TECHNICAL RESEARCH REPORT

Intrusion Detection with Support Vector Machines and Generative Models

by John S. Baras, Maben Rabi

TR 2002-22

Intrusion Detection with Support Vector Machines and Generative Models

John S. Baras and Maben Rabi

Institute for Systems Research and
Department of Electrical and Computer Engineering
University of Maryland, College Park MD 20742, USA.
baras,rabi@isr.umd.edu

Abstract. This paper addresses the task of detecting intrusions in the form of malicious attacks on programs running on a host computer system by inspecting the trace of system calls made by these programs. We use 'attack-tree' type generative models for such intrusions to select features that are used by a Support Vector Machine Classifier. Our approach combines the ability of an HMM generative model to handle variable-length strings, i.e. the traces, and the non-asymptotic nature of Support Vector Machines that permits them to work well with small training sets.

1 Introduction

This article concerns the task of monitoring programs and processes running on computer systems to detect break-ins or misuse. For example, programs like `sendmail` and `finger` on the UNIX operating system run with administrative privileges and are susceptible to misuse because of design short-comings. Any user can pas specially crafted inputs to these programs and effect *'Buffer-overflow'* (or some such exploit) and break into the system. To detect such attacks, the execution of vulnerable programs should be screened at run-time. This can be done by observing the trace (sequence of operating system calls; with or without argument values) of the program. In [3], S. Hofmeyr et.al. describe a method of learning to discriminate between sequences of system calls (without argument values) generated by normal use and misuse of processes that run with (root) privileges. In their scheme, a trace is flagged to be anomalous if its similarity to example (training) traces annotated as normal falls below a threshold; the similarity measure is based on the extent of partial matches with short sequences derived from the training traces. From annotated examples of traces, they compile a list of subsequences for comparing (at various positions) with a given trace and flag anomalous behavior when a similarity measure crosses a threshold. In [15], A. Wespi et. al. use the Teiresias pattern matching algorithm on the traces in a similar manner to flag off anomalous behavior. In both of the above, the set of subsequences used for comparison has to be learnt from the annotated set of traces (sequences of system calls) because, no other usable information or formal specification on legal or compromised execution of programs is available.

The approach advocated in this article is to obtain a compact representation of program behavior and use it (after some reduction) to select features to be used with a Support Vector Machine learning classifier.

Let $\mathscr{Y}$ be the set of all possible system calls. A trace $\mathcal{Y}$ is then an element of $\mathscr{Y}^*$ which is the set of all strings composed of elements of $\mathscr{Y}$. For a given program, let the training set be $\mathscr{T} = \{(\mathcal{Y}_i, L_i) | i = 1, \ldots, T\}$, where $L_i \in \{0, 1\}$, is the label corresponding to trace $\mathcal{Y}_i$, 0 for normal traces and 1 for attack traces. The detection problem then is to come up with a rule $\hat{L}$, based on the training set, that attempts to minimize the probability of misclassification $P_e = Pr[\hat{L}(\mathcal{Y}) \neq L(\mathcal{Y})]$. What is of more interest to system administrators is the trade-off between the probability of detection $P_D = Pr[\hat{L}(\mathcal{Y}) = 1 | L(\mathcal{Y}) = 1]$ and the probability of false alarms $P_{FA} = Pr[\hat{L}(\mathcal{Y}) = 1 | L(\mathcal{Y}) = 0]$ that the classifier provides. These probabilities are independent of the probabilities of occurrence of normal and malicious traces.

Annotation (usually manual) of live traces is a difficult and slow procedure. Attacks are also rare occurrences. Hence, traces corresponding to attacks are few in number. Likewise, we dont even have a good representative sample of traces corresponding to normal use. Hence, regardless of the features used, we need to use non-parametric classifiers that can handle finite (small) training sets. Support Vector Machine learning carves out a decision rule reflecting the complicated statistical relationships amongst features from finite training sets by maximizing true generalization (strictly speaking, a bound on generalization) instead of just the performance on the training set. To use Support Vector Machines, we need to map each variable length trace into a (real-vector-valued) feature space where Kernel functions (section 4) can be used. This conversion is performed by parsing the raw traces into shorter strings and extracting models of program execution from them.

2 MODELS FOR ATTACKS

The malicious nature of a program is due to the presence of a subsequence, not necessarily contiguous, in its trace of system calls. For the same type of attack on the host, there are several different combinations of system calls that can be used. Furthermore, innocuous system calls or sequences can be injected into various stages of program execution (various segments of the traces). Thus the intrinsic variety of attack sequences and the padding with harmless calls leads to a polymorphism of traces for the same plan of attack. Real attacks have a finite (and not too long) underlying *attack* sequence of system calls because they target specific vulnerabilities of the host. This and the padding are represented in a 'plan of attack' called the *Attack Tree* [12].

2.1 Attack Trees

An Attack Tree ($\mathscr{A}$)[12] is a directed acyclic graph (DAG) with a set of nodes and associated sets of system calls used at these nodes. It represents a hierarchy

of pairs of tasks and methods to fulfill those tasks. These nodes and sets of system calls are of the following three types:

1. $\mathscr{V} = \{v_1, v_2, \ldots, v_{k_1}\}$, the nodes representing the targeting of specific vulnerabilities in the host system, and a corresponding collection of subsets of $\mathscr{Y}$: $\mathscr{Y}^{\mathscr{V}} = \{\mathscr{Y}_1^v, \mathscr{Y}_2^v, \ldots, \mathscr{Y}_{k_1}^v\}$ representing the possible system-calls that target those vulnerabilities.
2. $\mathscr{P} = \{\wp_1, \wp_2, \ldots, \wp_{k_2}\}$, the set of instances where padding can be done along with a corresponding collection of subsets of $\mathscr{Y} \cup \{\epsilon\}$ (ϵ is the null alphabet signifying that no padding system-call has been included): $\mathscr{Y}^{\mathscr{P}} = \{\mathscr{Y}_1^{\wp}, \mathscr{Y}_2^{\wp}, \ldots, \mathscr{Y}_{k_2}^{\wp}\}$.
3. $\mathscr{F} = \{f_1, f_2, \ldots, f_{k_3}\}$, the final states into which the scheme jumps after completion of the attack plan along with a collection of subsets of $\mathscr{Y} \cup \{\epsilon\}$: $\mathscr{Y}^{\mathscr{F}} = \{\mathscr{Y}_1^f, \mathscr{Y}_2^f, \ldots, \mathscr{Y}_{k_3}^f\}$; a set that is not of much interest from the point of view of detecting attacks.

