

A Comparison Of Input Representations In Neural Networks: A Case Study In Intrusion Detection

Zhen Liu, German Florez and Susan M. Bridges

Mississippi State University

zliu@cs.msstate.edu

gf24@cs.msstate.edu

bridges@cs.msstate.edu

Abstract- Recently intrusion detection techniques have shifted from user-based and connection-based to process-based intrusion detection. Forrest et al. [2] presented one of the first papers analyzing sequences of system calls issued by a process for intrusion detection. Warrender et al. [11] do a comparison of the accuracy and performance of different algorithms for analysis of sequential calls. In this paper we present a comparison of two different encoding methods for three types of neural networks. We apply these classifiers to implement an anomaly detection module for UNIX processes.

1. INTRODUCTION

With the wide use of computers, the emergence of electronic commerce, and the rapid growth of the Internet, computer security has become more and more important. In 1988, Morris released a program that succeeded in infecting thousands of UNIX hosts on the Internet. From then on, systems have been unable to defend themselves against outside attacks that take advantage of new vulnerabilities. In 2000, for example, companies like Yahoo or Ebay were attacked by DDOS (distributed denial of service), costing them millions of dollars.

An intruder can implement an attack by exploiting software errors in privileged programs to gain root privilege, exploiting vulnerabilities in the system configuration to access confidential data, or relying on a legitimate system user to download and run a legitimate-looking Trojan horse program that inflicts damage [5].

Intrusion detection refers to a broad range of techniques that have been developed over the past several years to protect against malicious attacks. All of these methods are based on analysis of some type of audit data such as TCP/IP packet data, TCP/IP connection data, statistical data about CPU usage, or number of processes created by a user among others.

We present a comparison of two different encoding methods for three types of neural networks, and we apply these classifiers to implement an anomaly detection module for UNIX processes. The paper is organized as follows: section 2 is a brief discussion of system call analysis techniques, section 3 presents a survey of neural network topologies that we are studying; section 4 describes the data sets used, section 5 explain two different encoding strategies and data preprocessing techniques, section 6 presents experiments and results, and section 7 contains our conclusions.

2. SYSTEM CALL ANALYSIS

A system call is usually a request to the operating

system (kernel) to do a hardware/system-specific or privileged operation [3]. Generally it is implemented using the `syscall()` low-level command, which accepts as parameters the number of the system call and its arguments.

One of the first works analyzing sequences of system calls for intrusion detection is described in [2]. This system uses the concept of window size (the length of the sequence) to compare exact matches between a normal profile and the new trace of a process. This information is stored in a tree structure and corresponds to the database of normal behavior. When a new (unseen) sequence is presented, the system tries to find it in the tree. If it is not present, a counter of anomalies is increased. If this value exceeds a user threshold, an alarm is fired. This is known as the *stide* algorithm.

The *t-stide* algorithm [11] improves the *stide* algorithm by storing the frequency for each sequence in the database. Then, the rare matched sequences in the data presented to the system will be also labeled as anomalous.

Markov processes are widely used to model systems in terms of state transitions. Some intrusion detection algorithms that exploit the Markov property implement Hidden Markov models (using the Baum-Welch algorithm), Markov chains, or Sparse Markov trees. These methods do not use sequences of events (system calls) within an interval of time (window size); instead, they analyze transitions from (and to) each system call [11].

One of the most used rule-based algorithms in the intrusion detection field is the *Repeated Incremental Pruning to Produce Error Reduction* (RIPPER). It is a rule learning system developed by William Cohen [1]. This algorithm performs classification by creating a list of rules from a set of labeled training examples. RIPPER is also used for anomaly detection to predict system calls, where the classification of each training example corresponds to the last call of the sequence [8].

The same concept of prediction of sequence calls can be implemented using neural networks [7]. The main advantages of such an approach are the lack of dependence on any statistical assumption, noise tolerance, and abstraction.

A comparison of the accuracy of different algorithms for system call analysis can be found in [11].

