

Classification of Anomalous Traces of Privileged and Parallel Programs by Neural Networks

Zhen Liu, Susan M. Bridges, and Rayford B. Vaughn
Mississippi State University
Department of Computer Science
Center For Computer Security Research
Box 9637
Mississippi State, MS 39762
(662) 325-2756 fax: (662) 325-8997
<zliu, bridges, vaughn>@cs.msstate.edu

Abstract- The focus of intrusion detection has recently shifted from user-based and connection-based to process-based intrusion detection. Substantial research has been done in the analysis of system call logs using different methods including neural networks. Detection is based on the classification of short sequences as anomalous or normal. The classification of interest, however, is the status of the program trace, not just the short sequences. In this paper we report the results of a comparative study of three different methods for on-line classification of program traces based detection of anomalies in sequences of system calls by neural networks. These results demonstrate that methods that use information about the locality of anomalies are more effective than those that only look at the number of anomalies.

1. INTRODUCTION

The wide use of computers, the emergence of electronic commerce, the rapid growth of the Internet, the advent of computer crime and computer-based threats to national security have combined to make computer security a focus of research. Intrusion detection has become an essential and critical component of modern computer systems. Intrusion detection techniques are often classified as either misuse detection or anomaly detection techniques. Misuse detection systems are usually constructed to recognize the signature of intrusion patterns that have been recognized and reported by experts. However, some of the most dangerous intrusions are those that use previously unknown methods or that mask illegal behavior to deceive the detection system. Anomaly detection methods are designed to counter this kind of challenge. Anomaly detection methods represent patterns of normal behavior, with the assumption that an intrusion will usually include some deviation from this normal behavior. Observation of a deviation will then result in an intrusion alarm. This paper focuses on host-based anomaly detection for privileged and parallel programs.

The focus of research in intrusion detection has recently shifted from user-based and connection-based intrusion detection to process-based intrusion detection. Process-based monitoring intrusion detection tools analyze

the behavior of executing processes for possible intrusive activity. The premise of process monitoring for intrusion detection is that most computer security violations are made possible by misusing programs. When a program is misused its behavior will differ from its normal usage. Therefore, if the behavior of a program can be adequately captured in a compact representation, the behavioral features can be used for intrusion detection. Two possible approaches to monitoring process behavior are: instrument programs to capture their internal states or monitoring the operating system to capture external system calls made by a program [13]. The second method has been the focus of work by Stephanie Forrest and her research group [6]. They, along with many other researchers, have developed techniques for detecting intrusions based on traces of system calls of privileged programs.

Our research group is currently investigating the use of system call analysis for anomaly detection in high performance Linux clusters. Clusters of workstations have become widely used computational resources in sensitive environments. The power of the computational resources afforded by these clusters combined with the sensitivity of the applications that they run make them attractive targets for intrusions.

This paper describes a method for anomaly detection that uses neural network classifiers to classify program traces as normal or anomalous. The neural network examines short sequences of system calls for anomalies. The level of alarm raised by the intrusion detection system depends on the number and pattern of these anomalous sequences. We compare different methods for classifying a program trace as normal or anomalous.

2. SYSTEM CALL ANALYSIS

System calls provide a rich source of information about the behavior of a program. A system call is usually a request to the operating system (kernel) to perform a hardware/system-specific or privileged operation [1]. Many different algorithms have been used to analyze this type of data.

Hofmeyr, Forrest, and Somayaji [6] describe a method called sequence time-delay embedding (stide), in which a profile of normal behavior is built by enumerating all unique, contiguous sequences of a predetermined, fixed length k that occur in the training data. The learning process for this method is very fast. It uses a tree representation of sequences and just creates a new path in the tree when it encounters a new sequence. Only one pass through the data is needed. During classification, the sequences are compared with the patterns in the tree. When the sequence window is small, this process is also very fast. The disadvantage of enumerating sequences is its lack of generalization.

Warrender, Forest, and Pearlmutter [14] compared several alternative data models for intrusion detection via system call analysis including Hidden Markov Models, stide, and the Ripper rule-based algorithm [3]. In general, they found that Hidden Markov Models were the most accurate but the training time was unacceptably long.

