Mississippi State University

Scholars Junction

Theses and Dissertations

Theses and Dissertations

12-13-2002

A Trusted Environment for MPI Programs

German Florez-Larrahondo

Follow this and additional works at: https://scholarsjunction.msstate.edu/td

Recommended Citation

Florez-Larrahondo, German, "A Trusted Environment for MPI Programs" (2002). *Theses and Dissertations*. 335.
https://scholarsjunction.msstate.edu/td/335

This Graduate Thesis - Open Access is brought to you for free and open access by the Theses and Dissertations at Scholars Junction. It has been accepted for inclusion in Theses and Dissertations by an authorized administrator of Scholars Junction. For more information, please contact scholcomm@msstate.libanswers.com.

A TRUSTED ENVIRONMENT FOR MPI PROGRAMS

By

German Florez

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

Mississippi State, Mississippi

December 2002

Copyright by

German Florez

2002

A TRUSTED ENVIRONMENT FOR MPI PROGRAMS

By

German Florez

Approved:

Susan M. Bridges Professor of Computer Science (Major Professor)	Rayford B. Vaughn Associate Professor of Computer Science (Committee Member)
Eric Hansen Assistant Professor of Computer Science (Committee Member)	Donna S. Reese Associate Professor of Computer Science (Committee Member)
Julian E. Boggess Associate Professor of Computer Science Graduate Coordinator Department of Computer Science	A. Wayne Bennett Dean of the College of Engineering

Name: German Florez

Date of Degree: December 13, 2002

Institution: Mississippi State University

Major Field: Computer Science

Major Professor: Dr. Susan M. Bridges

Title of Study: A TRUSTED ENVIRONMENT FOR MPI PROGRAMS

Pages in Study: 102

Candidate for Degree of Master of Science

Several algorithms have been proposed to implement intrusion detection systems (IDS) based on the idea that anomalies in the behavior of a system might be produced by a set of actions of an intruder or by a system fault. Almost no previous research has been conducted in the area of anomaly detection for high performance clusters.

The research reported in this thesis demonstrates that the analysis of sequences of function calls issued by one or more processes can be used to verify the correct execution of parallel programs written in C/C++ with the Message Passing Interface (MPI) in a cluster of Linux workstations. The functions calls were collected via library interposition. Two anomaly detection algorithms previously reported to be effective methods for anomaly detection in sequences of system calls, Hidden Markov Model and sequence matching, were implemented and tested. In general, the simpler sequence matching algorithm outperformed the Hidden Markov Model.

As a result of our experiments, MPIguard, the first distributed-IDS approach for a high-performance environment was implemented and a new dataset for the anomaly detection community was generated. MPIguard is highly portable and can be used as performance analysis tool or as a fault detection mechanism.

DEDICATION

To my family.

ACKNOWLEDGMENTS

I would like to express gratitude to Dr. Susan M. Bridges for her invaluable feedback on my ideas and for helping me through the master's program. I am also grateful to the members of my committee and the members of the Center for Computer Security Research at Mississippi State University, especially Yoginder Dandass for providing the source code of some parallel applications, Dr. Anthony Skjellum for the useful discussions about clusters and Linux security, Dr. Rayford Vaughn for his advice and guidance concerning my research work, and Miguel Torres for implementing the anomaly generation and attack programs.

This work was supported in part by the National Science Foundation Grant CCR9988524 and Army Research Laboratory Grant DAA 17-01-C-0011.

TABLE OF CONTENTS

Page

DEDICATION ii

ACKNOWLEDGMENTS iii

LIST OF TABLES vii

LIST OF FIGURES viii

CHAPTER

I. INTRODUCTION 1

1.1 Motivation 4
1.2 Hypothesis and main goals 4
1.3 Organization 5

II. LITERATURE REVIEW 6

2.1 Intrusion detection systems 6
2.2 Sandboxing 8
2.2.1 Certified code 8
2.2.2 Java 9
2.2.3 Janus 10
2.2.4 Execution monitor 11
2.3 Monitoring a cluster of computers 11
2.4 System call analysis 15
2.5 Process monitoring 19
2.5.1 Library interposition 19
2.5.2 Other techniques 22

III. SYSTEM OVERVIEW 23

3.1 A test environment 25
3.2 Interposition library 25
3.3 Profiler 29

CHAPTER Page

3.4 Analyzer 33
3.5 Networking 34
3.6 Adding new components 34

IV. ANOMALY DETECTION WITH FUNCTION CALLS 35

4.1 Hidden Markov Models 35
4.1.1 Description 35
4.1.2 The Baum-Welch algorithm 37
4.1.3 Anomaly detection with HMMs 39
4.2 Sequence matching 41
4.2.1 Description 42
4.2.2 Anomaly detection with sequence matching 43

V. OFFLINE DETECTION 45

5.1 Datasets 45
5.2 Artificial anomalies 46
5.2.1 Description 46
5.2.2 Hidden Markov Models 49
5.2.3 Sequence matching 55
5.3 Real attacks 57
5.3.1 Description 57
5.3.2 Hidden Markov Models 61
5.3.3 Sequence matching 69

VI. ONLINE DETECTION 71

6.1 Simulating online detection 71
6.1.1 Sequence matching 71
6.1.2 Hidden Markov Model 74
6.2 The MPI/C program's *profile* 80
6.3 MPIguard's *Analyzer* 84
6.4 Monitoring complex applications 84
6.4.1 Implementing LLCBench2 85
6.4.2 Monitoring LLCbench2 87
6.4.3 LU factorization 88
6.5 Performance 92

VII. CONCLUSIONS 97

Page

REFERENCES 99

LIST OF TABLES

TABLE Page

5.1 Average number of anomalies with real attacks 61

5.2 Training time for the HMM with *ring* 63

5.3 Training time for the HMM with *ring-modified* 65

6.1 Size of LLCbench2 traces 88

6.2 Number of false positives for LLCbench2 88

6.3 Performance of MPIguard for LU-factorization 95

6.4 Performance of MPIguard for LLCbench2 95

LIST OF FIGURES

FIGURE Page

1.1 Programs behavior 3

2.1 Simple state-transition rule 9

2.2 Java security 10

2.3 The Umpire manager 13

2.4 The automatic counting profiling tool 14

3.1 Simple security policy 24

3.2 Head node architecture of the Linux cluster 26

3.3 Template for any MPI or C function in MPIguard 27

3.4 Configuration file with the functions to be analyzed by MPIguard 27

3.5 Automatic source code for MPISend 28

3.6 Interaction between target process and Profiler 29

3.7 Example of the *Profiler*'s output 30

3.8 Comparison of latency with the LLCbench benchmark 31

3.9 Comparison of bandwidth with the LLCbench benchmark 32

3.10 Summary of LLCbench Bandwidth in one processor 32

3.11 Online analysis of the target process 33

4.1 An ergodic Hidden Markov Model with 4 states and 5 symbols 36

FIGURE Page

4.2 Reestimation of the HMM model using a joint event . . . 38

4.3 Example of a trained HMM model . . . 40

4.4 *CAAC* is not tagged as anomalous by the HMM . . . 41

4.5 Function calls . . . 42

4.6 Subsequences of function calls of length 3 . . . 42

4.7 Normal behavior represented by a sorted tree . . . 43

5.1 Set of possible function calls . . . 47

5.2 Anomaly data generation with *addition* . . . 48

5.3 Artificial anomalies for *ring* . . . 49

5.4 Quality of the HMM . . . 51

5.5 Detecting anomalies of *IS* using an HMM with 5 states . . . 52

5.6 Detecting anomalies of *IS* using an HMM with 8 states . . . 52

5.7 Detecting anomalies of *IS* using an HMM with 40 states . . . 53

5.8 Detecting anomalies of *ring* using an HMM with 5 states . . . 54

5.9 Detecting anomalies of *ring* using an HMM with 8 states . . . 54

5.10 Detecting anomalies of *ring* using Sequence Matching . . . 56

5.11 Detecting anomalies of *IS* using Sequence Matching . . . 56

5.12 Daemon attack for *ring* . . . 58

5.13 New function included in *ring* to handle files . . . 59

5.14 Interposer attacks for *ring -modified version-* . . . 60

FIGURE Page

5.16 Detecting daemons attacks in *ring* with a 8-state HMM 62

5.16 Detecting daemons attacks in *ring* with a 14-state HMM 63

5.17 Detecting daemons attacks in *ring* with a 20-state HMM 64

5.18 Interposer attack in *ring-modified* with a 8-state HMM 65

5.19 Interposer attack in *ring-modified* with a 17-state HMM 66

5.20 Interposer attack in *ring-modified* with a 20-state HMM 66

5.21 Detecting interposer attacks on processor 4 with a 8-state HMM 67

5.22 Detecting interposer attacks on processor 4 with a 17-state HMM 68

5.23 Detecting interposer attacks on processor 4 with a 20-state HMM 68

5.24 Detecting daemon attacks in *ring* using Sequence Matching 69

5.25 Detecting interposer attacks in *ring-modified* using Sequence Matching . . 70

6.1 Comparing normal behavior of *ring-modified-* using Sequence Matching . . 72

6.2 Detecting *File attack* in *ring-modified-*with Sequence Matching 72

6.3 Detecting *MPI attack* in *ring-modified-*Sequence Matching 73

6.4 Monitoring *ring-modified-* on processor 3 using a 17-state HMM 74

6.5 Detecting *File attack* in *ring-modified-* using a 17-state HMM 75

6.6 Detecting *MPI attack* in *ring-modified-* using a 17-state HMM 75

6.7 Monitoring *ring-modified-* on processor 3 using an HMM, *LFC=9* 78

6.8 Detecting *File attack* in *ring-modified-* using a 17-state HMM, *LFC=9* . . . 78

6.9 Detecting *MPI attack* in *ring-modified-* using a 17-state HMM, *LFC=9* . . 79

FIGURE Page

6.10 Comparing the execution of *ring* with *profile_p2* 81

6.11 Comparing the execution of *slaves* nodes of *ring* with *profile_p2* 82

6.12 Comparing the execution of *slaves* nodes of *ring* with *profile_p3* 83

6.13 The *main* function of LLCbench2 87

6.14 Distribution of the subroutines of LLCbench2 88

6.15 Detection *MPI-attack* at the beginning of LLCbench2 89

6.16 Detection *MPI-attack* at the ending of LLCbench2 90

6.17 Monitoring LU-factorization on 4 processors 91

6.18 Monitoring LU-factorization using a *profile* with 10 executions 92

6.19 Detecting *interposer* attacks in LU-factorization 92

6.20 Comparison of MPI_Send latency with LLCbench using MPIguard 93

6.21 Comparison of the bandwidth with LLCbench using MPIguard 94

CHAPTER I

INTRODUCTION

Several algorithms have been proposed to implement intrusion detection systems (IDS) based on the idea that anomalies in the normal behavior of a system might be produced by a set of actions of an intruder or by a system fault. Different components of an information system can be modeled by using different types of data. Network traffic, resource usage, operating system calls and log events are among the most important. However, we know of no previous work in IDS architectures that applies artificial intelligence techniques to the analysis of data collected from a high performance environment to detect abnormal behavior.

One of the first algorithms used to analyze traces from UNIX processes was implemented by Forrest and Longstaff [12]. They also produced and published a complete dataset of process traces that is widely used for testing system call analysis algorithms. Such algorithms generally are used to analyze the behavior of standard programs like *sendmail* or *lpr*. The same technique might be used to detect anomalies from any program given a considerable library of sample traces.

However, "as user-level library functions replace operating system calls as the preferred way for programs to efficiently use system services, the range of program activities

observable via system call tracing will decrease further. This will increase the need to monitor within applications and their libraries" [28, p.3]. Others data streams have also been used by researchers to implement IDSs. Examples include CORBA method invocations [36] and language library calls [18].

Actual UNIX and Linux systems support collection of information in real time from any process, including parallel programs implemented in C with a Message Passing Interface (MPI) architecture. With such techniques a tool can be deployed to restrict the execution of programs by enforcing a security policy on the network, memory and file accesses. This method is known as *sandboxing* because the target process is confined inside a safe environment. Another method used to control the execution of a process is a real-time host-based anomaly detector. Such a technique allows detection of deviations of the current process from a database that represents the "normal" behavior of the program. The database can be described in term of rules, state-transition machines, performance statistics, etc.

Figure 1.1 (taken from Ko, Fink and Levitt [19]) shows the behavior model of privileged programs. As we can see, the dotted lines correspond to security-relevant behavior of the program that can be gathered with audit trails, and it is different from the expected behavior (a benign program) and the actual behavior (possibly bad behavior). Some techniques control the expected behavior of a program (e.g. analysis of the source code), whereas others only check footprints of potentially bad behavior. As an example, in the model proposed by Ko, Fink and Levitt [19], sandboxing can be implemented by using

audit trials to create a specification-based model of intended behavior. The same model can be applied to a real time anomaly-detector.

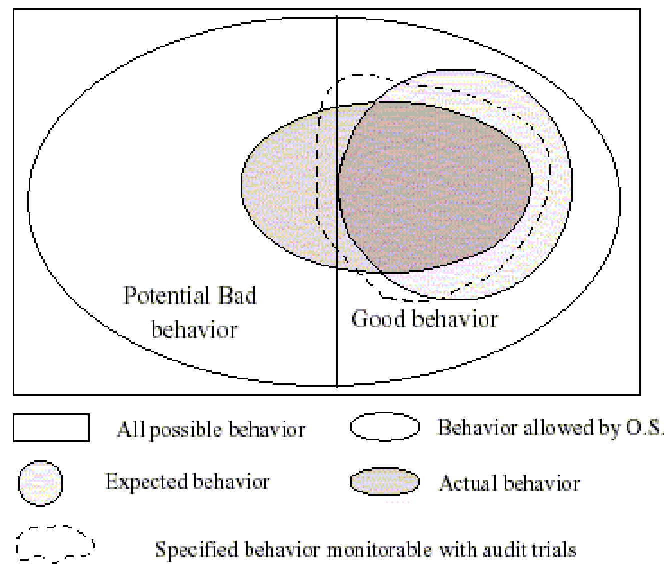

Figure 1.1 Programs behavior

A cluster of workstations is a special environment of interest because generally parallel programs running in such a distributed environment represent large and periodic tasks, and they can be modeled by using the same techniques used to create a signature of a process in a single host. Thus, we present the design of an anomaly detection model for a cluster of workstations that applies some of the concepts of *sandboxing*, performance monitoring and intrusion detection.

1.1 Motivation

As previously mentioned, we know of no work that has been published that applies artificial intelligence techniques to the analysis of data collected from a high speed network to detect abnormal behavior. Furthermore, given the amount of data produced in such an environment, many of the algorithms described in the literature cannot be used for detecting anomalies in (near) real time.

Data management and retrieval have also become interesting problems. Warrender et al. compared different algorithms for analysis of system calls and concluded that "perhaps a disproportionate amount of attention has been directed to the data modeling problem, and that equal attention should be paid to considering what are the most effective data streams to monitor"[43, p.145].

Finally, although a number of applications have been implemented to gather data from the execution of an MPI program via library interposition (Rabenseifner [30],Vetter and Supinski [41] among others), none were developed with intrusion detection as an objective. We have created a flexible tool to collect data useful not only for our anomaly detection framework, but also for the research community in the field of anomaly detection.

1.2 Hypothesis and main goals

We want to demonstrate that sequences of function calls can be used to verify the correct execution of an MPI parallel program in a cluster of Linux workstations.

The main goals of this research are:

1. Deploy an anomaly detector in a cluster of workstations able to verify the correct execution of parallel applications written in the C/MPI language. This detector should be accurate and produce low overhead.

2. Compare and contrast the detection rate and the performance of different machine learning algorithms for the anomaly detection task using function call traces of parallel programs. Function calls are captured using an interposition library mechanism.

3. Generate and document a new dataset for the IDS community.

1.3 Organization

The remainder of this document is organized as follows:

- Chapter II presents a survey of algorithms and concepts used in this research.
- Chapter III provides a brief analysis of MPIguard.
- Chapter IV describes three algorithms used for anomaly detection.
- Chapter V shows the accuracy and performance of our offline detection algorithms.
- Chapter VI presents MPIguard in online mode.
- Finally, Chapter VII reviews the most important characteristics of MPIguard and discusses limitations and future work.

CHAPTER II

LITERATURE REVIEW

2.1 Intrusion detection systems

An intrusion detection system (IDS) is an important component of the computer and information security framework, and its main goal is to differentiate between normal activities of a system and behavior that can be classified as suspicious or intrusive. Generally, it consists of a set of sensors, a set of analyzers and a user interface [1]. An IDS can be classified given the method used to detect the intrusion as an anomaly based or misuse based detection system [37]. An anomaly detection system assumes that an intrusion modifies the system behavior from its normal pattern. This approach can use statistical methods, sequence analysis or predictive pattern generation among others. In a misuse detection system, the IDS assumes that an intrusion can be detected by matching the current activity with a set of intrusive patterns (generally defined by security experts or "underground" web sites). Examples of methods used for misuse detection include expert systems, keystroke monitoring, and look-up tables for state-transition analysis.

Others classifications include the type of architecture of the IDS: centralized or decentralized [37], the level of the detection: application, host, network, or multinetwork; and the time in which the intrusion is detected: offline or online [1].

