graphs" below. In addition, in any system call window, only a certain subset of the system calls in the window are important in the modeling. That is we can ignore some of the system calls. This is equivalent to placing wild cards in the system call subsequence. Again, which system calls we can ignore depends on the context.

In this paper, we present a context dependent window size model for system call data. To model system calls with context dependent window sizes, we use *sparse Markov transducers* (SMTs) [3][1]. SMTs are an extension of probabilistic suffix trees [15, 16, 13]. SMTs use a mixture technique to estimate the optimal choices of context dependent window sizes and placement of wildcards based on the data. SMTs use *sparse prediction trees* which explicitly define the context dependent choices of window sizes and wild card placements. SMTs perform a mixture over possible trees. The mixture allows the estimate of the "best" tree that fits the data corresponding to the "best" choices of window sizes and placement of wild cards. Since there are exponentially many trees, SMTs employ an efficient method for computing the mixture and efficient data structures to keep the model in memory.

We perform experiments using SMTs and traditional methods over real intrusion detection data. We perform experiments over the 1999 DARPA Intrusion Detection Evaluation data created by MIT Lincoln Labs [12] as well as data collected by the University of New Mexico [17]. We compare the method presented in this paper with two traditional methods of intrusion detection, *stide* and *t-stide* [17]. We also compare the context dependent models to prediction models with different fixed window sizes. We show that our method performs significantly better than the traditional methods.

2 Entropy Modeling

We can use an information theoretic framework for choosing the optimal system call window size. In this framework we estimate how well different window sizes

[1]Sparse Markov transducers have been applied in [3] to biological sequence analysis problems in genomics.

will perform by computing the regularity of the data. A detailed description of applying information theory to anomaly detection is presented in [10].

The basic premise for anomaly detection is that there is intrinsic regularity in audit data that is consistent with the normal behavior and distinct from the abnormal behavior. We will show empirically the more regular the data, the better the performance of the anomaly detection algorithm. The process of building an anomaly detection model should therefore involve first measuring the regularity of the data under different models. In our case, we are interested in measuring the regularity under different window sizes. Thus the regularity of the data can help choose the best window size.

To model the system call traces, we use a prediction model whereby the probability of the nth system call is estimated from the previous $n-1$ system calls. The way the prediction model is trained is for each $n-1$ sequence of system calls present in the trace, we keep counts of the following system call. The prediction for a specific system call given a sequence of $n-1$ preceding system calls is simply the count of that system call divided by the total count[2]. Here the length n is the window size of the modeling algorithm. We will measure the regularity of the data for the prediction model under different values of n.

We propose to use an information theoretic measure to measure regularity of data, *conditional entropy*, to help us choose the value of n. The more regular the data, the lower the entropy. Briefly, the definition of conditional entropy is:

$$H(X|Y) = -\sum_{x,y \in X,Y} P(x,y) \log_2 P(x|y) \qquad (1)$$

where $P(x,y)$ is the joint probability of x and y and $P(x|y)$ is the conditional probability of x given y. In the context of system calls we let $X = \Sigma$ be the set of system calls and $Y = \Sigma^{n-1}$ be the set of system call sequences of length $n-1$. We use the notation (x,y) to represent a sequence of length n. Let the set of all sequences in the trace be denoted D. We use the notation $|(x,y)|$ to be the number of times the sequence appears in D and $|D|$ be the total number of sequences in the trace. We can then write the joint probability as $P(x,y) = \frac{|(x,y)|}{|D|}$. The conditional probability $P(x|y)$ is the prediction of our model. Since $P(x,y) = 0$ for a sequence (x,y) that does not occur in the trace and conditional entropy is additive, we can represent the conditional entropy for a window size n as:

$$\begin{aligned} H_n(X|Y) &= -\sum_{x\in\Sigma, y\in\Sigma^{n-1}} P(x,y)\log_2 P(x|y) \qquad (2) \\ &= -\sum_{(x,y)\in D} \frac{1}{|D|} log_2 P(x|y) \end{aligned}$$

[2]In order to avoid zero probabilities, we add a small initial count to each system call.

Equation (2) can be computed efficiently.

We computed the conditional entropy values ($H_n(X|Y)$) for each of the system call data sets presented later in this paper using different window sizes (values of n) and plotted them in Figure 1. The conditional entropy was computed with cross validation. The data was split into two portions. The prediction models was trained on the first portion of the data and the entropy was computed over the second portion. Then the first and second portions are reversed and we again compute the entropy. Since entropy is additive, the total entropy is the sum of the entropy on each portion. If we compare the entropy of a window size and the performance of the model with that window size we notice that in general the lower the entropy, the better the performance of the model as shown in Figure 5. The specific method for evaluating the models and details about the data set are given in section 5. In order to pick the optimal window size we can use equation (2) to compute the entropy under different window sizes and pick the window size with the minimum entropy.