There may be multiple system calls issued while at a state with possible restrictions on the sequence of issue. The basic attack scheme encoded in the Attack Tree is not changed by modifications such as altering the padding scheme or the amount of padding (time spent in the padding nodes). Given an attack tree, it is straightforward to find the list ($\mathscr{L}^{\mathscr{A}} \subset \mathscr{Y}^*$) of all traces that it can generate. But given a trace, we don't have a scheme to check if it could have been generated by $\mathscr{A}$ without searching through the list $\mathscr{L}^{\mathscr{A}}$. Our intrusion detection scheme needs to execute the following steps:

1. Learn about $\mathscr{A}$ from the training set $\mathscr{T}$.
2. Form a rule to determine the likelihood of a given trace being generated by $\mathscr{A}$.

These objectives can be met by a probabilistic modeling of the Attack Tree.

2.2 Hidden Markov Models for Attack Trees

Given an Attack Tree $\mathscr{A}$, we can set up an equivalent Hidden Markov model H^1 that captures the uncertainties in padding and the polymorphism of attacks. The state-space of H^1, $\mathscr{X}^1 = \{x_1^1, x_2^1, \ldots, x_n^1\}$ (the superscript 1 corresponding to the attack model (abnormal or malicious program) and the superscript 0 corresponding to the normal program model) is actually the union: $\{x_n^1\} \cup \mathscr{V} \cup \mathscr{P} \cup \mathscr{F}$ with x_n^1 being the start state representing the start node with no attack initiated and $n = 1 + k_1 + k_2 + k_3$. We now need to describe the statistics of state transitions (with time replacing the position index along a trace) to reflect the edge structure of $\mathscr{A}$ and to also reflect the duration of stay in the vulnerability and padding nodes. The only allowed state transitions are the ones already in $\mathscr{A}$ and self-loops at each of the states. The picture is completed by defining conditional output probabilities given the state of system calls in a way that captures the information in $\mathscr{Y}^{\mathscr{V}}$ and $\mathscr{Y}^{\mathscr{P}}$. Thus we have, $\forall\, x_i^1, x_j^1 \in \mathscr{X}^1, \forall\, y_l \in \mathscr{Y} \cup \{\epsilon\}$ and

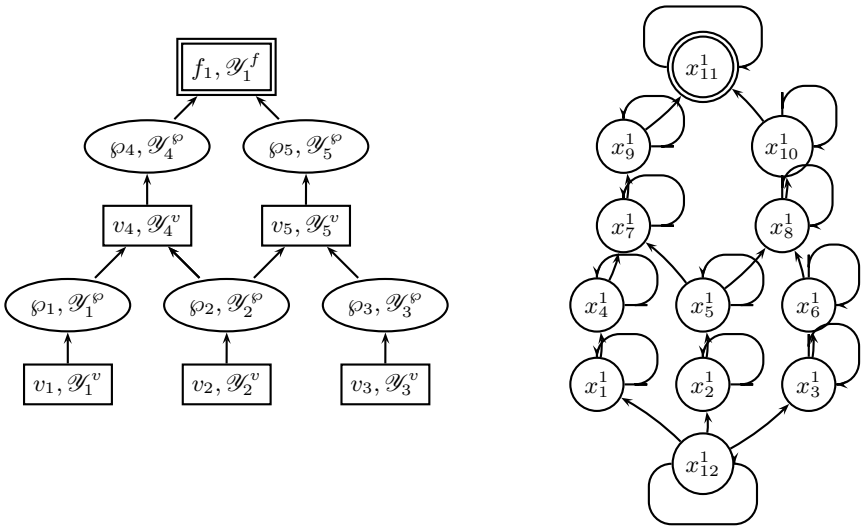

Fig. 1. An Attack Tree and its equivalent HMM with $k_1 = 5, k_2 = 5, k_3 = 1, n = 12$.

$\forall\, t \in \mathbb{N}$

$$P[X(t+1) = x_i^1 | X(t) = x_j^1] = q_{ji}^1, \tag{1}$$

$$P[Y(t+1) = y_l | X(t) = x_j^1] = r_{jl}^1. \tag{2}$$

We can write down a similar HMM for the normal traces also. This *normal* HMM, H^0 has as its state-space a set $\mathscr{X}^0$ in general bigger than $\mathscr{X}^1$, and certainly with a different state transition structure and conditional output probabilities of system calls given the state. The associated probabilities are as follows. $\forall\, x_i^0, x_j^0 \in \mathscr{X}^0, \forall\, y_l \in \mathscr{Y}$ and $\forall\, t \in \mathbb{N}$

$$P[X(t+1) = x_i^0 | X(t) = x_j^0] = q_{ji}^0, \tag{3}$$

$$P[Y(t+1) = y_l | X(t) = x_j^0] = r_{jl}^0. \tag{4}$$

We would like to represent the probabilities for the above HMMs as functions of some vector θ of real-valued parameters so as to be able to use the framework of [4] and [5]. In the next section, we use these parametric HMMs to derive a real valued feature vector of fixed dimension for these variable length strings that will enable us to use Support Vector Machines for classification.

