1-1-2003

A Lightweight Intrusion Detection System for the Cluster Environment

Zhen Liu

A LIGHTWEIGHT INTRUSION DETECTION SYSTEM FOR THE CLUSTER ENVIRONMENT

By

ZHEN LIU

A Thesis
Submitted to the Faculty of
Mississippi State University
in Partial Fulfillment of the Requirements
for the Degree of Master of Science
in Computer Science
in the Department of Computer Science and Engineering

Mississippi State, Mississippi

August 2003

A LIGHTWEIGHT INTRUSION DETECTION SYSTEM FOR

THE CLUSTER ENVIRONMENT

By

Zhen Liu

Approved:

Susan Bridges Professor of Computer Science and Engineering (Director of Thesis and Graduate Coordinator of the Department of Computer Science and Engineering)	Julian E. Boggess Associate Professor of Computer Science and Engineering (Committee Member)
Rayford Vaughn Associate Professor of Computer Science and Engineering (Committee Member)	A. Wayne Bennett Dean of the Bagley College of Engineering

Name: Zhen Liu

Date of Degree: August 2, 2003

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Susan M. Bridges

Title of Study: A LIGHTWEIGHT INTRUSION DETECTION SYSTEM FOR THE CLUSTER ENVIRONMENT

Pages in Study: 78

Candidate for Degree of Master of Science

As clusters of Linux workstations have gained in popularity, security in this environment has become increasingly important. While prevention methods such as access control can enhance the security level of a cluster system, intrusions are still possible and therefore intrusion detection and recovery methods are necessary. In this thesis, a system architecture for an intrusion detection system in a cluster environment is presented. A prototype system called pShield based on this architecture for a Linux cluster environment is described and its capability to detect unique attacks on MPI programs is demonstrated.

The pShield system was implemented as a loadable kernel module that uses a neural network classifier to model normal behavior of processes. A new method for generating artificial anomalous data is described that uses a limited amount of attack data in training the neural network. Experimental results demonstrate that using this method rather than randomly generated anomalies reduces the false positive rate without

compromising the ability to detect novel attacks. A neural network with a simple activation function is used in order to facilitate fast classification of new instances after training and to ease implementation in kernel space.

Our goal is to classify the entire trace of a program's execution based on neural network classification of short sequences in the trace. Therefore, the effect of anomalous sequences in a trace must be accumulated. Several trace classification methods were compared. The results demonstrate that methods that use information about locality of anomalies are more effective than those that only look at the number of anomalies.

The impact of pShield on system performance was evaluated on an 8-node cluster. Although pShield adds some overhead for each API for MPI communication, the experimental results show that a real world parallel computing benchmark was slowed only slightly by the intrusion detection system. The results demonstrate the effectiveness of pShield as a light-weight intrusion detection system in a cluster environment. This work is part of the Intelligent Intrusion Detection project of the Center for Computer Security Research at Mississippi State University.

ACKNOWLEDGEMENTS

I am deeply grateful to Dr. Susan Bridges for directing my graduate study and research work during last two years. In this research, she has often given me insights and guided me back to the right path. She always encouraged me to learn new things and think independently. I would like to thank Dr. Lois Boggess for her valuable neural networks course and useful advice. I also want to express my appreciation to my committee members, Dr. Rayford Vaughn and Dr. Julian E. Boggess, for their invaluable aid, direction, and ideas in this area. A special thanks to German Florez and Miguel Torres for their help and insightful discussion. I would also thank other faculty members and students in the Center for Computer Security Research (CCSR).

TABLE OF CONTENTS

CHAPTER Page

LIST OF TABLES

LIST OF FIGURES

FIGURE Page

Chapter I

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. Many different techniques and tools have been designed to enforce computer security. They fall into three general categories, prevention (firewall, cryptography, etc.), detection (IDS, Tripwire, etc.), and reaction (disk backup, automatically changing firewall policy).

Intrusion detection is an essential and critical component of modern computer systems. One of the reasons is that it is not technically feasible to build a system without any vulnerability [9]. It is also very difficult to test the security capabilities of a system since it is almost impossible to anticipate all intrusion patterns. In addition, attackers sometimes use completely unknown patterns that are unexpected and difficult to detect. Intrusions can also originate from authorized system users who choose to abuse their access rights and circumvent protection mechanisms [9].

Jain and Sekar [25] list several methods that an intruder can use to implement an attack.

- Exploit software errors in privileged programs to gain root privilege.
- Exploit vulnerabilities in the system configuration to access confidential data.

- Rely on a legitimate system user to download and run a Trojan horse program that inflicts damage.

Many techniques have been developed over the past several years to protect against malicious attacks. These techniques use audit data such as TCP/IP packet data, TCP/IP connection data, statistical data about CPU usage, number of processes a user created, or the bash command log of users. Recently some researchers have tried to find more resources that can be used to detect intrusions, including "the number of references to particular memory locations, the time spent executing different parts of a program, the frequency and quantity of communication among nodes of a multiprocessor system, and I/O resource usage" [33]. Moreover, Forrest and Hofmeyr [15] used sequences of system calls invoked during execution of a program to build normal profiles for privileged programs. Their techniques are based on the assumption that a system or a program under attack will exhibit changes in behavior and that these changes can be observed by looking at the sequences of system calls used by these processes. Other researchers [1, 25] have also shown that the damage caused by an attack can ultimately be inflicted via system calls. "It is thus possible to identify (and prevent) damage if we can monitor every system call made by every process, and launch actions to preempt any damage, e.g., abort the system call, change its operands (e.g., open a different file from one that is specified) or even terminate the process" [25].

Process-based intrusion detection has become the focus of recent intrusion detection research, although user-based and connection-based intrusion detection continues to have an important role in intrusion detection techniques [20]. Process-based

monitoring intrusion detection tools analyze the behavior of executing processes to detect possible intrusive or misused activity. “The premise of process monitoring for intrusion detection is that most computer security violations are made possible by misusing programs” [19]. Based on the assumption that a program’s behavior will differ from its normal usage if a program is misused, intrusion detection can be applied by observing the behavior of the program. A compact representation of the behavior of a program must be developed to effectively distinguish the normal and abnormal behavior. “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” [19]. The second is the method used by Forrest and Hofmeyr [15].

Forrest and Hofmeyr [15] demonstrated that the representation of a program’s behavior using sequences of system calls provides a powerful approach for detecting intrusions. This thesis extends their idea to the detector of anomalies in a cluster environment. A new real-time intrusion detection prototype system architecture is presented and the ability of a neural network to analyze system call data is explored and different methods to generate artificial anomaly data for training are compared. The remainder of this chapter explains the motivation for this work and summarizes the contributions of this thesis. Finally, an outline of subsequent chapters is presented.

1.1 Motivation

As computer systems have become increasingly complex, they have also become more unpredictable and unreliable. Today dozens of programs are run on any given

computer. This situation becomes worse in a cluster environment where it is difficult to correctly configure a cluster environment that involves more and more software participation. Each of the software components may have its own vulnerability. New vulnerabilities are found almost every day on most major computer platforms. Configuration errors can prevent components from working together correctly. Moreover, network failure and CPU misuse may lead an entire system into abnormal behavior. Many monitoring tools have already been designed to help system administrators identify system failures. However, with the increasing size of clusters, it is difficult for the administrator to monitor all of the hosts. Therefore, we need a tool to automatically detect abnormal behavior of the system.

1.2 Contributions

In Somayaji's work [39], he presented a method for learning program behavior by analysis of system calls and demonstrated that this method is capable of detecting buffer overflows, Trojan horse code, and kernel flaws. The work described in this thesis extends Somayaji's work by using a different method to collect the system call log and by using neural networks to learn a normal profile for a program from the system call traces. The method has also been ported to a cluster environment and its capability to detect unique attacks on MPI programs is demonstrated. A new method for generating artificial anomalies is presented and its effectiveness in improving the performance of the neural network classifier is shown. Several methods for monitoring the alarm level of a program trace are also compared.

System calls provide a rich source of information about the behavior of a program. In this thesis, we demonstrate the capability to detect unique attacks on an MPI program in real-time by using a built-in-kernel neural network to analyze the sequence of system calls. A prototype system has been built in a Linux cluster in the Department of Computer Science and Engineering, Mississippi State University. The prototype system consists of two main components: a data collection and response module (LKM) and a detection module (neural network).

In SunOS and Solaris, BSM can be used to generate the system call log. But in other operating systems there are no such tools to collect the system call logs. Researchers have conducted a lot of work in this area. Some of them change the kernel source code, while others use debugging mechanisms provided by the operating system to obtain logs at the user level [25]. In this thesis, we use a loadable kernel module (LKM) to fulfill this task.

A kernel module is used to collect the system call log. The kernel module is a loadable kernel module that intercepts system calls to perform pre-call and post-call processing. Our module generates a system call log and can provide an additional layer of fine-grained security control. Its key features are that it can be set up to be non-bypassable since it is in the kernel, and it is easy to install requiring no modification to the kernel or to the applications that are being monitored.

We use neural networks to build the detection module. The use of neural networks for constructing classifiers has become popular in recent years. Compared with other approaches to classifier construction such as template matching, statistical analysis and

rule-based decision trees, neural network models typically have the advantages of relatively low dependence on domain specific knowledge and efficient learning algorithms available for classifier training [24]. Ghosh, Schwartzbard, and Schatz have shown that neural networks can be used to analyze sequences of system calls [20]. We show that by using attack patterns derived from known attacks, the performance of the classifier can be improved without losing the capability to detect novel attacks.

A cluster is a group of workstations in which each node has its own operating system and cooperates with other nodes by communicating on a high speed or normal TCP/IP network. A cluster faces the same problems of susceptibilities to attack and system failure as traditional systems. We implemented our prototype system in a cluster environment and demonstrated its ability to detect attacks on MPI programs. Since our method is based on learning a program's normal behavior, this system is also able to detect abnormal behavior caused by system errors, hardware failures or program misuse.

1.3 Overview

The goal of this thesis is to demonstrate effective methods that can be used to create a real-time intrusion detection system in a cluster environment. A prototype system called pShield has been designed and implemented to demonstrate the capability of our methods. The remainder of this thesis proceeds as follows. Chapter 2 introduces the work that other researchers have done in this area. Chapter 3 gives the high level design of our system. The system architecture and important data structures are also described. Chapter 4 describes different methods used in building and training the detection module and

evaluates the system's performance. Different trace classification methods are described and compared. A new algorithm to generate artificial anomalies is introduced and compared with other methods. The overhead added to the operating system by pShield is also evaluated. Chapter 5 summarizes our work, analyzes its strengths and shortcomings, and presents ideas for future work.

Chapter II

Literature Review

Using sequences of system calls as the data source for an intrusion detection system has been shown to be a good way to detect many attacks. This method can detect buffer overflows, format string attacks, Trojan horse programs, and symbolic link attacks [26]. A great deal of work has been done in this area since it was first introduced by Stephanie Forrest and her colleagues [15]. Researchers have used different methods to obtain the system call log from the operating system and different algorithms to analyze the data. Markov chain models were shown to yield high accuracy but training was reported to require days [43]. The Stide algorithm is fast and its performance is as good as other more complicated learning algorithms [15]. The Ripper, a rule learning method, provides a good tradeoff in accuracy and efficiency [28]. A number of different IDS models have been used. Some perform misuse detection while others do anomaly detection. Different models have different characteristics.

2.1 Intrusion detection

Intrusion detection can be defined as the detection of outside intruders “who are using a computer system without authorization” and inside intruders “who have

legitimate access to the system but are abusing their privileges" [32]. Intrusion detection systems are usually built to identify unauthorized behaviors of outside or inside intruders and to enforce the security policy of computer systems.

Intrusion detection systems can be classified in several ways. One classification divides the systems into host-based intrusion detection systems and network-based intrusion detection systems. The main difference between these two types of systems is that they deal with different data sources. In host-based intrusion detection systems, the audit data includes information such as I/O activity, CPU usage of a program, CPU usage of a user session, bash commands used by a user, and resources used by a program (signal, pipe, memory etc.). Network-based intrusion detection systems process TCP/IP audit data including TCP/IP header data and TCP/IP packet content data. IDSs can also be classified as misuse detection or anomaly detection systems, based on their data analysis models. Misuse detection is based on the knowledge of system vulnerabilities and known attack patterns, while anomaly detection assumes that an intrusion will always reflect some deviations from normal patterns. The advantages of misuse detection are the potential for low false alarm rates, and the information they are able to impart to a system security officer about a detected attack. Such information is often encoded in the rules or patterns central to the functionality of such systems. However, misuse detection has several disadvantages. Since the set of anomalous patterns is based on known attacks, new attacks cannot be discovered and patterns corresponding to the attack must be manually constructed. Moreover, a sophisticated attacker can easily fool a misuse detection system by using a stealthy attack [8]. Anomaly detection is an approach to

address the shortcomings of misuse detection. The IDS constructs a normal profile for each object (a user, a session, a program, CPU usage etc). Normal profiles have been represented by several methods including a rule base [28], a classifier [26] or a finite state machine [37]. However, most commercial products are misuse intrusion detection systems, because anomaly detection often gives many false positive alarms and in the real world an administrator cannot accept a high false positive alarm rate. Recent commercial products tend to combine misuse and anomaly detection to obtain better performance.

In order to obtain acceptable performance, an IDS requires not only a good analysis algorithm, but also high quality data for training and evaluation. A learning phase is required for anomaly detection. A good data set can be used to generate an accurate normal profile, which leads to good performance. In misuse detection, high quality data can give more detailed information about normal and intrusive behavior and can help the security expert define useful intrusion patterns. Two well-known data repositories for intrusion detection exist. The DARPA-MIT (http://ideval.ll.mit.edu) data repository includes Tcpdump, BSM, and other system logs which were collected over long period for a real network. The goal of the DARPA-MIT project was to provide a good data source for evaluating different intrusion detection systems [8]. Another dataset is from the University of New Mexico (http://www.cs.unm.edu/immsec/systemcalls.htm) [43]. This data set includes system call data that contains many intrusions such as buffer overflow, symbolic link attacks, and Trojan horse programs.

2.2 System call analysis

On UNIX and UNIX-like systems, user programs do not have direct access to hardware resources; instead, one program called the kernel runs with full access to the hardware, and regular programs must ask the kernel to perform tasks on their behalf [1]. The system call interface is the only way that a user space program can communicate with the kernel. “Intrusion detection and prevention could be achieved if the OS could monitor every system call made by every process, and prevent malicious or unexpected invocations of system calls from being completed” [1]. A normal profile for each application can be built regarding the usage of system calls. An immediate detection and response system can potentially be implemented by having the operating system monitor the system calls invoked during the execution of the application. This could allow prevention to be conducted before the real damage occurs. Two approaches for confining the system call interface have been described in the literature. A description of these two methods follows.

2.2.1 Access control database

Bernaschi, Gabrielli and Mancini [1] created a security-enhanced operating system called REMUS. The basic idea is to add more access control capability into the original OS. They present a complete classification of the UNIX system calls according to their level of threat. Various threats including user-to-root and denial of service attacks are considered during their system call analysis. The focus of their considerations refers mainly to threats by which an intruder tries to gain direct access to the system as a

privileged user. In their system, they have defined an access control database. The programs are restricted by the rules in the database. The rules define the constraints on the arguments used by system calls. Bernaschi, Gabrielli and Mancini list the following key issues addressed by their work [1].