3 Program Call Graphs

The motivation for a context dependency of the window size stems from the underlying mechanism of how a process executes. A system call *trace* is a sequence of all of the system calls that a process of a given program makes during its lifetime. The system calls in the trace depend on the execution path of the process. A process execution path depends on many factors such as inputs to the process as well as the state of the system. These factors determine which execution path a process takes at each possible branch point.

We can model the set of all possible execution paths of a program using a "call graph". The call graph models the program structure and defines the possible execution paths. A call graph is a graph where each path through the graph is a possible path of execution of a process of a program. The nodes of the graph correspond to possible branch points of the program and the edges of the graph are labeled with the system calls between the branch points. There is a defined start node for the graph and at least one end node. We can view an execution path of a process as being a path through the call graph associated with a program. A system call trace is simply the system calls along the edges of the execution path of the process.

An example of a call graph and an execution path is shown in Figure 2. Although a call graph exists for every program, it is very difficult to obtain this graph. The graph depends on the source code of the program as well as the compiler used and the specifics of the operating system. Even with the source code available, it is impractical to assume that we can recreate the call graph from the observed system call traces. Although it is impossible to determine

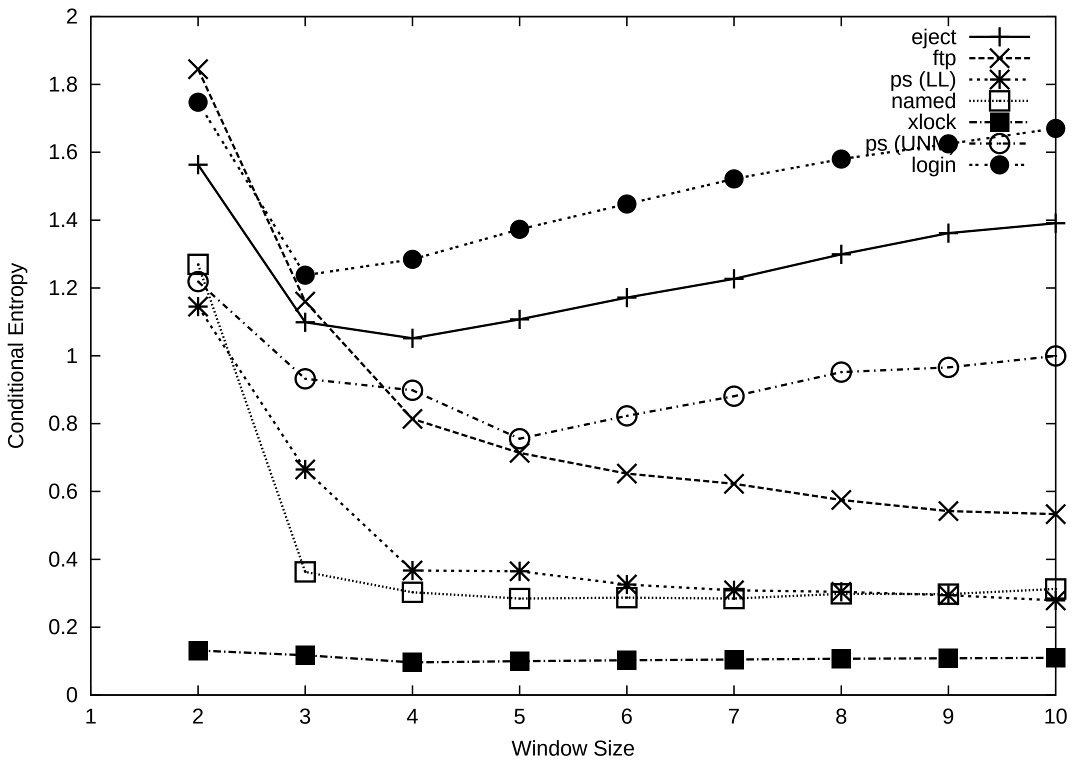


Figure 1. Conditional Entropy under different window sizes for different data sets. Notice that most of the entropy curve has a minimum. This is the optimal window size. Some of the curves do not have a minimum (ftp and ps (LL)) because the data for those processes is extremely regular.

the specific call graph for a program, we can reasonably assume that a call graph exists for each program and use this to motivate the context dependency of the optimal window size.

Applying the call graph framework to intrusion detection, there are a set of execution paths that correspond to exploits. The goal of the system call modeling method is to be able to determine whether a short subsequence of system calls corresponds to a normal execution path or an exploit execution path. If we had access to the programs call graph and it was labeled with normal and exploit paths, we could match the subsequence to where it uniquely occurs in the call graph. Notice that the tradeoff between longer and shorter sequences is made explicit in the context of the call graph. A longer sequence can more likely identify a unique portion of the call graph, however, it is often too long to fit within a single edge and must span some branch points. For this sequence to be observed multiple times, the states of the different processes where the longer sequence occurs will all have to force the execution paths to be the same with regard to those branches. This can introduce noise into our model. Shorter sequences on the other hand, span fewer branches. However, these shorter sequences can occur in multiple points in the call graph causing it to be difficult to determine uniquely where the short subsequence came from and whether the short subsequence corresponds to an exploit trace or a normal trace.