3 REAL VALUED FEATURE VECTORS FROM TRACES

Since we are dealing with variable length strings, we would like to extract the features living in a subset of an Euclidean space on which kernel functions are

readily available enabling use of Support Vector Machines[14][1]. In [4] and [5], each observation $\mathcal{Y}$ is either the output of a parametric HMM (Correct Hypothesis H_1) or not (Null Hypothesis H_0). Then we can compute the Fisher score:

$$U_{\mathcal{Y}} = \nabla_\theta \log\left(P[\mathcal{Y}|H_1, \theta]\right) \tag{5}$$

as the feature vector corresponding to each $\mathcal{Y}$, θ being the real-vector valued parameter. What is not clear in this set-up is how, given only the training set $\mathscr{T}$, the Fisher score is computed. For instance, the i^{th} entry of $U_{\mathcal{Y}}$ will look like

$$\begin{aligned}\left(U_{\mathcal{Y}}\right)_i &= \frac{\partial}{\partial\theta_i}\log\left(P[\mathcal{Y}|H_1, \theta]\right) \\ &= \frac{1}{P[\mathcal{Y}|H_1, \theta]} \times \frac{\partial}{\partial\theta_i}\left(P[\mathcal{Y}|H_1, \theta]\right)\end{aligned} \tag{6}$$

This could clearly depend on θ. To use some feature like the Fisher score, we need to identify a real-valued vector parameter θ and to completely specify the computation of $U_{\mathcal{Y}}$.

Let the sets of malicious and normal traces in the training set be:

$$\begin{aligned}\mathcal{M} &= \{\, \mathcal{Y} \,|\, (\mathcal{Y}, L(\mathcal{Y})) \in \mathscr{T}, L(\mathcal{Y}) = 1\} \\ \mathcal{N} &= \{\, \mathcal{Y} \,|\, (\mathcal{Y}, L(\mathcal{Y})) \in \mathscr{T}, L(\mathcal{Y}) = 0\}\end{aligned}$$

Let n_1, n_0 be the sizes of the state-spaces of the attack and normal HMMs respectively. For H^1 we compute an estimate of probabilities $\hat{H}^1 = \{\hat{q}^1_{ij}, \hat{r}^1_{lj}\}$, based on the Expectation Maximization algorithm[10][2]. We obtain an updated set of estimates $\tilde{H}^1 = \{\tilde{q}^1_{ij}, \tilde{r}^1_{lj}\}$ that (locally) increases the likelihood of $\mathcal{M}$ (i.e. of the traces in $\mathcal{M}$) by maximizing the auxiliary function as below:

$$\tilde{H}^1 = \arg\max_{\underline{H}} \sum_{\mathcal{Y}\in\mathcal{M}} E_{\hat{H}^1}\left[\log P\left(\mathcal{Y}; \underline{H}\right)|\mathcal{Y}\right] \tag{7}$$

This step can be executed by the following(in the same manner as equation (44) of [10]):

$$\tilde{q}^1_{ji} = \frac{\hat{q}^1_{ji}\left(\sum_{\mathcal{Y}\in\mathcal{M}} \frac{\partial}{\partial\hat{q}^1_{ji}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)}{\sum_{k=1}^{n_1} \hat{q}^1_{jk}\left(\sum_{\mathcal{Y}\in\mathcal{M}} \frac{\partial}{\partial\hat{q}^1_{jk}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)} \tag{8}$$

$$\tilde{r}^1_{ji} = \frac{\hat{r}^1_{ji}\left(\sum_{\mathcal{Y}\in\mathcal{M}} \frac{\partial}{\partial\hat{r}^1_{ji}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)}{\sum_{k=1}^{s} \hat{r}^1_{jk}\left(\sum_{\mathcal{Y}\in\mathcal{M}} \frac{\partial}{\partial\hat{r}^1_{jk}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)} \tag{9}$$

where for simplicity the null output ϵ is not considered. A variant of this idea is a scheme where instead of the summation over all $\mathcal{Y} \in \mathcal{M}$, we repeat the update

separately for each $\mathcal{Y} \in \mathcal{M}$ (in some desired order) as follows:

$$\tilde{q}^1_{ji} = \frac{\hat{q}^1_{ji}\left(\frac{\partial}{\partial \hat{q}^1_{ji}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)}{\sum_{k=1}^{n_1} \hat{q}^1_{jk}\left(\frac{\partial}{\partial \hat{q}^1_{jk}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)} \tag{10}$$

$$\tilde{r}^1_{ji} = \frac{\hat{r}^1_{ji}\left(\frac{\partial}{\partial \hat{r}^1_{ji}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)}{\sum_{k=1}^{s} \hat{r}^1_{jk}\left(\frac{\partial}{\partial \hat{r}^1_{jk}} P\left(\mathcal{Y}; \hat{H}^1\right)\right)} \tag{11}$$

$\hat{H}^1$ is set equal to the update $\tilde{H}^1$ and the above steps are repeated till some criterion of convergence is met. We will now specify the initial value of $\hat{H}^1$ with which this recursion gets started. The acyclic nature of the Attack Tree means that, with an appropriate relabelling of nodes, the state-transition matrix is upper triangular:

$$q^1_{ji} > 0 \Leftrightarrow i \geq j$$

or block-upper triangular if some states (padding states for instance) are allowed to communicate with each other. As initial values for the EM algorithm, we can take(the equi-probable assignment):

$$\hat{q}^1_{ji} = \frac{1}{n_1 - j + 1}\,, \quad \forall i \geq j \tag{12}$$

noting that equation (8) preserves the triangularity. Similarly, we can take:

$$\hat{r}^1_{jl} = \frac{1}{s} \quad \forall\, l, j. \tag{13}$$

Since we want to be alert to variations in the attack by padding, it is not a good idea to start with a more restrictive initial assignment for the conditional output probabilities unless we have reliable information, such as constraints imposed by the Operating System, or 'tips' of an 'expert-hacker' . Such system-dependent restrictions, in the form of constraints on some of the probabilities q^1_{ji}, r^1_{jl} further focus our attention on the real vulnerabilities in the system. To further sharpen our attention, we can augment $\mathcal{M}$, by adding to it, its traces segmented by comparing with the traces in $\mathcal{N}$ and using any side information; essentially an attempt at stripping off padding. These segmented traces would be given smaller weights in the EM recursion (7). Going further in that direction, we can, instead of using the EM algorithm use various segmentations of the traces in $\mathscr{T}$ (into n_1 parts) and estimate the probabilities $\{q^1_{ji}, r^1_{jl}\}$. Even though we face difficulties such as a large number of unknowns, a relatively small training set, and the problem of settling on a local optimum point in the EM algorithm, we are banking on the robustness of the Support Vector Machine classifier that uses the parameters of the generative model. We can compute similar estimates($\hat{H}^0$) for the HMM representing the normal programs even though they do not, in general, admit simplifications like triangularity of the state-transition matrix.

The parameter vector we are interested in is the following:

$$\theta = \left[q_{11}, q_{12}, \cdots, q_{21}, \cdots, q_{NN}, r_{11}, \cdots, r_{sN} \right]^T \tag{14}$$

N being the larger of n_1, n_0; setting to zero those probabilities that are not defined in the smaller model. This vector can be estimated for the two HMMs $H^1, H^0 : \hat{\theta}^1, \hat{\theta}^0$ simply from $\hat{H}^1, \hat{H}^0$.

For any trace, be it from $\mathscr{T}$ or from the testing set, we can define the following feature vector:

$$U_{\mathcal{Y}} = \left[\nabla_\theta \log \left(P[\mathcal{Y}|H^1, \theta] \right) \mid_{\theta=\hat{\theta}^1} \right] \tag{15}$$

This vector measures the likelihood of a given trace being the output of the Attack Tree model and can be the basis of a *Signature-based* Intrusion Detection Scheme. On the other hand, we can use the information about normal programs gathered in H^0 to come up with

$$U_{\mathcal{Y}} = \begin{bmatrix} \nabla_\theta \log \left(P[\mathcal{Y}|H^1, \theta] \right) \mid_{\theta=\hat{\theta}^1} \\ \nabla_\theta \log \left(P[\mathcal{Y}|H^0, \theta] \right) \mid_{\theta=\hat{\theta}^0} \end{bmatrix} \tag{16}$$

which can be used for a *Combined Signature and Anomaly-based* detection. Something to be kept in mind is that the parameter vector (and hence the feature vectors) defined by (14) will contain many useless entries (with values zero) because we do not use the triangularity of the state-transition matrix for H^1 or any system dependent restrictions and because we artificially treat (in (14)) the HMMs to be of equal size. Instead, we can define different(smaller) parameter vectors $\theta_{\mathcal{M}}$ and $\theta_{\mathcal{N}}$ for the malicious and normal HMMs respectively and considerably shrink the feature vectors. Also for each feature vector in (15) and the two 'halves' of the vector in (16), there is a constant scaling factor in the form of the reciprocal of the likelihood of the trace given the HMM of interest(as displayed in equation (6)). This constant scaling tends to be large because of the smallness of the concerned likelihoods. We can store this likelihood as a separate entry in the feature vector without any loss of information. A similar issue crops up in the implementation of the EM algorithm: the forward and backward probabilities needed for the computations in (8), (9), (10) and (11), tend to become very small for long observation sequences, making it important to have a high amount of decimal precision

4 THE SVM ALGORITHM AND NUMERICAL EXPERIMENTS

Support Vector Machines (SVMs)[14] are non-parametric classifiers designed to provide good generalization performance even on small training sets. A SVM maps input (real-valued) feature vectors ($x \in X$ with labels $y \in Y$) into a (much) higher dimensional feature space ($z \in Z$) through some nonlinear mapping (something that captures the nonlinearity of the true decision boundary).

In a feature space, we can classify the labelled feature vectors (z_i, y_i) using hyper-planes:

$$y_i[< z_i, w > +b] \geq 1 \tag{17}$$

and minimize the functional $\Phi(w) = \frac{1}{2} < w, w >$. The solution to this quadratic program can be obtained from the saddle point of the Lagrangian:

$$L(w, b, \alpha) = \frac{1}{2} < w, w > - \sum \alpha_i \left(y_i[< z_i, w > +b] - 1 \right) \tag{18}$$

$$w^* = \sum y_i \alpha_i^* z_i, \quad \alpha_i^* \geq 0; \tag{19}$$

Those input feature vectors in the training set that have positive α_i^* are called *Support Vectors* $\mathcal{S} = \{z_i | \alpha_i^* > 0\}$ and because of the Karush-Kuhn-Tucker optimality conditions, the optimal weight can be expressed in terms of the Support Vectors alone.

$$w^* = \sum_{\mathcal{S}} y_i \alpha_i^* z_i, \quad \alpha_i^* \geq 0; \tag{20}$$

This determination of w fixes the optimal separating hyper-plane. The above method has the daunting task of transforming all the input raw features x_i into the corresponding z_i and carrying out the computations in the higher dimensional space Z. This can be avoided by finding a symmetric and positive semi-definite function, called the Kernel function, between pairs of x_i

$$K : X \times X \to \mathbb{R}^+ \cup \{0\}, \; K(a, b) = K(b, a) \quad \forall a, b \in X \tag{21}$$

Then, by a theorem of Mercer, a transformation $f : X \to Z$ is induced for which,

$$K(a, b) =< f(a), f(b) >_Z \quad \forall a, b \in X \tag{22}$$

Then the above Lagrangian optimization problem gets transformed to the maximization of the following function of α_i:

$$W(\alpha) = \sum \alpha_i - \frac{1}{2} \sum \alpha_i \alpha_j y_i y_j K(x_i, x_j) \tag{23}$$

$$w^* = \sum y_i \alpha_i^* z_i, \quad \alpha_i^* \geq 0; \tag{24}$$

the support vectors being the ones corresponding to the positive αs. The set of hyper-planes considered in the higher dimensional space Z have a small estimated VC dimension[14]. That is the main reason for the good generalization performance of SVMs.