1. Provide a complete analysis of the critical system calls from the security viewpoint;
2. Detect illegal invocation of critical system calls before they complete so as to prevent attackers from hijacking control of any privileged process;
3. Allow an efficient check of the argument values of the system calls;
4. Implement a secure OS by means of lightweight extensions of the kernel, in particular without requiring changes in existing data structures and algorithms; and
5. Support, thanks to the immediate detection of possible attacks, other extensions of the OS to confine and tolerate intrusive processes running together with legitimate processes. This could allow a safe analysis of the attack while the intrusion is in progress.

The difficulty with this method is defining a good access control database.

2.2.2 Sequence analysis

Instead of looking at the arguments of system calls, Forrest, et al. [15] use the sequence order of system calls to detect anomalies. They ignore everything about the system calls except for their type and relative order.

Consider the following short trace of one run of a program,

open, read, write, close, close.

To learn the profile of this program, Forrest, et al [15] divide this trace into small sequences with a user defined window size. A sliding window is used to slide along the whole trace producing short sequences. For example, if we define a window size of three, the trace above generates the sequences shown in Table 2.1.

After the small sequences are generated, different learning algorithms can be used to create a normal profile for the program. An overview of artificial intelligence methods that have been used in system call analysis is given in next section.

Table 2.1 Sequences of system calls

open	read	write
read	write	close
write	close	close

2.3 Artificial intelligence in system call analysis

Many researchers have used artificial intelligence algorithms to analyze system call logs and build classifiers using these algorithms. An overview of different approaches is given below.

2.3.1 Enumerating sequences

Forrest, Hofmeyr, Somayaji, and Longstaff first reported system call logs as an intrusion detection data source in 1996 [15]. In their paper, they gave a method that depends only on enumerating sequences formed by normal traces and subsequently

monitoring for unknown patterns. Two different methods of enumeration were tried: lookahead pairs and contiguous sequences. They showed that contiguous sequences of some fixed length have better discrimination than lookahead pairs.

In the contiguous sequences method, the unique contiguous sequences extracted from a trace of an application based on a predetermined fixed window size are used to build the normal profile for this application. The sequences are stored as trees to save space and to speed up comparisons. Building such a database requires only a single pass through the data. At classification time, sequences from the test trace are compared to those in the normal database. Any sequence not found in the database is called a mismatch. Any individual mismatch could indicate anomalous behavior, or it could be a sequence that was not included in the normal training data [23].

The learning process of this method is very fast. It just creates a new path in the tree when it finds a new sequence. The learning process needs only one pass through the data. During classification, sequences are compared with the patterns in the tree. When the sequence window is small, this process is very fast; therefore it can be used for real-time detection. The disadvantage of enumerating sequences is its lack of generalization.

2.3.2 Rule learning systems

RIPPER (Repeated Incremental Pruning to Produce Error Reduction) is a rule learning system developed by William Cohen [4]. RIPPER extracts rules from training data that are first order hypotheses. For example, class A :- a1 = x, a2 = y (a1, a2 are attributes in the training set) means class A is chosen if attribute a1 is x and a2 is y.

Lee and Stolfo adapted RIPPER to intrusion detection [28]. Like enumerating sequences, this method also divides the long trace into small sequences according to a user specified window k. The advantage of RIPPER is it is able to generalize the system call sequence information in the training set to a set of concise and accurate rules. Warrender et al. [43] compared several methods of analysis sequence of system calls. They found that the rule sets learned by RIPPER contained 200 to 280 rules with 2 or 3 attributes per rule compared to a set of about 1500 entries generated when using enumerating sequences [28].

2.3.3 Hidden Markov Model

"A Hidden Markov Model (HMM) describes a doubly stochastic process. The states of an HMM represent some unobservable condition of the system being modeled. In each state, there is a certain probability of producing any of the observable system outputs and a separate probability indicating the likely next states" [43]. During learning, the system constructs a finite state machine with a probability on each edge. During classification, the HMM detects if a system call in the test trace requires unusual state transitions and/or symbol outputs.

Warrender, Forrest and Pearlmutter found that a Hidden Markov Model gave better accuracy on average than the RIPPER or enumerating sequences methods, but the computational time was very expensive. "Calculations for each trace in each pass through the training data take $O(TS^2)$, where T is the length of the trace and S is the number of states in the HMM. It often required days to train an HMM" [43].

2.3.4 Program specification

Wagner and Dean [42] showed how static analysis might be used to automatically derive a model of application behavior. The idea is to perform static analysis on the source code of the program and use some specification language to model the program's normal behavior. This method can accurately capture the characteristics of a program. It not only checks the sequence of calls, but also checks the arguments of these calls. The result of static analysis is a host-based intrusion detection system with three advantages: "a high degree of automation, protection against a broad class of attacks based on corrupted code, and the elimination of false alarms" [42]. However, the authors do not present any experimental results, so the performance of this method cannot be judged.

2.3.5 Inductive sequential patterns

The Time-based Inductive Machine (TIM) has been proposed by Teng, Chen and Lu [40] to learn sequential patterns automatically from audit data for real-time anomaly detection. The original system they designed was not to process system call events, but it could be easily adapted to this area. The format of the sequential rules inferred from audit trails by TIM can be illustrated with the following example: $A - B \rightarrow (C = 90\%; D = 10\%)$. This rule is interpreted to mean that if event A is directly followed by event B, then the next event will be C or D with the probabilities of 90% and 10%, respectively. Then any event sequence that does not match the normal sequential patterns inductively learned by TIM will be marked as an anomaly.

The main advantage of introducing an inductive learning mechanism to anomaly detection is that sequential patterns can be learned automatically and updated adaptively. This allows new audit data to be used to train the system to find new normal patterns.

2.3.6 Neural networks

Ghosh, Schwartzbard and Schatz [20] 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. The authors compared their method with the Enumerating Sequences method and demonstrated that the neural networks have better generalization on the data.

2.4 System call capture

All the approaches mentioned above are “based on the following observation about attacks: regardless of the nature of an attack, damage can ultimately be affected only via system calls made by processes running on the target system. It is thus possible to identify (prevent) damage if we can monitor every system call made by every process, and launch actions to preempt any damage, e.g., abort the system call, change its operands or even terminate the process” [25]. Several methods can be used to intercept the system calls in a UNIX system.

2.4.1 Library interposition

The library interposition method places a new or different library function between the application and its reference to a library function. “This technique allows a programmer to intercept function calls to code located in shared libraries by directing the dynamic linker to first attempt to reference a function definition in a specified set of libraries before consulting the normal library search path” [6]. This is useful for testing new libraries or for inserting debugging code. For a detailed description refer to [6].

On Solaris and Linux, a shared object can be interposed by setting the LD_PRELOAD environment variable before the execution of a program that we want to be interposed. When a function call is made that is undefined in the application, the dynamic linker will first check for definitions of this function in the objects listed in the LD_PRELOAD variable, and then check the usual library search path.

This approach has the benefit of being easy to implement. An important drawback of this approach is that these wrapper functions can be bypassed. For instance, it is possible for a program to directly invoke a system call using a lower level mechanism such as the assembly command INT 0x80 in x86 Linux.

2.4.2 Debug mechanism

Almost all versions of UNIX provide a mechanism for one process to trace and control the execution of another process and/or access its memory using the function *ptrace()*. The *ptrace()* function allows a parent process to control the execution of a child process. Its primary use is for the implementation of breakpoint debugging. The child process behaves normally until it encounters a signal (see *signal (5)* in man page), at

which time it enters a stopped state and its parent is notified via the *wait (2)* function. When the child is in the stopped state, its parent can examine and modify its "core image" using *ptrace()*. Also, the parent can cause the child either to terminate or continue, with the possibility of ignoring the signal that caused it to stop (Unix man page).

Goldberg, et al. [21] and Jain and Sekar [25] used this technique to build their monitoring program. The disadvantage of this method is that it adds substantial overhead to the operation of the system. To monitor a process, this method requires two more passes through the kernel. Another disadvantage is that the monitor is in user space. Therefore, the function *ptrace()* cannot access important process specific information such as the environment variables. These problems arise because *ptrace()* was designed for debugging.

2.4.3 Kernel methods

In kernel methods, system call interception is implemented within the operating system kernel and all of the extension code runs in kernel mode. This approach has been adopted by several researchers [16, 18, 31]. Extensions that use kernel-level interposition have a broad range of capabilities. "Extensions can provide security guarantees (for example, patching security flaws or providing access control lists), modify data (transparently compressing or encrypting files), re-route events (sending events across the network for distributed systems extensions), or inspect events and data (tracing, logging)" [31]. The major advantage of this method is that it cannot be bypassed. All programs must call the system call interface to access low level functionality implemented by the

kernel. Another advantage is that it does not need to change the code of programs. Only the kernel needs to be changed. The disadvantage is that a security hole in the monitor program is much more dangerous than in other methods. It can cause the system to crash or grant root privilege.

There are two different kernel methods:

1. Kernel patching: Kernel patching patches the kernel source code. Therefore, after patching the user must recompile the kernel. The method is not flexible, but it cannot be circumvented because the kernel image is patched and is loaded into memory at the first start up of the system.
2. Loadable Kernel Module: LKMs are used by the Linux kernel to expand its functionality. An LKM can access all the variables in the kernel and do whatever kernel patching can do. This method is more flexible because it does not require changes to the kernel source code. A privileged user can load it or unload it anytime. The disadvantage is the attacker can install malicious code in the kernel before the tool is loaded. This can be solved by adding the tool into a start script.

2.5 Data preprocessing

2.5.1 Data representation

When we train a neural network using sequences of system calls, we must decide how to encode the data for input to the network. Two possibilities are the following [29]:

1. Divide the system call trace into a fixed length window of size k. Enumerate all observed system call sequences as the encoding based on the window.
2. Again divide the trace into sequences. But compute the frequency of occurrence of different system calls in a window as the encoding.

In previous work [29] comparing these two methods, we found that the first option generally works better.

2.5.2 Artificial anomaly generation

A major difficulty in using machine learning methods for anomaly detection lies in making the learner discover boundaries between known and unknown classes. In order to train neural networks, it is necessary to expose them to both normal data and anomalous data.

Fan et al. [13] have developed a method called distribution-based artificial anomaly generation. Since the exact decision boundary between the known and anomalous instances is not known, they assume that the boundary may be very close to the existing data. “To generate artificial anomalies close to the known data, a useful heuristic is to randomly change the value of one feature of an example while leaving the other features unaltered” [13].

“The training space is divided into different regions. Some regions of known data in the training space may be sparsely populated. Some may be dense” [13]. To amplify sparse regions, Fan et al. [13] proportionally generate more artificial anomalies around sparse regions depending on their density using the algorithm presented in

Figure 2.1. Ghosh et al. [20] also generate artificial anomalies for training neural networks by randomly generating data spread throughout the training space.

Input: D ; Output: D'

1. let F = set of all features of D
2. let V_f = set of unique values of some feature $f \in F$
3. let $D' = \phi$
4. for each $f \in F$:
   - let $countV_{\max}$ = the number of occurrences of the most frequently occurring value in V_f
   - for each $v \in V_f$
     - let $countV$ = the number of occurrences of v in D
     - loop i : $countV < i \leq countV_{\max}$:
       - let d=a randomly chosen datum $d \in D$
       - let v_f = the value of feature f for d
       - replace v_f with a randomly chosen value $v' s.t.\ v' \neq v \wedge v' \neq v_f$ to create d'
       - $D' \leftarrow D \cup \{d'\}$
5. return D'

Note: The algorithm can be modified to take a factor, n, and produce $n \times |D|$ artificial anomalies

Figure 2.1 Distribution-based artificial anomaly generation algorithm (from [13])

2.6 Neural network learning algorithm

Neural networks are weighted directed graphs in which the vertices are artificial neurons and the edges represent the weighted connections between neuron outputs and inputs. “Knowledge is acquired by the network through a learning process. Interneuron connection strengths known as synaptic weights are used to store the knowledge” [22]. Weights are the primary means of long-term storage in neural networks and a learning

phase is used to update the weights and tailor the knowledge stored in the network. “A neural network derives its computing power through its massively parallel distributed structure and its ability to learn and therefore generalize. Generalization refers to the neural network producing reasonable outputs for inputs not encountered during training (learning)” [22]. This property is useful for building intrusion detection systems because it may enable a system to detect unknown attacks while maintaining a low false positive rate.

Back-propagation is probably the most widely used algorithm for generating classifiers and it is often used for benchmarking other learning algorithms [22]. 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 commonly 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. A detailed description of the back-propagation algorithm can be found in [10, 22].

Several key parameters used during construction of back-propagation networks are the learning rate, number of epochs, momentum and the number of hidden nodes.

Different parameter values will cause different performance. In standard back-propagation, a learning rate that is too low will make the network learn very slowly. A learning rate that is too high will make the weights and objective function diverge, so there is no learning at all. In other words, a high learning rate will cause the network to jump around the desired solution and never converge to it. The number of hidden nodes will also affect the capability of the back-propagation network. Too few hidden nodes make the network unable to find the correct function within the training set. Too many hidden nodes make the network learn much more slowly and limit its ability to generalize. The back-propagation learning algorithm may also need regularization to prevent overfitting. Overfitting refers to the problem that occurs when a classifier memorizes the training instances rather than generalizing the mapping from input to outputs. Many regularization methods have been developed to prevent overfitting including early stopping and weight decay [24].

Chapter III

Prototype System Architecture and Implementation

In this thesis, we demonstrate the feasibility of detecting unique attacks in real-time in a cluster environment using an intrusion detection system based on behavior analysis of programs. A prototype system called pShield has been implemented based on the proposed architecture and its ability to detect attacks on MPI programs is analyzed. Since analysis of sequences of system calls has been shown by others to be an effective approach for intrusion detection, we extend this method into the cluster environment. The prototype system has been implemented as a loadable kernel module in Linux and can easily be ported to other Unix-like system. In this chapter we will introduce the architecture of the prototype system and describe how it has been implemented in Linux. The first section introduces the basic functionality of the prototype and discusses why a kernel implementation was chosen. The architecture of the prototype system is described in Section 3.2. Section 3.3 introduces important data structures used in pShield.

3.1 Introduction to the prototype system

Much of recent research in intrusion detection concerns network-based detection. However, network-based intrusion detection systems have difficulty detecting some kinds of attacks that are based on exploitation of security holes in programs. There are two

reasons for using host-based IDSs in addition to network-based IDSs. First, efforts to capture application layer exploits often require substantial overhead for the NIDSs. In a fully saturated network, the NIDSs will lose some packets and the performance will degrade. Second, some attacks will not have a signature in the network packets and cannot be detected by NIDSs. Therefore we need host-based IDSs to detect and respond to such attacks.