Ideally, for any given subsequence, we would like to take the shortest subsequence that uniquely (or almost uniquely) identifies the location of the call graph that generated this subsequence. Because the branch points occur in different places, the optimal length of the subsequence depends on the specific system calls in the subsequence. Hence, the optimal window size is context dependent.

Another common feature of call graphs is that they often have a branch which affects a single system call. An example of a branch in a call graph is shown in Figure 3. In this portion of the call graph, there are two possible traces through it, “ioctl mmap open mmap unlink” and “ioctl mmap close mmap unlink”. Because there are two possibilities, the amount of observed system call traces from this portion of the call graph are split into two parts. As discussed in the context of longer sequences, this is not optimal for modeling. Intuitively, we want to be able to group these two sequences into a single category. We can do this by including wild cards in the subsequence. Both of the sub-

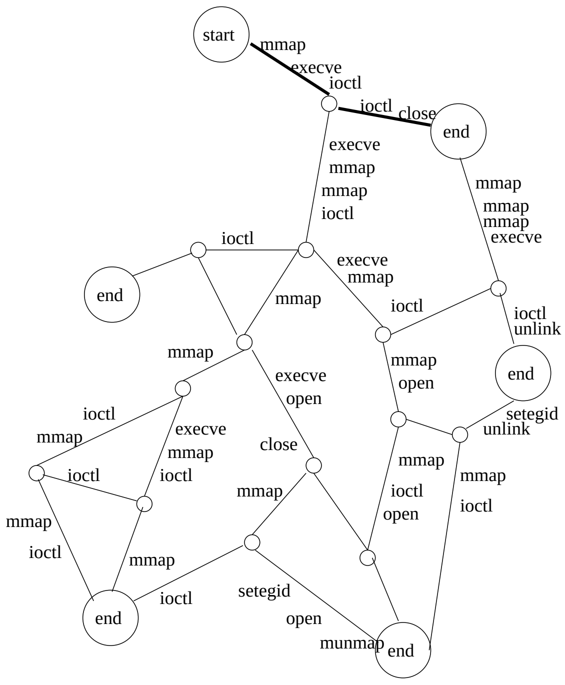


Figure 2. A sample call graph and execution trace. Note the execution path is marked on the graph in bold. The system call trace of the execution path is the set of system calls along the edges of the graph and in this case: $mmap\ execve\ ioctl\ ioctl\ close$.

sequences can fit into the model of "ioctl mmap * mmap unlink". Again, the placements of the wildcards are context dependent relating to the call graph. This motivates the incorporation of context dependent wild cards into the model.

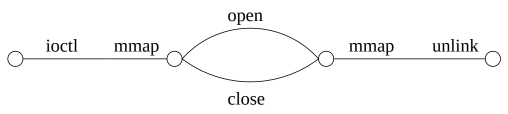


Figure 3. A portion of a call graph corresponding to a single system call branch. Note the system call subsequence corresponding to this call graph is "ioctl mmap * mmap unlink".

4 Sparse Markov Transducers

In order to determine whether a system call trace subsequence corresponds to an exploit or normal trace, we build a probabilistic prediction model which predicts the last (nth) system call given the previous ($n-1$) system calls in the subsequence. In this model, this can be represented as a probability estimate of the last system call conditional on the sequence of previous system calls. The size of the window and the placement of the wild cards correspond to the length of the conditioning sequence and the specific positions in the conditioning sequence on which the probability is conditioned. To model this type of probability distribution, we use sparse Markov transducers.

The prediction model is equivalent to computing the probability

$$P(X_n|X_{n-1}X_{n-2}X_{n-3}X_{n-4}...X_1) \tag{3}$$

where X_k are random variables over the set of system calls Σ. In this probability distribution the last system call X_n is conditional on the $n-1$ previous system calls.

As motivated by call graphs, the probability distribution is conditioned on some of the system calls and not others. The different window sizes correspond to different lengths of the conditional sequence. For example, if the optimal window size for a given context is $n = 4$, then the probability distribution would be conditional only on the first 3 system calls in the sequence ($X_3X_2X_1$). Depending on the specific system calls in the sequence, there is a different value of n.

Also dependent on the context is the placement of wild cards. For any given sequence, the conditioning sequence contains wild cards. For example, if the optimal window size for a given context is $n = 5$ with the third system call being a wild card, the conditioning sequence will be $X_4 * X_2X_1$ where the symbol $*$ represents a wild card. We use the notation $*^n$ to represent n consecutive wild cards.