Now that we have real vectors for each trace, we are at full liberty to use the standard kernels of SVM classification. Let $u_1, u_2 \in \mathbb{R}^n$,. We have the *Gaussian Kernel*

$$K(u_1, u_2) = \exp\left(-\frac{1}{2\sigma^2} (u_1 - u_2)^T (u_2 - u_2) \right), \tag{25}$$

the *Polynomial Kernel*

$$K(u_1, u_2) = (u_1^T u_2 + c_1)^d + c_2, \quad c_1, c_2 \geq 0, \quad d \in \mathbb{N} \tag{26}$$

or the *Fisher Kernel*

$$K(u_1, u_2) = u_1^T I^{-1} u_2; \;\; I = E_Y[U_Y U_Y^T] \tag{27}$$

Having described the various components of our scheme for intrusion detection and classification, we provide below a description of the overall scheme and experiments aimed to provide results on its performance. The overall detection scheme executes the following steps:

1. For the given T_1 attack traces of system calls $\mathcal{Y}_i$, we estimate using the EM algorithm a HMM model H^1 for an attack with n_1 states.
2. For given T_0 normal traces of system calls, $\mathcal{Y}_i$, we estimate a HMM model H^0 for the normal situation with n_0 states.
3. We compute the Fisher scores for either a signature-based intrusion detection or a combined signature and anomaly-based intrusion detection using equations (15) and (16).
4. Using the Fisher scores we train a SVM employing either one of the kernels (Gaussian, Polynomial, Fisher).
5. Given a test trace of system calls $\mathcal{Y}$, we let the SVM classifier decide as to whether the decision should be 1 (attack) or 0 (normal). The Fisher scores of $\mathcal{Y}$ are computed and entered in the SVM classifier.

We performed numerical experiments on a subset of the data-set for host based intrusion detection from the University of New Mexico [13][3]. We need to distinguish between normal and compromised execution on the Linux Operating system of the `lpr` program which are vulnerable because they run as a privileged processes. In the experiments, we tried various kernels in the SVMs. The performance evaluation is based on the computation of several points of the receiver operating characteristic (ROC) curve of the overall classifier; i.e. the plot of the curve for the values of the probabilities of correct classification (detection) P_D vs the false alarm probability P_{FA}.

In our experiments with HMMs (both attack and normal), we encountered two difficulties due to the finite precision of computer arithmetic (the `long double` data type of `C/C++` for instance is not adequate):

1. The larger the assumed number of states for the HMM, the smaller the values of the probabilities $\{q_{ji}\}$. For a fixed set of traces, like in our case, increasing the number of states from say, 5 to 10 or any higher value, did not affect the EM estimation (or the computation of the Fisher score) because, despite the attacks and normal executions being carried out in more than 5 (or n) stages, the smaller values of $\{q_{ji}\}$ make the EM algorithm stagnate immedeately at a local optimum.
2. Having long traces (200 is a nominal value for the length in our case) means that values of the *forward* and *backward* probabilities [10] $\alpha_t(j), \beta_t(j)$ become negligible in the EM algorithm as well as in the computation of the Fisher score. For the EM algorithm, this means being stagnant at a local optimum and for the computation of the Fisher score, it means obtaining score vectors all of whose entries are zero.

3. While computing the Fisher scores (15,16), if any element of θ is very small at the point of evaluation, the increased length of the overall Fisher score has a distorting effect on the SVM learning algorithm. For instance, while using linear kernels, the set of candidate separating hyper-planes in the feature space is directly constrained by this. This problem is actually the result of including the statistics of non-specific characteristics (background-noise, so to speak) like the transition and conditional output probabilities related to the basic system calls like `break, exit, uname` etc.

To combat these problems of numerical precision, one can go for an enhanced representation of small floating point numbers by careful book-keeping. But this comes at the cost of a steep increase in the complexity of the overall detection system and the time taken for computations.

We propose a solution that simplifies the observations and segments each trace into small chunks with the idea of viewing the trace as a (short) string of these chunks. This solution removes the floating point precision problems.

4.1 SVM classification using reduced HMMs

We describe a technique for deriving a reduced order Attack HMM (or a normal HMM) from the traces in the training set. We choose a small number of states to account for the most characteristic behavior of attacks (or of Normal program execution). We also use the observation that system-calls that constitute intrusions (attack system calls from the set $\mathscr{Y}^{\mathscr{V}}$) are not exactly used for padding (i.e. $\mathscr{Y}^{\mathscr{V}} \cap \mathscr{Y}^{\mathscr{P}} \approx \emptyset$). For every trace $\mathcal{Y}$, we can compute the ratio of the number of occurrences of a system-call s and the length of that trace. Call this number $\rho_s(\mathcal{Y})$. We can also compute the ratio of the position of first occurrence of a system-call s and the length of the trace (same as the ratio of the length of the longest prefix of $\mathcal{Y}$ not containing s and the length of $\mathcal{Y}$). Call this number $\delta_s(\mathcal{Y})$. Calculate these ratios $\rho_s(\mathcal{Y})$,$\delta_s(\mathcal{Y})$ for all system calls $s \in \mathscr{Y}$, and for all T_1 malicious traces in $\mathscr{T}$.