3. NEURAL NETWORKS

3.1 Back-Propagation

Back-propagation (BP) is probably the most widely used algorithm for generating classifiers and is often used

for benchmarking other learning algorithms [9]. A back-propagation neural network is a feed-forward multi-layer neural network. It has two stages: a forward pass and a backward pass. The forward pass involves presenting a sample input to the network and letting activations flow until they reach the output layer. The activation function can be any function. The linear sum, sigmoid function and Gaussian function are three often used functions. During the backward pass, the network's actual output (from the forward pass) is compared with the target output and error estimates are computed for the output units. The weights connected to the output units can be adjusted in order to reduce those errors. We can then use the error estimates of the output units to derive error estimates for the units in the hidden layers. Finally, errors are propagated back to the connections stemming from the input units. Detailed descriptions of the back-propagation algorithm can be found in [4][9].

We have implemented two versions of the back-propagation algorithm for analyzing sequences of system calls.

3.2 Radial Basis Function Networks

Radial basis function (RBF) networks are another commonly used type of feed-forward network. RBF networks have been proven to be universal approximators and have the properties of fast convergence, easy solution to regularization, and robustness to outliers [4].

A pattern-classification problem cast in a high-dimensional space is more likely to be linearly separable than in a low-dimensional space [9]. The radial basis function network uses this characteristic as its basis. It transforms the input vector into a higher dimension feature space and does the classification in this space. The input layer is made up of source nodes (sensory units) that connect the network to its environment. The second layer, the only hidden layer in the network, applies a nonlinear transformation from the input space to the hidden space (often a higher dimension space). The output layer is linear, supplying the response of the network to the activation pattern applied to the input layer [9].

We have implemented two versions of radial basis functions networks, using K-means to find the RBF centers in the data set and K-nearest neighbors to compute the deviation of each center of the RBF.

3.3 Self-Organizing Map

Self-organizing map networks are based on competitive learning; the output neurons of the network compete among themselves to be activated or fired, with the result that only one output neuron, or one neuron per group, is activated each time. In a self-organizing map, the neurons are placed at the nodes of a lattice that is usually one or two-dimensional. A self-organizing map is characterized by the formation of a topographic map of the input patterns in which the spatial locations of the neurons in the lattice are indicative of intrinsic statistical features

contained in the input patterns [9]. This type of neural network is used for unsupervised learning. In order to be used for supervised learning it makes use of an algorithm like LVQ (learning vector quantization) after SOM [9]. LVQ learning has the advantage of very fast convergence [6].

For decimal encoding experiments, we are using the SOM_PAK (The self-organizing map program package) and LVQ_PAK (The learning vector program package) from The Helsinki University of Technology, Laboratory of Computer and Information Science. (http://www.cis.hut.fi/research/som_lvq_pak.shtml, accessed on October 10, 2001). The source code for each of the packages was compiled on a Linux system.

For binary encoding we have implemented a version of SOM and LVQ algorithms.

4. DATA SETS

For these experiments we used traces of system calls from two different UNIX programs: lpr and sendmail. This data was downloaded from the University of New Mexico at <http://www.cs.unm.edu/~immsec/systemcalls.htm>.

In order to train the networks, it is necessary to expose them to both normal and anomalous data. Training will be most effective if the training data is spread throughout the input space. Because training data typically contains only a small number of anomalous patterns, we have used two methods to generate additional anomalous data.

The first method generates additional anomalous data by making random changes to sequences extracted from the data. The randomly generated data is spread throughout the input space and causes the network to generalize that most data is anomalous. The normal data tends to be localized in the input space causing the network to recognize a particular area of the input space as normal. Using data with randomly generated anomalies allows the neural network to learn to detect new unknown attacks that it has not seen before.