SMTs are used to model system call traces by estimating a context dependent "predictive" probability as motivated by the call graph framework. This is the probability of predicting the final system call in a subsequence given the previous subsequences. This probability estimation takes into account the context dependent nature of the data. Once we train this model over normal data from a given program, we then have a predictive probability distribution for that program. When evaluating new program traces to determine whether or not they correspond to exploits, we compute the predictive probability for each subsequence. If the subsequence probability is below some threshold, then we know that the subsequence trace is very unlikely to have originated from a normal process and we declare the process trace an exploit. The value of the threshold defines the tradeoff between the detection rate and the false positive rate of the system. Higher thresholds will mean more traces will be reported as exploits meaning a higher detection rate at a potentially higher false positive rate. Likewise lower thresholds will mean fewer traces will be reported as exploits which gives a lower false positive rate at a potentially lower detection rate. Because of this tradeoff, we evaluate the system under many different thresholds as described in more detail in section 5.

In brief our approach is as follows. We define a type of prediction suffix tree called a *sparse prediction tree* which is representationally equivalent to sparse Markov transducers. These trees probabilistically map input strings to a probability distribution over the output symbols. The topology of a tree encodes the context dependent length and positions of the wild-cards in the conditioning sequence of the probability distribution. We estimate the probability distributions of these trees from the set of examples. Since *a priori* we do not know the optimal window sizes or positions of the wild-cards, we do not know the best tree topology. For this reason, we use a mixture (weighted sum) of trees and update the weights of the tree weights based on their performance over the set of examples. We update the trees so that the better performing trees get larger weights while the worse performing trees get smaller weights. Thus the data is used to choose the positions of the wild-cards in the conditioning sequences. We use an efficient algorithm for updating the mixture weights and estimating the sparse Markov transducer presented in [3]. The algorithm computes the *exact* mixture weights for an exponential number of trees in a highly efficient manner. Efficient data structures are used to represent common subtrees exactly once, and hence the mixture can be computed in time proportional to the number of sequences (not proportional to an exponential number of trees).

4.1 Sparse Markov Trees

To model sparse Markov transducers, we use a type of prediction suffix tree called a sparse prediction tree. A sparse prediction tree is a rooted tree where each node is either a leaf node or contains one branch labeled with $*^n$ for $n \geq 0$ that forks into a branch for each element in Σ (each system call). Each leaf node of the tree is associated with a probability distribution over the system calls, Σ. Figure 4 shows a sparse Markov tree. In this tree, leaf nodes, $1, ...7$, each are associated with a probability distribution. The path from the root node to a leaf node represents the conditioning sequence in the probability distribution. We label each node using the path from the root of the tree to the node. Because the path contains the wild-card symbol $*$, there are multiple strings over Σ that are mapped to a single node. A tree associates a probability distribution over output symbols conditioned on the input sequence by following an input sequence from the root node to a leaf node skipping a symbol in the input sequence for each $*$ along the path. The probability distribution conditioned on an system call sequence is the probability distribution associated with the leaf node that corresponds to the system call sequence. The length of the conditioning sequence corresponds to the depth of a leaf node in the tree. Notice that the conditioning sequence length is different for different paths in the tree. Also, the wild cards are in different places for different paths in the tree. Thus a tree makes explicit the choices of context dependent length and placement of wild cards. As described later, the tree is trained with a training set of system call length $n-1$ subsequences $x_{n-1}x_{n-2}...x_1$ and their corresponding nth system call x_n.

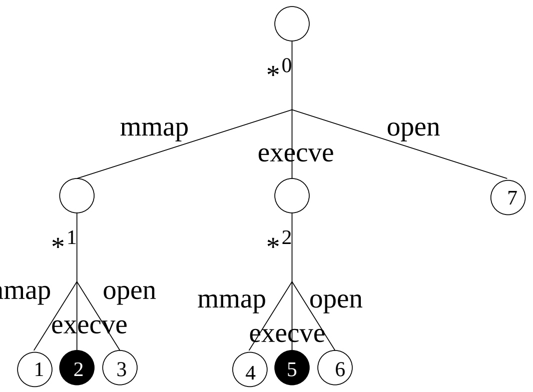


Figure 4. A sparse Markov tree.

For example, in Figure 4 the sets of input strings that correspond to each of the two highlighted nodes are $mmap * execve$ which corresponds to node 2 and $execve *^2 execve$ which corresponds to node 5. In our application the two nodes would correspond to any system call sequences $mmap * execve$ and $execve * * execve$ where $*$ denotes a wild-card. The node labeled 2 in the figure corresponds to many sequences including **mmap** $execve$ **execve** and **mmap** $mmap$ **execve**. Similarly for the node labeled 5 in the figure corresponds to the sequences **execve** $mmap\ mmap$ **execve** and **execve** $mmap\ open$ **execve**. Also **execve** $mmap\ open$ **execve** $mmap$ corresponds to node 5 because the prefix of the sequence corresponds to node 5. The probability corresponding to an input sequence is the probability contained in the leaf node corresponding to the sequence. In this example $P(open|mmap\ execve\ execve)$ would be the probability of the symbol $open$ in the probability distribution associated with node 2.