The accuracy of an IDS depends not only on the algorithm used, but also on the quality of the data used for training and testing. The creation of such data is not an easy task, because it involves a knowledge of networking, operating systems, programming languages and information security. The generation of normal data (attack free) depends on system wide features such as the topology and architecture of the network, the number and type of hosts, users, etc. Thus, one of the first steps to create a complete dataset is to define the best way to gather representative data of the system being studied. For instance, in order to create data for the DARPA-MIT intrusion detection evaluation three different approaches were considered [4]: [1]

1. Collect real operational Air Force network traffic.
2. Preprocess real traffic changing sensitive data
3. Create an abstract model of the traffic to generate normal and attack patterns on a private network (this last approach was used for the 1998 and 1999 evaluations).

Another well-known data set for (host-based) intrusion detection was created at the University of New Mexico [12]. This repository [2] contains real and synthetic executions of different UNIX programs, with some intrusions such as buffer overflows, symbolic link attacks, and Trojan horse programs. Finally, it is important to note that no work has been done in a high performance environment to create similar datasets and make them available to the research community

[1] Accessible from `http://ideval.ll.mit.edu`

[2] Currently stored in `http://www.cs.unm.edu/ immsec/systemcalls.htm`

2.2 Sandboxing

The process of verifying a program's behavior can be done at compilation time or execution time. At compilation time (static checking), the flow and properties of the algorithm are verified taking into account a set of rules representing the host policy (a typical example is the Java Byte-Code Verifier). On the other hand, the runtime checker performs a verification of all "dangerous methods," such as network connections and files accesses while the program is being executed.

2.2.1 Certified code

Certified code is the method used to verify security properties of untrusted source code. Before running the application, "the host checks annotations and proves that they imply the host's security policies" [42, p.1]. Examples of certified code include the Java Virtual Machine (JVM), Efficient Code Certification (ECC), and Proof-carrying code (PCC) [42]. However, those methods concentrate exclusively on standard safety properties. On the other hand, systems like the Walker's Secure Automata [42] can apply any policy (described in terms of states and transitions) at compilation time. Afterwards, the verification module in this architecture checks the resulting code and links the program with the rest of the application.

As an example, Figure 2.1 (taken from Walker [42]) shows a single policy for a remote applet that can be expressed as: do not perform a *send* operation over the network after reading a file in the host.

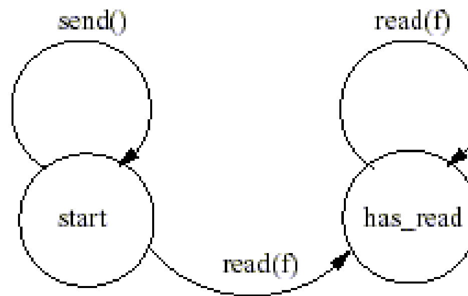

Figure 2.1 Simple state-transition rule

2.2.2 Java

Java contains a large set of security features that allow a host to run remote classes (and local ones) without risk. Basically, the Java Security Module is composed of the Byte-Code Verifier, the Applet Class Loader and the Security Manager. The Byte-Code Verifier is a static process that validates the untrusted code before it is executed. Also, Java is a type-safe language: "the compiler ensures that methods and programs do not access memory in ways that are inappropriate" [26] . The Applet Class Loader and the Security Manager handle the execution of the applet at runtime. In Figure 2.2 (taken from Sun Microsystems [26]) we can see that the Java compiler creates trusted byte-code in the local host, whereas the Byte-Code Verifier checks the untrusted code. Then, the applet class loader fetches the applet's code from the remote host, creates a namespace (to prevent replacing system-levels components) and forbids the execution of native class loader processes. While the applet is being executed, the Security Manager controls the socket operations, files access, thread integrity and verifies the access to other Java packages.

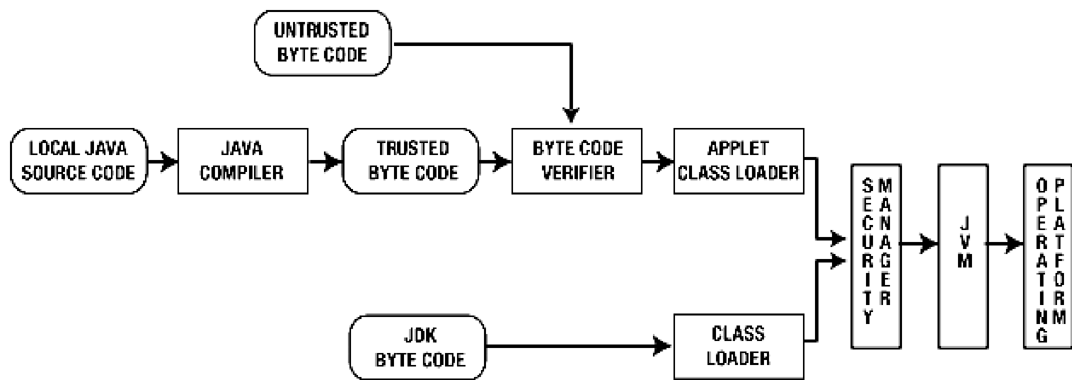

Figure 2.2 Java security

2.2.3 Janus

Janus is a set of tools that allows the execution of Linux processes in a restricted environment [14]. Such a restriction is defined by security policies written by the system administrator, and its main objective is to disallow harmful system calls. Janus is built upon the following assumption: "An application can do little harm if its access to the underlying operating system is appropriately restricted" [14, p.1] . The policy modules are dynamically loaded at runtime. Thus, each host, user or even application can be assigned a different policy. An example of a security policy is:

1. The application cannot execute *chdir* (change directory).
2. Access paths containing the string ".." are denied.
3. The application can read some generic files, like shared libraries or configuration files, but access to other files is restricted.

Janus itself is intended to be secure, because the design and the implementation is based on simplicity, and the whole Janus architecture has less that 4500 lines of code, and a detailed code review can be done to assure a secure (trusted) environment.

2.2.4 Execution monitor

The Execution Monitor restricts the execution of privileged programs- those with the ability to bypass the kernel's security mechanisms, like *sendmail* or *finger* [19]. For each target program, a policy (a database of logical expressions) is defined to allow and disallow different system calls. The trace of a program produced by the operating system (using tracing tools llike the Sun BSM *log*) is transformed into logical expressions that are compared with the policy in real time. If the system finds a mismatch, a violation is reported.

2.3 Monitoring a cluster of computers

A large collection of tools available to monitor the behavior of a cluster of workstations has been designed to analyze network information. Although networking is the most critical component in a high-performance computer, monitoring the communication among workstations is not enough to describe the system behavior. "Also, unlike in general purpose networks, in most cluster setups it is possible -and worthy- to install custom agents in the nodes" [2, p.59].

Typical intrusion detection systems face the problem of large data volume due to the speed of communication and processors. This situation often causes the number of false positive and false negative alarms to become unacceptably high. The possibility of detecting an intrusion in such an environment in real time is a difficult task. In addition, the generation of useful information may require computational resources that the cluster could spend on others tasks. For that reason, the definition of the level and the correct representation of the data that should be generated (and stored) for the IDS is a complex problem.

Interposition library techniques (discussed in some detail in the following chapters) has been used successfully to monitor MPI (Message Passing Interface) programs. Some of the implementations try to detect typical programming errors in MPI, whereas others use this technique to create logs and statistical reports. For instance, Umpire "monitors the MPI operations of an application by interposing itself between the application and the MPI runtime system using the MPI profiling layer" [41, p.1] . This tool checks for specific behavior patterns to detect, among others, deadlock and resource exhaustion. Figure 2.3 (taken from Vetter and Supinski [41]) describes this process.

Function calls from the MPI application are captured by the MPI profiling layer and are transmitted to a central agent, the Umpire Manger, via shared memory to analyze and verify the execution of the parallel program. The shared memory corresponds to a specific region of memory that is accessible by the MPI profiler and the Umpire Manager. This reduces overhead dramatically.

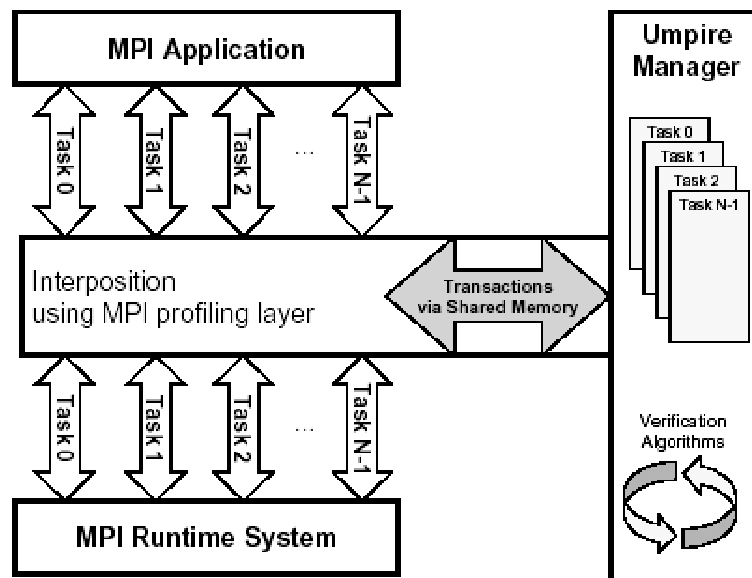

Figure 2.3 The Umpire manager

As another example, Rabenseifner [30] implements an automatic MPI counter profiling by using the concept of an interposition library to gather the number of function calls, the time spent and the total number of bytes at the end of a parallel job. This information is stored in a file and it is used to create statistical reports each week and month. Figure 2.4 (taken from Rabenseifner [30]) shows the general architecture of the Automatic Counting Profiling.

This tool is able to gather information such as number of processes, wall clock time of the application, and number of calls to each MPI function (because of the amount of information, the counter is only incremented after 128 calls)

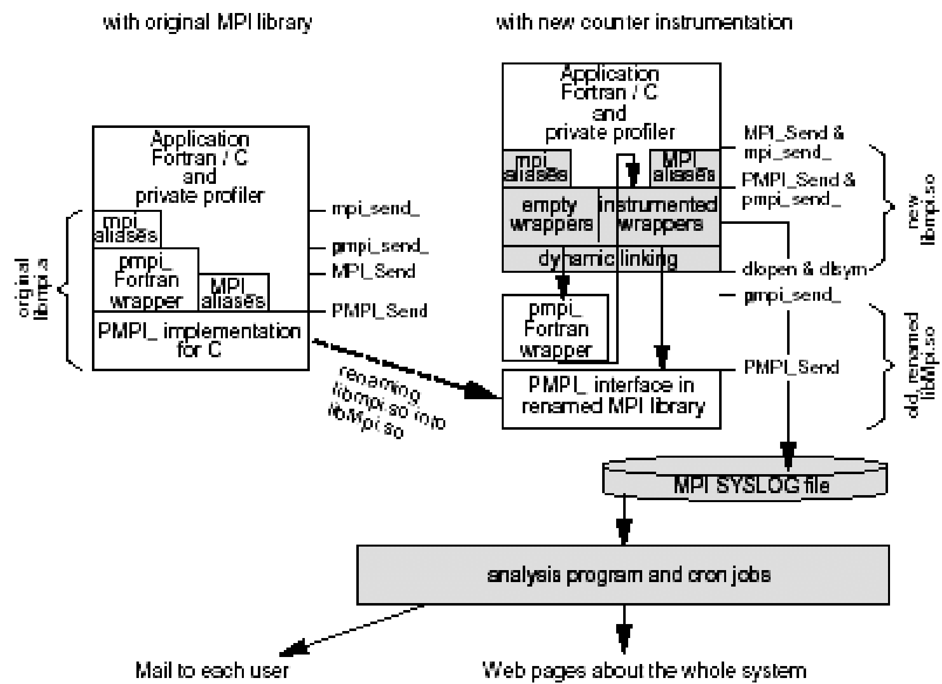

Figure 2.4 The automatic counting profiling tool

2.4 System call analysis

"A system call is usually a request to the operating system (kernel) to do a hardware/system specific or privileged operation" [15] . Generally it is implemented by using the *syscall()* low-level command, which accepts as parameters the number of the system call and its arguments. A UNIX-like system has more than 150 system calls.

Function libraries are high-level implementations of system services, allowing the execution of language primitives, complex functions and system calls. Thus, we can consider that the function calls from the C standard library are placed in a layer above the defined system calls. We also can assume that the problems faced by researchers analyzing system calls are similar to the problems encountered when analyzing functions calls.

In the field of machine learning and data mining, many algorithms are used to solve the problem of analysis of temporal sequences of events. However, the brief discussion presented here describes only some of the algorithms used for the system call analysis task in the intrusion detection area.

Forrest and Longstaff [12] reported one of the first research projects taking this approach. They used sequences of system calls within a window size (the length of the sequence) to compare exact matches between the normal profile and the new trace of a process. This is known as the *stide* algorithm. An extended overview is described in [35]. Given the following trace of system calls (without parameters):

open, read, mmap, mmap, open, read, mmap

We can generate several consecutive sequences of length k. With k=3 we have:

open, read, mmap
read, mmap, nmap
mmap, mmap, open
mmap, open, read

This information is stored in a tree structure and corresponds to the database of normal behavior. When a new (unseen) sequence is presented, the system tries to find it in the tree. If it is not found in the tree, a counter of anomalies is increased. If this value exceeds a user threshold, an alarm is fired. An additional measure to detect an anomaly in the stide algorithm is the difference between the abnormal sequence (i.e. not presented in the tree) and the normal ones. This can be achieved finding the minimal Hamming distance [12]. An improvement over the *stide* algorithm is *t-stide* algorithm: it stores the frequency for each sequence in the database. Then, rarely matched sequences (for instance sequences with a frequency of 0.001%) in the data presented to the system will be also labeled as anomalous [12]. Another frequency-based algorithm is the *EMERALD* system [29], where the model compares short-term frequencies with the database of historical distribution.

Somayaji [33] extends some of this concepts to create pH, a system able to detect and respond to changes in sequences of system calls. When an process is behaving unusually, pH responds by slowing down the system calls of such a process.

Eschenauer [7] implements host-based anomaly detection using the same concept, but he also describes other footprints that can be analyzed, such as the EIP (program counter) from the system call issuer.

Markov processes are widely used to model systems in terms of state transitions. Some intrusion detection algorithms that exploit the Markov property implement Hidden Markov Models(HMM), Markov chains, and sparse Markov trees. Lane [38] used HMMs to profiler user identities for the anomaly detection task. An open problem with this profiling technique is the ability to select appropiate model parameters. Others experiments performed by Warrender, Forrest and Pearlmutter [43] compared the HMM with algorithms such as *s-tide* and RIPPER. They concluded that the Hidden Markov Model exhibited the best performance of the models considered but was the most computationally expensive.

Eskin et al. [8] constructed a probabilistic prediction model able to determine the last system call of a given sequence by using sparse Markov trees. An important contribution of this work is the use of information theory to find the minimum entropy of the data set. Experimental results showed that the system achieves the best performance using the minimum entropy to compute the value of the window size.

One of the most widely used rule-based algorithms in the intrusion detection field is the Repeated Incremental Pruning to Produce Error Reduction (*RIPPER*) [3]. This is a rule learning system developed by William Cohen. His algorithm performs classification by creating a list of rules from a set of labeled training examples. RIPPER also is used for anomaly detection by Lee and Stolfo [23] to predict system calls, where the classification label of each training example corresponds to the last call of the sequence. Each generated rule is associated with a number called *confidence*, given by the equation

$$\frac{100M}{T} \tag{2.1}$$

where M corresponds to the number of records that satisfies the rule's conditions, and T is the number of times the prediction of the rule was corrected. If a new (unseen) sample is presented to the system and violates high-confidence rules (for instance, with a confidence greater than 80%) the sample is labeled as an anomaly.

The same concept of prediction of sequence calls can be implemented using neural networks [20]. The main advantages of such an approach are the lack of dependence on any statistical assumption, noise tolerance, and abstraction.

In previous work [24] we have successfully applied three different neural network topologies (multilayer perceptron trained with backpropagation, radial basis functions trained with a perceptron and self-organizing maps) to detect attacks on well-known UNIX programs, such as *lpr* and *sendmail*. Multiple self-organizing maps have also been for used network intrusion detection in [32].

We have demonstrated [10, 11] that the accuracy of such neural networks can be improved by applying Adaboost (a boosting by resampling technique). In some experiments we were able to improve the classification rate by 15% or more, achieving in some cases 100% accuracy in the detection of anomalies.

Finally, Wespi et al. [44] present an intrusion detection system using pattern matching with fixed and variable length sequences. The first method uses a static table to try to find an exact (or similar) match of the new sequence, whereas the last method uses a tree to store each normal pattern, taking into account that several patterns start with the same string. The intrusion detection described in their work executes filtering, reduction and

aggregation of the sequences (for the experiments, they used audit events from the FTP daemon). Their results show that the concept of variable length sequences improves the IDS accuracy.

2.5 Process monitoring

Operating systems give the user several tools to control the execution of a process in real-time. We are interested in the implementation of a dynamic library to collect information from function calls, but others mechanisms can be used to achieve such a goal.