One of the key drawbacks of the n -gram approach used by Forrest's group [6] is the inability to generalize from past observed behavior. Thus, if the normal program behavior is not adequately captured, future unseen normal behavior may be classified as anomalous, thus contributing to the false positive rate.

Ghosh, Schwartzbard and Schatz [4] have used neural networks to analyze system log data. The goal in using neural networks for intrusion detection is to be able to generalize from incomplete data and to be able to classify online data as being normal or anomalous. An artificial neural network is composed of simple processing units, or nodes, and connections between them. The functionality of a neural network is to correctly map the input to output after training. Ghosh, Schwartzbard, and Schatz [4] compared their method with the Enumerating Sequences approach and showed that the neural networks have better generalization on the data.

We have investigated several machine-learning approaches for learning program behavior with the goal of identifying models that are sufficiently sensitive and that also produce few false alarms [9]. In our previous work, we have shown that neural networks are effective models for robust intrusion detection.

3. NEURAL NETWORKS FOR SYSTEM CALL ANALYSIS

The use of neural networks for constructing classifiers has become popular in recent years. When compared with other approaches of classifier construction such as template matching, statistical discriminant analysis and rule-based decision trees, neural network models have the advantage of relatively low dependence on domain specific knowledge and have efficient learning algorithms available for classifier training [5].

In the research described in this paper, we have used a classical backpropagation neural network as our classifier.

Different types of neural networks were evaluated in our previous work including backpropagation neural networks, radial basis function networks, and self-organizing maps (SOM) [9]. The number of hidden nodes in the network determines the computational overhead when using a network to classify new instances. Our previous results indicate that the backpropagation model has accuracy similar to that of a radial basis function network with many fewer hidden nodes. Both outperformed the SOM.

We use a sliding window to divide a trace of calls (a sequence of calls from one run of a program) into a set of small sequences of calls. For example, suppose we had a normal trace consisting of the following sequence of calls:

execve, brk, open, fstat, mmap, close, open, mmap, munmap and a window size of 4. We slide the window across the sequence, and for each call we encounter, we record the calls that precedes it at different positions within the window, numbering the calls from 0 to $w-1$, with 0 being the current system call. The trace above yields the following instances:

position 3	position 2	position 1	current
			<i>execve</i>
		<i>execve</i>	<i>brk</i>
	<i>execve</i>	<i>brk</i>	<i>open</i>
<i>execve</i>	<i>brk</i>	<i>open</i>	<i>fstat</i>
<i>brk</i>	<i>open</i>	<i>fstat</i>	<i>mmap</i>
<i>Open</i>	<i>fstat</i>	<i>mmap</i>	<i>close</i>
<i>fstat</i>	<i>mmap</i>	<i>close</i>	<i>open</i>
<i>mmap</i>	<i>close</i>	<i>open</i>	<i>mmap</i>
<i>close</i>	<i>open</i>	<i>mmap</i>	<i>munmap</i>

We use a binary representation of the identifiers assigned of these system calls as the input for the neural network. There are about 250 different system calls in the Linux system. Therefore, if we define a window size of 10, there are 80 inputs for the neural network.

Because we seek to determine whether an input string is anomalous or normal, we use a single output node to indicate the status of the inputs string with a value of 0 indicating normal and a value of 1 indicating anomalous. We use a single hidden layer in our network. The appropriate number of hidden nodes was determined empirically. Network weights are initialized to random values prior to training. These random initial weights can have a large and unpredictable effect on the performance of a trained network. In order to avoid poor performance due to bad initial weights, we trained 10 networks with different initial weights for each number of hidden nodes. We then tested each network with test data and retained the best network, discarding the others.

In order to train the networks, it is necessary to expose them to both normal data and anomalous data. For the experiments described in this paper, we have used artificial anomalies that were randomly generated for training data in a manner similar to that described by Ghosh, et al. [4].

4. DATASETS

We have used one data set from the University of New Mexico repository (traces of *sendmail*) and one dataset from our laboratory that traces a parallel program under attack [15].