A sparse prediction tree, T, can be used to compute a context dependent predictive probability for system call windows. For a training example pair containing a system call x_n and an input sequence $x_{n-1}x_{n-2}...x_1$, we can determine the conditional probability for the example, denoted $P_T(x_n|x_{n-1}x_{n-2}...x_1)$. As described above, we first determine the node u which corresponds to the conditioning sequence $(x_{n-1}x_{n-2}...x_1)$. Once that node is determined, we use the probability distribution over output symbols associated with that node. The prediction of the tree for the example is then:

$$P_T(x_n|x_{n-1}x_{n-2}...x_1) = P_T(x_n|u) \tag{4}$$

4.2 Training a Prediction Tree

A prediction tree is trained from a set of training examples of system calls trace subsequences. The conditioning sequences are the first $n-1$ sequences and the prediction is the nth subsequence.

Each leaf node keeps counts of each system call that reaches the leaf node. We smooth each count by adding a constant value to the count of each output symbol. The predictor's estimate of the probability for a given output is the smoothed count for the output divided by the total count in the predictor.

For example, consider the prediction tree in Figure 4. We first initialize all of the predictors (in leaf nodes $1, ..., 7$) to the initial count values. If for example, the first element of training data is the system call $mmap$ (x_n) preceded by the sequence $mmap\ open\ execve$ ($x_{n-1}x_{n-2}, ...$), we would first identify the leaf node that corresponds to the sequence. In this case the leaf node would be node 2. We then update the predictor in node 2 with the system call $mmap$ by adding 1 to the count of $mmap$ in node 2. Similarly, if the next nth system call is $execve$ and corresponding preceding sequence is $mmap\ execve\ execve\ mmap$, we would update the predictor in node 2 with the system call $execve$. If the next system call is $open$ and corresponding sequence is $mmap\ mmap\ mmap\ mmap\ execve$, we would update node 1 with the system call $open$.

After training on these three examples, we can use the tree to output a prediction for a sequence by using the probability distribution of the node corresponding to the sequence. For example, assuming the initial count is 0, the prediction of the the input sequence $mmap\ mmap\ execve$ which correspond to the node 2 and would give the probability for $execve$ as .5 and the probability of $mmap$ as .5. The probability of $execve$ (.5) is the count (1) of $execve$ in the node divided by the total count (2) in the node. Similarly, the probability of $mmap$ (.5) is the count (1) of $mmap$ divided by the total count (2).

4.3 Mixture of Sparse Prediction Trees

In general, we do not know the optimal size of the window or where to put the wild-cards. Thus we do not know which tree topology can best estimate the distribution. Intuitively, we want to use the training data in order to learn which tree predicts most accurately.

We use a Bayesian mixture approach for the problem. Instead of using a single tree as a predictor, we use a mixture technique which employs a weighted sum of trees as our predictor. We then use a Bayesian update procedure to update the weight of each tree based on its performance on each element of the dataset. In this way, the weighted sum uses the data to make the best prediction.

Our algorithm is as follows. We initialize the weights in the mixture to the prior probabilities of the trees. Then we update the weight of each tree for each training example in the training set based on how well the tree performed on predicting the last symbol in the window. At the end of this process, we have a weighted sum of trees in which the best performing trees in the set of all trees have the highest weights.

Specifically, we assign a weight, w_T^t, to each tree in the mixture after processing training example t (denoted with superscript t). The prediction of the mixture after training example t is the weighted sum of all the predictions of the trees divided by the sum of all weights:

$$P^t(x_n^t|x_{n-1}^t...x_1^t) = \frac{\sum_T w_T^t P_T(x_n^t|x_{n-1}^t...x_1^t)}{\sum_T w_T^t} \tag{5}$$

where $P_T(x_n^t|x_{n-1}^t...x_1^t)$ is the prediction of tree T for sequence $x_n^t x_{n-1}^t x_{n-2}^t...x_1^t$.

One way to define the prior probability of a tree w_T^1, is using the topology of the tree. Intuitively, the more complicated the topology of the tree the smaller its prior probability.

4.4 General Update Algorithm

We use a Bayesian update rule to update the weights of the mixture for each training example. The mixture weights are updated according to the evidence which is simply the probability of the final system call x_n^t given the input sequence $x_{n-1}^t x_{n-2}^t...x_1^t$, $P_T(x_n^t|x_{n-1}^t x_{n-2}^t...x_1^t)$. The prediction is obtained by updating the tree with the training example and then computing the prediction of the training example. Intuitively, this gives a measure of how well the tree performed on the given example. The unnormalized mixture weights are updated using the following rule:

$$w_T^{t+1} = w_T^t P_T(x_n^t|x_{n-1}^t x_{n-2}^t...x_1^t) \tag{6}$$

with w_T^1 is defined to be the prior weight of the tree. Thus the weigh of a tree is the prior weight times the evidence for each training example:

$$w_T^{t+1} = w_T^1 \prod_{i=1}^{t} P_T(x_n^t|x_{n-1}^t x_{n-2}^t...x_1^t) \tag{7}$$

After training example t we update the weights for every tree T. Since the number of possible trees are exponential in terms of the maximum allowed tree depth, this update algorithm requires exponential time.