2.5.1 Library interposition

The link editor (*ld*) in a Linux operating system builds dynamically linked executables (although it can also build statically linked programs). Building "incomplete" executables, the link editor allows the incorporation of different objects in real time. The communication between the main program and the objects is done by shared memory operations. Such (shared) objects are called dynamic libraries: "A dynamic library consists of a set of variables and functions which are compiled and linked together with the assumption that they will be shared by multiple processes simultaneously and not redundantly copied into each application" [5, p.1]. The dynamic library is linked without an entry point (i.e. without an inclusion of the program prologue *crt0*) [13]. In other words, a dynamic library cannot be executed by itself. As an example, take the simplest C program: the "Hello world" application (*hello.c*), which contains [13]:

printf ("Hello World");

Generally, the user compiles this program with a command like this:

cc -o hello hello.c

Such a command line invokes:

ld -e Start -dc -dp -o hello /lib/crt0.o hello.o -lc

The *ld* command creates an incomplete *hello* executable, requiring the inclusion of the C standard library, *libc.so.V* (V is the current version) at execution time. When *printf* is executed, *ld.so*, the execution-time linker, searches in the symbol table of the *hello* application and then the *libc.so* library for the definition of any command named "printf" with the same parameters. Because this is a standard C function, *printf* from the *libc.so* library is executed. Afterwards, the execution-time environment saves a pointer with the reference of such a function to avoid searching again [13].

In a Linux system, the link editor uses the LD_PRELOAD environment variable to search for the user's dynamic libraries [5]. Using this feature, the operating system gives the user the option of interposing a new library. Interposition is "the process of placing a new or different library function between the application and its reference to a library function" [5]. Thus, the library interposition technique allows intercepting the function calls without the modification or recompilation of the dynamically linked target program. By default the C compilers in LINUX use dynamic linking. Furthermore, "most parts of the Linux libc package are under the Library GNU Public License, though some are

under a special exception copyright like *crt0.o*. For commercial binary distributions this means a restriction that forbids statically linked executables" [15]. The user functions (i.e. the functions inside the interposition library with the same prototype as the "real" ones) are able to check, record and even modify the arguments and the response of the original function call. Other advantages include [5]:

1. A subset of the library can be profiled instead of the whole library.
2. Different levels of profiling can be generated.
3. Nesting levels inside the library can be controlled.

The main disadvantage of the interposition library technique is that it can be bypassed by calling functions at a lower level (for instance, executing system call interruptions) [17]. The process of searching for the symbol table in the new interposition library and the allocation of memory does increase the execution time of the target process. As an example, Kuperman and Spafford ([21]) present some experiments that show that functions not defined in the shared library are slowed by 340 nanoseconds, whereas functions defined are slowed by 6.53 microseconds on a Red Hat Linux 5.2 machine.

The interposition library technology is widely used and many wrappers and tools have been implemented. Curry [5] shows an application of this technique in detail with the Shared Library Interposer (SLI) library. In his work, Curry describes the new problems that arise because of the interception of calls from the *libc.so*. As an example, the interposer program might use *libc* functions to perform the profiling, and these calls should not be taken into account.

2.5.2 Other techniques

UNIX and Linux operating systems provide a large collection of mechanisms to trap system and function calls from any process. For instance, to monitor kernel calls, the OS provides the tools *strace*, *trace* and *truss* [21]. However they only record kernel level functions, and the trap mechanism produces too much overhead [5].

Interception of system calls can be built inside the OS kernel by adding code extensions. This allows low interception overhead and speed (used for real-time applications), but the system itself is “less secure” by adding extensions, and the implementation requires super-user privileges (it requires kernel modification), so it will affect any process in the upper layers of the OS [17].

By using the *ptrace()* special system call a target process can be monitored by another program. When the kernel reads a system call from the target program, the monitor awakes and takes control. Before the kernel processes the call, the monitor can do a pre-call process [14]. After the system call is executed by the kernel, it is able to do a post-call process. The monitoring process can extend Linux (and other POSIX system) capabilities to modify system call data and it is considered very efficient. However, overhead is required to provide a secure monitoring process, and *ptrace()* traces every system call, not just the interesting ones. Also in many operating systems it is impossible to abort a system call without killing the process being monitored [14]. A similar monitoring mechanism can be implemented with the */proc* interface instead of *ptrace()* [14].

CHAPTER III

SYSTEM OVERVIEW

We present MPIguard, a new architecture to verify the correct execution of parallel programs (written in C/MPI) over a Linux cluster by monitoring library calls issued in each host. The system is designed as a host-based anomaly detector similar to sandboxing architectures . It is able to control in (near) real-time a process being executed.

A program might violate the policy of the system (stored as a database of legal traces of the program) because of the following conditions:

1. Corrupted MPI programs
2. Corrupted data.
3. Deadlocks.
4. Failure of cluster components (such as networking or disk failures).

Modification of a program's source code is very unlikely in a "closed" cluster environment, where the security policies of one (or more) central nodes control the access of the users to the cluster. The risk of corrupted MPI programs is higher for "enterprise" or "non dedicated" clusters, where each node is able to connect to the Internet to access a remote database or a remote service. However, we assume that an unauthorized user cannot modify MPIguard by changing setup files or removing components.

MPIguard uses an interposition library technique to store on disk the function calls from the standard C and the MPI/PRO libraries (although many others can be profiled) in a Linux node. This trace file represents the normal behavior of each one of the MPI programs that the system administrator wants to control. Somayaji defines normal behavior as "behavior that is observed when we are reasonably certain no activity is occurring that requires non-routine direct human administrative observation and interaction" [33, p.92]. Note that this definition of normal behavior includes not only security-related events, but also misconfigurations and failures in the system.

When an MPI program is being executed, the current sequence of function calls is compared with the normal trace of the program. If MPIguard detects that the program is behaving unusually an alarm is fired.

It is important to note that our system can be adapted to include several policy and recovery rules, like the one presented in Figure 3.1. In this example any memory request of more than 2000 Kbytes of memory with the function *malloc* (memory allocation) is forbidden.

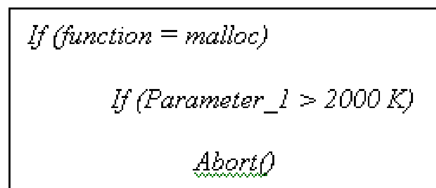

Figure 3.1 Simple security policy

MPIguard assumes that the parallel program follows the Single Program Multiple Data (SPMD) paradigm, i.e. the same program is executed in all the nodes of the cluster. This implies that the anomaly detector in each one of the nodes of the cluster is trained with the same dataset [1]. However, a parallel application running different processes in each node (no SPMD) could be monitored with MPIguard by using a trace file for each of the nodes.

3.1 A test environment

MPIguard was implemented and tested with a Linux cluster built at Mississippi State University in the Computer Science Department. This cluster contains one head node (*microcosm0*) and 8 slaves, each with 4 processors. The nodes are fully connected with Ethernet and Giganet (high-speed) networks. Figure 3.2 shows this architecture.

3.2 Interposition library

This library captures function calls (with their parameters) from a MPI/PRO executable. It performs two main processes:

1. Send information about the function to the *Profiler* and the *Analyzer*.
2. Execute the function.

MPIguard automatically generates the source code needed to gather information from any function. Figure 3.3 shows the template used to generate the code for any MPI/C function.

[1] Many of the real-world applications written in MPI follow the SPMD paradigm.

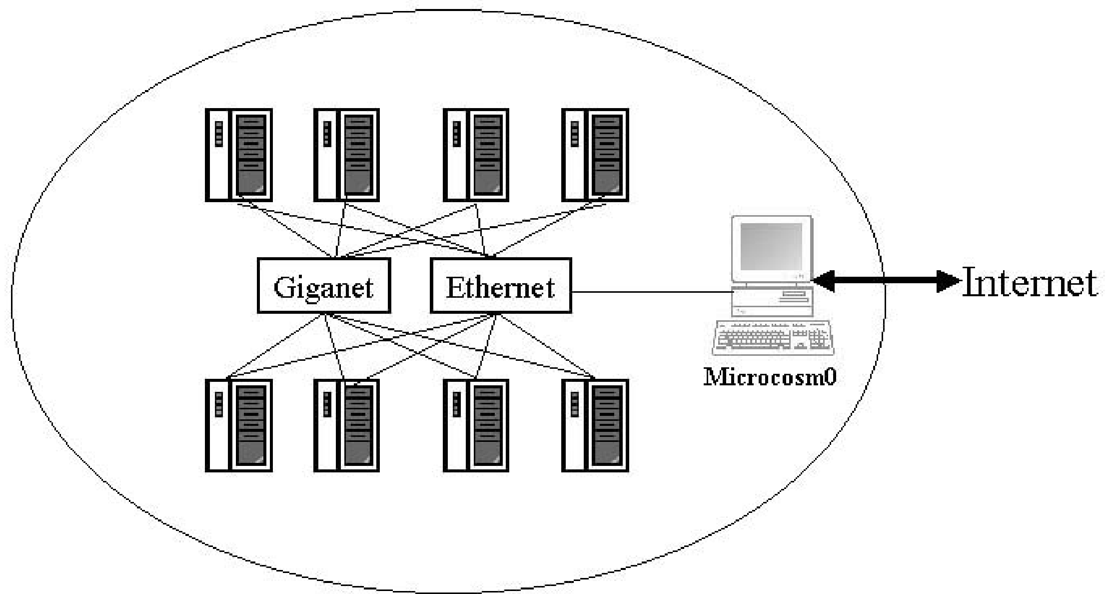

Figure 3.2 Head node architecture of the Linux cluster

The system administrator interacts with MPIguard indicating which functions and parameters will be analyzed. Basically, a configuration file is used to describe the type, the name and the parameters of the functions. Figure 3.4 shows an example of this file. *Code* is an internal code for each function (this value is stored on disk instead of the real name of the function) and *Parameter to store in disk* is the parameter (an integer that generally corresponds to some buffer's size) that will be written in the log file. A value of -1 indicates that the function's parameter is not important for the user or the function has no parameters.

Using the template described in Figure 3.3 and the information of MPI_Send in Figure 3.4, MPIguard is able to generate the source code shown in Figure 3.5.

```
__FUNCTYPE __MPIAPI_C __FUNCREALNAME (__FUNCPARAM)
{
      typedef __FUNCTYPE(*function_type) (__FUNCPARAM);
      static function_type function=NULL;
      static char* function_name=__FUNCNAMESTRING;

      __TYPERETVAL __DEFINERETVAL

      if (!function) {
            __HANDLEMPILIBRARY
            __OPENMPILIBRARY
            function = (function_type) dlsym(__HANDLER,function_name);
            __CLOSEMPILIBRARY
      }

      if ((!ThisLibraryCall) && (DoProfile)) {
            ThisLibraryCall=TRUE;
            //execute the funtion and then profile
            __ASSIGNRETVAL ((*function)(__FUNCNOTYPEPARAM));
            PROFILE(__FUNCID,
                  __FUNCTOLOGPARAM);
            ThisLibraryCall=FALSE;
      }
      else //do not profile, only execute
            __ASSIGNRETVAL ((*function)(__FUNCNOTYPEPARAM));

      __RETURNRETVAL
}
```

Figure 3.3 Template for any MPI or C function in MPIguard

TYPE	NAME	CODE	C-LIKE HEADER	PARAMETERS	PARAMETER TO STORE IN DISK
int	MPI_Scatterv	85	void *sendbuf, int *sendcount, int displs, MPI_Datatype sendtype, void *recvbuf, int recvcount, MPI_Datatype recvtype, int root, MPI_Comm comm	sendbuf, sendcount, displs, sendtype, recvbuf, recvcount, recvtype, root, comm	*sendcount
int	MPI_Send	86	void *buf, int count, MPI_Datatype datatype, int dest, int tag, MPI_Comm comm	buf, count, datatype, dest, tag, comm	count
int	MPI_Wait	93	MPI_Request *request, MPI_Status *status	request, status	-1

Figure 3.4 Configuration file with the functions to be analyzed by MPIguard

```
int MPIAPI_C MPI_Send (void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm)
{
	typedef int(*function_type) (void *buf,int count,MPI_Datatype datatype,int dest,int tag,MPI_Comm comm);
	static function_type function=NULL;
	static char* function_name="MPI_Send";

	int  retval;

	if (!function){
		void* handle;
		handle=dlopen("/usr/lib/libmpipro.so",RTLD_LAZY);
		function = (function_type) dlsym(handle,function_name);
		dlclose(handle);
	}

	if ((!ThisLibraryCall) && (DoProfile)){
		ThisLibraryCall=TRUE;
		//execute the funtion and then profile
		retval = ((*function)(buf,count,datatype,dest,tag,comm));
		PROFILE(86,
			count);
		ThisLibraryCall=FALSE;
	}
	else //do not profile, only execute
		retval = ((*function)(buf,count,datatype,dest,tag,comm));

	return (retval);
}
```

Figure 3.5 Automatic source code for MPISend

3.3 Profiler

This process receives messages from the interposition library and collects information about the system calls. Its main goal is to store the program's normal behavior on disk, although it can be used to generate the trace of any MPI/C program. This feature allows storing the traces of different executions of the same program for off-line analysis.

Figure 3.6 shows the interaction between the *Profiler* and the target process using the interposition library. As we can see, the interposition library sends information in real time to the *Profiler* via shared memory.

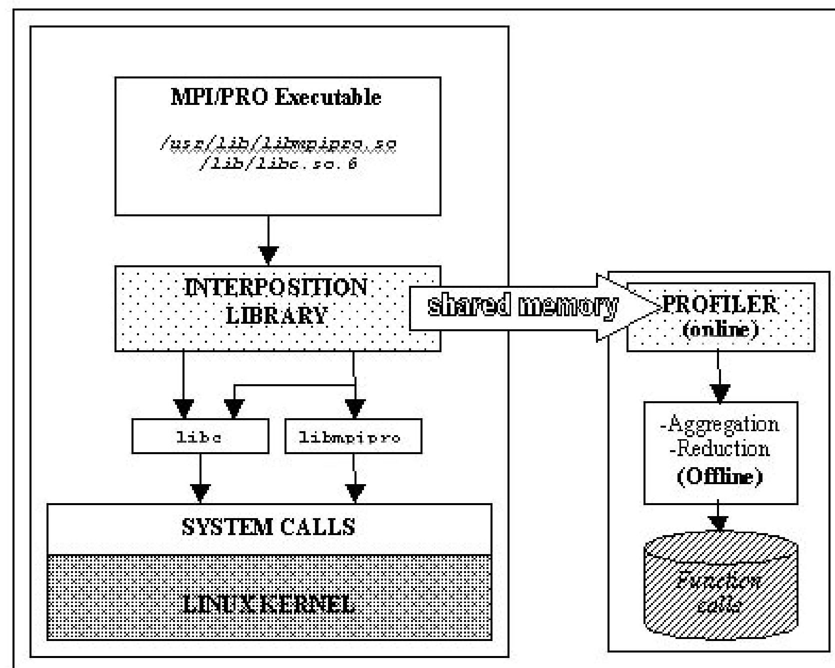

Figure 3.6 Interaction between target process and Profiler

An example of the *Profiler*'s output can be seen in Figure 3.7. It is an ASCII file containing the process identifier (PID), the internal code of the function call and the optional integer parameter.

PID	CODE	SIZE
5693	4	1029
5693	5	1
5693	6	33257
5693	4	1029
5693	5	1
5693	6	33257
5693	79	1
5693	72	1
5693	86	1

Figure 3.7 Example of the *Profiler*'s output

As was mentioned before, including a new library does increase the execution time of the process. We have executed LLCbench [27], a parallel performance benchmark implemented in the University of Tennessee at Knoxville, to measure the overhead produced by the interposition library and the *Profiler*. We have used 4 processors and averaged the results of 20 experiments.

Figure 3.8 shows the latency of MPI_Send (function that sends a message to one processor and blocks the current task until the message is received) with different packet sizes. Small packets result in an overhead of a few microseconds and for large packets the overhead is almost imperceptible.

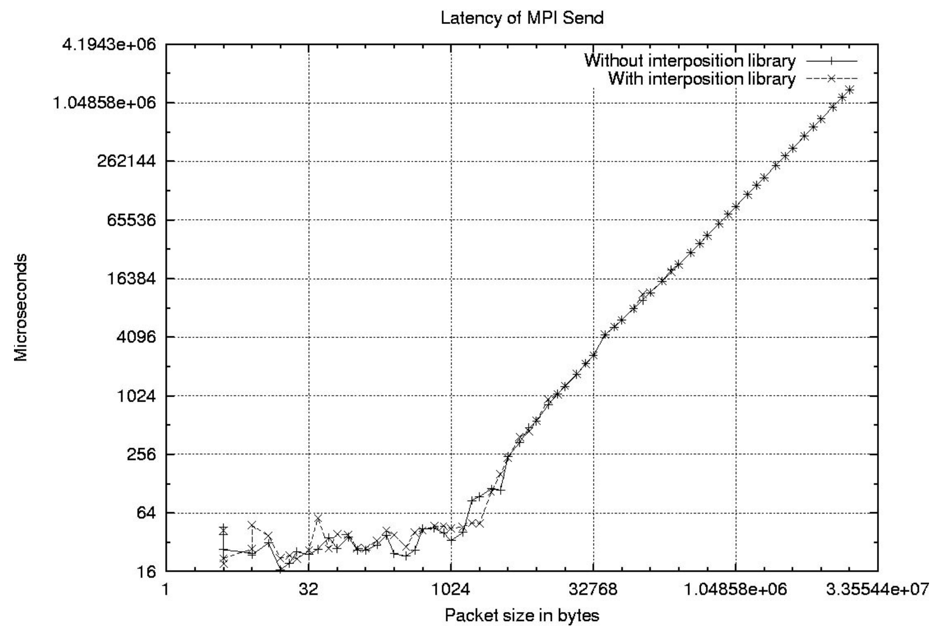

Figure 3.8 Comparison of latency with the LLCbench benchmark

Figure 3.9 shows the average bandwidth between the master and one slave processor using unidirectional transmissions. For small packets (up to 1024 bytes) the shape of the graph of the benchmark run without the interposition library indicates a somewhat better bandwidth and for large packets the reduction in the bandwidth is very close to 0.