Other researchers [15, 19, 20, 23, 26, 28, 37, 38, 39, 43] have already shown that system call analysis can be effectively used to detect attacks such as buffer overflow, Trojan horse, etc. In this thesis, we describe the design of a real-time host-based IDS for the MPI environment which can help an administrator detect both intrusive and abnormal program behavior. The system has two fundamental components: a mechanism that captures system calls and an analysis engine. Both reside in kernel space. A user space program has been created to collect the system call log for training the analysis module. To use the system, a system call log for the program to be monitored must be collected. A neural network analysis module is then trained using this log. Finally, the trained module is loaded into the kernel to monitor the program. Our system supports monitoring of several programs at the same time and programs can be added and removed from the monitoring list at run-time without deleting and reloading the system.

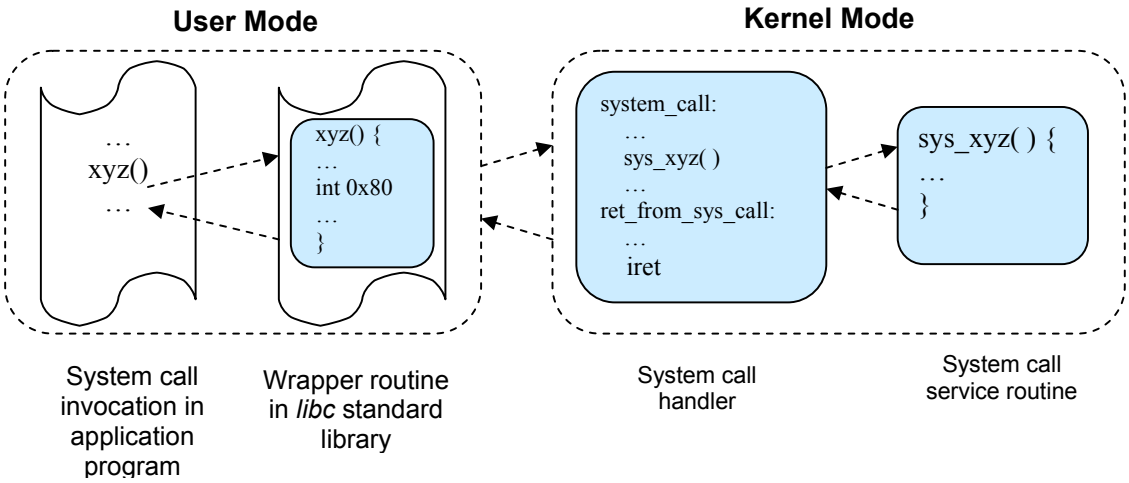
The most important parts of the prototype system reside in the kernel space. In section 2.4, different methods for capturing the system call log were discussed. The *strace* program is a user space utility for debugging that uses the *ptrace()* system call to monitor the system calls of another process. However, other research groups have

reported that *strace* often causes security-critical programs like *sendmail* to crash, and when it worked, *strace* would slow down monitored programs by 50% or more [39]. Interposition of the Libc library is an alternative way to capture the system call log. Somayaji [39] tried this method and showed that it could not detect system calls made by buffer overflow attacks since the "shell code" of such attacks typically makes system calls without using library routines. Therefore the kernel method was determined to be best choice with regard to the performance and effectiveness. Two kernel methods are available. One is kernel patching and the other is LKM. LKM was chosen because it is more flexible than kernel patching. You can easily load and unload the module at any time without recompilation of the kernel. Figure 3.1 compares the invocation of a system call in Linux before and after our pShield system is loaded. Our system adds one more function call than kernel patching. In Chapter 4, results are presented to demonstrate that although pShield adds one more function call, its impact on performance is acceptable.

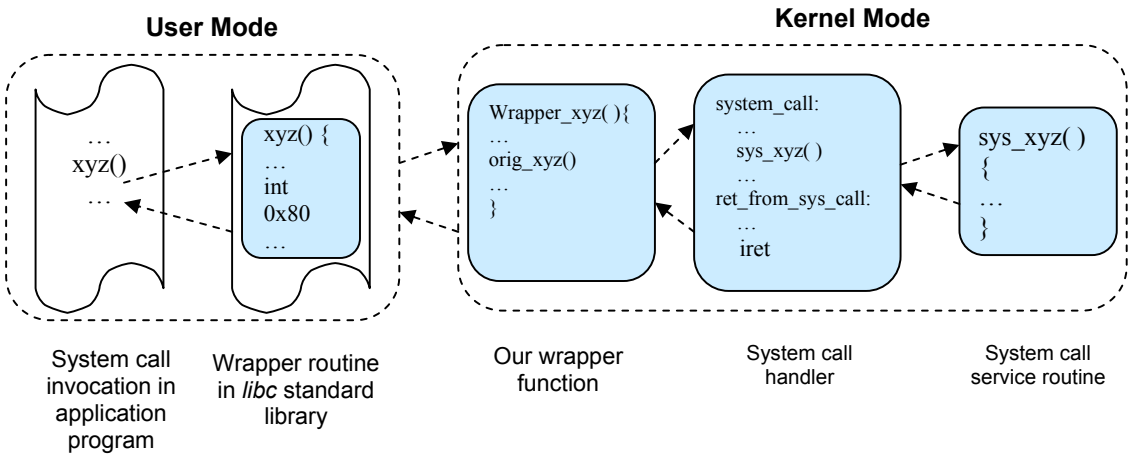
3.2 System architecture

We have implemented pShield on a cluster (known as Microcosm) in the Computer Science and Engineering Department at Mississippi State University. Figure 3.2 shows the structure of this cluster. A sensor was installed on each node and all sensors detect intrusions independently. The final results are combined at the head node. The operating system installed on each node is RedHat 7.1 Linux. The code of pShield was compiled with the 2.4.2 kernel source tree. The MPI environment used in all

experiments was MPI/Pro 1.5. The head node is a 4 CPU SMP computer and the compute nodes are dual CPU SMP computers.



(a) Invoking a system call in original Linux (from [2])



(b) Invoking a system call in Linux with pShield

Figure 3.1 A comparison of a system call invocation (a) without and (b) with pShield loaded

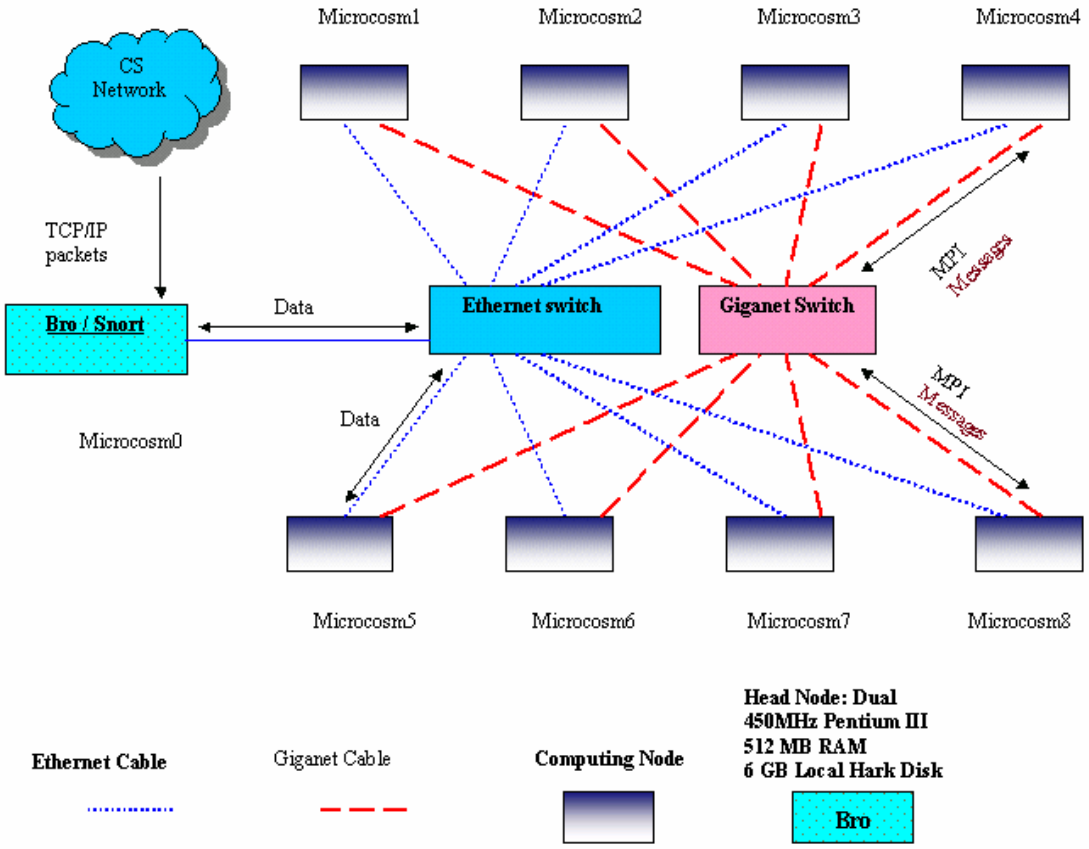


Figure 3.2 Structure of the Microcosm cluster

Figure 3.3 illustrates the design of each sensor. The system combines three components. The first component intercepts the original system call interface. Whenever a user application calls the system call interface, it actually invokes the wrapper function. The wrapper function forwards the system call information to the analysis module. Depending on the result from the analysis module, control flow can be transferred to the original system call or an appropriate response can be generated prior to the return. Different responses can be generated depending on the results of the analysis module.

Currently we just generate an alarm when the analysis module detects an intrusion. More complicated responses can be easily implemented. From Figure 3.1 (b) we can see that code can be inserted before and after the execution of the real system call. It is possible to prevent any possible side effects of system calls that intruders currently exploit for their attacks by means of checks made by the OS kernel before the system call is completed. We can delay or abort the system call or even kill the intrusive process.

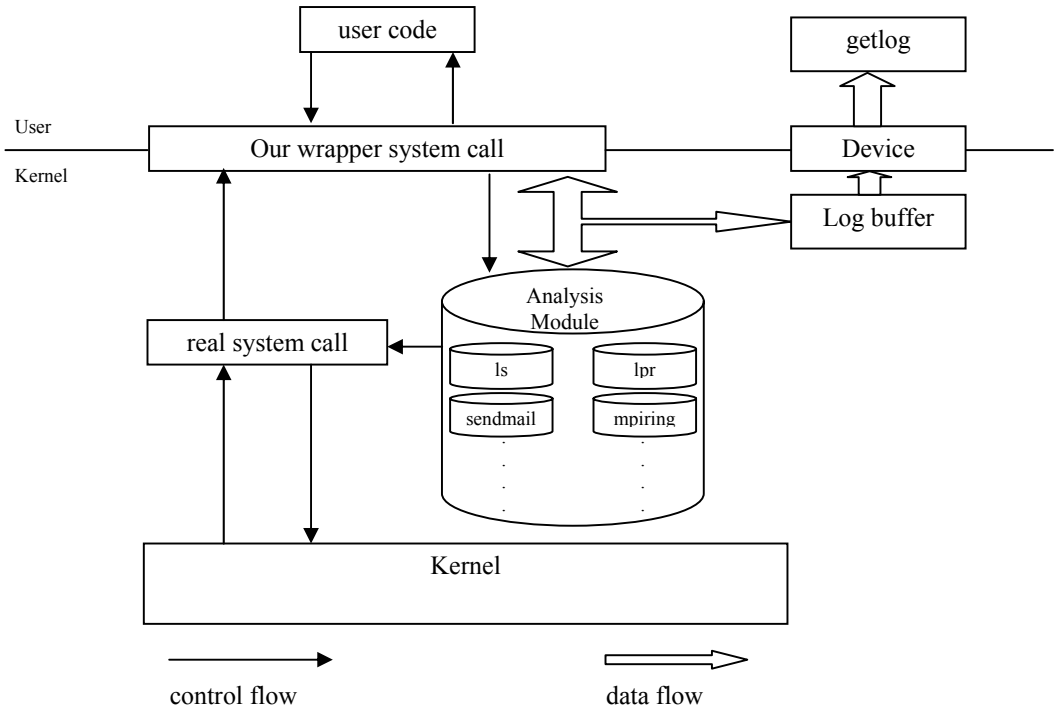


Figure 3.3 System architecture

The analysis module is composed of a set of neural networks for the programs being monitored. Each program profile is composed of a trained neural network, a sliding window size, and a threshold for identifying the anomalies. The analysis module receives the sequences from the interception module and determines whether each is normal or

anomalous. If the user wants to obtain a system call log for training, a user space program is provided that reads system call information from a device we have created and writes the system call log to a disk file. We have chosen to use a device rather than directly writing the log file to disk for two reasons. First, the overhead associated with writing to disk is quite high. Second, it might be possible for an intruder to conduct a denial of service attack by generating a long log that consumes all available disk space.

3.2.1 Data collection module

A system call is implemented in the Linux kernel. When a program executes a system call, arguments are packaged and handed to the kernel, which assumes execution of the program until the call completes. A listing of system calls for a specific version of the Linux kernel can be found at */usr/include/asm/unistd.h*. The system call number is an index in an array of a kernel structure called *sys_call_table[]*. This structure maps the system call numbers to the needed service function. In Linux 2.4.* and earlier kernels, *sys_call_table* is defined as a global variable and can be accessed by any kernel module. We can intercept the system call by modifying the *sys_call_table* and add processing before and after the execution of the original system call.

A ring buffer is maintained by this component. The data collection module feeds the sequences of system calls into this buffer. If the data in this buffer exceeds the buffer size, new data will overwrite old data.

3.2.2 Analysis module

The analysis module combines a group of neural networks. Each neural network records the normal behavior of a program. This module receives system call information used by a program from the data collection module and analyzes it within the execution context of this program.

3.2.3 Log generating program (*getlog*)

The user space program *getlog* is used to read the log from a ring buffer into a log file. Whenever *getlog* reads a data item from the ring buffer, this data item is removed from the buffer so that the data collection module is able to add another log item to the buffer. The user can also use this program to update the profile in the kernel or to update the system configuration. A new device has also been created to enable communication between a user program and the kernel.

3.3 Data structures

In the Linux kernel, processes and kernel-level threads are both implemented in terms of *tasks*. A task is a kernel-schedulable thread of control and is represented internally by a *task_struct* structure. If a task has its own virtual address space, it is a complete, single-threaded process. If a task shares its address space with another task, it is one thread of a multi-threaded process [2]. Like the Linux kernel, our system does not distinguish between processes and threads and instead treats running programs as tasks.