However, SMTs can be computed efficiently in both time and space. An efficient update algorithm that computes the exact mixture weights is presented in [3]. The efficient algorithm stores and updates weights in the nodes of the tree and uses those weights to compute the mixture of sparse Markov trees. The algorithm for node weight updates does not require exponential time.

5 Experiments over Audit Data

We applied SMTs to detect intrusions based on the analysis of process system calls and compared them to baseline methods. We examined two sets of system call data containing intrusions. The arguments to the system calls are ignored for the analysis. In both of these sets, there was a set of clean traces and a set of intrusion traces.

The first set of data is from the BSM (Basic Security Module) data portion of the 1999 DARPA Intrusion Detection Evaluation data created by MIT Lincoln Labs [12]. The data consists of 5 weeks of BSM data of all processes run on a Solaris machine. We examined three weeks of traces of the programs which were attacked during that time. The programs attacked were: *eject*, *ps*, and *ftp*.

The second set of data was obtained from Stephanie Forrest's group at the University of New Mexico. This data set is described in detail in Warrender et al. [17]. This data contains up to 15 months of normal traces for certain programs as well as intrusion traces. The data provides normal and intrusion traces of system calls for several processes. We examine the data for the processes that were attacked with a "user to root" attack. The processes examined correspond to the programs: *named*, *xlock*, *login*, and *ps*.

Tables 1 and 2 summarize the data sets and list the number of system calls and traces for each program. Traces from each program in each data set were separated into a disjoint training and testing portion. The training set contained approximately $2/3$ of the traces and the test set contained the remaining traces. We train and test on different sets of data in order to simulate how the method may work in practice, i.e. testing a model against data that has not been observed when building the model.

5.1 Baseline Comparison Methods.

We compare our method against two methods, *stide* and *t-stide*, shown to be effective in detecting intrusions in system call data when trained over clean data in experiments performed on the University of New Mexico data set [17]. We also compare our context based method to fixed window size prediction models of different sizes.

The sequence time-delay embedding (stide) algorithm keeps track of what sequences were seen in the training data and detects sequences not seen in training. The method builds a model of normal data by making a pass through the training data and storing each unique contiguous sequence of a predetermined length in an efficient manner. We used a length of six because that is the length of the sequences used in the published results of the method.

When the method is used to detect intrusions, the sequences from the test set are compared to the sequences in the model. If a sequence is not found in the normal model, it is called a *mismatch* or anomaly.

The threshold sequence time-delay embedding (t-stide) algorithm is an extension of the stide algorithm which incorporates a threshold. In addition to unknown sequences, rare sequences are also counted as mismatches. In this method, any sequence accounting for less than 0.001% of the total number of sequences is considered rare.

To detect intrusions, these methods compare the number of mismatches in a local region of 20 consecutive sequences. A threshold is set for these local regions between 1 and 20. If the number of mismatches reaches or exceeds the local mismatch threshold, the process is declared an intrusion.

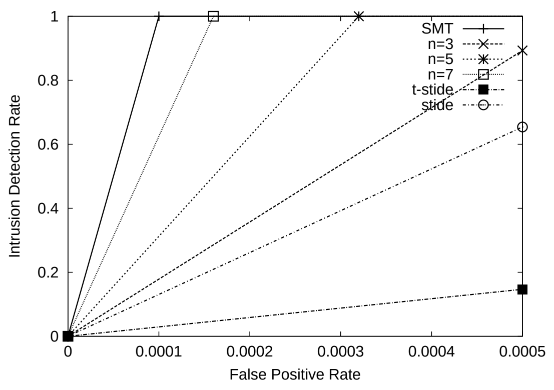
5.2 Experimental Results

We compare the performance of the method presented in this paper with the baseline methods described above. We empirically show that the methods presented in this paper outperform the baseline methods when trained over the same dataset.

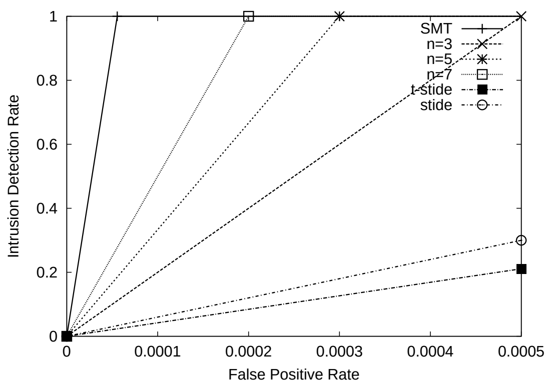
If a process trace contains an anomaly, we declare that process an intrusion. We consider an intrusion detected if either the intrusion process is detected, or one of the processes spawned by the intrusion is detected.