Different information can be extracted from the log files generated by our *Profiler* tool, and it can be used for performance monitoring or debugging. For example, Figure 3.10 shows a summary of the function calls executed by LLCbench-bandwidth in one processor.

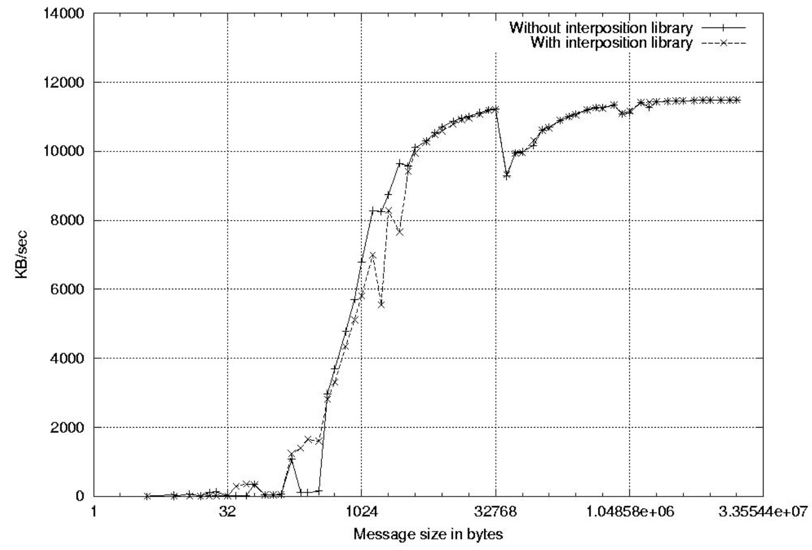

Figure 3.9 Comparison of bandwidth with the LLCbench benchmark

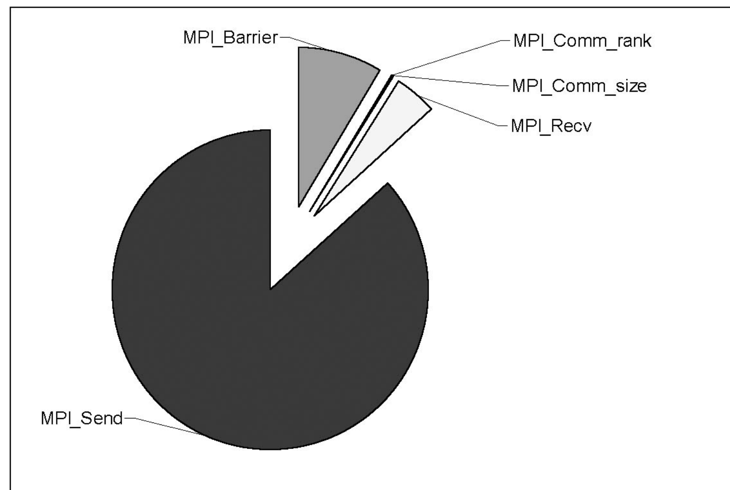

Figure 3.10 Summary of LLCbench Bandwidth in one processor

3.4 Analyzer

The *Analyzer* is the principal component of MPIguard and its position in the MPI-guard architecture is illustrated in Figure 3.11. This application is trained with the current executable's normal behavior (generated by the *Profiler*).

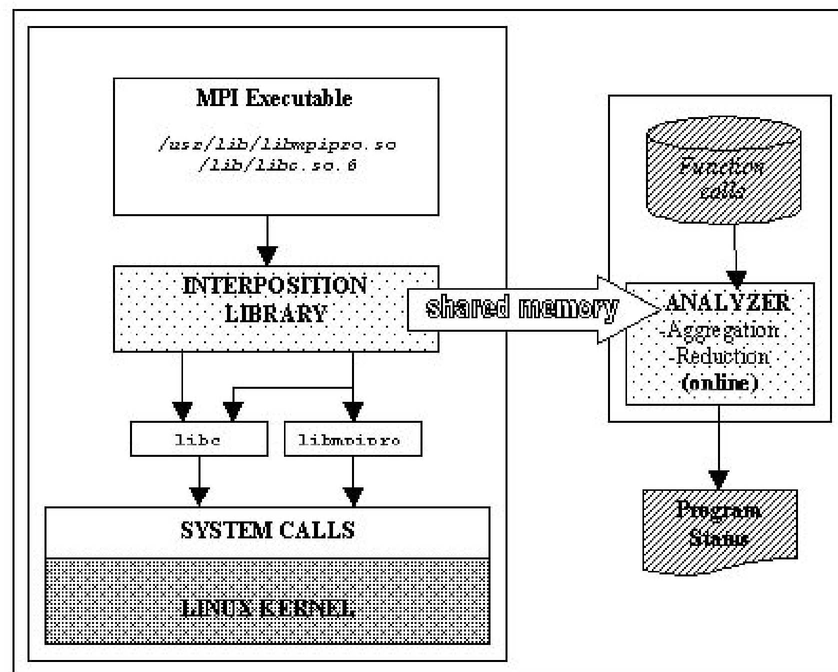

Figure 3.11 Online analysis of the target process

When MPIguard is running, the interposition library sends sequences of function calls to the *Analyzer* in real-time, and the *Analyzer* compares the current trace with the *profile* stored on disk for the program that is being executed. If the sequences are being recognized as suspicious activities, an alarm is fired. In Chapter IV we will explain the algorithms that

we have used to detect deviations from normal behavior of parallel programs in the nodes of a Linux cluster.

3.5 Networking

If the anomaly detector is being executed in real time in each node of the cluster (i.e. the interposition library and the *Analyzer* are running), MPIguard can collect all the process status information at a given time and present it to the system administrator as an ”overall” state of the system. This communication would not dramatically increase the normal network traffic in the cluster.

3.6 Adding new components

Each component in the MPIguard system can be seen as a software agent. The *Profiler* and the *Analyzer* could potentially receive information from other applications, not only the instance of the interposition library being executed. Such agents might be able to monitor CPU usage, authentications and networking. Several architectures using agents for intrusion detections are described in the literature (e.g. [16, 34]).

CHAPTER IV

ANOMALY DETECTION WITH FUNCTION CALLS

Several algorithms can be implemented to find patterns from a process using information about its function calls (for example, by comparing timings or relative frequencies, or analyzing the parameters [33]). In our research, we are using Hidden Markov Models and sequence matching to detect abnormal program behavior using only the type of the function call and its relative order in a trace.

4.1 Hidden Markov Models

Hidden Markov Models (HMM) are used for modeling sequences of events and are widely used for speech recognition and DNA sequencing.

4.1.1 Description

"A Hidden Markov Model describes a doubly stochastic process. An HMM's states 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, p.135]. Figure 4.1 (adapted from [31]

shows an ergodic [1] HMM with 4 states. In each state there is a probability of observation for each of the elements of the alphabet *a,b,c,d* and *e*.

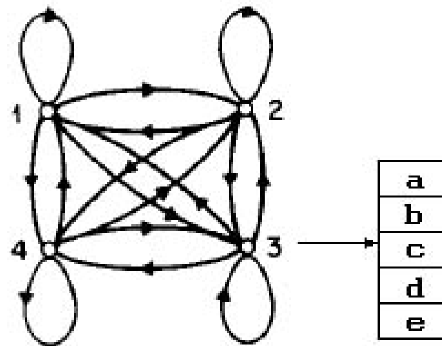

Figure 4.1 An ergodic Hidden Markov Model with 4 states and 5 symbols

The elements of an HMM are [31]:

1. *N*, the number of states.
2. *M*, the number of distinct observation symbols per state (the alphabet size).
3. *A*, the state transition probability distribution.
4. *B*, the observation symbol probability distribution.
5. π, the initial state distribution.

For convenience, an HMM model can be expressed as

$$\lambda = (A, B, \pi) \tag{4.1}$$

Rabiner [31] describes three different problems that must be solved with an HMM when we have an observation sequence $O = O_1 O_2 O_3 ... O_\tau$:

[1]Every state can be reached from any other state.

1. How do we compute $P(O|\lambda)$?
2. How do we choose the state sequence that best explain O?
3. How do we maximize $P(O|\lambda)$ adjusting $\lambda = (A, B, \pi)$?

4.1.2 The Baum-Welch algorithm

The Baum-Welch algorithm is generally used to train the transition and symbol probabilities of an HMM [31] (attempting to solve problem 3). This is an iterative algorithm that computes *A, B* and π based on the concept of forward-backward probabilities. The forward procedure finds the probability of the partial observation sequence from the first event to some event *O(t)* at time *t*, whereas the backward procedure finds the probability of the partial observation from *O(t+1)* to the end.

The forward variable can be defined as

$$\alpha_t(i) = P(O_1 O_2 ... O_t, q_t = Si|\lambda) \tag{4.2}$$

It corresponds to the probability of the partial observation sequence *O* until time *t* and state S_i at time *t*, given the model λ .

The backward variable is the probability of the partial observation from *t+1* to the end, given the state S_i at time *t* and the model λ , and it can be expressed as

$$\beta_t(i) = P(O_{t+1} O_{t+2} ... O_\tau | q_t = S_i, \lambda) \tag{4.3}$$

The Baum-Welch algorithm updates the model λ using the probability of being in state S_i at time *t* and state S_j at time *t+1*, given the model and the observations. This variable, $\xi_t(i, j)$ is illustrated in Figure 4.2 (taken from [31]).

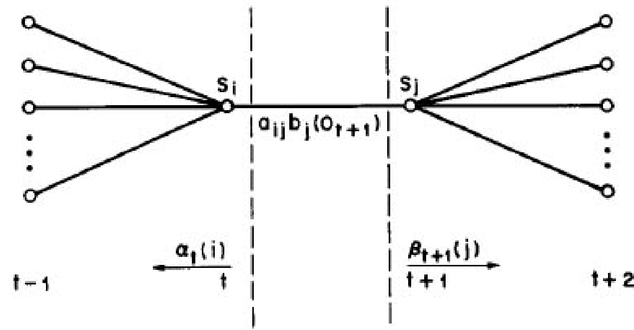

Figure 4.2 Reestimation of the HMM model using a joint event

With ξ we can compute the probability of being in state S_i at time *t* given the model and observation sequence

$$\gamma_t(i) = \sum_{j=1} \xi(i, j) \tag{4.4}$$

The initial state distribution can be computed as the expected frequency (number of times) in state S_i at time $t = 1$

$$\overline{\pi}_i = \gamma_1(i) \tag{4.5}$$

The state transition probability distribution is given by the expected number of transitions from the state S_i to state S_j divided by the expected number of transitions from state S_i

$$\overline{a}_{ij} = \frac{\sum_{t=1} \xi_t(i, j)}{\sum_{t=1} \gamma_t i} \tag{4.6}$$

The observation symbol probability distribution can be computed as the expected number of times in state j and observing symbol v_k divided by the expected number of times in state j

$$\overline{b}_i(k) = \frac{\sum_{\substack{t=1 \\ s.t.\ O_t = v_k}} \gamma_t(j)}{\sum_{t=1} \gamma_t(j)} \tag{4.7}$$

Thus, $\xi(i,j)$ can be computed using the new values of $\overline{\pi}_i,\ \overline{a}_{ij}$ and $\overline{b}_i(k)$. This process is repeated several times until some limiting point is reached.

4.1.3 Anomaly detection with HMMs

In order to train the HMM with the Baum-Welch algorithm, we should specify the observation sequence *O*. This is *NT*, the function call trace of the normal MPI/C program produced by the *profiler*. No further preprocessing should be done. With an optimal model λ , we can assume that the probabilities *A* and *B* generalize the normal behavior of the process.

Given a new observation $\overline{O}$ (that corresponds to the trace of an unseen instance of the MPI/C program, named *UT*), we can apply the following algorithm [43]:

1. Using the model λ
2. For each one of the observations $\overline{O_t}$.
    - For each one of the states S_i (if the state can be reached from the previous one, i.e., if the probability of moving to the current state is greater than some user threshold θ) .
        - If the probability of producing the symbol $\overline{O_t}$ in the current state $B(i, \overline{O_t})$ is less than θ then the function call in the trace *UT* is labeled as anomalous
    - If $\overline{O_t}$ could not be produced by any state (i.e. the function call in the trace was tagged as anomalous in each state S_i) then the counter of anomalies *C* is increased.
3. If *C* exceeds some user threshold ρ then the entire trace *UT* is tagged as anomalous.

Although it could be said that comparing the probability of moving to the current state S_i and the probability of producing a symbol $\overline{O_t}$ in the current state with the same user threshold θ has no mathematical foundation, we wanted initially to include the smallest

number of parameters that we could conceive for the anomaly detection task with the HMM.

Figure 4.3 shows an example of an ergodic HMM with two states and 3 possible symbols. Using the algorithm described before, with $\theta = 0.3$ and $\rho = 0$, Figure 4.4 shows that the trace *CAAC* could be produced by that HMM, since there is at least one possible state producing every symbol $\overline{O_t}$. For instance, in step *iii.*, *A* could not be produced in state 1, but it could be produced in state 2 with a probability of 0.4.

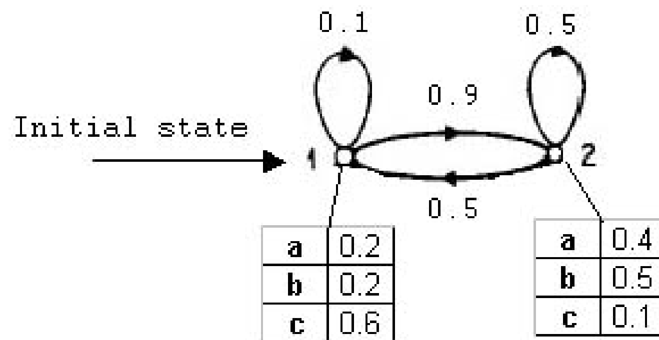

Figure 4.3 Example of a trained HMM model

An improvement on this algorithm counts the number of abnormal function calls using some *tolerance factor* μ . If the number of abnormal function calls between the current observation $\overline{O_t}$ and the last μ observations is equal to μ , then the counter of anomalies *C* is increased. This concept is similar to the *locality frame count LFC*, used by Somayaji [33],

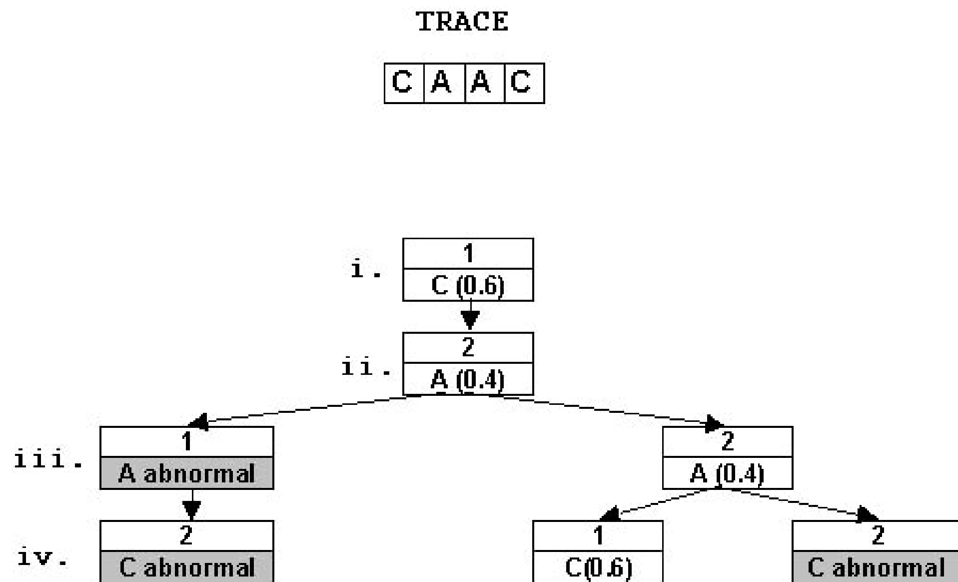

Figure 4.4 *CAAC* is not tagged as anomalous by the HMM

where the main idea is to count the number of anomalies in the last *LF* calls to estimate the delay needed to penalize the system calls of the anomalous process.

As an example, if μ is 20, and we have only detected 5 anomalies in the last 20 observations, the total counter of anomalies of the MPI/C program is not incremented. This algorithm is suitable for on-line detection.

4.2 Sequence matching

The sequence matching algorithm is the simplest approach to detect abnormal behavior and security violations of a program.