Our system has three fundamental data structures used for storing for the log, the tasks and the profile. The structure *log_buffer_t* is a ring buffer that the system uses to

store the log data. The structure *PROGRAMProfile_t* contains the data for each executable profile. The structure *PROGRAMMonitoring_t* maintains the tasks that are currently being monitored. Together, these hold the data needed for our system to monitor programs and detect intrusions.

```
typedef struct m_entry {
        unsigned int pid;
        unsigned int syscallID;
        char
processImageName[FILENAMEMAXLENGTH];
} log_entry_t;

typedef struct m_log_buffer {
        unsigned int count;
        unsigned int numReferences;
        unsigned int head;
        unsigned int tail;
        log_entry_t logQueue[RINGBUFFERSIZE];
        spinlock_t devLock;
} log_buffer_t;
```

Figure 3.4 Data structure log_buffer_t definition

The ring buffer is shared by all monitored processes. Linux is a nonpreemptive kernel. This means that Linux cannot arbitrarily interleave execution flows while they are in privileged (kernel) mode. In a single processor computer, there is only one process running at a time and the running of a system call cannot be interrupted or interleaved with other kernel programs. Therefore we can run and modify data structures without fear of being interrupted or having another thread alter those data structures. But in an SMP computer, two processes in two different processors may access the buffer at the same time which will cause concurrency problems. Therefore, we define a spinlock to constrain the access to the buffer so that only one process can access the buffer at a time.

```
typedef struct NNLayer {
        int numOfNodes;
        double **weightMatrix;
        double *out;
} NNLayer_t;

typedef struct BPNNClassifier {
        int numOfLayers;
        NNLayer_t *allLayers;
} BPNNClassifier_t;

typedef struct PROGRAMProfile {
        char comm[FILENAMEMAXLENGTH];
        int windowSize;
        int threshold;
        BPNNClassifier_t *classifier;
        struct PROGRAMProfile *nextProfile;
} PROGRAMProfile_t;
```

Figure 3.5 Data structure *PROGRAMProfile_t* definition

Multiple processes may run the same executable, e.g., there may be several running instances of *bash* and *vi* in the system at the same time. Our system maintains one *PROGRAMProfile_t* structure per executable and these are shared among the processes running that executable. Thus, the *PROGRAMMonitoring* for two processes running *vi* will both point to the same profile. Each of the profiles contains a neural network as its classifier and defines the sliding window size and the threshold used to identify abnormal behavior. We use the executable file name to distinguish the profile structure. The structure *BPNNClassifier* is the back-propagation neural network. In cases where some different executables use the same neural network as their classifier, the *PROGRAMProfile* will point to the same classifier. For example, *pico* and *vi*, two similar text editors, may have the same neural network for their profile.

When monitoring a process, we count the anomalous sequences generated by this process. In a Unix-like operating system, *fork()* creates a new process with a different PID and the same executable image name. When a process calls *execve()*, a new process with the same PID but a different executable name is created. Whenever a new process is encountered whose executable name is in the name list that we are monitoring, we create a new *PROGRAMMonitoring_t* record and insert it into the monitoring program list. When a process's PID is in the monitoring program list, we keep using the *PROGRAMMonitoring* structure to test it. When a process calls *execve()*, the situation is much more complicated. The process has a *PROGRAMMonitoring* record in the list, but the name has changed. The problem encountered now is to determine if we should change the profile for this process since the executable name has changed. We manage this with two solutions. First, if the program we are monitoring is a program that seldom calls *execve()* and the neural network has been trained with such *execve()* traces, we view this run as part of the run of the calling process. Therefore we keep using the calling process profile. For programs that often use *execve()* and for which the traces were not included in the training data, we change the profile for this process. In this case, if no profile matches this executable name, pShield generates an alarm to notify the administrator that a process is trying to execute an insecure program. In a buffer overflow attack, *execve()* is usually used to spawn a new *sh* with a higher privilege. With our two solutions, this attack can be detected. Using the first solution, the behavior of this new *sh* will be much different than that of the program profile. Using the second solution, pShield will notify the administrator that an insecure program was executed.

```
typedef struct PROGRAMMonitoring {
        unsigned int pid;
        PROGRAMProfile_t *profile;
        int *currentSeq;
        int *tmpSeq;
        int currentPosition;
        unsigned long totalCalls;
        int totalAnomaly;
        int contAnomaly;
        int bucketAnomaly;
        struct PROGRAMMonitoring *nextProgram;
} PROGRAMMonitoring_t;

typedef struct m_program_list {
        unsigned int numOfProgram;
        //contain the programs' name that we want observe
        char **nameList;
        char **profileList;
} program_list_t;
```

Figure 3.6 Data structure *PROGRAMMonitoring_t* definition

Chapter IV

Experiments and Results

Our prototype system is a host-based intrusion detection system in Linux. In order to demonstrate the effectiveness of our approach we must show that it can effectively detect intrusions with an acceptable performance penalty. Several methods to improve the classifier's accuracy are introduced and experimental results are presented. In section 4.1, we will introduce our experimental environment and attacks on MPI programs that have been monitored. A description of the datasets used in our experiments is presented. Our implementation of a neural network classifier for intrusion detection is described in section 4.2. In section 4.3, we describe different ways to conduct online and offline detection. In order to train a neural network, we need to expose the network to a training space that contains both normal and anomalous instances. Section 4.4 presents several methods we have used to enhance the training space by generating artificial anomalous data. The results of all experiments are presented and discussed in section 4.5. In section 4.6, the performance penalty added to the operating system by pShield is measured and discussed.

4.1 Introduction to experiments

Our prototype system is built in Linux with kernel version 2.4.2. Experiments

were conducted on the cluster illustrated in Figure 3.2. The system design was described in Chapter 3. We installed a sensor on each node of the cluster. Each of the sensors detects intrusions independently. In this thesis, the capability to detect intrusions in a cluster environment is demonstrated by monitoring MPI programs running under normal conditions and under attack.

4.1.1 Attack Scenarios on MPI Programs

In our experiments, we have implemented attacks on MPI programs and shown that our system can effectively detect such attacks. Several attack scenarios for the MPI environment were designed by Miguel Torres (another graduate student at MSU and a member of the CCSR). The attacks used in our experiments can be divided into two major types: daemon attacks and library interposition attacks.

1. Daemon attack: Daemon processes are spawned and left running in memory in the background after an MPI program terminates normally.

   A cluster environment provides rich computational resources with very low cost. The daemon attack we have implemented simulates an attack that allows unauthorized users to steal computational resources from the cluster. This is a Trojan horse-like attack. An attack changes the program executable file and steals computational resources in the cluster. In our experiments, we have run two instances of this attack on the ring program described in section 4.1.3, ringFile and ringFork. Ringfile spawns a daemon, consumes system memory, and writes data to a file. This simulates a computational procedure in which computation is done in background and saved to a file for later reference by

the attacker. RingFork is a denial-of-service attack that keeps spawning new processes in the system and consumes all the memory in the system. This attack will cause the machine to crash some time after the MPI program terminates normally.

2. Library interposition attack: The MPI shared library is intercepted by a new library that contains malicious code. Attacks used in our experiments are the following:
    - MPI_Init and MPI_Finalize: These functions are in charge of starting and terminating MPI library usage. The attack we have implemented randomly generates a daemon process that is left running in the background of the computer memory after the program finishes its normal execution.
    - MPI_Comm_rank: This MPI function returns the rank of a process in the communicator of which it is a member. A communicator is a collection of processes that can send messages to each other. The implemented attack makes a call to the MPI_Comm_size function that returns the size of the communicator. Then the process generates a random number between 0 and the size of the communicator and returns that value as the rank rather than the correct one. This has the effect of confusing the identities of the processes.
    - MPI_Recv and MPI_Send: These MPI functions receive and send a memory block that corresponds to a structure of an MPI data type between

processes. These interposer attacks change the actual information contained in the memory block that is transferred.

- MPI_Reduce: This MPI function is a collective communication operation, in which all the processes in a communicator contribute data that are combined using a binary operation. This attack changes the kind of operator used in the binary operation to another valid binary operator.

4.1.2 Description of Datasets

Three sets of data were used in our experiments. One dataset is the *Sendmail* data from the University of New Mexico archive. This dataset has been widely used by other research groups conducting system call analysis research [15, 23, 28, 38, 43]. Two additional datasets, MPI ring and LU factorization, were collected in the Microcosm cluster environment in the Department of Computer Science and Engineering at Mississippi State University. These two datasets were used to test the capability of our methods to detect intrusions in a cluster environment.

4.1.2.1 *Sendmail* dataset

We have used the *sendmail* data from UNM in our experiments to test the effectiveness of neural networks as intrusion detection classifiers. Synthetic data for *sendmail* was collected at UNM on Sun SPARCstations running unpatched SunOS 4.1.1 and 4.1.4 with the included *sendmail* [43]. The dataset can be found at http://www.cs.unm.edu/immsec/systemcalls.htm. There are a total of 640 processes in the normal data. This dataset contains five kinds of intrusion data that include a total of 16 traces of attack instances. The attacks that appear in the data are the following [43]:

- 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.

  8LGM Advisory: search for "[8lgm]-Advisory-16.UNIX.sendmail-6-Dec-1994".

  3 traces.

- 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. The *uudecode* program 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. The *sendmail* program 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 traces.

- error condition - forwarding loops: A local forwarding loop occurs in sendmail when a set of $HOME/.forward files form a logical circle.

  5 traces.

- 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 has tested both modes. They also varied the number of commands issued as root after a successful attack.

  4 traces.

- unsuccessful intrusions - sm5x, sm565a: These are attack scripts for which SunOS 4.1.4 has patches.

  2 traces.

4.1.2.2 MPI ring program

The first test program for the cluster environment is the MPI ring program. There are two major arguments for this program: 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. For each execution of the program, four machines were used. Each program trace collected from a single machine can be viewed as a normal run of the MPI program. We implemented the daemon attack on this ring program and collected data for the ringFile attack with various sizes of data written to disk and different numbers of write operations. We also ran the

MPI ring program with the malicious library attack under different conditions. Sometimes, the attack caused the MPI program to crash and freeze because the changes in the communication rank and size prevented MPI processes from communicating with each other correctly. Table 4.1 gives detailed statistics about the MPI ring dataset.

Table 4.1 Statistical summary of the MPI ring dataset

	Normal ring	ringFile	ringFork	Interpostion 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 was chosen to spawn the daemon process. The three other 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 do the computational work. This depends on the design of the MPI library. RingFork generates many processes that do little work. Most of these processes have fewer than 10 system calls in their trace. Since a window size of 10 was used during our experiments, it is impossible to classify these traces. However, as shown in section 4.5, the attack can still be detected from the main process which spawns the daemon process.

4.1.2.3 LU factorization

MPI ring is too simple to represent real parallel computational work. Therefore, we have used another more complicated program provided by Yogi Dandass (Department

of Computer Science and Engineering at Mississippi State University) in our experiments. This program uses the LU factorization method to solve systems of linear equations such as Ax = b. There are two major parameters for this program. One is the size of the matrix A (n). The other is the block size for the communication (k). In our experiments, various values for these two arguments were used to generate the training and test data. We used all combinations of n = 1024, 512, 256, 128, 64 and k = 32, 16, 8, 4, 2 to execute the LU factorization program and obtained a total of 5*5 = 25 executions of the program. Since the program randomly initializes some internal parameters at the beginning, some differences are present in the behaviors of two different runs of the program with the same parameters. Therefore, we executed the program two times with each set of parameters. Attacks were also implemented on this program. The two attacks used to generate the datasets were both daemon attacks: one writes data to a file and the other spawns many processes. We ran the file attack 8 times and the fork attack 6 times.

4.1.3 ROC curves for analysis of classifier performance

When building an intrusion detection system, the key component is the analysis module. A good analysis module will lead to good performance of the intrusion detection system. A neural network is used as the analysis module in our prototype system and the Receiver Operating Characteristic (ROC) curves have been used to evaluate performance of the neural network classifier. “The ROC curve is a good way of visualizing a classifier’s performance in order to select a suitable operating point, or decision threshold. When comparing a number of different classification schemes it is often desirable to obtain a single figure as a measure of the classifiers’ performance” [3].

The ROC curve is a plot of the true positive rate against the false positive rate for different possible thresholds for a classifier. An ROC curve demonstrates several things [3]:

- It shows the tradeoff between sensitivity and specificity.
- The closer the curve follows the left-hand border and then the top border of the ROC space, the more accurate the classifier is.
- A curve around the 45-degree diagonal of the ROC space indicates the classifier is poor.
- The area under the curve is a measure of overall performance of a classifier, with a greater area indicating better overall performance.

4.2 Using a backpropagation neural network for intrusion detection

The goal in using artificial neural networks for intrusion detection is to take advantage of their generalization capability to classify online data as being normal or intrusive. An artificial neural network is composed of simple processing units, or nodes, and connections between them. The connection between any two units has some weight, which is used to determine how much one unit will affect the other. The backpropagation network consists of three types of units, input units, hidden units, and output units. By assigning a value to each input node, and allowing the activations to propagate through the network, a neural network performs a functional mapping from one set of values (assigned to the input units) to another set of values (appearing in the output units). The mapping function is stored in the weights of the network.

4.2.1 Training the neural network

In our work, a classical feed-forward multi-layer perceptron network was implemented - a backpropagation neural network. The use of different types of neural networks was investigated in our previous work [29]. These included backpropagation neural networks, radial basis function networks, and self-organizing maps. 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 backpropogation neural network achieves accuracy similar to that of radial basis function networks with many fewer hidden nodes. Therefore we have chosen a backpropagation model for the analysis module of our system. Four major issues need to be addressed in order to use backpropagation neural networks: determining how to encode the data for input to the network, selecting a network topology, assigning meaning to the output values, and determining how to conduct supervised learning with the neural network.

Because our goal is to create a real-time intrusion detection system, we must reduce the computational overhead as much as possible. Therefore, we decided to use a simple binary encoding of each system call and to use a sequence of these binary values as the input value to the neural network. As mentioned in Chapter 2.2.2, we use a sliding window to step through the trace and extract the unique sequences from these traces for training. Each system call has an identifying number (this number might change with 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. Because we seek to determine whether an

input string is anomalous or normal, we use a single output node to indicate if an input string is normal or anomalous (a value of 0 indicates normal and a value of 1 indicates anomalous). After defining the input and output nodes, we need to find an appropriate network topology. With an input layer, a hidden layer, and an output layer, a neural network can be constructed to compute any arbitrarily complex function [22]. Researchers have shown that a single hidden layer has the same capability as one with several hidden layers [22]. In order to reduce the time for training and online detection, a single hidden layer is used in our network.

Since, with different training data, the optimal number of hidden nodes is unknown, we trained neural networks with 4, 8, 16, 32, and 64 hidden nodes. For different numbers of hidden nodes, we trained the network for different numbers of epochs. Before training, network weights were initialized randomly. 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, for each number of hidden nodes, we trained 10 networks with different initial weights. We then tested each network with test data and retained the best network, discarding the others.

4.2.2 Embedding the neural network in the kernel

In order to build a lightweight intrusion detection system, the computational overhead must be reduced as much as possible without compromising the accuracy of the IDS. Moreover, in Linux kernel programming, the *libc* math library cannot be accessed. Therefore, we need to use a simple activation function that retains accuracy and reduces computation.

In order to reduce computational requirements, a simple sigmoid function described by Tveter [41] is used instead of the standard sigmoid function

($y = 1 / (1 + e^{-x})$).

The formula of this simple sigmoid function is the following and it can be computed very quickly:

$$y = (x / 2) / (1 + |x|) + 0.5$$

The derivative is: $1/(2*(1+|x|)*(1+|x|))$.