Synthetic data for *sendmail* were collected at UNM on Sun SPARCstations running unpatched SunOS 4.1.1 and 4.1.4 with the included *sendmail*. There are a total of 640 processes in the normal data. We randomly chose 330 of the processes as the normal training data. This data set also includes 16 attack traces for five different kinds of intrusions. The attacks appearing in the data are described as follows [15]:

- sunsendmailcp intrusion: The sunsendmailcp (sscp) script uses a special command line option to cause *sendmail* to append an email message to a file. By using this script on a file such as */.rhosts*, a local user may obtain root access. (3 trace instances)
- decode intrusion: In older *sendmail* installations, the alias database contains an entry called "decode," which resolves to *uudecode*, a Unix program that converts a binary file encoded in plain text into its original form and name. *uudecode* respects absolute filenames, so if a file "bar.uu" says that the original file is *"/home/foo/.rhosts"* then when *uudecode* is given "bar.uu", it will attempt to create *foo's .rhosts* file. *sendmail* will generally run *uudecode* as the semi-privileged user daemon, so email sent to decode cannot overwrite any file on the system; however, if the target file happens to be world-writable, the decode alias entry allows these files to be modified by a remote user. (2 trace instances)
- error condition - forwarding loops: A local forwarding loop occurs in *sendmail* when a set of *\$HOME/.forward* files form a logical circle. (5 trace instances)
- syslogd intrusion: The syslogd attack uses the syslog interface to overflow a buffer in *sendmail*. A message is sent to the *sendmail* on the victim machine, causing it to log a very long, specially created error message. The log entry overflows a buffer in *sendmail*, replacing part of *sendmail's* running image with the attacker's machine code. The new code is then executed, causing the standard I/O of a root-owned shell to be attached to a port. The attacker may then attach to this port at his or her leisure. This attack can be run either locally or remotely; UNM tested both modes. They also varied the number of commands issued as root after a successful attack. (4 trace instances)
- unsuccessful intrusions - sm5x, sm565a: These are attack scripts for which SunOS 4.1.4 has patches. (2 trace instances)

We are currently investigating the use of sequences of system calls to detect intrusions in parallel programs

running on a cluster of Linux workstations. The workstations within these clusters exchange information not only through TCP/IP networks, but also through special high-speed switches that facilitate rapid communication. We have described and implemented several scenarios that could potentially be used to mount attacks against a Linux high performance cluster [10]. We used traces of attacks on a parallel MPI ring program to test trace classification methods using neural networks.

The MPI ring program has two major arguments: the data size for each data transfer and the number of send operations for each loop. The program was executed several times using different values for the parameters and data was collected in each node. Four machines were used for each execution of the program. Each program trace in a machine can be viewed as a normal run of the MPI program. We implemented two daemon attacks on this ring program (called *ringFork* and *ringFile*). These attacks spawn a daemon process that consumes system memory or disk resources. The behavior of the attacks was varied for the runs by changing parameters such as the size of data written to disk or the number of write operations. We also ran the MPI ring program with a malicious interposition library attack under different conditions. Sometimes, the attack caused the MPI program to crash or freeze. This attack randomly changes the communication rank and prevents MPI processes from communicating with each other correctly. Table 1 gives detailed statistics about the MPI ring dataset. More detail about these attacks can be found in [10].

Table 1 Statistics of MPI ring dataset

	Normal ring	ringFile	ringFork	Interposition Lib
Num Of Runs	13	9	2	5
Num of Traces	85	9	2	20
Num of Processes	255	45	2056	80
Syscall per Pro. Min-Max	22-20818	12-18701	5-4039	5-8343

In the *ringFile* and *ringFork* attacks, only one of the four machines is chosen to spawn the daemon process. The other three machines represent normal runs. Therefore, the total number of normal traces is equal to 85 ($13*4+(9+2)*3$). Each trace of a normal ring program contains several processes. Some are used for communication and others primarily perform computations. This depends on the design of the MPI library. *RingFork* generates many processes that do little work. The traces of most of these processes have fewer

than 10 system calls. Since a window size of 10 was used during our experiments, it is impossible to classify these traces. However, as shown in section 7, the attack can still be detected from the main process which spawns the daemon process.