4.2.1 Description

We used the *sliding window* concept that is widely used in the analysis of system calls [12, 35, 33, 24, 10, 11] to create subsequences of the function calls of an MPI/C application. The sliding window divides a trace of N function calls into a set of small sequences, each one of length ω (the window size). As an example see in Figure 4.5 a program trace. With $\omega = 3$ the first sequence can be obtained with positions 1, 2 and 3 in the trace. The second sequence corresponds to positions 2, 3 and 4. The third is created using positions 3, 4 and 5 and so on. The result of this process is a database like to the one presented in Figure 4.6

MPI_Init	MPI_Send	MPI_Receive	MPI_Wait	MPI_Send

Figure 4.5 Function calls

MPI_Init	MPI_Send	MPI_Receive
MPI_Send	MPI_Receive	MPI_Wait
MPI_Receive	MPI_Wait	MPI_Send

Figure 4.6 Subsequences of function calls of length 3

Somayaji has created a formal definition of these subsequences [33]. Let C be the alphabet of possible system calls, $c = |C|$, $T = t_1, t_2, \ldots, t_\tau | t_i \in C$, τ the length of T, ω

the window size ($1 \leq \omega \leq r$), P the *profile* (set of patterns associated with T and ω). A sequence P_{seq} is defined as

$$P_{seq} = \{\langle si, s_{i+1}, \ldots, s_j \rangle : s_i, s_{i+1}, \ldots, s_j \in C, 1 \leq i, j \leq \tau, j - i + 1 = \omega, \} \quad (4.8)$$

This database can be stored as a sorted tree to perform efficient comparisons. Figure 4.7 shows an example of a *profile* with a window of length 4.

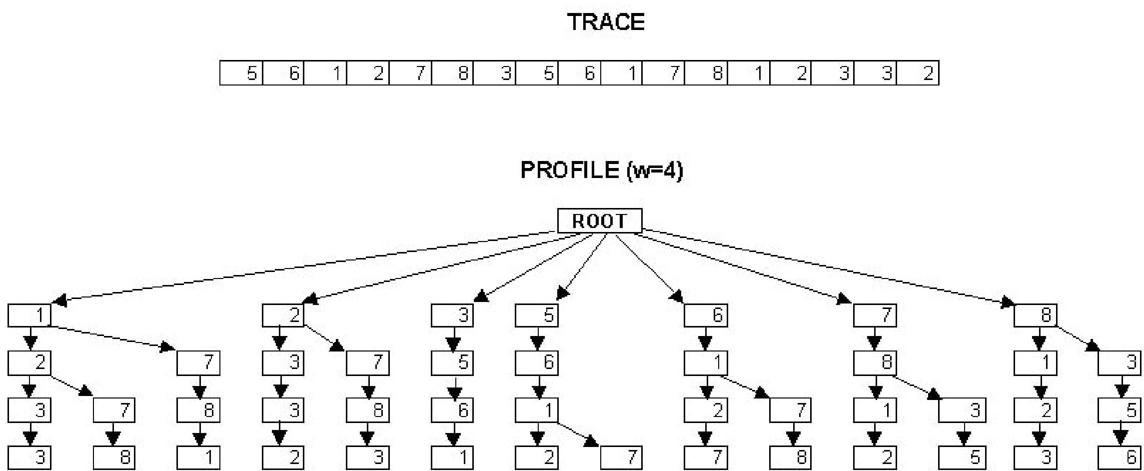

Figure 4.7 Normal behavior represented by a sorted tree

4.2.2 *Anomaly detection with sequence matching*

With exact matching, if a sequence $\overline{S} =< \overline{S_i}, \overline{S_{i+1}}, ..., \overline{S_j} >$ (from the unseen trace *UT*) is not contained in P_{seq} , the sequence $\overline{S}$ is tagged as anomalous. Researchers have demonstrated that this method is very efficient and it is able to detect security violations and software failures [33, 35, 43, 12]. Some modification of this method include:

1. Compute the *Hamming distance* between the new sequences $\overline{S}$ and each one of the sequence of P_{seq} [35].

2. Compute the frequency of each sequence in P_{seq} . If the sequence $\overline{S}$ (from *UT*) matches with a sequence in P_{seq} with very low frequency, $\overline{S}$ is still tagged as anomalous [43].

3. Compute the mismatches rate per sequence [12]. It is important to observe that the number of anomalies reported by the sequence matching detection algorithm might exceed the number of abnormal function calls.

   As an example take a *profile* containing sequences of length 3 created with all the possible combinations of the alphabet *a,b,c,d,e*. When a new trace such as *abXde* is compared with the *profile*, the sequences *abX, bXd, Xde* will be flagged as anomalous. However, the only anomalous symbol is *X* [2].

   The maximum number of mismatches for a sequence of length *L* and window *k* is

$$k(L-k)+(k-1)+(k-2)+...+1=k(L-(k+1)/2) \tag{4.9}$$

[2]In contrast, the number of anomalies detected by the HMM does correspond to the number of function calls that can not be generated with the model

CHAPTER V

OFFLINE DETECTION

The main goal of MPIguard is to detect deviations of an MPI/C program from its expected normal behavior, called the *profile*. In order to achieve a (near) on-line detection, we first analyzed the accuracy and efficiency of the algorithms described in Chapter IV. However we do not know, apriori, any decision boundary between normal and abnormal instances of MPI/C programs.

5.1 Datasets

We are using MPIguard's *Profiler* tool to gather MPI function calls from two programs: *IS* ("Interger Sort", included in the NAS parallel benchmark suite [45]) and a very simple MPI program called *ring*. It is important to observe that both programs satisfy the SPMD constraint of MPIguard.

IS sorts keys in parallel with a problem size of 2^{23} numbers. This program generates a small trace consisting of about 40 calls in each node using 9 function calls. The *ring* application sends messages from one processor to the next one defined in its MPI_COMM_WORLD (default handler for MPI communication). The last processor sends a message to the first one creating a "closed ring" architecture. This process is

repeated several times. This program generates a large trace with about 6400 calls in each node, but using only 5 function calls.

Both programs were executed using 4, 8 and 16 processors in the Linux cluster at Mississippi State University (Figure 3.2). The detection algorithms were run on an Intelx386, 64MB RAM with Linux Mandrake.

5.2 Artificial anomalies

Using MPIGuard's *profiler* we can generate a trace of the function calls issued by any MPI/C program (Figure 3.7). If we can assure that the program is executed under normal conditions, we can use such trace as a representation of the program's normal behavior (*profile*).

5.2.1 Description

Assuming that possible anomalies of a MPI/C program will generate patterns that are similar to the *profile* ([9, 22, 24]), a simple but useful heuristic that can be used to create a synthetic trace representing abnormal behavior is to randomly change the value of some of the calls in the "normal" trace. Both the type of the system call and its argument can be modified.

Torres [39] has implemented two programs that interact with MPIguard's *Profiler* files to generate synthetic datasets. Both programs receive as input a configuration file containing the set of possible functions that can be generated by the MPI/C program. Figure 5.1

shows an example of the contents of this file. It is a subset of the *Profiler*'s configuration file (Figure 3.4).

NAME	CODE	PARAMETER
MPI_Alltoallv	6	*sendcounts
MPI_Irecv	72	count
MPI_Irsend	73	count
MPI_Isend	74	count
MPI_Issend	75	count
MPI_Wait	90	-1

Figure 5.1 Set of possible function calls

Torres includes in his programs *randomlib*, a C library to generate random numbers based on Fibonacci sequences implemented at Florida State University [25]. This is one of the best algorithms to create random numbers. The artificial anomalies are created either by modification or by addition:

1. *Modification*: The type of the function call (internal code) is changed for any other valid code with a probability ϕ (see Figure 5.1).

   If the function call is going to be changed, the value of its parameter will also be modified with a probability of 0.5. Since we assume that the value of the parameter corresponds to the size of a transmission or a buffer, the new value is a power of 2 less than 65536.

   If the function call does not change, there is a probability of $0.3\overline{3}$ of changing the value of its parameter. Not all the functions have parameters. If this is the case, the value of the parameter is -1.

2. *Addition*: A new (valid) function call is created with a probability ϕ and the value of its parameter is generated as a power of 2 less than 65536. In order to create more diversity, the value of the parameters of the function calls are changed with a probability of 0.5.

In our research we have generated anomalies using the *addition* method. An example can be seen in Figure 5.2.

NORMAL

CODE	SIZE
20	-1
21	0
28	128
30	64
35	-1
12	512

ABNORMAL

CODE	SIZE
20	-1
21	0
28	128
26	256
30	64
35	-1
20	-1
12	512

Figure 5.2 Anomaly data generation with *addition*

Figure 5.3 shows the distribution of artificial anomalies in pattern space that were injected into a dataset containing a trace of the normal execution of *ring* in one node. The x-axis corresponds to the identifier of the function call, the y-axis corresponds to the position in the trace and the z-axis shows the size of the parameter that was stored on disk by the *Profiler*. Although we do not believe that real deviations from an MPI/C program's *profile* contain such a high number of anomalies, the artificial anomaly generator using *addition* helps us to demonstrate the ability of detection of our IDS.

Both *IS* and *ring* were executed with 4 processors. Selecting any two processors other than the master node (*rank=0*) two datasets $Normal_{p1}$ and $Normal_{p2}$ were generated. Applying the artificial anomaly generation algorithm to $Normal_{p1}$ and $Normal_{p2}$ we have created $Abnormal_{p1}$ and $Abnormal_{p2}$.

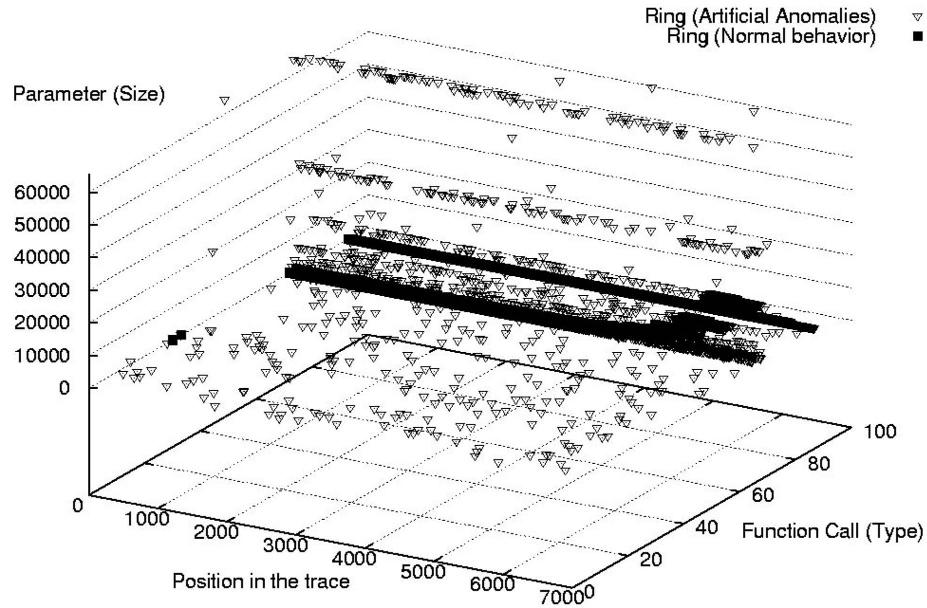

Figure 5.3 Artificial anomalies for *ring*

5.2.2 Hidden Markov Models

As explained in section 4.1.3, an HMM λ trained with a dataset representing normal behavior of an MPI/C program can be used as a *profile* to detect anomalies. Detection with this algorithm is very efficient since the trace generated by the MPIguard's *Profiler* can be used directly as input for the detection algorithm without preprocessing.

By training the HMM using the Baum-Welch algorithm we can obtain optimal values for A , B and π . However, we must specify the number of states in the model *N* and the threshold for detection θ (the minimum allowed probability of transition between states and the minimum allowed probability of producing a symbol) and ρ (the maximum allowed number of anomalous function calls in the trace). We believed we could use HMMs

with few states because the alphabet size for normal behavior of *IS* and *ring* is less than 10. Since we do not know of any useful heuristic to determine the parameters of the model, we conducted a set of experiments to determine the impact of θ and ρ on the accuracy of the detection. The results of these experiments are plotted in graphs, where the *x-axis* correspond to the user threshold θ and the *y-axis* correspond to the percent of anomalies detected in the trace.

The overall shape of the graphs help us to determine the capabilities of the HMM. For example, in Figure 5.4, the ability of the HMM to represent normal behavior can be described as $X \cdot Y$ where X is the maximum θ that can be used to distinguish between normal and abnormal behavior (when θ is greater than X, several function calls are labeled as anomalous no matter what trace is presented to the HMM) and Y is the difference between the number of anomalies detected for normal and abnormal traces when θ is lower than X. Y is related with the accuracy of the HMM and X is related with its generalization capability. If Y is small (close to 0) the HMM is not able to differentiate between normal and abnormal behavior. If X is small, the probabilities of the matrices A, B and π are very small.

It is important to observe that the area $X \cdot Y$ gives us an idea of the behavior of the HMM, but it cannot be used to estimate its detection accuracy. In the next chapter we will compute the total number of anomalies instead of the percent of anomalies in the trace to compute false positive and false negative rates.

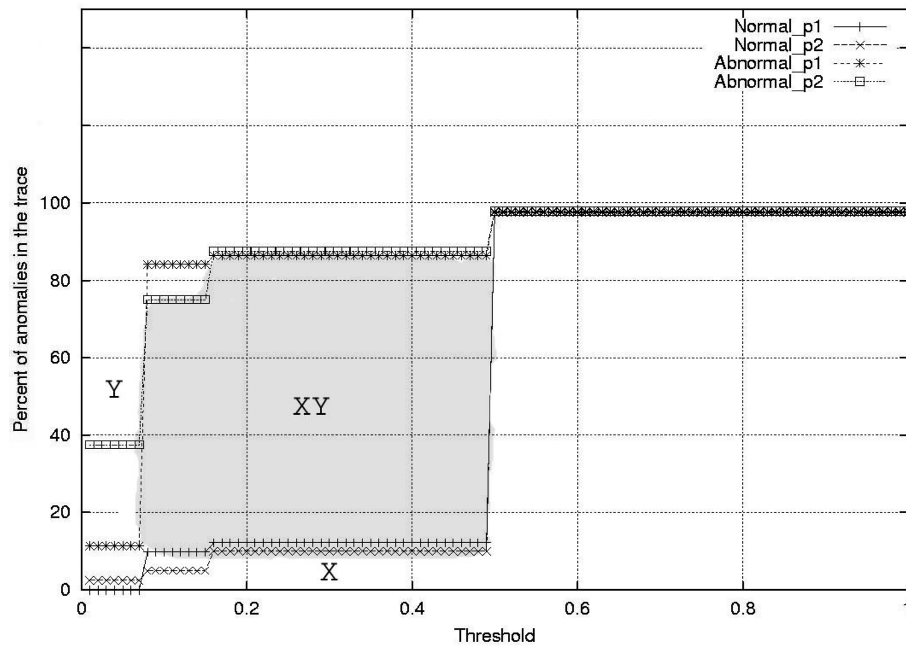

Figure 5.4 Quality of the HMM

Figure 5.5 shows the number of anomalies detected by the algorithm using $Normal_{p1}$ for *IS* as training data for an HMM with 5 states and $0 \leq \theta \leq 1.0$. When $\theta > 0.5$ the algorithm detects almost 100% percent of the anomalies for each trace, even for the training dataset $Normal_{p1}$. Thus, the number of false positives is overwhelming. However, by using $0.2 \leq \theta < 0.5$ there is a distinction between normal and abnormal traces and by selecting an appropriate ρ (e.g. $\rho = 40$) we can accurately classify the program trace with no false positives or false negatives.

Figure 5.6 and Figure 5.7 show results from the same experiment with 8 and 40 hidden states. Increasing the number of states does not change the detection capability of λ.