The second method for generating additional anomaly data is based on small intrusion patterns. To find small intrusion patterns we first extract normal sequences from normal training data. We go through the anomaly training data to find those sequences that occur only in attack data. Most of sequences in attack data are normal. The anomaly sequences only occur in a small part of the trace of a program run that includes an intrusion. After finding these anomaly sequences, we find the common subsequences in them. For example, if we have these intrusion sequences:

2	3	4	50	12	3
3	4	50	12	3	6
4	50	12	3	6	45

4 50 12 3 is a small pattern that is common in these sequences. We can view this pattern as the signature of this

intrusion and generate more anomaly patterns based on this pattern. We randomly insert some system calls before or after this small pattern within the window size. Generating data based on small intrusion patterns will cause the network to pay more attention to the patterns that are more likely to be intrusions. This will cause the neural network to detect all the intrusions presented in the training data without increasing the false-positive rate.

5. INPUT REPRESENTATION

In order to use a neural network effectively for this problem it is important to develop a proper encoding for the input sequences. We use a sliding window to divide a trace of N system calls (a sequence of calls from one run of a program) into some small sequences of calls. For example, suppose we had as a normal trace the following sequence of calls:

open, read, write, close, close

and a window size (i.e. sequence length) of 3 (actual traces are much longer than this simple example). We slide the window across the sequence, and for each call we encounter, we record what calls precede it at different positions within the window, numbering them from 0 to *window_size-1*, with 0 being the current system call. For this trace, the sequence is described in Table 1.

<i>open</i>	<i>read</i>	<i>write</i>
<i>read</i>	<i>write</i>	<i>close</i>
<i>write</i>	<i>close</i>	<i>close</i>

TABLE 1. SEQUENCES OF SYSTEM CALLS

In our experiments, we used a window size of 10.

5.1 Binary encoding

We use a binary representation of these system calls as the input of neural network. Each system call has an identifying number (this number might change between different operating systems). We use the binary representation (with 8 bits) of those numbers as input for our neural networks. Therefore, if you define a window size of 3, there are 24 inputs for the neural network.

For example, in a Linux operating system the identifying numbers for *open*, *read*, *write*, *close* might be 10, 11 and 12 and 13 respectively (*open* is encoded as 00001010, *read* is transformed to 00001011 and so on) Thus, the binary encoding for the sequence described in Table 1 is shown in Table 2 (each bit corresponds to an input for the neural network).

00001010	00001011	00001100
00001011	00001100	00001101
00001100	00001101	00001101

TABLE 2. EXAMPLE OF BINARY REPRESENTATION

5.2 Decimal encoding

For this encoding, system calls are assigned consecutive numbers (following the order that the system call is presented in the database). The example presented in Table 3 uses only 4 system calls, and this is the dimension of the input space used by the neural network, (i.e. the number of input units).

We compute the frequency of each system call in each sequence divided by the maximum frequency of any call in the sequence (*window_size*). Table 3 shows an example of the decimal encoding. In the first sequence (*open*, *read*, *write*) the frequency for *close* is 0, and the others calls have a frequency of 1 divided by *window_size*.

<i>Open</i>	<i>Read</i>	<i>Write</i>	<i>Close</i>
1/3	1/3	1/3	0
0	1/3	1/3	1/3
0	0	1/3	2/3

TABLE 3. EXAMPLE OF DECIMAL REPRESENTATION

The main differences between the two input representations are:

- Binary encoding uses 8 bits for each system call, whereas decimal encoding uses a floating-point number (between 0 and 1).
- In decimal encoding, each input unit corresponds to one and only one system call.
- Using binary encoding, the total number of input units of the neural network is $8 * window_size$. Using decimal encoding, the total number is N (the number of different system calls presented in the training and test data).
- Decimal encoding does not take into account the order of the system calls in the sequence. It only computes frequencies.
- Binary approach encodes each system call using Linux identifying numbers. These numbers depend on the operating system version. Decimal encoding does not use this identifier.

6. EXPERIMENTS AND RESULTS

The dataset for our *lpr* experiments contains 2703 normal traces, with 1001 anomalous traces. With the binary encoding experiments we used 1500 normal traces and 500 anomalous traces for training; and 403 normal traces and 501 anomalous traces for testing. With decimal encoding we use 18 normal traces and 101 anomalous traces for training, and 45 normal traces and 301 anomalous traces for testing.