We compare the anomaly detection methods in both sets of experiments using ROC curves which graph the false positive rate versus the detection rate [14]. The detection rate is the percentage of intrusions which are detected. In order to be consistent with previous published results on these data sets, the false positive rate is defined to be the percentage of normal system calls which are declared anomalous [17]. The threshold of the methods is varied to obtain multiple points on the ROC curve. The ROC curves have few points because of the small amount of intrusion traces in each data set. In the ROC curves, the optimal detector is the graph closest to the y-axis, i.e. having the highest detection rate with minimum false positive rate.

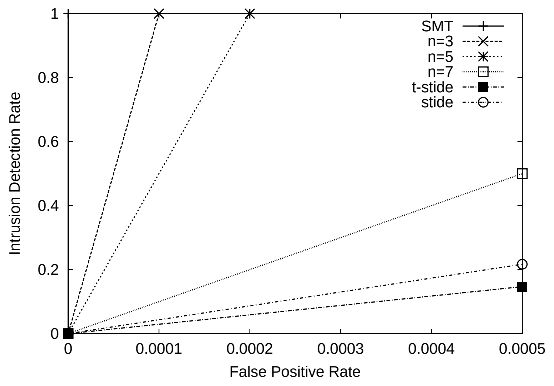
Examining Figure 5, we notice that different prediction models have different levels of performance. We notice that the optimal window size is different for each process. This is consistent with the entropy modeling experiments. Also note that in most cases the fixed window size methods are outperformed by the context dependent window size methods as expected.



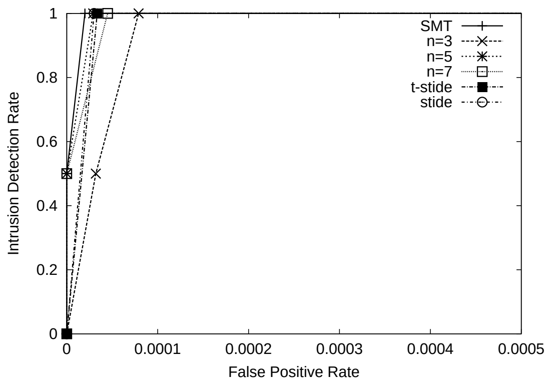
(a)



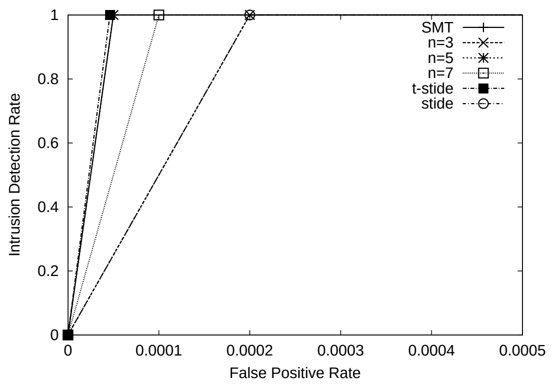
(b)



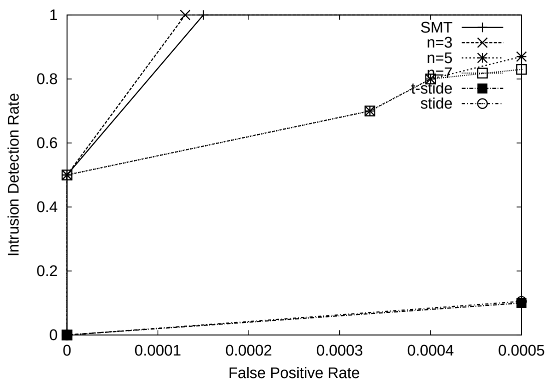
(c)



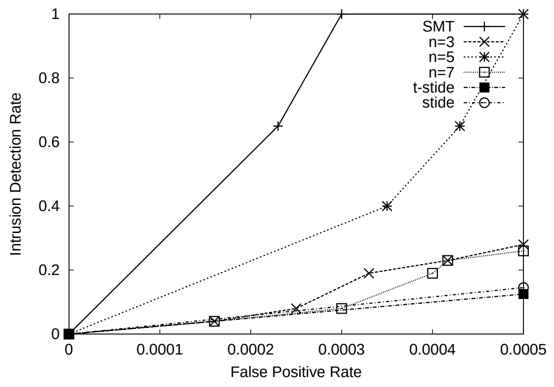
(d)



(e)



(f)



(g)

Figure 5. ROC curves showing the comparison of SMTs, fixed window size prediction models, stide and t-stide trained over the same data. The fixed window sizes used a window size of length 3, 5, and 7. The curves are obtained by varying the detection threshold. Notice that in general the best performing fixed window size corresponds to the minimum of entropy curve (Figure 1). The curves show the comparison trained over a different set of data: (a) ftpd, (b) ps (LL), (c) eject, (d) xlock, (e) named, (f) login, and (g) ps (UNM).