5. TRACE CLASSIFICATION METHODS

Our neural network classifies each sliding window sequence as normal or anomalous. However, the goal is to classify an entire program trace as normal or anomalous. We introduce several different methods for online and offline detection of program traces. In offline detection, all of the system calls that a program generated for one complete run is available at the time of classification. For online detection, we only know the calls that the program used before the current call; we do not know which calls will follow the current one. Therefore online and offline detection require different techniques for classification. The four methods we have used for online and offline detection are described below. All are based on classification of sliding window sequences.

Offline:

- Vector Count: Unique sequences are extracted from the traces and classified. A threshold is defined for the total number of anomalous sequences needed to fire the alarm.

Online:

- Total Anomalies: The number of anomalous sequences within the trace is counted. If the number becomes greater than a threshold an alarm is fired; otherwise it is classified as normal.
- Max Cons Anomalies: The number of consecutive anomalous sequences is counted. When the maximum number of consecutive anomalies becomes larger than a threshold, an alarm is generated.
- Max Burst Counter: A variable called BC (burst counter) is maintained that remembers the total number of anomalous sequences seen so far. After encountering a normal sequence, the value of the variable is decreased at a slow rate. This is similar to Ghosh et al.'s leaky bucket method [4]. Since the output of our neural network is either 0 or 1 and theirs is a continuous number, we cannot use the leaky bucket method directly. Our method has an effect similar to the LFC method of Somayaji [12] but with much less computational overhead. The assumption of Max Burst Counter is that an anomalous sequence seen long ago should only have a small effect on classification.

Other researchers have reported that anomalous sequences tend to occur in clusters [4][6][8][12][14]. The advantage of using MaxBurstCounter is that it allows occasional anomalous behavior that is to be expected

during normal system operation. However it is quite sensitive to large numbers of temporally co-located anomalies which one would expect if a program were really being misused. Though anomalous sequences tend to occur locally, they are not necessarily continuous. MaxBurstCounter represents the locality characteristic of anomalies without the requirement that they are consecutive.

6. ROC CURVES FOR ANALYSIS OF CLASSIFIER PERFORMANCE

When building an intrusion detection system, the key component is the analysis module. We have used Receiver Operating Characteristic (ROC) curves to evaluate the performance of the neural network classifier with different trace classification techniques [2].

ROC curves allow one visualize a classifier's performance over a variety of decisions thresholds. The ROC curve is a plot of the true positive rate against the false positive rate for different possible thresholds for a classifier. These curves provide an effective means of evaluating the tradeoff between sensitivity (true positive rate) and specificity (false positive rate). An ideal ROC curve follows the left-hand border and top border of the ROC space. The worst case performance is a curve that follows the 45 degree diagonal. A higher area under the ROC curve indicates higher overall accuracy and robust performance for a variety of decision thresholds.

7. EXPERIMENTAL RESULTS

We have evaluated different trace classification methods for a neural network classifier using the two data sets described above.

In our experiment with the *sendmail* data set, we randomly chose 330 processes from the normal data for training. The remaining 310 processes are used as testing data. All attack traces were used for testing. We extracted 1249 vectors from the training data and 1130 vectors from test data. Vectors representing anomalous behavior were generated randomly to provide additional training data. There are 254 "normal" vectors that appear in the test data, but not in the training data. This means that some unseen normal behavior is present in the test data. This matches the fact that in the real world program behavior is variable and it is not typically possible to obtain a training data set that includes all the possible behavior of a program.

Figure 1 shows the results using different trace classification methods. We can see that using either MaxBurstCounter or MaxConsAnomalies yields better results than using TotalAnomalies. This confirms results by other researchers that indicate that anomalous sequences tend to occur in clusters.