For every $s \in \mathscr{Y}$, find the median of $\rho_s(\mathcal{Y})$ over all T_1 malicious traces in $\mathscr{T}$. Call it $\hat{\rho}^1_s$. Similarly, compute the medians $\hat{\delta}^1_s \ \forall s \in \mathscr{Y}$. We prefer the median over the mean or the mode because we want to avoid being swayed by outliers. We now propose a scheme for identifying attack states $\{v\}$. Choose $\gamma_1, \gamma_2 : 0 < \gamma_1, \gamma_2 < 1$. Find subsets $\{s_1, s_2, \ldots, s_k\}$ of $\mathscr{Y}$ such that

$$|\hat{\rho}^1_{s_i} - \hat{\rho}^1_{s_j}| < \gamma_1 \ , \ |\hat{\delta}^1_{s_i} - \hat{\delta}^1_{s_j}| < \gamma_2 \ , \ \forall i, j \ \in \ \{1, 2, \ldots, k\} \tag{28}$$

Increase or decrease γ_1, γ_2 so that we are left with a number of subsets equal to the desired number of states n_1. In practice, most, if not all, of these subsets are disjoint. These subsets form the attack states. However, the alphabet is no longer $\mathscr{Y}$ but $\mathscr{Y}^*$. Thus, for the state $x_j = \{s_1, s_2, \ldots, s_k\}$, all strings of the form '$w_1, s_{\pi(1)} 1, w_2, s_{\pi(2)}, w_3, \ldots, w_k, s_{\pi(k)}, w_{k+1}$' are treated as the same symbol corresponding to it (with $w_1, w_2, w_3, \ldots, w_k, w_{k+1} \in \mathscr{Y}^*$ and with π a permutation on $\{1, 2, \ldots, k\}$ such that $\hat{\delta}^1_{s_{\pi(i)}}$ is non-decreasing with i). We call this symbol (also a regular expression) y_j.

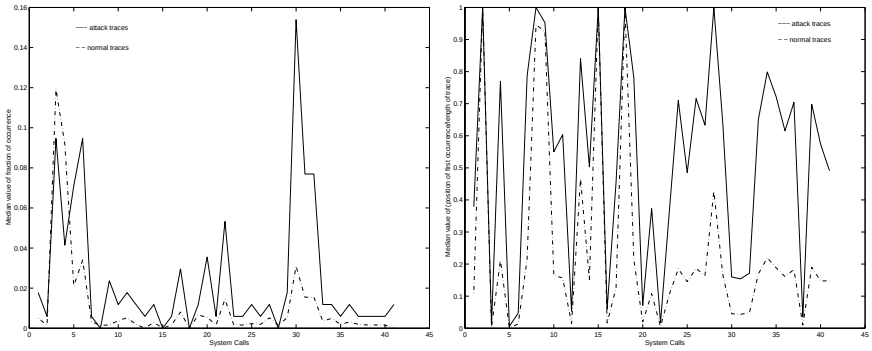

Fig. 2. Plots of the values of $\hat{\rho}_s^1$ and $\hat{\delta}_s^1$ over normal (dashed lines) and attack (solid lines) sequences used in obtaining reduced HMMs for the `lpr` (normal) and `lprcp` (attack) programs (the system call index has been renamed to ignore those system calls never used).

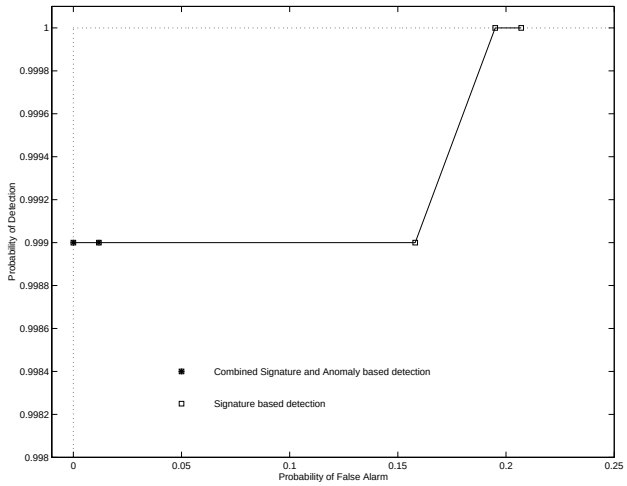

Fig. 3. Plot of ROC (for different number of hidden states) using SVMs and reduced HMMs using the computation of the medians $\hat{\rho}_s^1$,$\hat{\rho}_s^0$,$\hat{\delta}_s^1$ and $\hat{\delta}_s^0$. We used the trace dataset of `lpr` (normal) and `lprcp` (attack) programs.

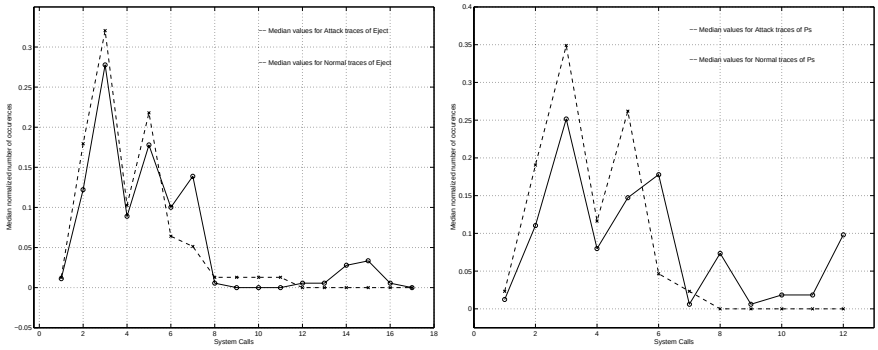

Fig. 4. Plots of the values of $\hat{\rho}_s^1$ and $\hat{\rho}_s^0$ over normal (dashed lines) and attack (solid lines) sequences used in obtaining reduced HMMs for the `lpr` (normal) and `lprcp` (attack) programs (For the `Eject` program: sys_call_12 = `pipe`, sys_call_13 = `fork` For the `Ps` program: sys_call_10 = `fork`, sys_call_11 = `fcntl`)