The dataset for our *sendmail* experiments contains 172 normal traces and 3 anomalous traces. With the binary encoding experiments we used 74 normal traces and 1 anomalous trace for training, and 98 normal traces and 2 anomalous traces for testing. With decimal encoding we used 91 normal traces and 1 anomalous trace for training;

whereas we used 172 normal traces and 3 anomalous traces for testing.

6.1 Experiments with sendmail

Figures 1, 2 and 3 show the accuracy of the neural networks that used binary encoding for the sendmail traces.

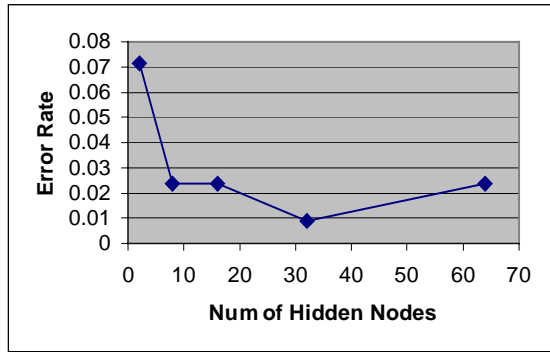


Figure 1 (Binary) BPNN for *sendmail*. Error rate with testing data

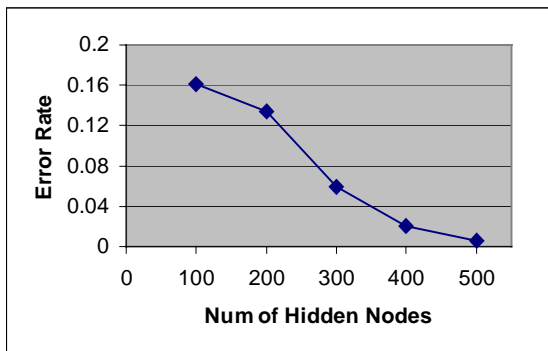


Figure 2 (Binary) RBF for *sendmail*. Error rate with testing data

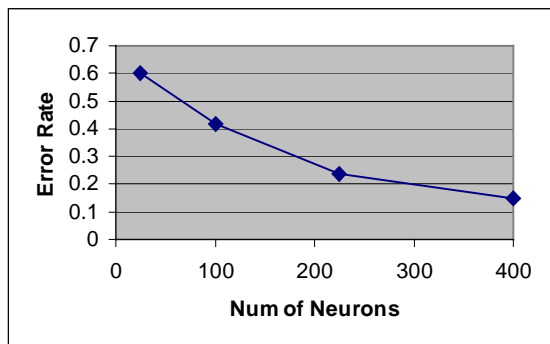


Figure 3 (Binary) SOM for *sendmail*. Error rate with testing data

Figures 4, 5 and 6 show the accuracy of the classifiers when using decimal encoding with *sendmail*.