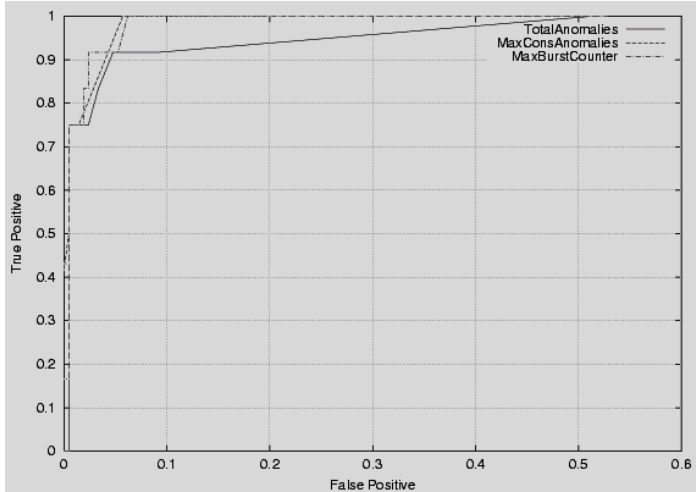


Figure 1 ROC curves with *sendmail* dataset

For our experiment with the ring data set, we trained the neural network with real normal data and artificially generated random sequences of sequence calls for anomalous data. Four runs of the ring program were selected as the training set. Since the program was executed on 4 machines, each run contains 4 traces for a total of 16 traces used for training. The remaining 69 normal traces were used as test data. All attack traces were used in testing. Table 2 shows the results with a normal trace that was not used in training. The results for MaxBurstCounter shows the maximum value of the burst counter variable during the trace while those for MaxConsAnomalies show the maximum number of anomalous sequences that occurred continuously.

Table 2 Comparison of online classification methods with unseen normal run

Pid	Total Anomalies	MaxBurst Counter	MaxCons Anomalies	Total calls
932	1	5	1	9198
933	4	18	3	26
934	1	5	1	6813

Table 3 Comparison of online classification methods with *RingFile* attack

Pid	Total Anomalies	MaxBurst Counter	MaxCons Anomalies	Total calls
1303	9	30	6	5596
1304	3	15	3	22
1305	4	9	1	3714
1306	3	14	2	15
1307	24	102	3	54

Table 3 shows the results with a *RingFile* attack. In the *RingFile* attack, a daemon process is generated after the MPI program finishes. Therefore in Table 3, processes 1306 and 1307 are the real abnormal processes. Process

1303 contains the call of the daemon function to generate the daemon process. Therefore, it also represents an anomaly. Figure 2 shows the distribution of anomalous sequences in process 1303. The anomalous sequences are clustered at the end of the run where the daemon process is generated.

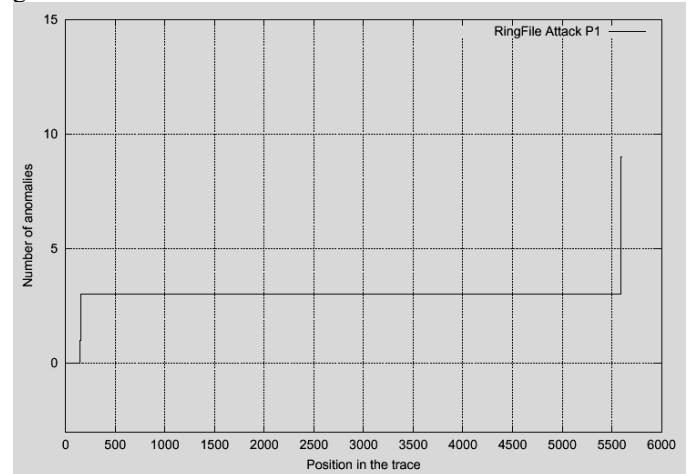


Figure 2 Anomalies distribution in first process of ring program under *RingFile* attack

Figure 3 shows the results using different trace classification methods. We varied the threshold in each method to generate the ROC curve. The off-line VectorCount method has the best performance, since it contains all the information for the entire trace. The results in Figure 3 show that the MaxBurstCounter method yields a lower false positive rate than the other two online methods, but its true positive rate improves much slowly than the other two with the increasing false positive rate. The VectorCount off-line method exhibits the best performance.