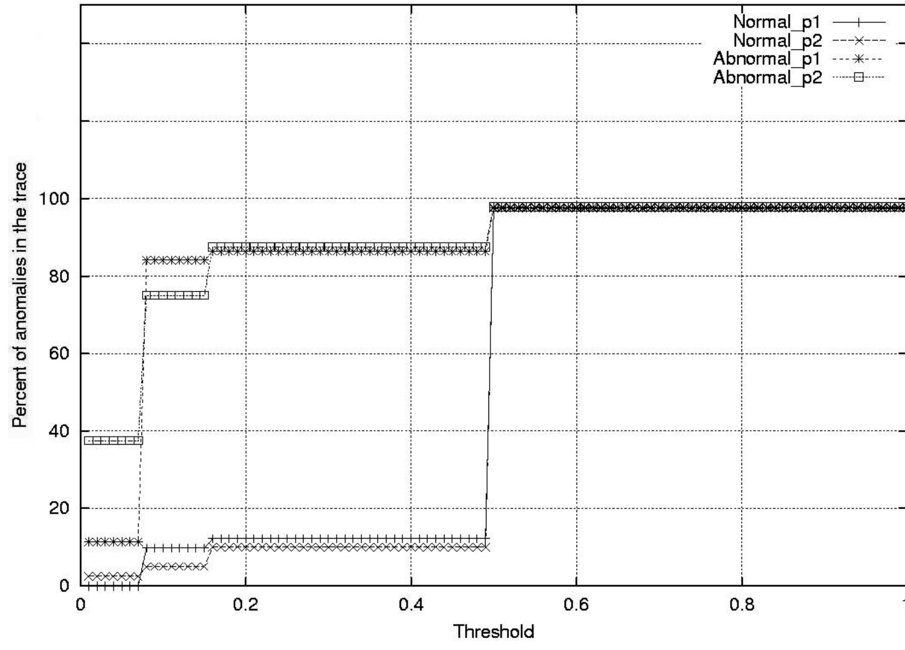

Figure 5.5 Detecting anomalies of *IS* using an HMM with 5 states

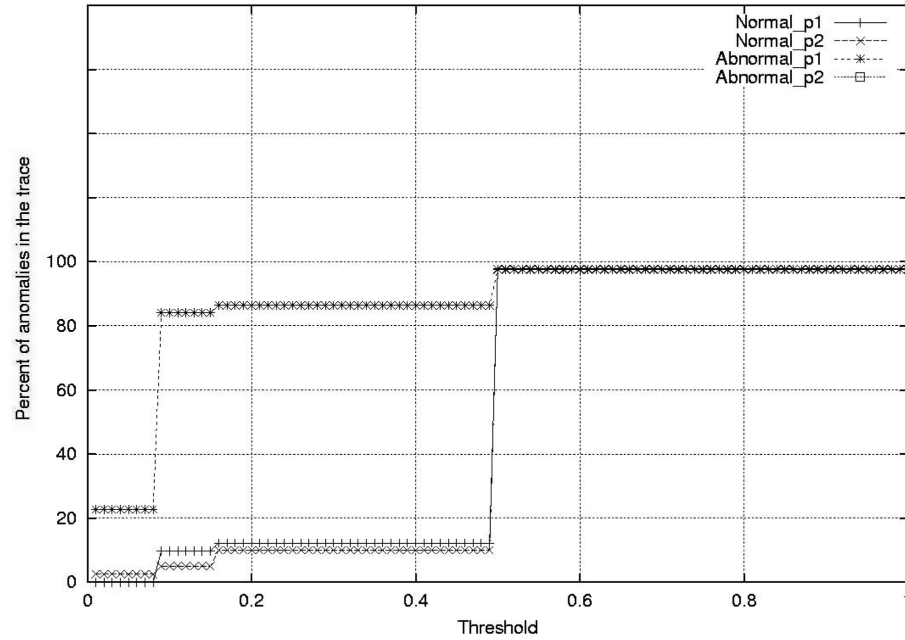

Figure 5.6 Detecting anomalies of *IS* using an HMM with 8 states

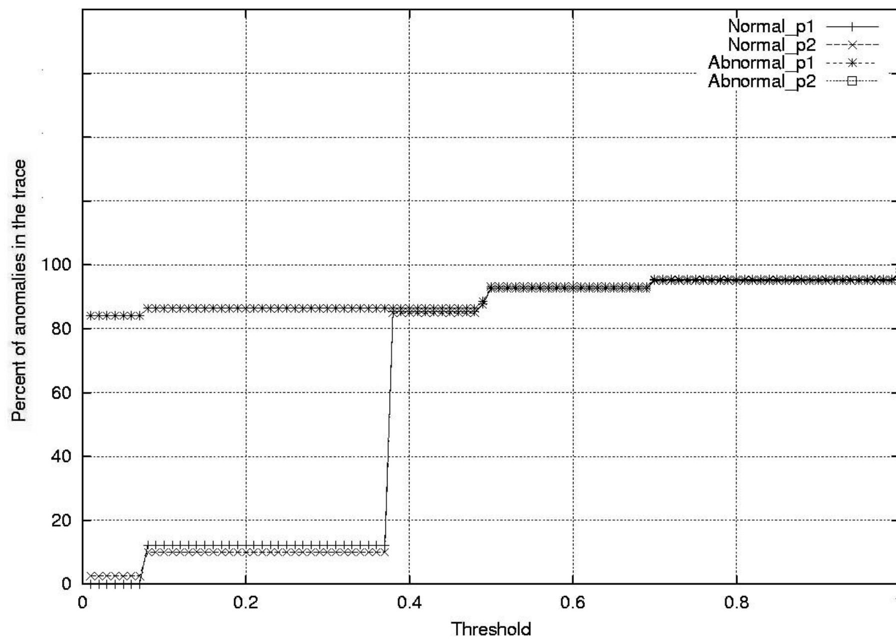

Figure 5.7 Detecting anomalies of *IS* using an HMM with 40 states

Figure 5.8 and Figure 5.9 show the detection accuracy using traces from the *ring* program with 5 and 8 states (we were not able to find λ with 40 states). With these models X is very large (above 0.8) and by choosing $\rho = 40$ we can achieve perfect accuracy. Again, increasing the number of states seems not to affect the representation of normal behavior by the HMM.

Since the alphabet needed to represent normal behavior of *IS* and *ring* contains few symbols, training the HMM is not as computationally expensive as we would expect. For *IS* with a small trace containing about 40 calls the Baum-Welch algorithm takes less than 1 second with 5 states and less than 2 seconds with 40 states and For *ring* with a large trace containing more than 6000 calls, the Baum-Welch algorithm takes under 240 seconds with

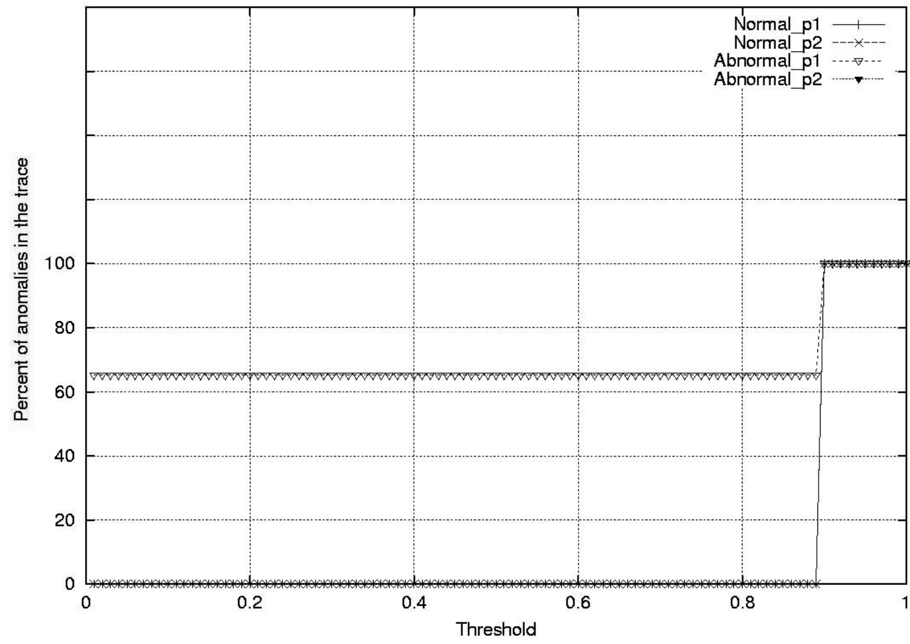

Figure 5.8 Detecting anomalies of *ring* using an HMM with 5 states

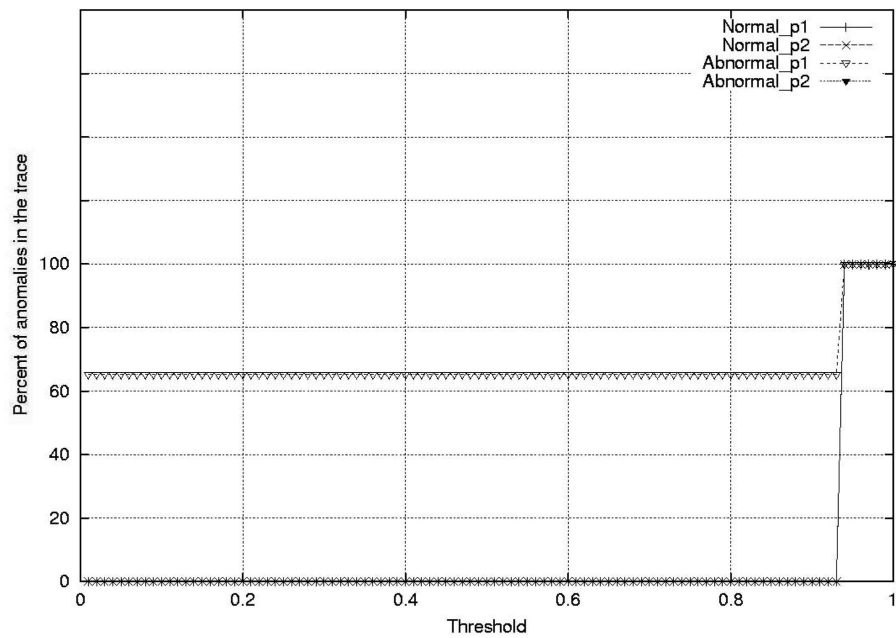

Figure 5.9 Detecting anomalies of *ring* using an HMM with 8 states

5 states and under 150 seconds with 8 states. Testing (detection) using the trained model takes less than 1 second for both *IS* and *ring*.

5.2.3 Sequence matching

Section 4.2.2 described how the *profile* created using sliding windows can be used to detect anomalies in an MPI/C program (see Figure 4.7). The only parameter needed for such an algorithm is the window size ω . The following experiments were conducted to find the impact of ω in the detection accuracy of the basic sequence matching algorithm. Figure 5.10 shows the percent of anomalies in each trace when comparing different traces of *ring* with the *profile* $Normal_{p1}$ using 4 window sizes. Note that we are including the function call traces of a third processor: $Normal_{p3}$ and $Abnormal_{p3}$. The number of anomalies detected for the traces $Normal_{p2}$ and $Normal_{p3}$ is 0 and the number of anomalies detected for the traces $Abnormal_{p1}$, $Abnormal_{p2}$ and $Abnormal_{p3}$ is greater than 20%. Increasing the sequence length in the algorithm does increase the number of anomalies detected. However we will see in the following chapter that when using the sequence matching algorithm with real attacks, the window size does not have such an impact on the detection rate.

Figure 5.11 shows the results the same experiment using *IS* traces. The behavior of the algorithm is similar to the one described for *ring* although the IDS reports that the dataset $Abnormal_{p3}$ contains more than 40% of anomalies.

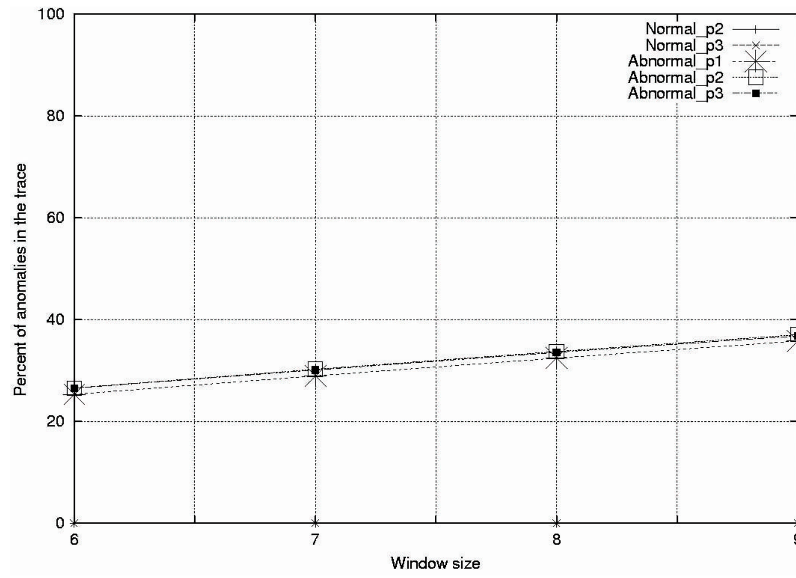

Figure 5.10 Detecting anomalies of *ring* using Sequence Matching

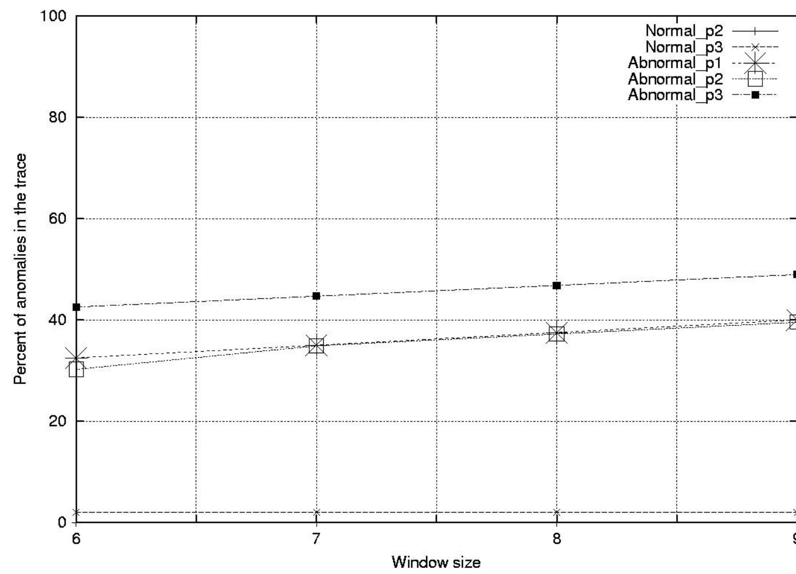

Figure 5.11 Detecting anomalies of *IS* using Sequence Matching

Creating the *profile* using sorted trees is a very efficient task. The running time for both training and testing using sequence matching takes under 1 second for both *ring* and *IS*.

5.3 Real attacks

Section 5.2 described a simple heuristic to insert artificial anomalies in the function call trace of an MPI program executed under normal conditions. Although the datasets created with these methods were useful to demonstrate the accuracy and efficiency of the Hidden Markov Model and sequence matching algorithms, we must test MPIguard using function call traces that represent possible anomalies of a parallel program written with C/MPI on Linux.

5.3.1 Description

Torres [40] has implemented a set of C routines that simulate anomalies in MPI programs. These routines were adapted to perform experiments with MPIguard and they can be divided into two main groups:

1. Anomalies are generated by modifiying the source code of an MPI program (*ring*). In the following chapters we will refer these anomalies as *daemon attacks*.
   - *RingMalloc*: Before the parallel execution of the program finalizes (but after the MPI_Finalize call) a background process is created (a daemon). This program attempts to allocate memory *N* times. We have used *N*=100 for all the example attacks.
   - *RingFile*: Before the parallel execution of the program finalizes (but after the MPI_Finalize call) a background process is created (a daemon). This program

attempts to allocate memory and read a file *N* times. If the file can be read, the process appends artificial data to it.

- *RingFork*: This program is similar to *RingMalloc*, but every time the process attempts to allocate memory, a new daemon is created. Since every new daemon is an exact copy of its parent the number of times this process is repeated, *M*, has to be very small. Otherwise, the attack will completely blocks the MPI program until the operating system is able to free resources. We have used *M* = 3.

Figure 5.12 shows the distribution of anomalies generated for the daemon attacks for *ring*. The x-axis shows the position in the trace and the y-axis displays the identifier of the function call. The daemon attacks creates anomalies at the beginning and at the end of the execution of the MPI application.

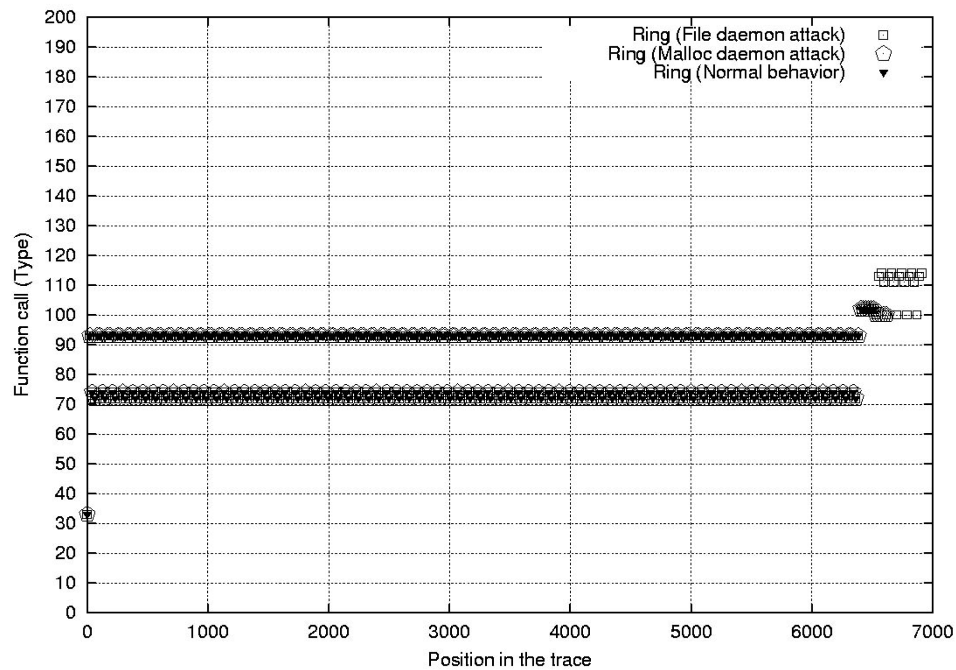

Figure 5.12 Daemon attack for *ring*

2. Anomalies are generated by including a new library between *libc/libmpi* and the stack of dynamic libraries of the run-time system (i.e. using the same concept of

interposition library implemented in MPIguard). In the following chapters we will refer these anomalies as *interposer attacks*.