Table 1. Lincoln Labs Data Summary

Program Name	# Intrusion Traces	# Intrusion System Calls	# Normal Traces	# Normal System Calls	% Intrusion Traces
ftpd	1	350	943	66842	0.05%
ps (LL)	21	996	208	35092	2.7%
eject	6	726	7	1278	36.3%

Table 2. University of New Mexico Data Summary

Program Name	# Intrusion Traces	# Intrusion System Calls	# Normal Traces	# Normal System Calls	% Intrusion Traces
xlock	2	949	72	16,937,816	0.006%
named	2	1,800	27	9,230,572	0.01%
login	9	4,875	12	8,894	35.4%
ps (UNM)	26	4,505	24	6,144	42.3%

In general, the methods presented in this paper outperform *t-stide* and *stide*. The main difference between our methods and t-stide and stide is the threshold. Our methods use a probabilistic threshold while the other methods use the number of consecutive mismatches. Empirically, the probabilistic threshold outperforms the number of mismatch threshold even when the window size is the same as shown in Figure 5.

6 Conclusion

We have shown that system call modeling methods can be improved by using dynamic window sizes. We have presented two methods for using window sizes estimated from the data. The first is using *entropy modeling* to determine the optimal window size. We have shown empirically, that the method can pick the optimal window size by measuring the regularity in the data.

The second method takes advantage of the context dependency of the optimal window size. We have presented a new method for modeling system call traces using sparse Markov transducers. This method takes advantage of the context dependency of the optimal window size. The method estimates the best window size depending on the specific system calls in the subsequence based on their performance over the training data. We have shown that this method outperforms traditional methods in modeling system call data.

Future work involves moving beyond sliding windows for modeling system call traces. Intuitively, we can better model system calls by looking at the underlying call graph. A direction for future work would be to explicitly attempt to construct a call graph from a set of traces and use this call graph to attempt to improve the predictions. In this way we may be able to incorporate more structural information into the model to boost performance.

Acknowledgments

This work was supported under DARPA F30602-00-1-0603.

References

[1] D. Denning. An intrusion detection model. *IEEE Transactions on Software Engineering*, SE-13:222–232, 1987.

[2] E. Eskin. Anomaly detection over noisy data using learned probability distributions. In *Proceedings of ICML 2000*, Menlo Park, CA, 2000. AAAI Press.

[3] E. Eskin, W. N. Grundy, and Y. Singer. Protein family classification using sparse markov transducers. In *Proceedings of the Eighth International Conference on Intelligent Systems for Molecular Biology*, Menlo Park, CA, 2000. AAAI Press.

[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. IEEE Computer Society, 1996.

[5] A. Ghosh and A. Schwartzbard. A study in using neural networks for anomaly and misuse detection. In *Proceedings of the Eighth USENIX Security Symposium*, 1999.

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

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

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

[9] W. Lee, S. J. Stolfo, and P. K. Chan. Learning patterns from unix processes execution traces for intrusion detection. In *In Proceedings of the AAAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, pages 50–56. Menlo Park, CA: AAAI Press, 1997.

[10] W. Lee and D. Xiang. Information-theoretic measures for anomaly detection. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[11] C. Marceau. Characterizing the behavior of a program using multiple-length n-grams. In *Proceedings of the New Security Paradigms Workshop 2000*, 2000.

[12] MIT Lincoln Labs. 1999 DARPA intrusion detection evaluation. In *http://www.ll.mit.edu/IST/ideval/index.html*, 1999.

[13] F. Pereira and Y. Singer. An efficient extension to mixture techniques for prediction and decision trees. *Machine Learning*, 36(3):183–199, 1999.

[14] F. Provost, T. Fawcett, and R. Kohavi. The case against accuracy estimation for comparing induction algorithms. In *Proceedings of the Fifteenth International Conference on Machine Learning*, July 1998.

[15] D. Ron, Y. Singer, and N. Tishby. The power of amnesia: Learning probabilistic automata with variable memory length. *Machine Learning*, 25:117–150, 1996.

[16] Y. Singer. Adaptive mixtures of probabilistic transducers. *Neural Computation*, 9(8):1711–1734, 1997.

[17] C. Warrender, S. Forrest, and B. Pearlmutter. Detecting intrusions using system calls: alternative data models. In *Proceedings of the 1999 IEEE Symposium on Security and Privacy*, pages 133–145. IEEE Computer Society, 1999.

[18] N. Ye. A markov chain model of temporal behavior for anomaly detection,. In *Proceedings of the 2000 IEEE Systems, Man, and Cybernetics Information Assurance and Security Workshop*, 2000.