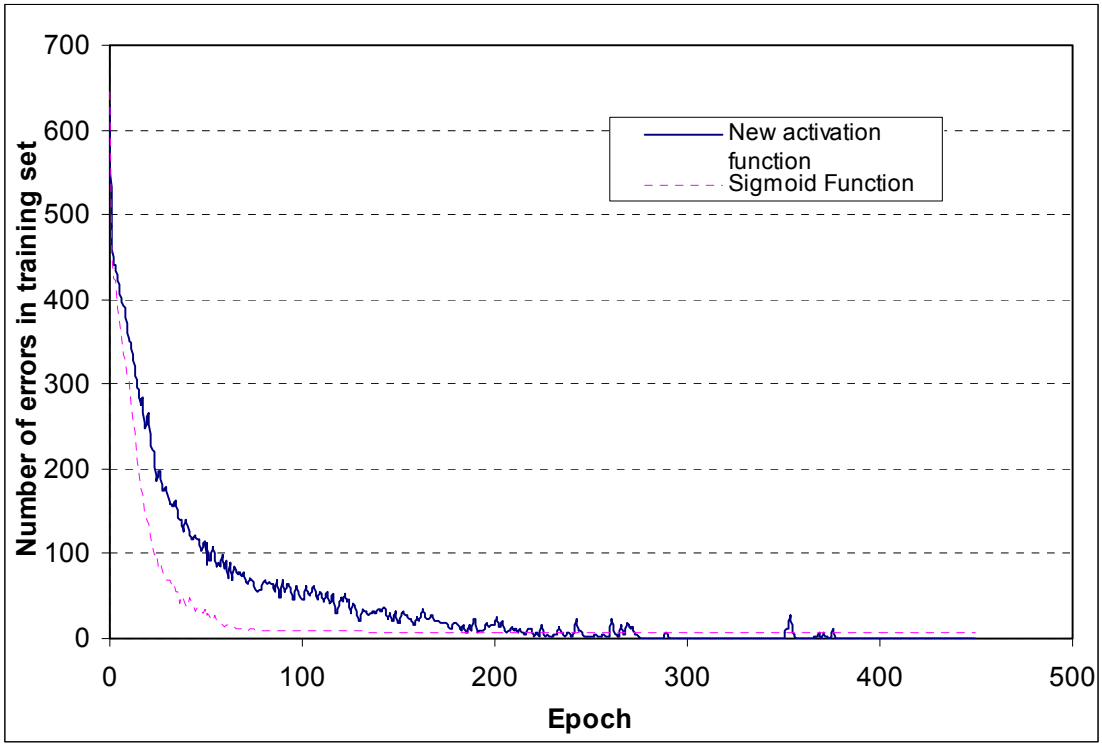


Figure 4.1 Comparison of convergence speed using simple sigmoid function and the standard sigmoid function

Tveter states that "this sigmoid function approaches its extremes more slowly. This means that if you are trying to output numerical values it will take more iterations to

reach your target value. But if you're doing a classification problem you really only care to get the correct output value greater than the other outputs and here these functions will save on CPU time without influencing the number of iterations required by very much" [41]. Since in our case we are using the neural network as a classifier, this simple activation function fulfills our requirements.

Table 4.2 Comparison of error rates using a simple sigmoid function and the standard sigmoid function (average of 20 networks)

	Error rate in test set	**Deviation**
Simple sigmoid function	7.28%	0.462%
Standard sigmoid function	8.02%	0.328%

Experiments were conducted to test Tveter's statement using the LU factorization dataset described in section 4.2.2. The training data included 10 runs of the normal program. The remaining 208 normal traces and all 16 intrusion traces were used as test data. Unlike the experiments in section 4.5 where the error rate is computed based on an entire trace, the error rate for this experiment was computed based on the extracted sequences. Since we use the neural network to classify each of the sequences and not the entire trace, the effect of different trace classification methods introduced in section 4.3 is removed. From Figure 4.1, we can see that the simple sigmoid function converges more slowly than the standard sigmoid function, but both can reach approximately the same extreme. We also trained 20 neural networks with these two different activation functions. As shown in Table 4.2, the simple sigmoid function produced performance comparable to that of the standard sigmoid function.

4.3 Trace classification methods

Our neural network classifies each window of sequences as normal or anomalous. However, the goal is to classify an entire program trace as normal or anomalous. In this section, we will introduce different methods for online and offline detection of intrusion in program traces. In offline detection, we already have all the system calls that a program generated for one complete run. The goal is to classify the entire trace as normal or anomalous. 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. 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:** We extract all unique sequences from the traces and classify these sequences. We define a threshold for the total number of anomalous sequences needed to fire the alarm.

Online:

- **Total Anomalies:** We count the number of anomalous sequences within the trace. If the number becomes greater than a threshold an alarm is fired; otherwise it is classified as normal.
- **Max Cons Anomalies:** We count the number of consecutive anomalous sequences. When the maximum number of consecutive anomalies becomes larger than a threshold, an alarm is generated.

- **Max Burst Counter:** We define a variable that remembers the total number of anomalous sequences seen so far. We call this variable BC (burst counter). After encountering a normal sequence, we decrease this variable at a slow rate. This is similar to Ghosh et al.'s leaky bucket method [20]. 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 [39] 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 observed that anomalous sequences tend to occur in clusters [20]. The advantage of using MaxBurstCounter is that it allows occasional anomalous behavior that would be expected during normal system operation, but is quite sensitive to large numbers of temporally co-located anomalies expected 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 be consecutive.

4.4 Artificial anomaly generation methods

The major challenge for anomaly detection systems is achieving an effective detection rate with an acceptable false positive rate [23]. These systems create a profile of normal behavior based on past behavior. Any deviation from this normal behavior is viewed as an anomaly. Anomaly detection systems suffer from a basic difficulty in

defining "normal." Methods based on anomaly detection tend to produce many false alarms because they are not capable of discriminating between abnormal patterns triggered by an otherwise authorized user and those triggered by an intruder. Misuse detection is generally more accurate in detection of known intrusive behavior and generates far fewer false alarms. But misuse detection also has the major disadvantage of not being able to detect new attacks that it has never seen before. Recently more and more new attacks are appearing and sophisticated attackers can use different mechanisms to hide or change their attack trace from the well-known ones. Therefore, the capability to detect new attacks has become more and more important. In this section, we describe our effort to find a good tradeoff between anomaly and misuse detection.

In order to train the neural 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. Ghosh, Schwartzbard, and Schatz [20] randomly generate anomalous data that covers the entire input space. 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. This is a pure anomaly detection method that does not use any attack information. In Unix-like systems there are at least 200 different system calls. If we define a window size of 10, there will 200^{10} possible instances in the training space. We hypothesize that it is impossible to adequately sample this training space using randomly generated sequences spread throughout the space.

Based on this hypothesis, we designed a new method, the Intrusion Pattern Based algorithm (IPB), to generate artificial anomalous data. We used both normal and intrusion data for training. Sequences that appear both in attack traces and in normal traces are eliminated from the anomalous data and additional anomalous data is generated based on the intrusion patterns extracted from the attack data using the methods described below.

IPB uses several methods to generate additional anomalous data based on small intrusion patterns. To find small intrusion patterns we first extract normal sequences from normal training data. We go through the anomalous training data to find those sequences that occur only in the attack data. Most sequences in attack data are normal. The anomalous sequences only occur in a small part of the trace of a program run that includes an intrusion. After finding these anomalous sequences, we find the common subsequences in them. For example, consider the following intrusion sequences:

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

The sequence 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 additional anomaly patterns based on this pattern. After finding this small intrusion pattern, IPB uses three methods to create the anomalous training data.

- First, we include all the known intrusion data in the training data to ensure that we can detect all the intrusions that we have seen.
- Second, we randomly insert some system calls before or after small patterns within the window size. This will cause the neural network to recognize

sequences that are similar with the known intrusion sequence, and which are more likely to be part of an intrusion.

- Third, we randomly select sequences from normal data and insert the small intrusion patterns into these sequences. This will simulate intrusions within different program contexts.

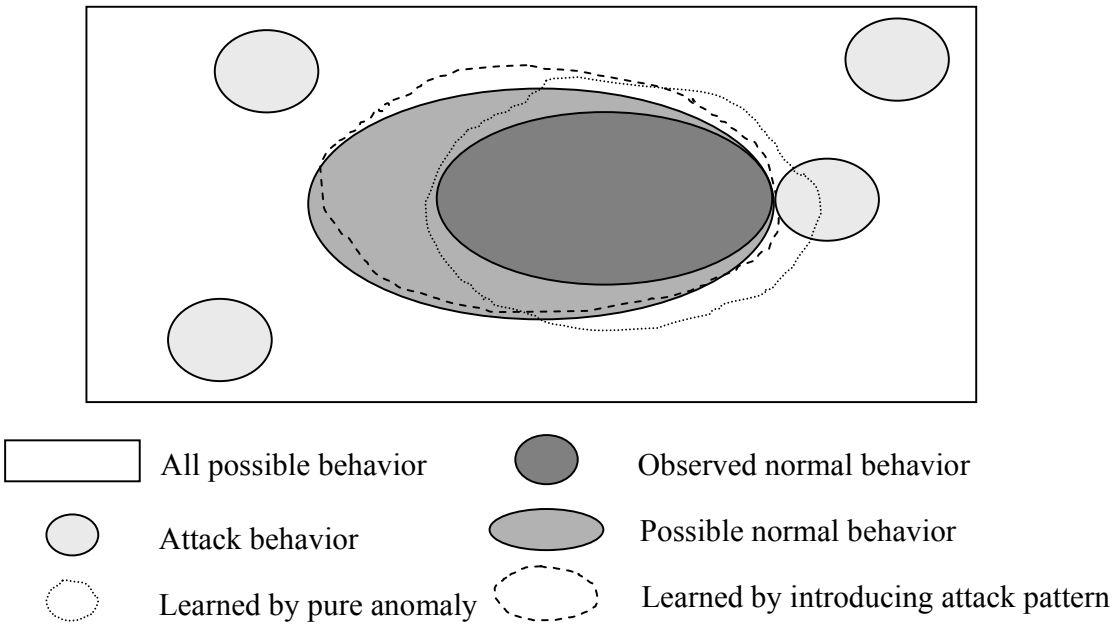


Figure 4.2 Program behavior represented in pattern space

Figure 4.2 illustrates the behavior of different artificial anomaly generation methods. The rectangle represents all possible instances in the training space. The shaded eclipse represents the normal instances that could occur in the system. The dark circle represents the normal instances that we have already observed. If we only use the observed normal instances plus randomly generated anomalies to train the neural

network, the neural network will just memorize the observed normal instances and any other unseen instances will be viewed as intrusions. The boundary between normal and anomaly learned by the neural network in this case is represented by the small dotted line. This behavior will cause the neural network to generate many false alarms. IPB utilizes known intrusive patterns to train the neural network. This allows the neural network to correctly detect previously seen attacks. The dashed line shows the boundary with IPB artificial anomaly generation. These artificial anomalies based on intrusive patterns allow the neural network to learn a more reasonable boundary. Because we used intrusion data when we trained our neural network, it is not pure anomaly detection. But unlike misuse detection systems which only record the intrusion signature, our experimental results indicate that our method can still detect novel attacks.

4.5 Experiments and results

In this section, the experiments that evaluate the performance of pShield are described and results are presented. Different trace classification methods and different artificial anomaly generation methods are compared and discussed.

4.5.1 Design of experiments

In this section, we describe the experiments that were conducted to evaluate different aspects of pShield. For each dataset available, the IPB method of generating additional anomalous data was compared with pure anomaly detection where anomalous data was generated randomly.

Ring program dataset:

- **Pure anomaly detection:**

  For this experiment, we trained the neural network with real normal data and randomly generated anomalous data. Four runs of the ring program were selected as the training set. Since the program was executed on 4 machines and each run contains 4 traces, a total of 16 traces were used for training. The remaining 69 normal traces were used as test data. All attack traces were used in testing.

- **IPB:**

  The same set of normal training data was used in this experiment. One attack from the ringFile attacks was also used by IPB to generate anomalous data. The normal test data includes all the traces that were not used in training. All attacks were used in testing.

LU factorization dataset:

- **Pure anomaly detection:**

  The training data includes 10 runs of the normal program. The remaining 208 normal traces and all 16 intrusion traces are used as test data.

- **IPB:**

  In addition to the normal data, two instances of the fork attack were used to generate additional anomalous data. The test set consists of 208 normal traces and all 16 instances of attacks.

***Sendmail* dataset:**

- **Pure anomaly detection:**

We randomly chose 330 processes from the normal data for training. The remaining 310 processes were used as test data. The test data also contains all attack traces.

- **IPB:**

  Four traces were chosen from the first three kinds of attacks to generate part of the training set using IPB. We use the remaining 12 attack traces as the test set. When testing, since we use a window size 10 and some of the processes have fewer than 10 system calls, only 210 of the 310 normal processes are used for testing.

4.5.2 Comparison of different trace classification methods

The comparison of different trace classification methods with different datasets is discussed in this section. In all experiments in this section we used the pure anomaly detection method to generate anomalous data.

4.5.2.1 Ring program dataset

We conducted the experiment as described in the previous section. Table 4.3 shows the results for a normal trace that was not used in training. The result for MaxBurstCounter shows the maximum value of the burst counter variable during the trace. MaxConsAnomalies shows the maximum number of anomalous sequences that occurred continuously.

Table 4.4 shows the intrusion detection results for data generated by the RingFile attack. In the RingFile attack, a daemon process is generated after the MPI program finishes. Therefore in Table 4.4, processes 1306 and 1307 represent the abnormal

processes. Process 1303 contains the call to the daemon function to generate the daemon process and, therefore, it also represents an anomaly. Figure 4.3 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.

Table 4.3 Comparison of online classification methods with an unseen normal run

Pid	TotalAnomalies	MaxBurstCounter	MaxConsAnomalies	Total calls
932	1	5	1	9198
933	4	18	3	26
934	1	5	1	6813

Table 4.4 Comparison of online classification methods with the RingFile attack

Pid	TotalAnomalies	MaxBurstCounter	MaxConsAnomalies	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

Figure 4.4 shows the intrusion detection results using different trace classification methods. We varied the threshold in each method to generate the ROC curve. The VectorCount method has the best performance, since it contains all the information for the entire trace. The results in Figure 4.4 show that the MaxBurstCounter method has a lower false positive rate than the other two online methods, but its true positive rate improves much more slowly than the other two with the increasing false positive rate.