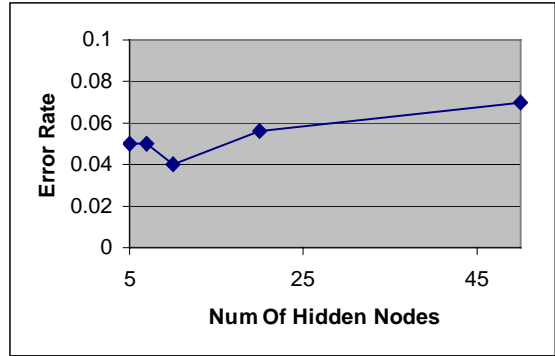


Figure 4 (Decimal) BPNN for *sendmail*. Error rate with testing data

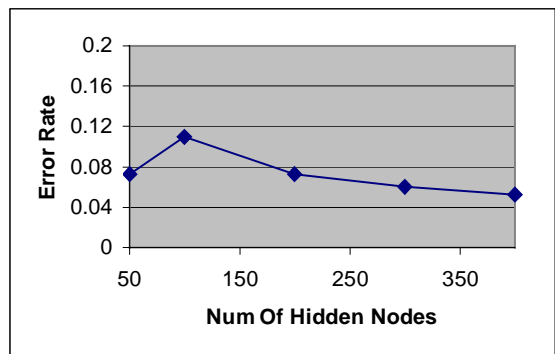


Figure 5 (Decimal) RBF for *sendmail*. Error rate with testing data

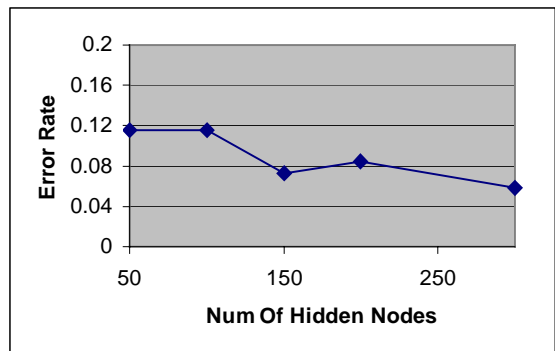


Figure 6 (Decimal) SOM&LVQ for *sendmail*. Error rate with testing data

Using the RBF and SOM & LVQ algorithms we can expect to further reduce the number of errors for the *sendmail* trace by increasing the size of the networks. However, with backpropagation, the classification error seems to increase with a large number of hidden units.

Figures 1-6 give the error rates for classification of sequences of size *window*. However, our goal is to label a program trace as normal or anomalous. We count the number of continuous anomalies of each trace. If such a number exceeds a user threshold, the trace is classified as an anomaly. Table 4 presents the confusion matrix for anomaly detection of *sendmail* traces using decimal input

representation with three different classifiers. *Negative* means normal and *Positive* means anomaly. *True Class* is the label in the real world, and *Predicted Class* is the class computed by our classifier. Our system was able to discriminate between normal and abnormal traces with a 100% true-positive rate, and a very low false-positive rate (as a matter of fact, using Backpropagation or RBF the false-positive rate is 0%)

		Backpropagation		Radial Basis		SOM&LVQ	
		Predicted Class		Predicted Class		Predicted Class	
		-	+	-	+	-	+
True Class	-	172	0	172	0	167	5
	+	0	3	0	3	0	3

TABLE 4. CONFUSION MATRIX FOR TRACE CLASSIFICATION USING DECIMAL ENCODING FOR SENDMAIL

6.2 Experiments with *lpr*

Figures 7, 8 and 9 show the accuracy of the neural networks by using binary encoding for the *lpr* traces (with artificial anomalies).

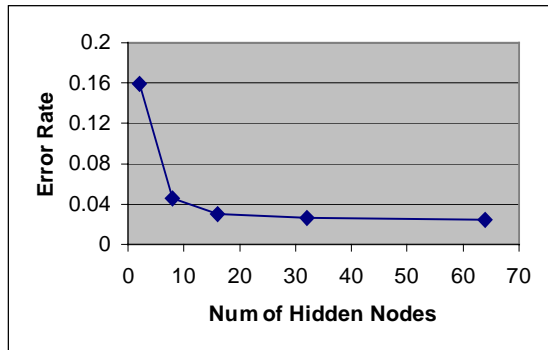


Figure 7 (Binary) BPNN for *lpr* Error rate with testing data

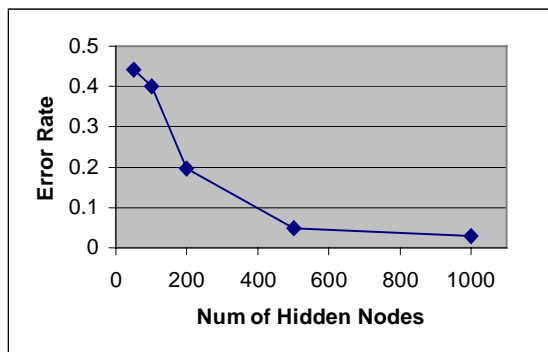


Figure 8 (Binary) RBF for *lpr*. Error rate with testing data

Figures 10, 11 and 12 show the accuracy of the neural networks by using decimal encoding for the *lpr*.