Now, we can assign numerical values for $\{q_{ji}\}$ and for $\{r_{jl}\}$. The transition probability matrix will be given a special structure. Its diagonal has entries of the form : τ_i and the first super-diagonal has its entries equal to $1-\tau_i$ and all other entries of the matrix are equal to 0. This is the same as using a *flat* or *left-right* HMM [10]. We set the conditional output probability of observing the compound output y_j corresponding to state x_j to be $\mu_j : 0 < \mu_j < 1$. We treat all other outputs at this state as the same and this *wild-card* symbol (representing $\mathscr{Y}^* - \{y_j\}$) gets the probability $1-\mu_j$. We can make the values μ_j all the same or different but parameterized in some way, along with the τ_is by a single variable so that we can easily experiment with detection performance as a function of the μ_j, τ_i. A point to be kept in mind all along is that we need to parse any given trace $\mathcal{Y}$ into n_1 (or more) contiguous segments. When there are different segmentations possible, all of them can be constructed and the corresponding feature vectors tested by the classifier.

The above steps can be duplicated for constructing the normal HMM also. A sharper and more compact representation is obtained if the Attack tree and the Normal tree do not share common subsets as states. In particular, consider a subset (of $\mathscr{Y}$) $x = \{s_1, s_2, \ldots, s_l\}$ that meets condition (28) for both the normal and attack traces:

$$|\hat{\rho}_{s_i}^L - \hat{\rho}_{s_j}^L| < \gamma_1^L \ , \ |\hat{\delta}_{s_i}^L - \hat{\delta}_{s_j}^L| < \gamma_2^L \ ,$$

$$\forall i, j \ \in \ \{1, 2, \ldots, k\}, \ 0 < \gamma_1^L, \gamma_2^L < 1, \ L \in \{0, 1\} \tag{29}$$

Then, x should clearly not be a state in either the Attack HMM or the Normal HMM. The signature based detection scheme would as usual use only the reduced attack HMM. The combined signature and anomaly-based approach would use both the attack and normal HMMS.

Now the overall detection scheme executes the following steps:

1. For the given T_1 attack traces of system calls $\mathcal{Y}_i$, we parse the $\mathcal{Y}_i$ into n_1 blocks and estimate using the reduced HMM model H^1 for an attack with n_1 states.
2. For given T_0 normal traces of system calls, $\mathcal{Y}_i$, we parse the $\mathcal{Y}_i$ into n_2 blocks and estimate a reduced HMM model H^0 for the normal situation with n_0 states.
3. We compute the Fisher scores for either a signature-based intrusion detection or a combined signature and anomaly-based intrusion detection using equations (15) and (16).
4. Using the Fisher scores we train a SVM employing either one of the kernels (Gaussian, Polynomial, Fisher).
5. Given a test trace of system calls $\mathcal{Y}$, we let the SVM classifier decide as to whether the decision should be 1 (attack) or 0 (normal). The Fisher scores of $\mathcal{Y}$ are computed and entered in the SVM classifier.

We performed numerical experiments on live `Lpr` and `Lprcp` (the attacked version of `Lpr`) traces in the data-set for host based intrusion detection [13][3]. We found that the quadratic programming step of the SVM learning algorithm did not converge when we used linear and polynomial kernels (because of very long feature vectors). On the other hand, SVM learning was instantaneous when we used the Gaussian kernel on the same set of traces. The value of the parameter σ in equation (25) made no significant difference. We used the Gaussian kernel (25) We selected a small training set (about one percent of the whole set of traces with the same ratio of intrusions as in the whole set). We trained the SVM with different trade-offs between the training-error and the margin (through the parameter c in [7]) and different number of hidden states for the Attack and Normal HMMs . We averaged the resulting P_D, P_{FA} (on the whole set) over different random choices of the training set $\mathscr{T}$.

We also performed experiments on the `eject` and `ps` attacks in the 1999 MIT-LL-DARPA data set [8]. We used traces from the first three weeks of training. In the case of the `eject` program attack, we had a total of 8 normal traces and 3 attack traces in the BSM audit records for the first three weeks. Needless to say, the SVM classifier made no errors at any size of the reduced HMMs. The interesting fact to observe was that the single compound symbol (28) (for the most reduced HMM) '`pipe*fork`' was enough to classify correctly, thus learning the Buffer-overflow step from only the names of the system calls in the traces. The ps trace-set can be said to have more statistical significance. We had 168 normal and 3 attack instances. However, for all sizes of reduced HMMs, all of the Fisher scores for the Attack traces were the same as for the Normal ones. Here, too, at all resolutions, the buffer-overflow step was learnt cleanly: All the reduced HMMs picked the symbol '`fork*fnctl`' to be part of their symbol set (28). Here too, the SVM made no errors at all. The plots of $\hat{\rho}_s^1$ and $\hat{\rho}_s^0$ in Fig.3 complete the picture. This data-set make us beleive that this approach learns efficiently buffer-overflow type of attacks. It also highlights the problem of a lack of varied training instances.

We used the SVM^{light}[7] program for Support Vector Learning authored by Thorsten Joachims.

5 SVM classification using gappy-bigram count feature vectors

Here, we present an algorithm that uses a simpler feature that avoids the estimation of the gradient of the likelihoods. For any trace $\mathcal{Y} \in \mathscr{Y}^*$, we can write down a vector of the number of occurrences of the so-called *gappy-bigrams* in it. A *bigram* is a string (for our purposes, over the alphabet $\mathscr{Y}$) of length two that is specified by its two elements in order. A gappy-bigram '$r\lambda s$' is any finite-length string (over the set $\mathscr{Y}$) that begins with the alphabet s and terminates with the alphabet $\grave{s}$. Let

$$\#_{s\grave{s}}(\mathcal{Y}) = \text{the number of occurences of the gappy} - \text{bigram '}\ s\lambda\grave{s}\ \text{' in}\ \mathcal{Y} \tag{30}$$

where

$$s, \grave{s} \in \mathscr{Y}\ ,\ \ \lambda \in \mathscr{Y}^* \cup \{\epsilon\}\ ,\ \ \epsilon\ \text{being the null string.} \tag{31}$$

We write down the T^2-long vector of counts $\#_{s\grave{s}}(\mathcal{Y})$ for all $(s, \grave{s}) \in \mathscr{Y} \times \mathscr{Y}$.

$$C(\mathcal{Y}) = \begin{bmatrix} \#_{s_1 s_1} \\ \#_{s_1 s_2} \\ \vdots \\ \#_{s_T s_T} \end{bmatrix} \tag{32}$$

We call the feature vector $C(\mathcal{Y})$, the count score of $\mathcal{Y}$ and use this to modify the earlier scheme using the Fisher score.