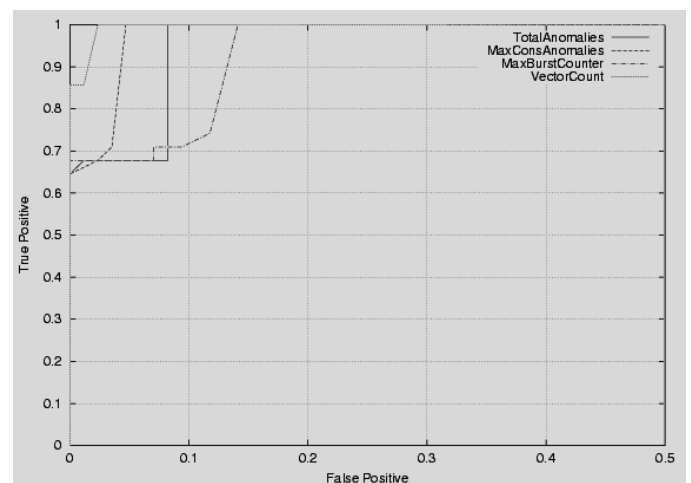


Figure 3 ROC curve of pure anomaly detection with different online and offline trace classification methods with ring program dataset

8. CONCLUSIONS

We have compared different trace classification methods based on neural network classifications of sliding window sequences. The MaxBurstCounter method gives the highest true positive rate when the false positive rate is restricted to 0%. However, as the number of false positives is increased, the MaxBurstCounter method is a little slower to converge to a 100% detection rate than the MaxConsAnomalies method. The performance of TotalAnomalies is substantially worse than that of the other two on-line methods. This method is dependent on the length of the trace and does not take the locality of anomalies into account. Although the VectorCount method is very accurate for off-line detection, it is particularly useful for real-time or near real-time detection.

Acknowledgements

This work was partially sponsored by National Science Foundation Grants CCR-0085749 and CCR-9988524 as well as Army Research Laboratory Grant DAAD 17-01-C-0011 and Office of Naval Research Grant N00014-01-1-0678.

REFERENCES

- [1] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, Inc. Sebastopol, California, 2001.
- [2] A. P. Bradley, "The Use of the Area under the ROC Curve in the Evaluation of Machine Learning Algorithms," *Pattern Recognition*, vol. 30, no. 7, July 1997, pp. 1145-1159.
- [3] W. W. Cohen, "Fast Effective Rule Induction," *Proceedings: 12th International Conference on Machine Learning*, Lake Tahoe, California, 1995, pp. 115-123.
- [4] A. K. Ghosh, A. Schwartzbard, and M. Schatz, "Learning Program Behavior Profiles for Intrusion Detection," *Proceedings: 1st USENIX Workshop on Intrusion Detection and Network Monitoring*, Santa Clara, California, 1999, pp. 51-62.
- [5] S. Haykin, *Neural Networks A Comprehensive Foundation*, 2nd Edition, Prentice-Hall, Upper Saddle River, New Jersey, 1999.
- [6] S. A. Hofmeyr, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal of Computer Security*, vol. 6, no. 3, 1998, pp. 151-180.
- [7] T. Kohonen, J. Hynninen, J. Kangas, J. Laaksonen, and K. Torkkola, "LVQ_PAK: The learning vector quantization program package version 3.1," http://www.cis.hut.fi/research/som_lvq_pak.shtml (current 15 Oct 2001).
- [8] W. Lee and S. J. Stolfo, "Data Mining Approaches for Intrusion Detection," *Proceedings: 7th USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 79-94.
- [9] Z. Liu, G. Florez and S. M. Bridges. "A Comparison of Input Representations in Neural Networks: A Case Study in Intrusion Detection," *Proceedings: International Joint Conference on Neural Networks (IJCNN)*, Honolulu, Hawaii, 2002, (published as CD).
- [10] M. Torres, R. B. Vaughn, G. Florez, Z. Liu, S. M. Bridges, 2003, High Performance Linux Cluster Attack Scenarios, submitted for publication.
- [11] A. Somayaji, "Automated Response Using System-Call Delays," *Proceedings: 9th USENIX Security Symposium*, Denver, Colorado, 2000, pp. 185-197.
- [12] A. Somayaji, *Operating System Stability and Security through Process Homeostasis*, doctoral dissertation, Department of Computer Science, University of New Mexico, 2002.
- [13] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proceedings: IEEE Symposium on Security and Privacy*, Oakland, California, 2001, pp. 156-169.
- [14] C. Warrender, S. Forrest, and B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *Proceedings: IEEE Symposium on Security and Privacy*, Los Alamitos, California, 1999, pp. 133-145.
- [15] Datasets. <http://www.cs.unm.edu/~immsec/systemcalls.htm>