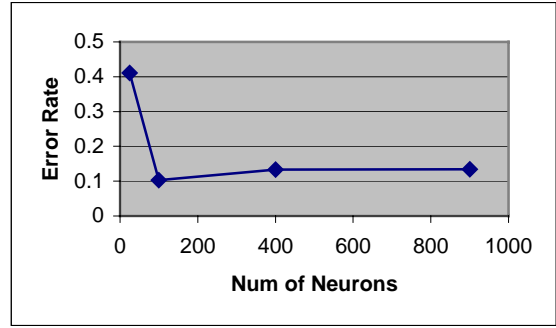


Figure 9 (Binary) SOM for *lpr*. Error rate with testing data

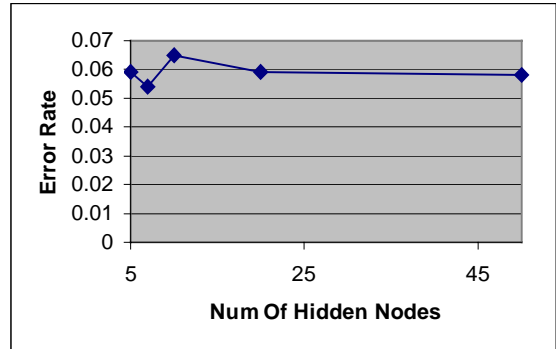


Figure 10 (Decimal) BPNN for *lpr*. Error rate with testing data

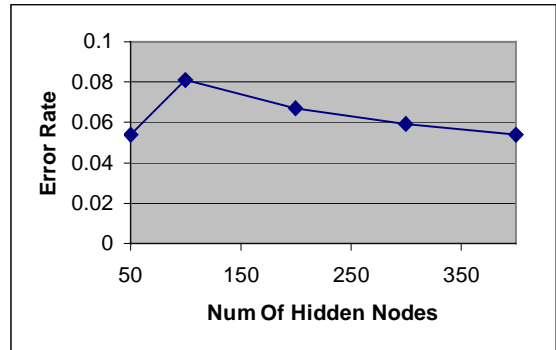


Figure 11 (Decimal) RBF for *lpr*. Error rate with testing data

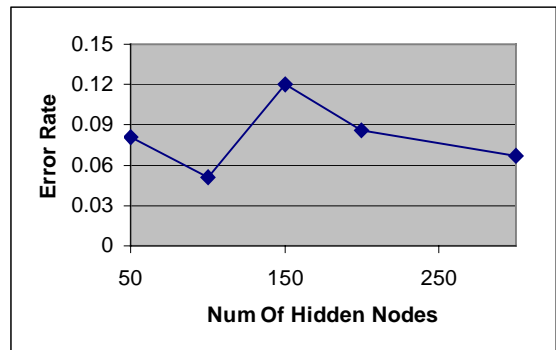


Figure 12 (Decimal) SOM&LVQ for *lpr*. Error rate with testing data

Table 5 presents the confusion matrix for decimal encoding for each one of the classifiers

		Backpropagation		Radial Basis		SOM&LVQ	
		Predicted Class		Predicted Class		Predicted Class	
		-	+	-	+	-	+
True Class	-	45	0	40	5	40	5
	+	1	300	1	300	1	300

TABLE 5. CONFUSION MATRIX FOR TRACE CLASSIFICATION USING DECIMAL ENCODING FOR LPR

Table 6 shows the confusion matrix for the *lpr* data using binary representation with different thresholds.