The new overall detection scheme executes the following steps:

1. We compute the count scores using equation (32).
2. Using the count scores we train a SVM employing either one of the kernels (Gaussian, Polynomial, Fisher).
3. Given a test trace of system calls $\mathcal{Y}$, we let the SVM classifier decide as to whether the decision should be 1 (attack) or 0 (normal). The count scores of $\mathcal{Y}_i$ are computed and entered in the SVM classifier.

We performed numerical experiments on live `Lpr` and `Lprcp` (the attacked version of `Lpr`) traces in the data-set for host based intrusion detection [13][3]. We found that the quadratic programming step of the SVM learning algorithm did not converge when we used linear and polynomial kernels (because of very long feature vectors). On the other hand, SVM learning was instantaneous when we used the Gaussian kernel on the same set of traces. The value of the parameter σ in equation (25) made no significant difference. Our experiments were of the following two types:

1. We selected a small training set (about one percent of the whole set of traces with the same ratio of intrusions as in the whole set). We trained the SVM with different trade-offs between the training-error and the margin (through the parameter c in [7]). We averaged the resulting P_D, P_{FA} (on the whole set) over different random choices of the training set $\mathscr{T}$. Our average (as well as the median) values of P_D, P_{FA} were 0.95 and 0.0.
2. We used the whole set of traces available for training the SVM with different tradeoffs (again, the parameter c in [7]) and used the leave-one-out cross-validation $\xi\alpha$ ([7]) estimate of P_D, P_{FA}. We obtained the following values for P_D, P_{FA} : $0.992, 0.0$.

We have only one measured point on the ROC curve. We also note that this detection system behaves like an anomaly-based intrusion detection system.

6 CONCLUSIONS

We have described a method for incorporating the structured nature of attacks, as well as any specific system-dependent or other 'expert-hacker' information, in the HMM generative model for malicious programs. Using the generative model, we have captured the variability of attacks and compressed into a vector of real values, the set of variables to be examined for flagging off attacks. We use these derived feature vectors in place of variable-length strings, as inputs to the Support Vector Machine learning classifier which is designed to work well with small training sets. We have presented a method for deriving reduced HMMs using the temporal correlations (28, 29) between system calls in traces. An alternative large-scale HMM classifier would need to use techniques from the area of large vocabulary speech recognition [6] to grapple with the numerical problems associated with full-scale generative models for attacks and normal program execution. We also presented the gappy-bigram count feature vector for SVM based classification. We need to develop versions of the above intrusion detection systems that work in real-time, and those that work on distributed programs like a network transaction.

7 Acknowledgments

This work was supported by the United States Army Research Office under contract number DAAD190110494 of the CIP-URI program. We express our thanks to Senni Perumal, Sudhir Varma and Shah-An Yang for suggestions and for assistance in the computer experiments. We also thank the referees for their useful comments.

References

1. N. Cristianini, J. and Shawe-Taylor. An introduction to Support Vector Machines and other kernel-based learning methods. Cambridge University Press (2000)

2. R. Elliot, L. Aggoun, J., Moore. Hidden Markov Models, Estimation and Control, Springer-Verlag.
3. S. Hofmeyr, S. Forrest, A. Somayaji. Intrusion detection using sequences of system calls. Journal of Computer Security **6** (1998) 151–180
4. T. Jaakkola, and D. Haussler. Exploiting generative models in discriminative classifiers. Advances in Neural Information Processing Systems II, San Mateo, CA. Morgan Kauffmann Publishers.
5. T. Jaakkola, and D. Haussler. Using the Fisher Kernel method to detect remote protein homologies. Proceedings of the Seventh International Conference on Intelligent Systems for Molecular Biology (1999)
6. F. Jelinek. Statistical methods for Speech Recognition. MIT Press, (1999)
7. T. Joachims. SVM^{light}. `http://svmlight.joachims.org/`.
8. MIT Lincoln Labs, The 1999 DARPA Intrusion Detection evaluation data corpus. `http:// www.ll.mit.edu /IST /ideval`
9. O. Sheyner, J. Haines, S. Jha, R. Lippmann, J. Wing, J. Automated generation and analysis of Attack Graphs. Proceedings of the IEEE Symposium on Security and Privacy, Oakland, CA, (May 2002)
10. L. Rabiner. A tutorial on Hidden Markov Models and selected application in Speech Recognition. Proceedings of the IEEE, vol: **77**, No: 2, (February 1989)
11. I. Rigoutsos and A. Floratos. Combinatorial pattern discovery in Biological sequences: the Teiresias algorithm. Bioinformatics, vol:**14**, no:1, pages:55-67 (1998)
12. B. Schneier. Attack Trees. Dr. Dobb's Journal, `http:// www.ddj.com /documents /s=896/ddj9912a/9912a.htm` (December 1999)
13. `http://www.cs.unm.edu/ immsec/data/`
14. V. Vapnik. Statistical Learning Theory. Wiley Inter-science. (1996)
15. A. Wespi, M. Dacier, H. Debar. Solutions périodiques, du An intrusion detection system based on the Teiresias pattern discovery algorithm. EICAR proceedings (1999)