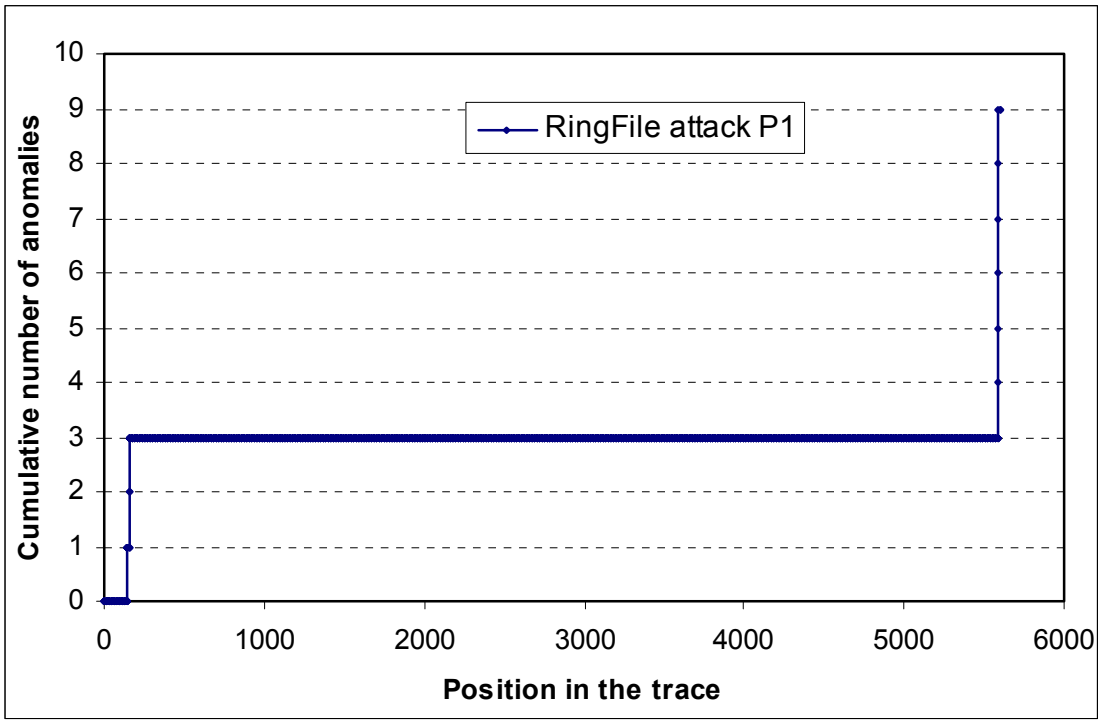


Figure 4.3 Distribution of anomalies in the first process of the ring program under the RingFile attack

4.5.2.2 LU factorization dataset

Figure 4.5 shows the intrusion detection results for LU factorization using anomaly detection. When we use the MaxBurstCounter method to classify the traces we can detect all attacks with no false positives. Other methods also have good performance. We extracted 833 sequences from the normal training data and 1048 sequences from the normal test data. Of the sequences extracted from the normal test data, 259 sequences are unique and are not found in the training data.

The longest process in the LU factorization data only used 4715 system calls. Though the running time of the ring program is much less than that of the LU

factorization program, the number of system calls for the longest run for normal ring program is 20818 (shown in Table 4.1). This illustrates that complicated MPI programs may use fewer system calls and spend more time on computation. The relatively small number of system calls used by LU factorization does not mean that its behavior is simple. We tested two runs of the LU factorization program with the same parameters. The first run contained 511 sequences and the second contained 562. There were 125 sequences in the second run that do not appear in the first trace. We conclude that complex programs may have much more complex communication patterns that make the program behavior more variable.

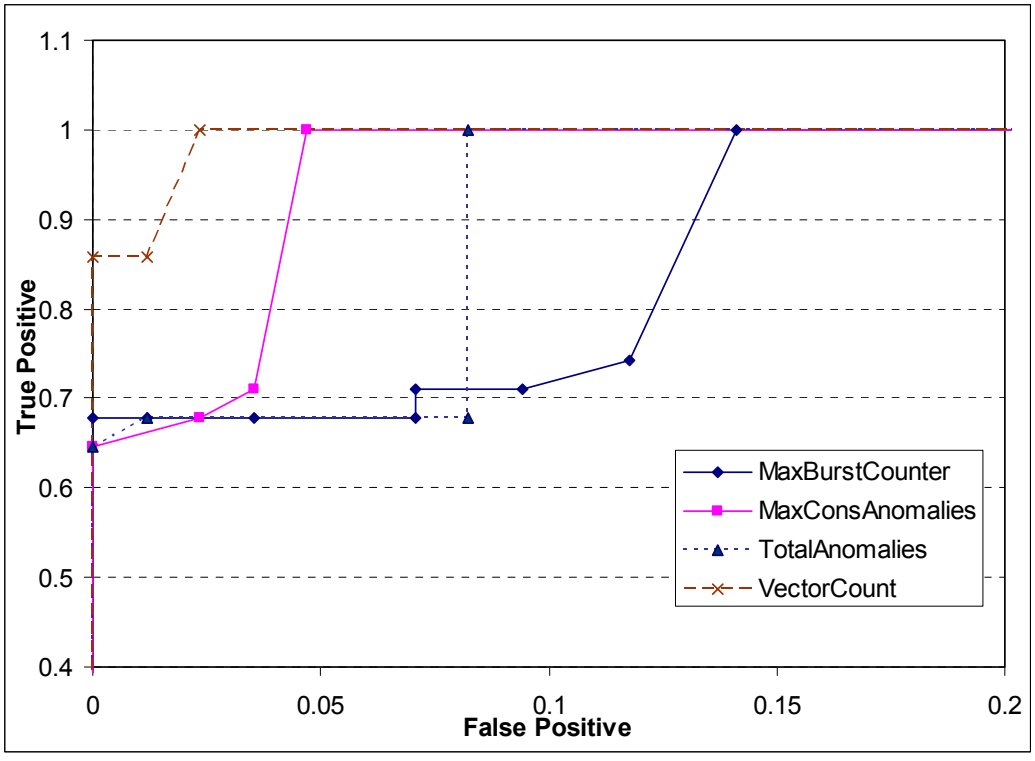


Figure 4.4 ROC curve of pure anomaly detection with different online and offline trace classification methods with the ring program dataset

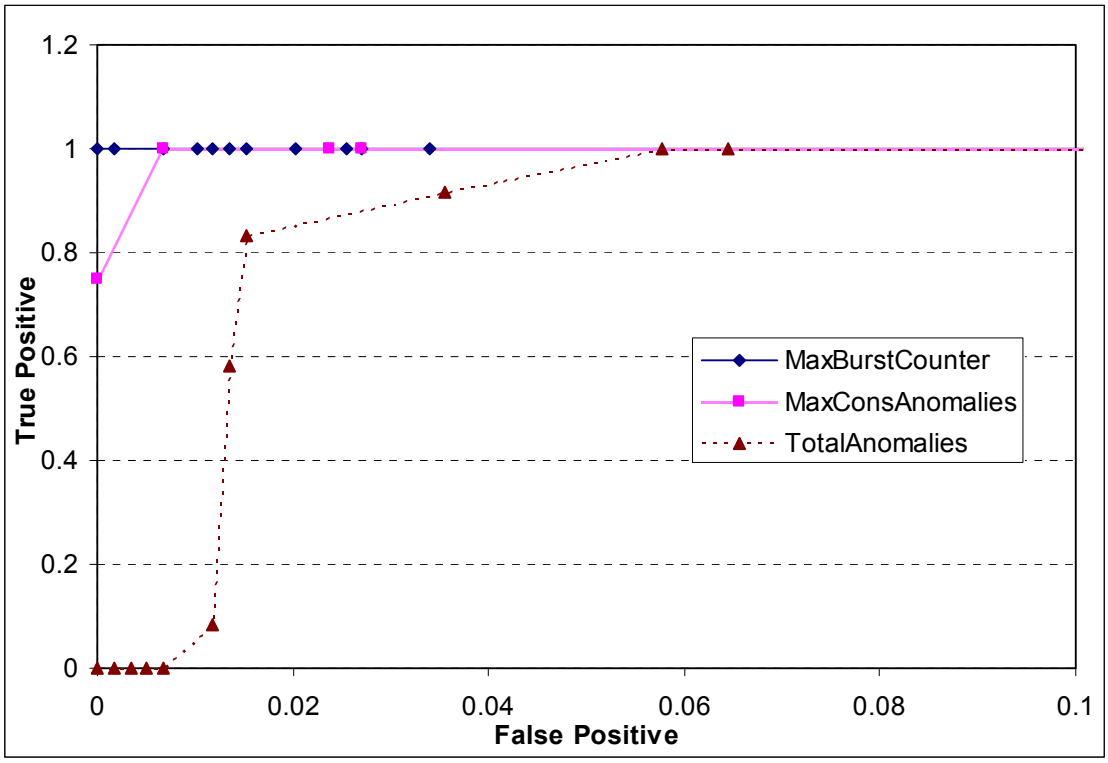


Figure 4.5 ROC curve of pure anomaly detection with different online and offline trace classification methods with the LU factorization dataset

4.5.2.3 *Sendmail* dataset

A total of 1249 vectors were extracted from the training data and 1130 vectors from test data. There are 254 vectors that appear in the test data, but not in the training data. This means there is some previously unseen normal behavior in the test data. This reflects the fact that, in the real world, program behavior is variable and we cannot usually obtain a training data set that includes all possible behavior of a program. Figure 4.6 shows the results using different detection methods with pure anomaly detection. We can see that using MaxBurstCounter and MaxConsAnomalies yield better results than using TotalAnomalies.

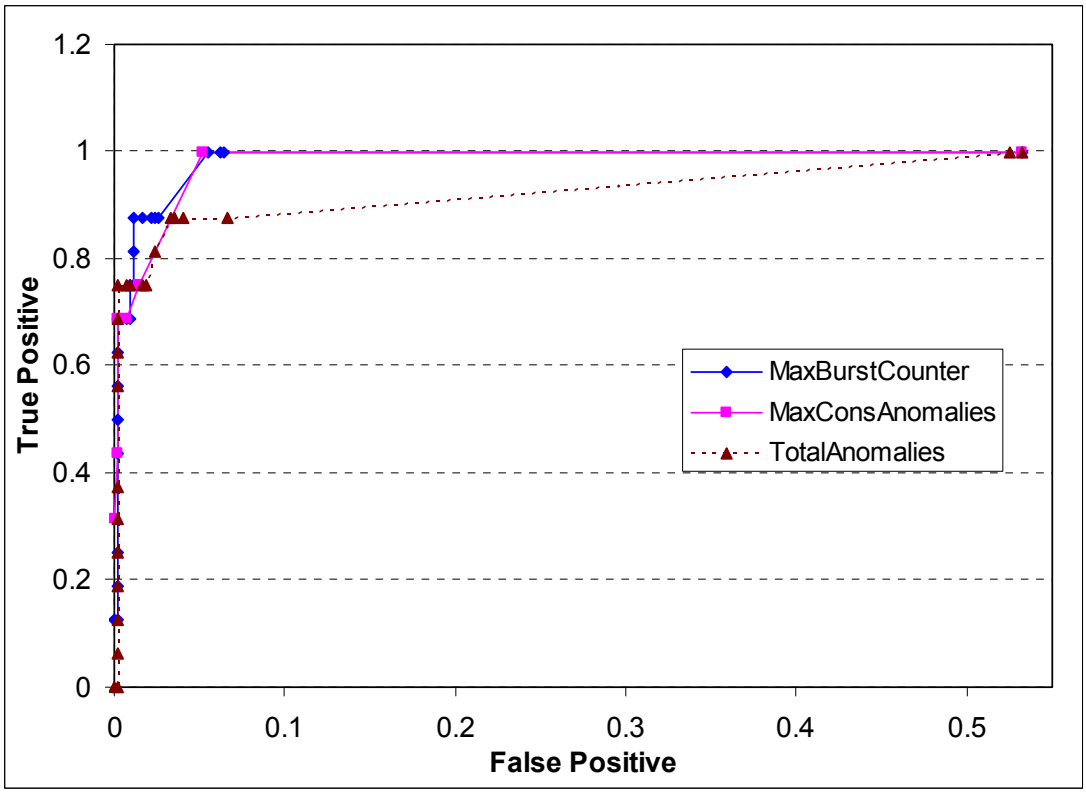


Figure 4.6 ROC curve for pure anomaly detection with the *sendmail* dataset

4.5.2.4 Conclusions regarding trace classification methods

We have compared different trace classification methods in this section. The MaxBurstCounter achieves the highest true positive rate of all methods when the false positive rate is required to be zero. However, MaxConsAnomalies achieves a detection rate of 100% true positives with a lower false positive rate than MaxBurstCounter. TotalAnomalies exhibited the worst performance of the online methods. Since a long trace contains more small sequences than a short trace, the classifier will generate more false positives in the long trace and the long trace is more likely to be classified as an intrusive trace. For the online methods, both MaxBurstCounter and MaxConsAnomalies

appear to outperform TotalAnomalies. Since our focus is real-time online intrusion detection, the offline VectorCount method was tested with only one dataset. From Figure 4.4, we can see that it outperformed all other methods. However, the capability of VectorCount needs to be tested with additional datasets to evaluate its ability for offline detection.

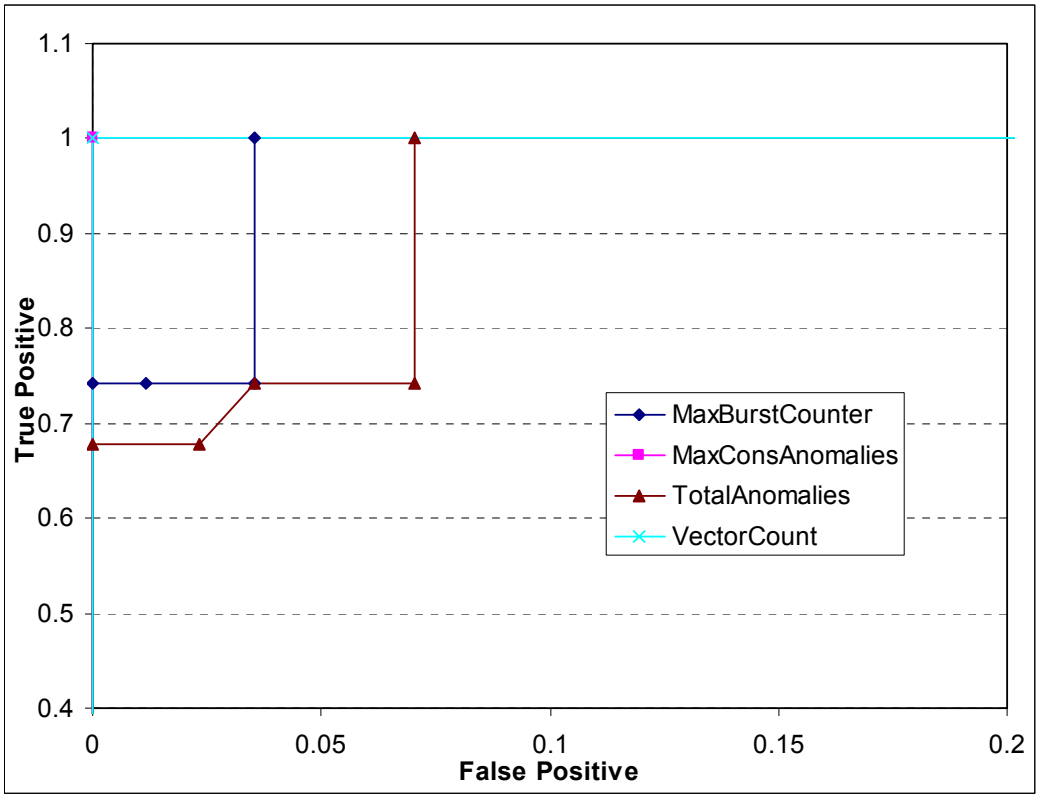


Figure 4.7 ROC curve of IPB with different online methods with the *ring* dataset

4.5.3 Comparison of different artificial anomaly generation methods

The comparison of different artificial anomaly generation methods with different datasets is presented in this section.

4.5.3.1 Ring program dataset

Using the IPB generation method, the false-positive rate decreases as expected without compromising the true positive rate. Although the ring-fork and interposition lib attacks do not appear in the training dataset, they can still be detected by our system. When using the VectorCount and MaxConsAnomalies methods, we can detect every attack with no false positives. With MaxBurstCounter and TotalAnomalies, we can see from Figure 4.7 that the performance is also improved over that of pure anomaly detection.