		Threshold=2		Threshold=3		Threshold=4	
		Predicted Class		Predicted Class		Predicted Class	
		-	+	-	+	-	+
True Class	-	1403	0	1401	2	1398	5
	+	1	500	1	500	0	501

TABLE 6. CONFUSION MATRIX FOR TRACE CLASSIFICATION USING BINARY ENCODING FOR LPR

We can see that the accuracy of our classifiers is very high. When the false positive rate is 0, our module detects most of the attacks. Because our modules used intrusion patterns to train the classifiers, the system can correctly identify all the intrusions that have been presented in the training data. The last trace in *lpr* intrusion data is not used to train our module. It is a new attack for our module. But from Table 6, we can see with a very low false positive rate 0.004%, all the intrusion can be detected.

7. CONCLUSIONS

We have demonstrated that neural networks are able to analyze sequences of system calls, and they can be used to deploy an intrusion detection system. By using two different encoding techniques (binary and decimal representation) the neural networks generate high true-positive rates and very low false-positive rates.

Using binary encoding the neural networks have lower error rates than decimal encoding, because this last representation does not take into account the order of the system calls in the sequence (it only computes the frequency). However, decimal encoding appears to handle noise well and the classifiers can be trained with fewer data. Decimal encoding does not take into account the order of the system call in the sequences. For that reason, there is a high probability of two sequences being identical, and the number of unique traces for training and testing is small.

Our experiments demonstrate that programs like *lpr* or *sendmail* generally do not use more than 45 system calls. Thus, the number of input neurons with decimal encoding is about 45, whereas the number of input neurons with binary encoding is 8×10 . As a consequence, binary

encoding uses a more complex topology, and the training and testing time is greater than for the neural networks with decimal encoding.

ACKNOWLEDGMENTS

This work was partially sponsored by the National Science Foundation Grant# CCR-9988524 and the Army Research Laboratory Grant# DAAD17-01-C-0011

We would like to thank Dr. Lois Boggess for her instruction in the Neural Networks course and her guidance for this project. Appreciation is also due to Dr. Rayford Vaughn and the Center for Computer Security Research at Mississippi State University for supporting this work.

REFERENCES

- [1] W. W. Cohen, "Fast Effective Rule Induction," *Proceedings of the Twelfth International Conference on Machine Learning*, Lake Tahoe, CA, 1995, pp. 115-123.
- [2] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, 1996, IEEE Computer Society Press, pp. 120-128.
- [3] S. Goldt, M. Sven, B. Scott and W. Matt, "The Linux Programmer's Guide," 1995, <http://www.ibiblio.org/mdw/LDP/lpg/node5.html> (Accessed on August, 20 2001)
- [4] A. K. Jain, R. P. W. Duin, and J. Mao, "Statistical Pattern Recognition: A Review," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. 22, no. 1, Jan. 2000, pp. 4-37.
- [5] K. Jain, and R. Sekar, "User-level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," 1999, <http://citeseer.nj.nec.com/jain00userlevel.html> (Accessed 10 July 2001)
- [6] T. Kohonen, J. Hynninen, J. Kangas, J. Laaksonen, and K. Torkkola. "LVQ_PAK: The learning vector quantization program package version 3.1," 1995, Espoo, Finland: Helsinki University of Technology. http://www.cis.hut.fi/research/som_lvq_pak.shtml (Accessed 15 Oct 2001).
- [7] S. Kumar and E. Spafford, "An Application of Pattern Matching in Intrusion Detection," Technical Report 94-013, Department of Computer Sciences, Purdue University, March 1994.
- [8] W. Lee and S. J. Stolfo, and P. K. Chan, "Learning Patterns from Unix Process Execution Traces for Intrusion Detection," *Proceedings of AAI-97 Workshop on AI Approaches to Fraud Detection and Risk Management*, 1997, AAAI Press, pp. 50-56.
- [9] H. Simon, *Neural Networks A Comprehensive Foundation*, 2nd Edition, Prentice-Hall, Upper Saddle River, New Jersey, 1999.
- [10] University of New Mexico. Computer Immune systems. <http://www.cs.unm.edu/~immsec/data/synth-sm.html> (Accessed on November 2001)
- [11] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *IEEE Symposium on Security and Privacy*, Oakland, CA, 1999, IEEE Computer Society, pp. 133-145.