- *File attack*: Every time the MPI program closes a file *F* with the *fclose* call, *F* is copied to a temporary file located in a directory with read permissions for any user.
- *MPI attack*: Every time the program executes the MPI calls MPI_Initialize or MPI_Finalize a daemon process is created with the same features as the *RingMalloc* attack.

Since the original version of *ring* does not include file management, we have added a small function to write data into files (note that this function does not affect the overall behavior of *ring*). Figure 5.13 shows this new function,

```
void MYprintf(char* s, int d)
{
        FILE *f;
        char name[30];
        sprintf(name,"lixo%d",getpid());
        if (!(f=fopen(name,"r+")))
        {
                f=fopen(name,"w");
                fprintf(f,"%s\n","lixo begins..");
        }
        else
                fseek(f,0,SEEK_END);

        fprintf(f,s,d);
        fclose(f);
}
```

Figure 5.13 New function included in *ring* to handle files

Figure 5.14 shows the distribution of anomalies for the interposer attacks with the *ring-modified* program.

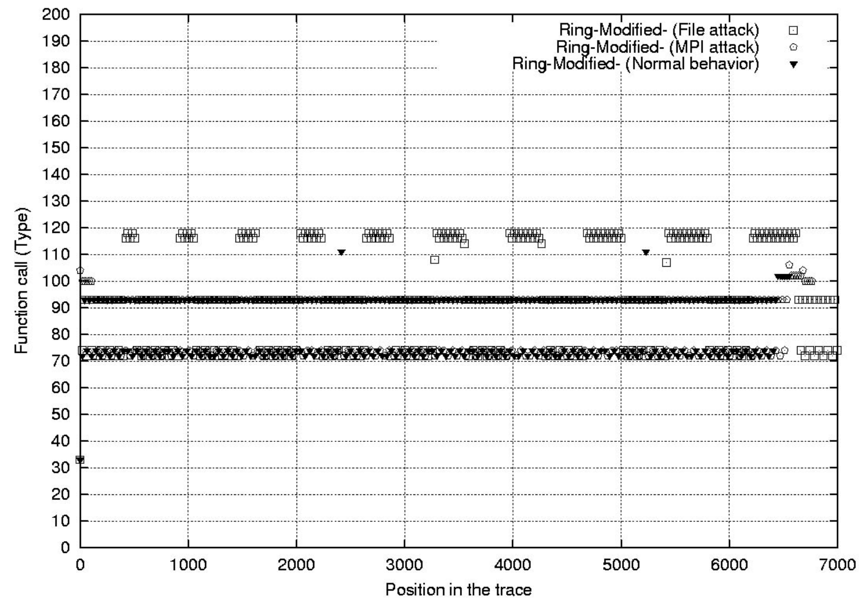

Figure 5.14 Interposer attacks for *ring -modified version-*

As explained before, MPIguard's *Profiler* can be configured to create a log of any *libc* or *libmpipro* function. In order to detect the daemon and interposer attacks, we need to include standard C functions from the *malloc* family (memory allocation), *fork* family (process bifurcation),and string and file management among others. It is important to observe that MPIguard's *profiler* only generates traces from the same level of execution in the program. As an example, a *malloc* call done by the MPI_Send function is not profiled since MPI_Send is being profiled. However, an explicit *malloc* in the source code will be profiled.

With *ring* the daemon attacks generate 14 different function calls and the interposer attacks generate 17 function calls. Table 5.1 shows the average of the number of anomalies generated on a slave node by the daemon and interposer attacks for 3 executions of *ring* and *ring-modified*.

Table 5.1 Average number of anomalies with real attacks

Attack	*Anomalies*
Daemon-RingMalloc	109
Daemon-RingFile	409
Daemon-RingFork	30
Interposer-File	5342
Interposer-MPI	217
Interposer-Both	5559

5.3.2 Hidden Markov Models

Figure 5.15 shows the accuracy of an HMM trained with normal behavior of *ring* on process 2 using 8 states when *ring* was executed with the daemon attacks. The HMM is able to distinguish between normal and abnormal behavior when the user threshold is small ($\theta < 0.09$). As an example, if $\theta = 0.1$ the number of anomalies detected for normal behavior with the 8-state HMM is 111 (all are false positives), 520 anomalies with the *RingFile* attack (111 false positives), 141 anomalies with the *RingFork* attack (111 false positives) and 220 anomalies with the *RingMalloc* (111 false positives) [1]. Thus, the HMM

[1] See Table 5.1

is detecting all the anomalies produced by the daemon attacks with 0% false negative rate but with 111 false positives. It is important to observe the impact of the user threshold in the detection: With $\theta > 0.4$ almost all the functions in the traces are flagged as anomalous for the HMMs with 8,14, and 20 states.

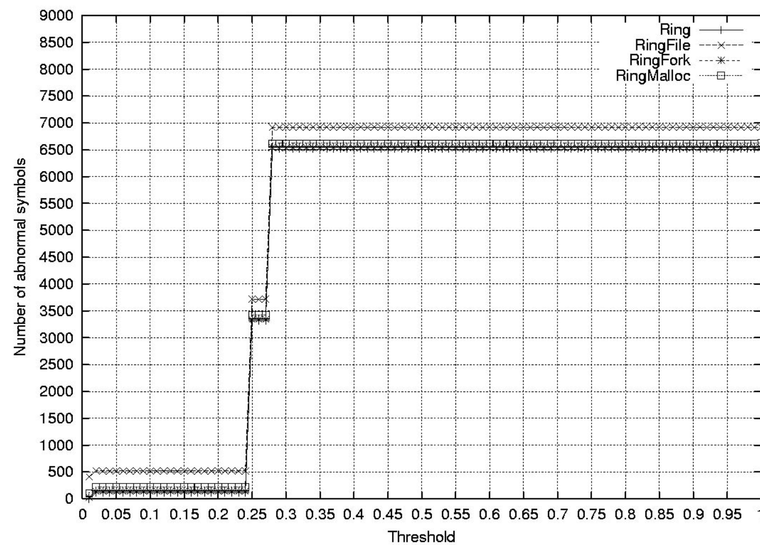

Figure 5.15 Detecting daemons attacks in *ring* with a 8-state HMM

Table 5.2 shows the best training time found using as input for the Baulm-Welch algorithm the normal behavior in process 2.

Figure 5.16 shows the detection of anomalies with a 14-state HMM and Figure 5.17 shows the detection with a 20-state HMM.

Figure 5.18 shows the accuracy of a 8-state HMM for *ring-modified* when the program is being executed with interposer attacks. The HMM distinguishes between normal and

Table 5.2 Training time for the HMM with *ring*

Number of states	*Duration(seconds)*
8	982
14	176
20	46

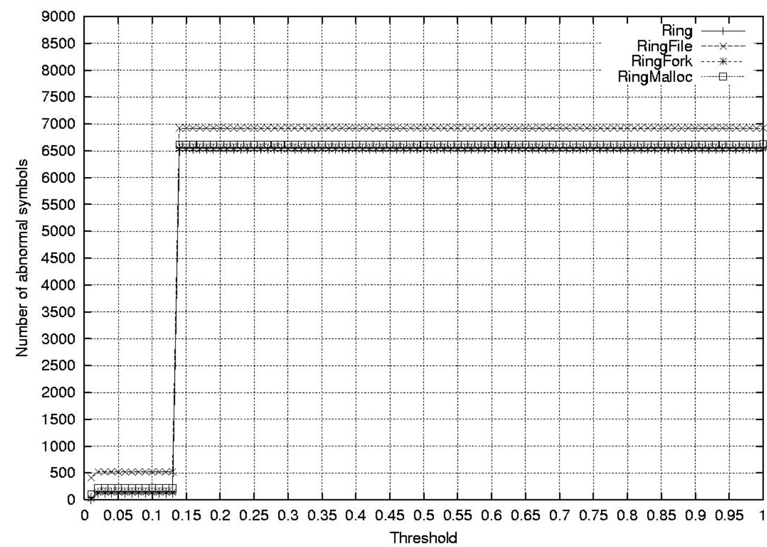

Figure 5.16 Detecting daemons attacks in *ring* with a 14-state HMM

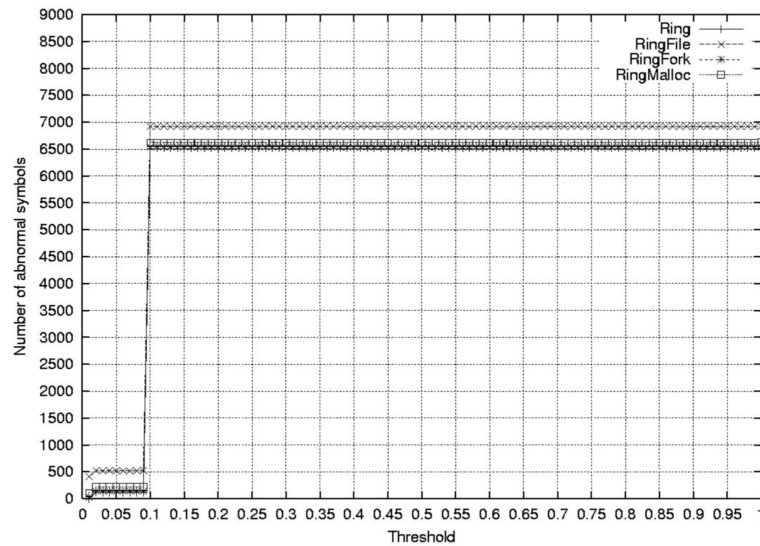

Figure 5.17 Detecting daemons attacks in *ring* with a 20-state HMM

abnormal behavior but the number of false positives is overwhelming. As an example, using a small threshold $\theta < 0.2$, the IDS detects 162 anomalies with normal behavior (all are false positives), 9201 anomalies with the *File attack* (3859 false positives) and 6550 anomalies with the *MPI attack* (6342 false positives). However, the HMMs can be used for detection since the difference between the number of anomalies of normal and abnormal behavior is fairly high (in the above example this difference exceeds the 3500 anomalies).

Table 5.3 shows the best training time found using as input for the Baulm-Welch algorithm the normal behavior in process 2.

The accuracy of a 17-state HMM is shown in Figure 5.19 and the accuracy of a 20-state HMM is presented in Figure 5.20.

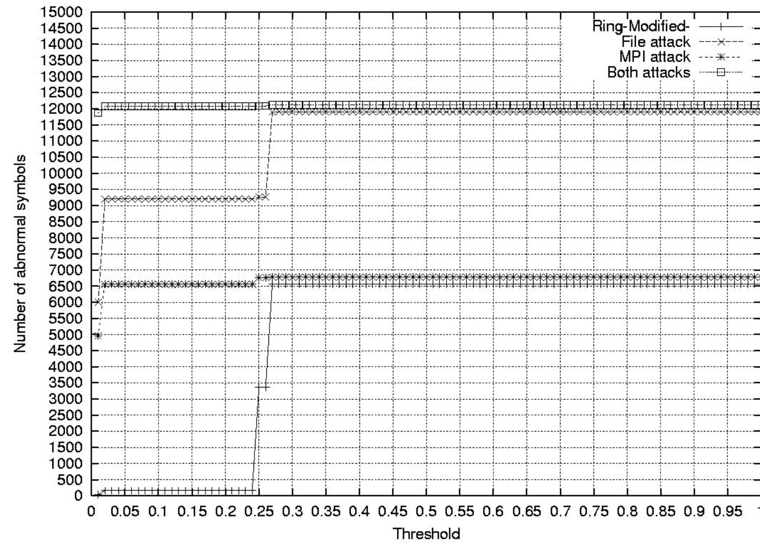

Figure 5.18 Interposer attack in *ring-modified* with a 8-state HMM

Table 5.3 Training time for the HMM with *ring-modified*

Number of states	*Duration(seconds)*
8	169
17	89
20	68

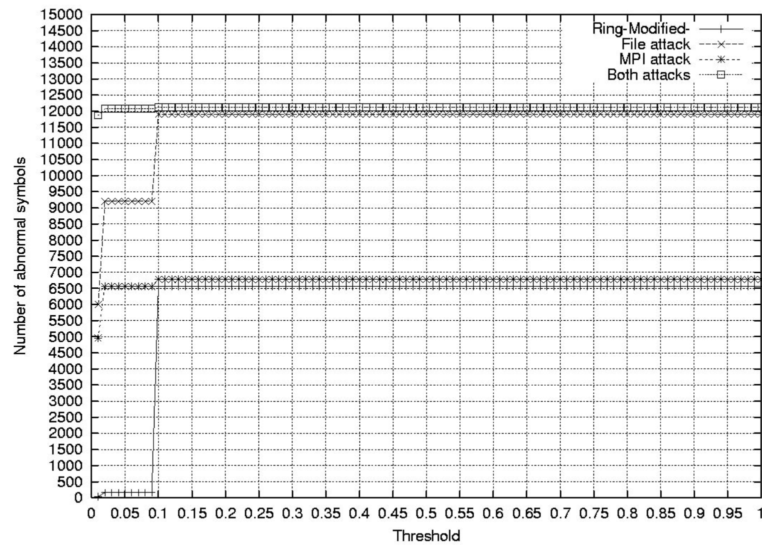

Figure 5.19 Interposer attack in *ring-modified* with a 17-state HMM

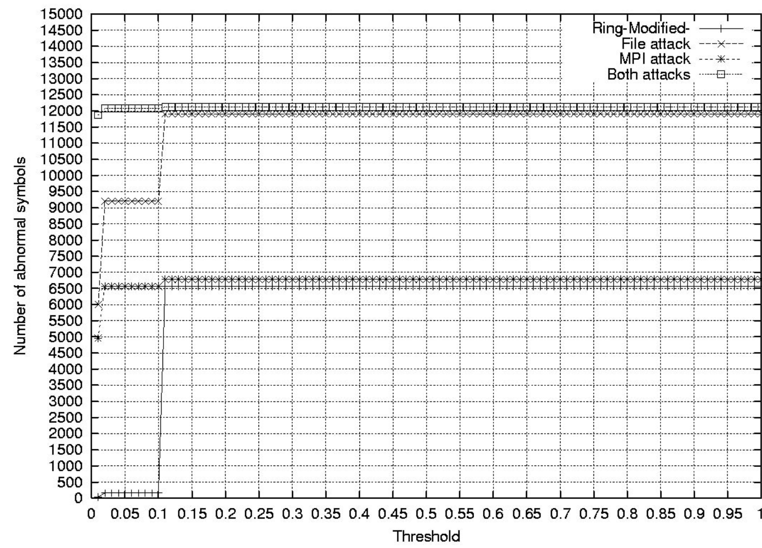

Figure 5.20 Interposer attack in *ring-modified* with a 20-state HMM

Finally, to test the generalization capability of the IDS, we computed the accuracy of the HMM on processor 2 when the model was created with the normal behavior on processor 2 (exploiting the SPMD property of *ring*). Figure 5.21 shows the detection with a 8-state HMM, Figure 5.22 presents the detection of a 17-state HMM and Figure 5.23 shows the accuracy of a 20-state HMM. The overall detection rate of the HMMs is fairly similar to the HMMs trained with the normal behavior of process 2.