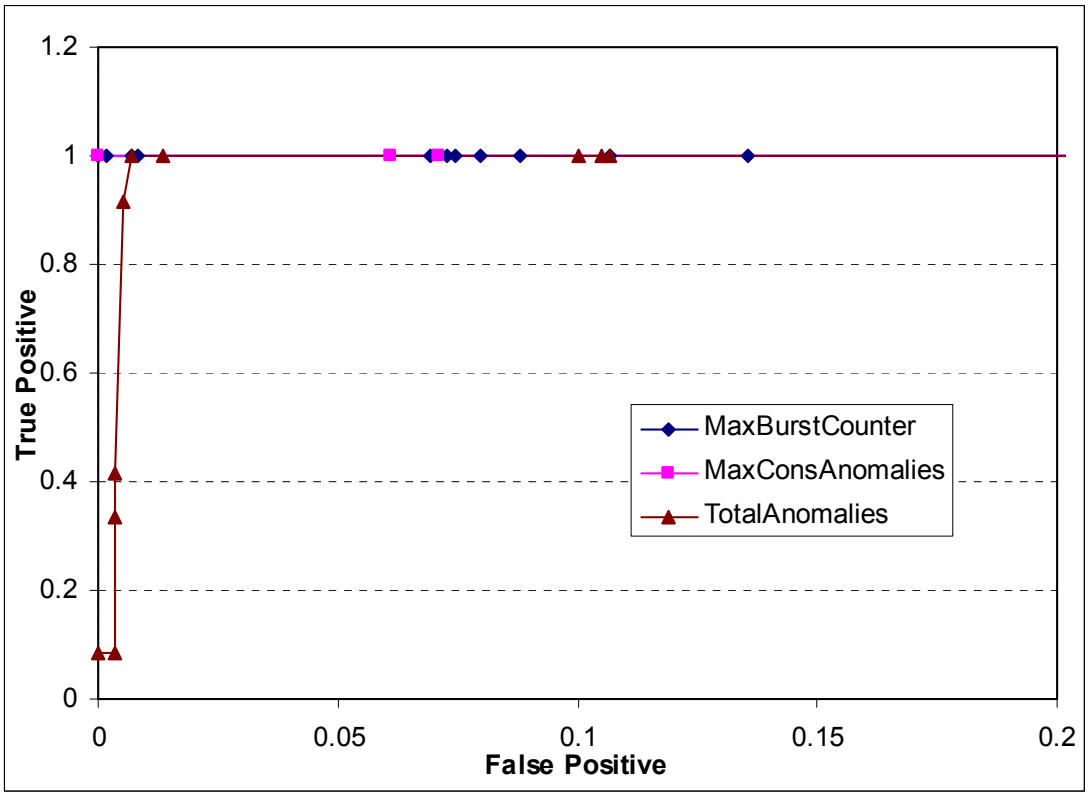


Figure 4.8 ROC curve of IPB detection with different online and offline trace classification methods with the LU factorization dataset

4.5.3.2 LU factorization dataset

Figure 4.8 illustrates that using IPB to generate data allows MaxBurstCounter and MaxConsAnomalies to achieve a 100% detection rate with a 0% false positive rate for this dataset. When the results in Figure 4.7 are compared to those in Figure 4.4, we can see that IPB improved performance for all datasets including TotalAnamalies.

4.5.3.3 *Sendmail* dataset:

Figure 4.9 shows the results with IPB and the *sendmail* dataset. The new attack can easily be detected with a low false positive rate. Figure 4.10 gives a direct comparison of the performance of IPB and pure anomaly detection using MaxBurstCounter. We can see that using IPB improves performance.

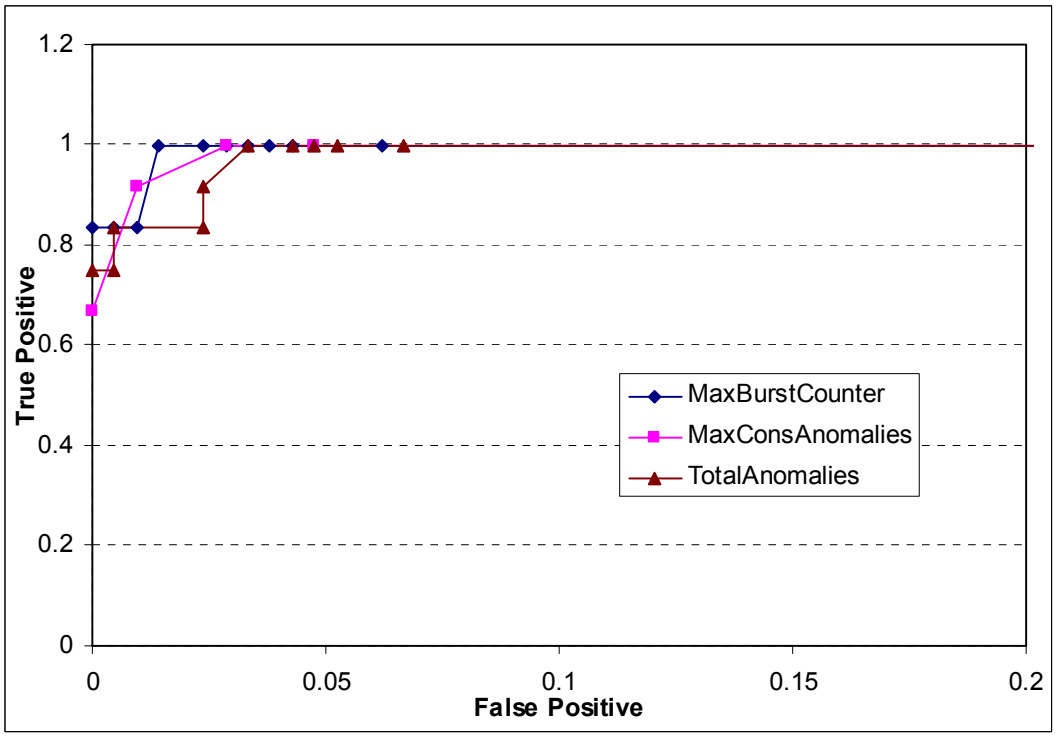


Figure 4.9 ROC curve for IPB with the sendmail dataset

4.5.3.4 Summary regarding artificial anomaly generation methods

We compared IPB with pure anomaly detection with different datasets. The IPB method for generating artificial anomalies has better performance than a pure anomaly detection method for each of the datasets. It effectively reduces the false positive rate without losing the capability to detect novel attacks. Fan et al. [13] proposed DB2 to generate the artificial anomalies. In order to train a neural network, the normal and anomalous data should be of similar size. DB2 will generate 10 times more anomalous data than normal data so that it is difficult to use to train the neural network. Therefore, we did not compare DB2 with IPB.

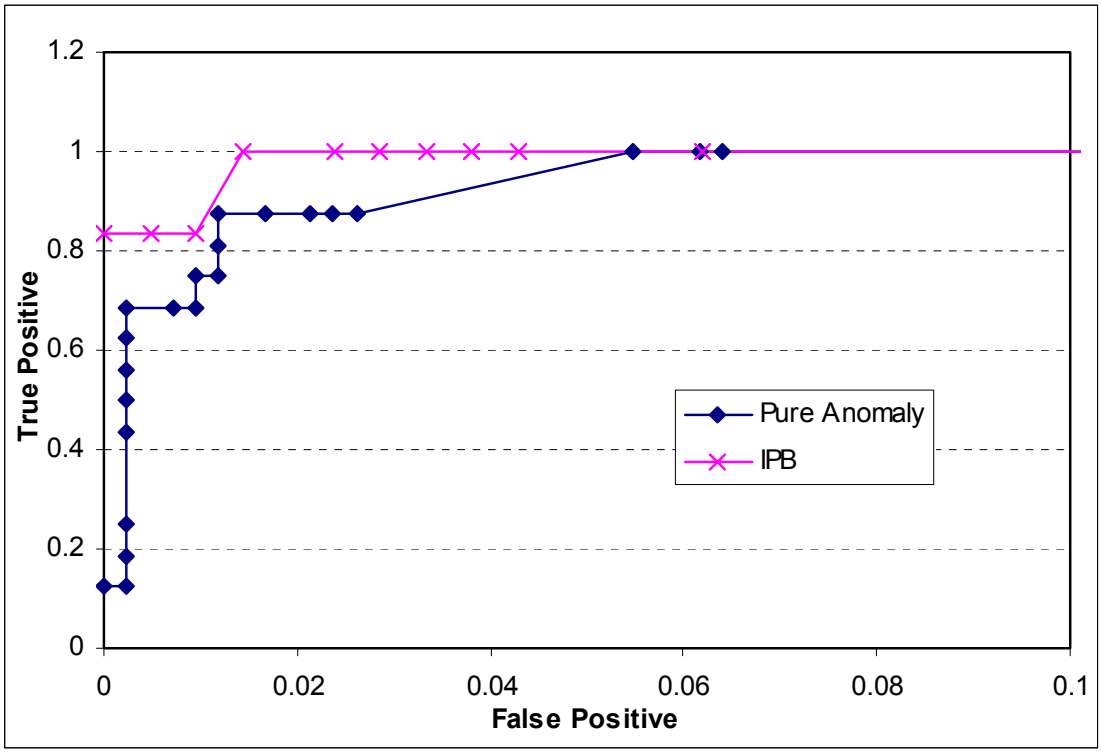


Figure 4.10 ROC curve for comparison of anomaly and IPB data generation methods using burst counter with the sendmail dataset

4.6 Performance evaluation

In order to be a lightweight real-time IDS, pShield must be fast and add little overhead to the operating system. In the MPI cluster environment, pShield should slow down the communication between nodes as little as possible. We tested our system with two different benchmarks to measure the performance of the system. One is the IS benchmark which simulates a real computational problem in the MPI cluster environment. The second, MPBench, is used to test the performance of each MPI call. The tests with the IS benchmark allow us to measure the effect of pShield on the performance of the entire cluster system. The tests with MPBench show the communication overhead added by pShield.

The first benchmark we used is the IS benchmark [36]. The NPB 2 implementation of the IS kernel benchmark is based on a bucket sort. "The number of keys ranked, number of processors used, and number of buckets employed are all presumed to be powers of two. This simplifies the coding effort and leads to a compact program. The number of buckets is a tuning parameter. On the systems tested, the best performance was obtained when the number of buckets was half that which gives best load balancing. Communication costs are dominated by an MPI alltoall, wherein each processor sends to all others those keys which fall in the key range of the recipient" [36].

We ran this benchmark under three conditions: without pShield loaded, with the module loaded but not monitoring the program, and with the module loaded and monitoring the benchmark program. In each case, we ran the benchmark 20 times. Table 6 shows the results. We can see that loading our module resulted in only 0.53% overhead,

and monitoring the benchmark program resulted in 1.55% overhead. Both are well under the 5% performance penalty that is usually considered acceptable.

Table 4.5 Results with the IS benchmark

	Average Time of 20 times (seconds)	Standard Deviation
without pShield loaded	5.5475	0.06
with pShield loaded and without monitoring	5.577	0.037
with pShield loaded and monitoring	5.634	0.035

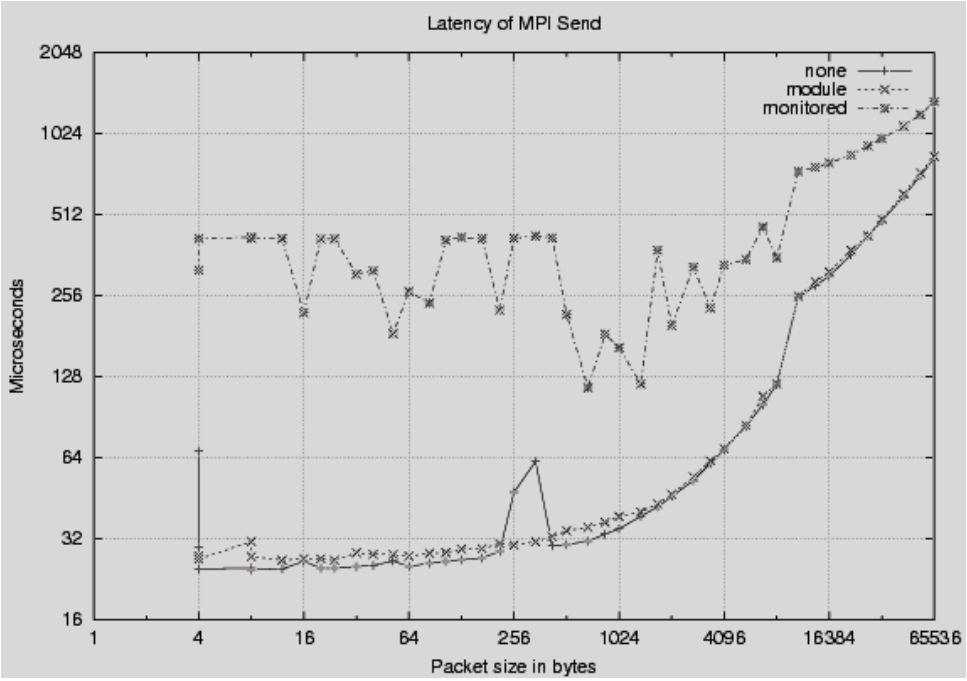


Figure 4.11 Latency of MPI send tested with MPBench

We also conducted an experiment to measure the influence of pShield on the MPI communication APIs that affect the performance of parallel computing. From Figures 4.11, 4.12, and 4.13 we can see that loading our module has no effect on the APIs, but if

we monitor the program, it does affect the performance. Figure 4.11 shows the effect on MPI send. The MPBench test for latency can properly be described as a measure of the time for an application to issue a send and continue computing. We can see when monitoring the benchmark program, the latency is notably increased, especially when sending small packets. When sending a very small packet, the time for the send operation is very short. Therefore the computation time for the neural network dominates. As the packet size increases, the proportion of computation time by the neural network decreases. The performance penalty when sending a 4 byte packet is severe – 1331%. With a 65536 byte packet, the performance penalty is reduced to 62%. In high performance computing environments, the data packets communicated are usually large (from several hundred thousand bytes to millions of bytes) so the overhead added by pShield is acceptable.

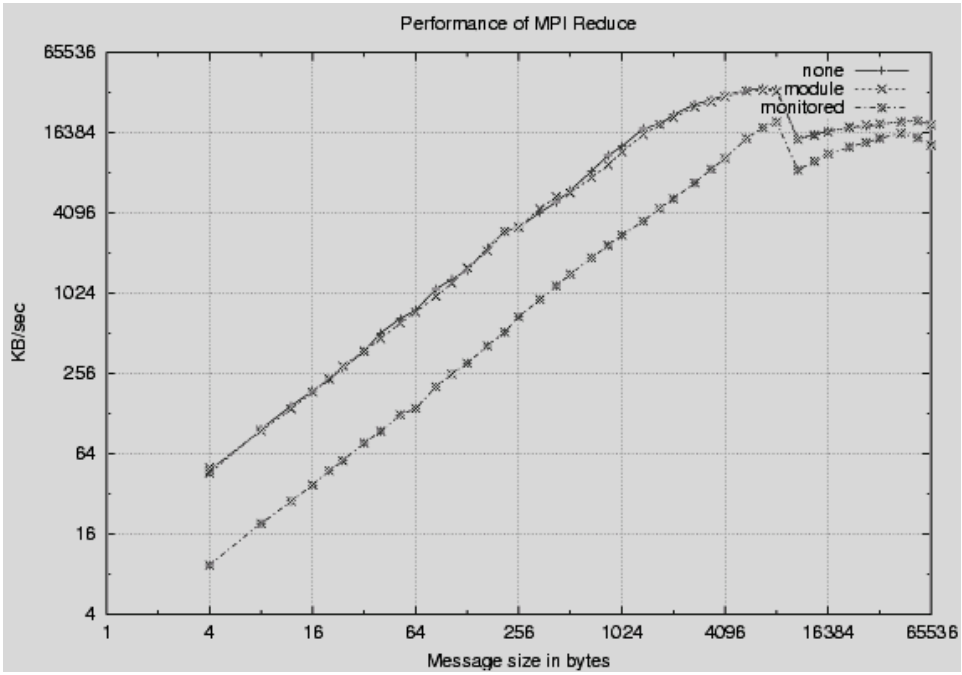


Figure 4.12 Performance of MPI reduce tested with MPBench

Figure 4.12 and 4.13 show the effect on performance of two heavily used MPI functions. When the packet size is small, the overhead added by pShield is obvious. But as the size increases, the performance penalty is alleviated.

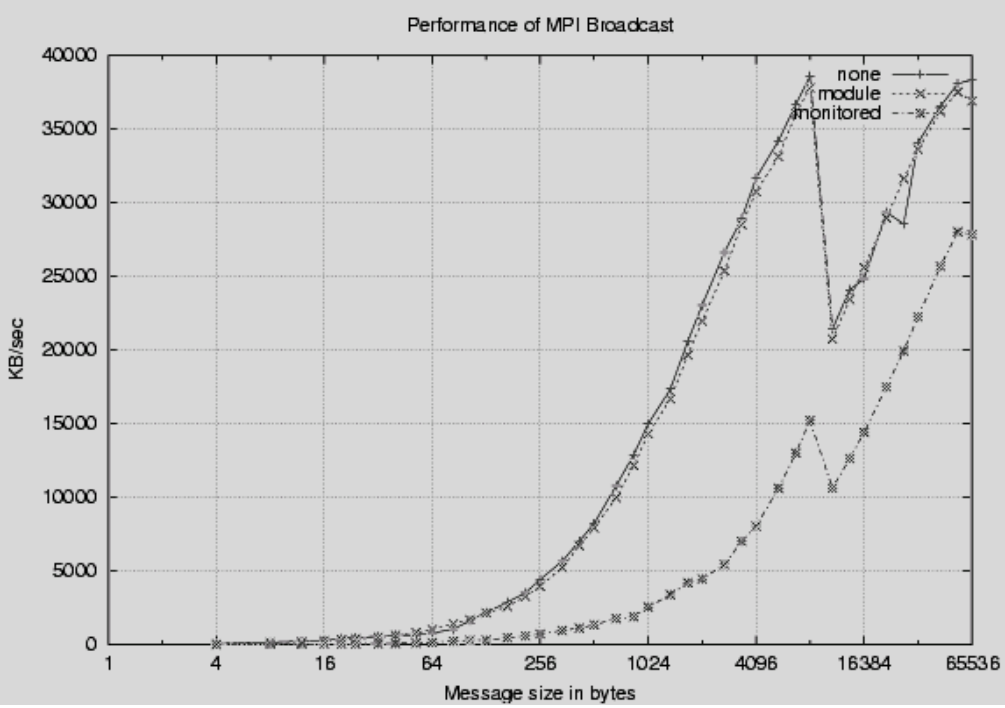


Figure 4.13 Performance of MPI broadcast tested with MPBench

Chapter V

Conclusions and Future Work

Loosely coupled clusters of workstations, connected together by high-speed networks for parallel applications have increased in popularity due to greater cost-effectiveness and performance and 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. As the size of a cluster becomes larger and larger, more and more software and workstations are involved in the cluster system, and the security problem becomes more difficult.

IDS is an active research area today and serves as a necessary security mechanism within an organization. It is a powerful tool when used with other security measures such as firewalls and virus protection products. A DARPA 1998 intrusion detection evaluation study found that novel attacks against systems are rarely detected by most IDSs that are based on misuse detection techniques. However, misuse detection is able to detect well-known attacks with great accuracy and low false alarm rates. To overcome the problems in current misuse and anomaly detection approaches, the generalization capability of IDSs should be improved to detect novel attacks as well as reduce the false positive rate.

Currently most cluster security research focuses on access control. In this thesis, we describe an effort to provide a fine-grained intrusion detection capability for a cluster. We also describe a method to overcome the problems in misuse and anomaly detection methods.

A system architecture for a host-based intrusion detection system in a cluster environment is presented in this thesis and a prototype system, pShield, is described. Several unique attacks on MPI program have been implemented on different real world programs ranging from a simple ring program to a complex real-world LU factorization program. The capability of pSheild to detect these attacks is demonstrated. All attacks can be effectively detected with very few false alarms.

A neural network is used as the classifier to learn the behavior of a program. In order to embed the neural network into the kernel, a simple sigmoid function is used instead of the standard sigmoid function. The generalization and learning capability of the neural network with the simple sigmoid activation function was investigated and the results show that using the simple sigmoid activation function yields a classifier with accuracy comparable to one that uses a standard sigmoid activation function.

In order to find a good tradeoff between misuse and anomaly detection, we proposed a new method, IPB, to generate artificial anomalous data for training the neural network. IPB takes advantage of the characteristics of known attacks and utilizes them to predict new attacks. Although, IPB utilizes attacks patterns during training, experiments have shown that the neural network classifier still has the ability to detect novel attacks

that were not in the training set. However, the false positive rate is improved greatly over methods that use only randomly generated anomalies for training.

The performance penalty on a cluster system with pShield running was investigated. Although our system imposes some overhead on the MPI communication API, the overall performance of a real-world program was only slightly slowed.

The results from [43] compared the performance using RIPPER, Stide, and HMM in detecting intrusions using sequences of system calls. However, in these experiments, the test unit for computing error rates is a sequence, and in our experiment it is trace. This makes it difficult to compare their results to ours. It is also difficult to compare our results to those of Ghosh, et al. [20], since our focus is on cluster security and we used different datasets.

In the future, we will integrate our detectors into the IIDS project under development by the CCSR. A mechanism to collect the information generated by pShield and transfer it to the decision module will be designed. Other researchers have used interposition library [39] and MPI library calls [14] to conduct the intrusion detection on a cluster. We will compare the performance of our method with theirs and determine which combination of sensors is most effective. Since our prototype system is a lightweight IDS in cluster environment, scalability issues should be investigated further. We will investigate the feasibility of learning the behavior of sessions rather than individual programs. We will also investigate the applicability of our method for fault detection in a cluster environment.

REFERENCES

[1] M. Bernaschi, E. Gabrielli, and L. V. Mancini, "REMUS: A Security-Enhanced Operating System," *ACM Transactions on Information and System Security*, vol. 5, no. 1, February 2002, pp. 36-61.

[2] D. P. Bovet and M. Cesati, *Understanding the Linux Kernel*, O'Reilly & Associates, Inc., Sebastopol, California, 2001.

[3] 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.

[4] W. W. Cohen, "Fast Effective Rule Induction," *Proceedings: 12th International Conference on Machine Learning,* Lake Tahoe*,* CA, 1995, pp. 115-123.

[5] C. Cowan, P. Wagle, C. Pu, S. Beattie, and J. Walpole, "Buffer Overflows: Attacks and Defenses for the Vulnerability of the Decade," *Proceedings: DARPA Information Survivability Conference and Exposition*, Hilton Head, South Carolina, 1999.

[6] T. W. Curry, "Profiling and Tracing Dynamic Library Usage via Interposition," *Proceedings: USENIX Summer 1994 Technical Conference,* Boston, Massachusetts, 1994, pp. 267-278.

[7] T. E. Daniels and E. H. Spafford, "A Network Audit System for Host-based Intrusion Detection (NASHID) in Linux," *Proceedings: 16th Annual Conference on Computer Security Applications,* New Orleans, Louisiana, 2000, pp. 178-187.

[8] K. J. Das, *Attack Development for Intrusion Detection Evaluation*, master's thesis, Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Massachusetts, 2000.

[9] D. Denning, "An Intrusion Detection Model," *Proceedings: 1986 IEEE Computer Society Symposium on Research in Security and Privacy,* Oakland, California*,* 1986, pp. 118-131.

[10] R. O. Duda, P. E. Hart, and D. G. Stork, *Pattern Classification and Scene Analysis: Part I Pattern Classification*, John Wiley & Sons, Inc, New York, 1998.

[11] D. Endler, "Intrusion Detection: Applying Machine Learning to Solaris Audit Data," *Proceedings: 1998 Annual Computer Security Applications Conference*, Los Alamitos, California, 1998, pp. 268-279.

[12] M. D. Ernst, A. Czeisler, W. G. Griswold, and D. Notkin. "Quickly Detecting Relevant Program Invariants," *Proceedings: 22nd International Conference on Software Engineering (ICSE 2000)*, Limerick, Ireland, 2000, pp. 449-458.

[13] W. Fan, M. Miller, S. J. Stolfo, W. Lee, and P. K. Chan, "Using Artificial Anomalies to Detect Unknown and Known Network Intrusions," *Proceedings: IEEE Intl. Conf. Data Mining*, San Jose, California, 2001, pp. 123-130.

[14] G. Florez, *A Trusted Environment for MPI Programs*, master's thesis, Department of Computer Science and Engineering, Mississippi State University, Mississippi, 2002.

[15] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff, "A Sense of Self for Unix Processes," *Proceedings: 1996 IEEE Symposium on Computer Security and Privacy,* Los Alamitos, California, 1996, pp. 120-128.

[16] T. Fraser, L. Badger, and M. Feldman, "Hardening COTS Software with Generic Software Wrappers," *Proceedings: IEEE Symposium on Security and Privacy*, Oakland, California, 1999, pp. 2-16.

[17] Y. Freund and R. E. Schapire, "A Decision-Theoretic Generalization of Online Learning and an Application to Boosting," *Journal of Computer and System Sciences*, vol. 55, no. 1, August 1997, pp. 119-139.

[18] D. P. Ghormley, D. Petrou, S. H. Rodrigues, and T. E. Anderson, "SLIC: An Extensibility System for Commodity Operating Systems," *Proceedings: 1998 USENIX Annual Technical Conference*, New Orleans, Louisiana, 1998, pp. 39-52.

[19] A. K. Ghosh and A. Schwartzbard, "A Study in Using Neural Networks for Anomaly and Misuse Detection," *Proceedings: 8th USENIX Security Symposium,* Washington, D.C., 1999.

[20] 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.

[21] I. Goldberg, D. Wagner, R. Thomans, and E. Brewer, "A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker," *Proceedings: 6th USENIX UNIX Security Symposium*, San Jose, California, 1996, pp. 1-12.

[22] S. Haykin, *Neural Networks A Comprehensive Foundation*, 2nd Edition, Prentice-Hall, Upper Saddle River, New Jersey, 1999.

[23] 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.

[24] 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, January 2000, pp. 4-37.

[25] K. Jain and R. Sekar, "User-level Infrastructure for System Call Interposition: A Platform for Intrusion Detection and Confinement," *Proceedings: ISOC Network and Distributed Systems Security Symposium,* 2000, pp.19-34.

[26] S. Jha, K. M. C. Tan, and R. A. Maxion, "Markov Chains, Classifiers and Intrusion Detection," *Proceedings: 14th IEEE Computer Security Foundations Workshop*, Cape Breton, Nova Scotia, Canada, 2001, pp. 206-219.

[27] C. Ko, G. Fink, and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring," *Proceedings: 10th Annual Computer Security Applications Conference*, Orlando, Florida, 1994, pp. 134-144.

[28] W. Lee and S. J. Stolfo, "Data Mining Approaches for Intrusion Detection," *Proceedings: 7th USENIX Security Symposium*, San Antonio, Texas, 1998, pp. 79-94.

[29] 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.

[30] J. Luo, *Integrating Fuzzy Logic with Data Mining Methods for Intrusion Detection*, master's thesis, Department of Computer Science and Engineering, Mississippi State University, 1999.

[31] T. Mitchem, R. Lu, and R. O'Brien, "Using Kernel Hypervisors to Secure Applications," *Proceedings: Annual Computer Security Application Conference*, San Diego, California, 1997, pp. 175-181.

[32] B. Mukherjee, L. Heberlein, and K. Levitt, "Network Intrusion Detection," *IEEE Network,* vol. 8, no. 3, May/June 1994, pp. 26-41.

[33] D. L. Oppenheimer and M. R. Martonosi, "Performance Signatures: A Mechanism for Intrusion Detection," http://www.cs.berkeley.edu/~davidopp/pubs/perfsig.html (current July 2001)

[34] A. Rubini and J. Corbet. *Linux Device Drivers, 2nd Edition,* O'Reilly & Associates, Inc., Sebastopol, California, 2001.

[35] S. J. Russell and P. Norvig. *Artificial Intelligence: A Modern Approach*, Prentice-Hall, Inc, New Jersey, 1995.

[36] W. Saphir, R. V. Wijngaart, A. Woo, and M. Yarrow "New Implementations and Results for the NAS Parallel Benchmarks 2," http://www.nas.nasa.gov/Software/NPB/Specs/npb2.2_new_implementations.ps (current October 2002)

[37] R. Sekar, M. Bendre, D. Dhurjati, and P. Bollineni, "A Fast Automaton-based Method for Detecting Anomalous Program Behaviors," *Proceedings: the 2001 IEEE Symposium on Security and Privacy*, Oakland, California, 2001, pp. 144-155.

[38] A. Somayaji, "Automated Response Using System-Call Delays," *Proceedings: 9th USENIX Security Symposium,* Denver, Colorado, 2000, pp. 185-197.

[39] A. Somayaji, *Operating System Stability and Security through Process Homeostasis,* doctoral dissertation, Department of Computer Science, University of New Mexico, 2002.

[40] H. Teng, K. Chen, and S. Lu, "Adaptive Real-time Anomaly Detection Using Inductively Generated Sequential Patterns," *Proceedings: 1990 IEEE Computer Society Symposium on Research in Security and Privacy*, Oakland, California, 1990, pp. 278-284.

[41] D. R. Tveter, "Backpropagator's Review," http://www.dontveter.com/bpr/bpr.html (current August 2002)

[42] D. Wagner and D. Dean, "Intrusion Detection via Static Analysis," *Proceedings: IEEE Symposium on Security and Privacy,* Oakland, California, 2001, pp. 156-169.

[43] C. Warrender, S. Forrest, and B. Pearlmutter, “Detecting Intrusions Using System Calls: Alternative Data Models,” *Proceedings: 1999 IEEE Symposium on Security and Privacy*, Los Alamitos, California, 1999, pp. 133-145.