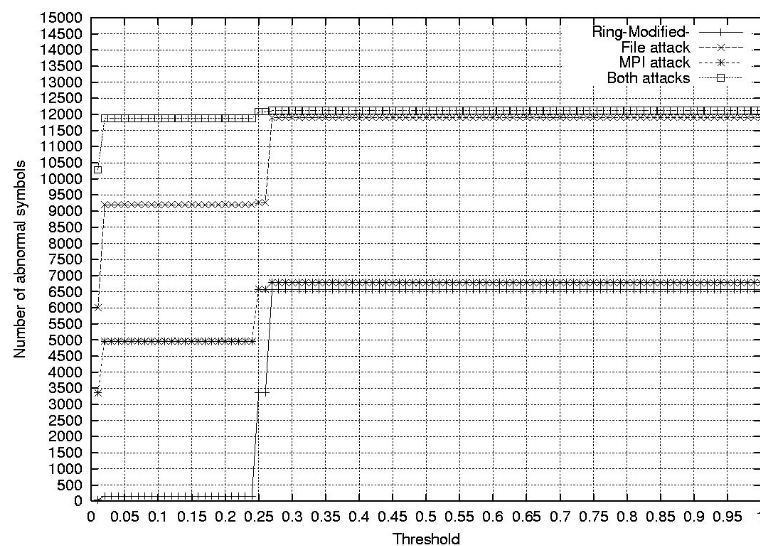

Figure 5.21 Detecting interposer attacks on processor 4 with a 8-state HMM

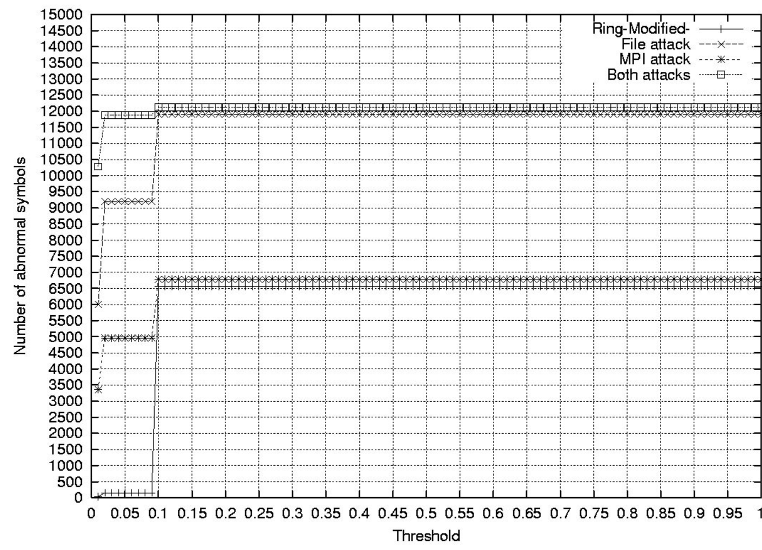

Figure 5.22 Detecting interposer attacks on processor 4 with a 17-state HMM

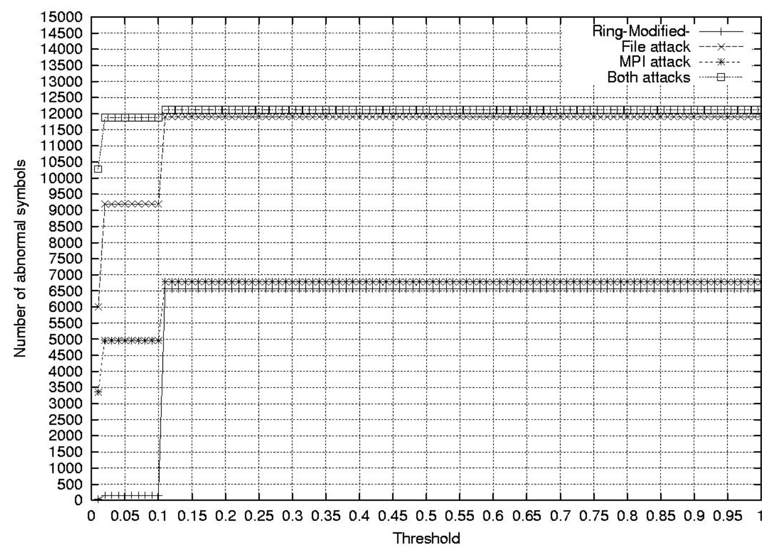

Figure 5.23 Detecting interposer attacks on processor 4 with a 20-state HMM

5.3.3 *Sequence matching*

Figure 5.24 shows the accuracy of the sequence matching algorithm using four different window sizes ($\omega = 6, 7, 8, 9$) for *ring* with the daemon attacks. The trace from process 2 was used as the *profile* of the application. Using sequences of length 9, the IDS detects 109 anomalous sequences for the *RingMalloc attack* (containing 109 anomalous function calls), 409 anomalous sequences for the *RingFile attack* (containing 409 anomalous function calls) and 30 anomalous sequences for the *RingFork attack* (containing 30 anomalies). Thus, the IDS detects 100% of the daemon attacks with 0 false negatives. Figure 5.25 shows the detection accuracy of the sequence matching algorithm of *ring-modified* on processor 4.

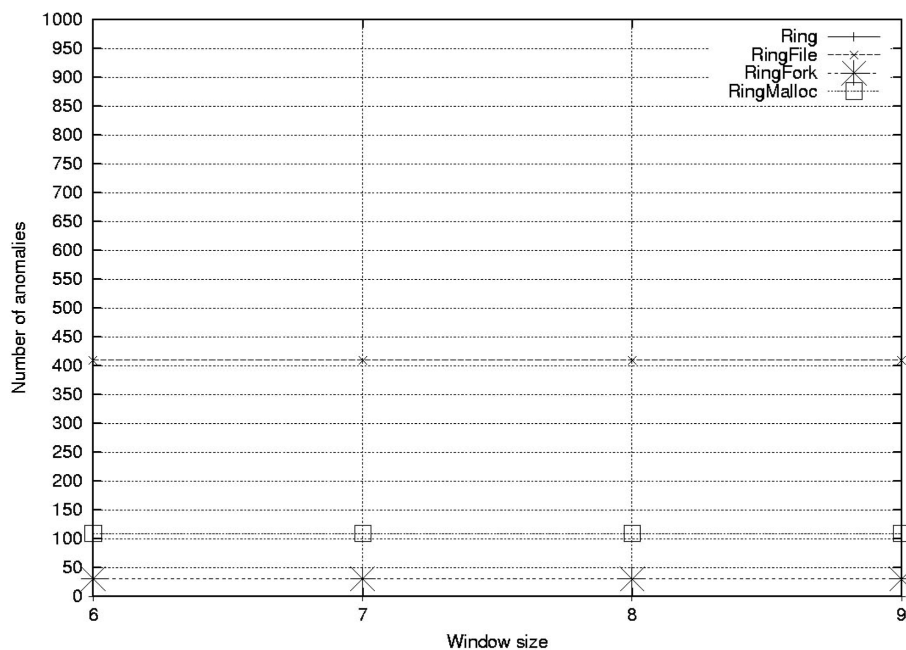

Figure 5.24 Detecting daemon attacks in *ring* using Sequence Matching

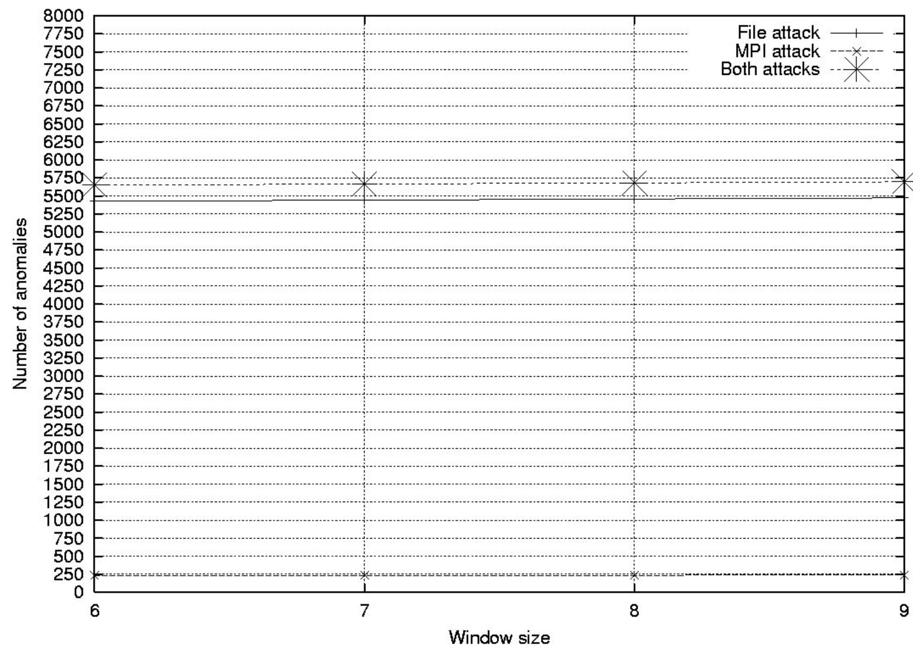

Figure 5.25 Detecting interposer attacks in *ring-modified* using Sequence Matching

The detection accuracy of the sequence matching algorithm for interposer attacks is again fairly high. As an example, using sequences of length 9 the IDS on processor 2 found 5466 anomalous sequences for the *File attack* (containing 5342 anomalous function calls [2]), and 230 anomalous sequences for the *MPI attack* (containing 217 anomalous function calls). Similar results were obtained with the others processors. However, the IDS detects up to 6 false positives in each processor when *ring-modified* is executed under normal conditions.

---

[2]The number of anomalous sequences detected by this algorithm might exceed the real number of anomalous function calls. See section 4.2.2 for details.

CHAPTER VI

ONLINE DETECTION

6.1 Simulating online detection

The results obtained in Section 5.3 lead use to believe that both the HMM and the sequence matching algorithm can be used to implement MPIguard's *Analyzer*. We conducted new experiments to determine the accuracy of the detection algorithms in real-time and the results were plotted where the *x-axis* corresponds to the position of the function call in the trace (it can be seen as a time dimension) and the *y-axis* corresponds to the number of anomalies identified by the IDS.

6.1.1 Sequence matching

Figure 6.1 compares the normal behavior of processor 3 with the *profile* created in processor 2 using the sequence matching algorithm with $\omega = 9$. It shows that the IDS only detects a chunk of about 10 anomalies at the end of the trace (false positives).

The *File attack* creates anomalies when a file *F* is closed. In *ring-modified* this occurs several times during the execution of the program and approximately at the same interval of time. Figure 6.2 shows that the sequence matching algorithm is able to detect these anomalies.

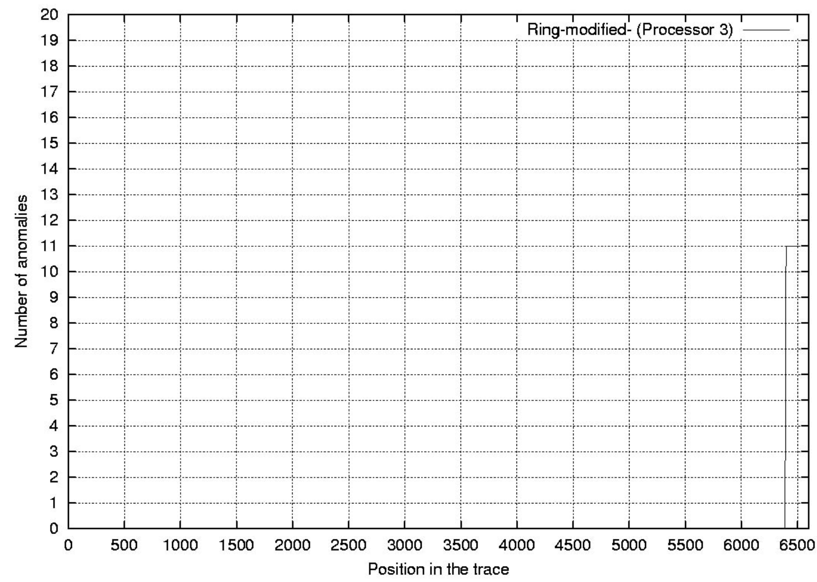

Figure 6.1 Comparing normal behavior of *ring-modified-* using Sequence Matching

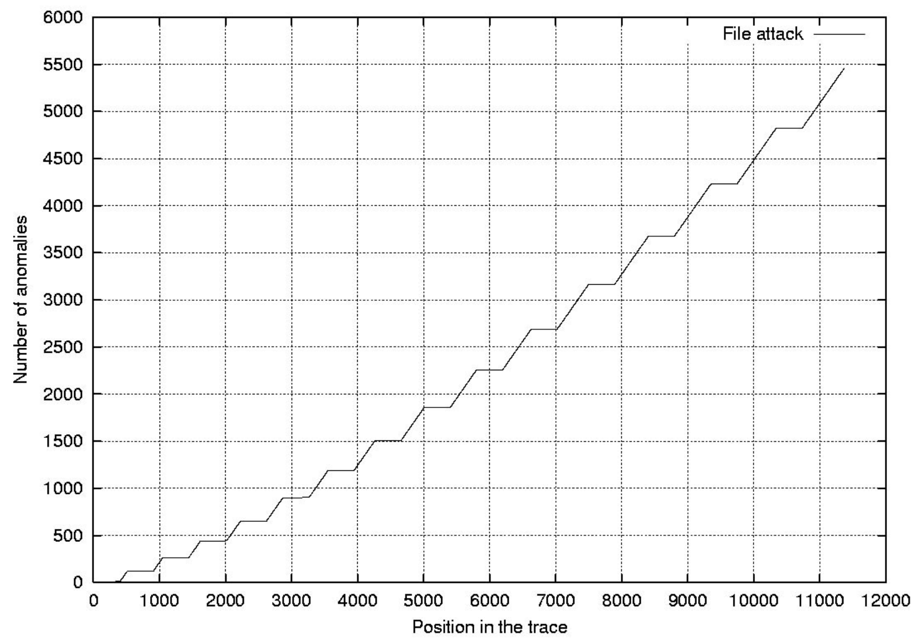

Figure 6.2 Detecting *File attack* in *ring-modified-*with Sequence Matching

The *MPI attack* creates anomalies when the functions *MPI_Init* and *MPI_Finalize* are executed in the MPI program. In Figure 6.3 we can see that at the beginning of the trace the IDS detects up to 120 anomalies. Afterwards, no anomalies are detected until the MPI program is finishing (at the end of the trace), where the IDS detects another 110 anomalies.

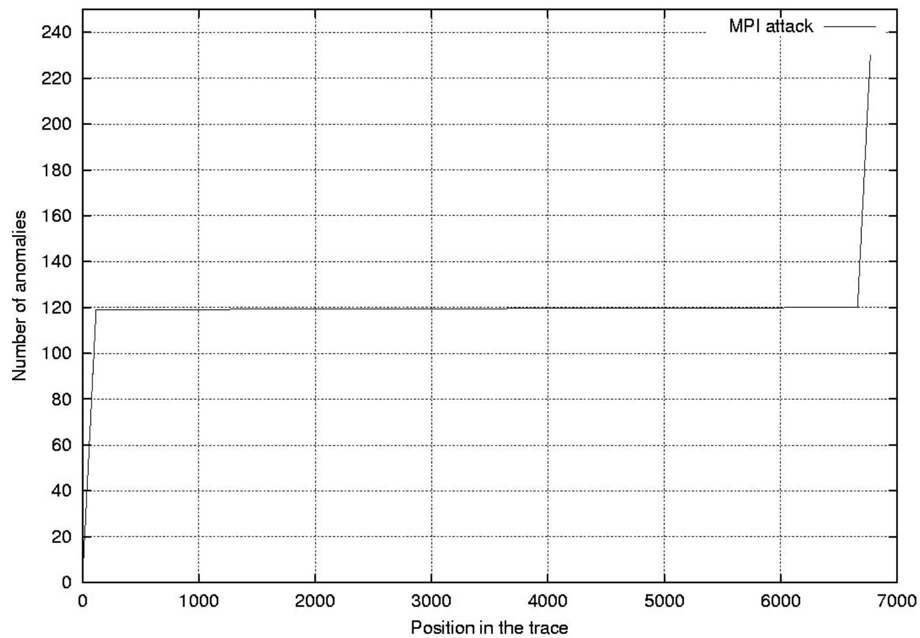

Figure 6.3 Detecting *MPI attack* in *ring-modified*-Sequence Matching

We concluded that the sequence matching algorithm is suitable for online detection resulting in few false positives or negatives.

6.1.2 Hidden Markov Model

Figure 6.4 compares the normal behavior of processor 3 with the profile created in processor 2 using an HMM with 17 states and $\theta = 0.05$. The HMM detects 2 anomalies every 500 function calls approximately, resulting in a classification rate of 99.6%. However, the total number of false positives with the *ring* trace (containing 6500 function calls) is 162.

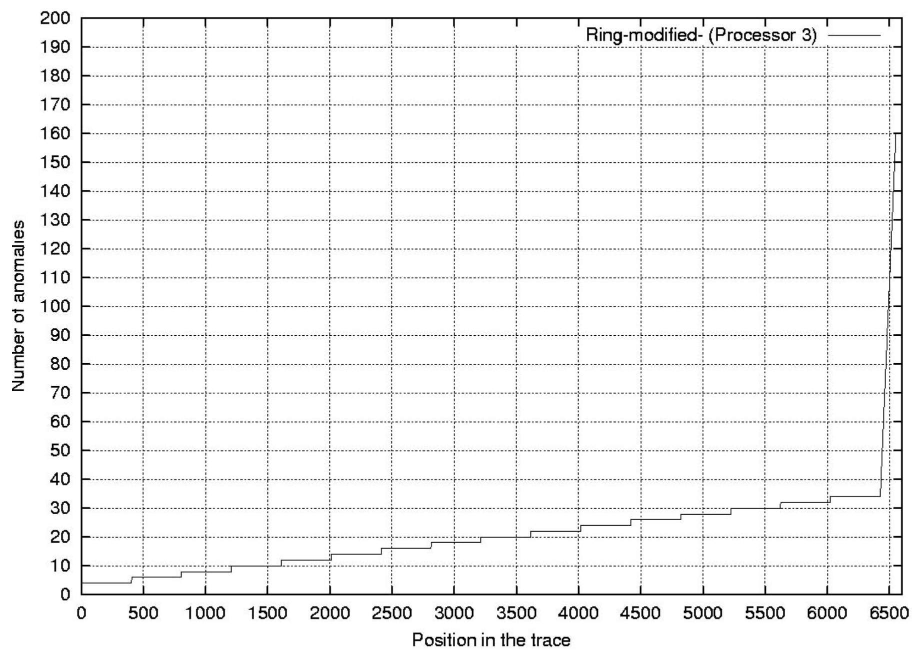

Figure 6.4 Monitoring *ring-modified-* on processor 3 using a 17-state HMM

Figure 6.5 shows the detection rate of the IDS with *File attack* and Figure 6.6 with the *MPI attack.* Almost every function call is being tagged as anomalous, resulting in a very high false positive rate.

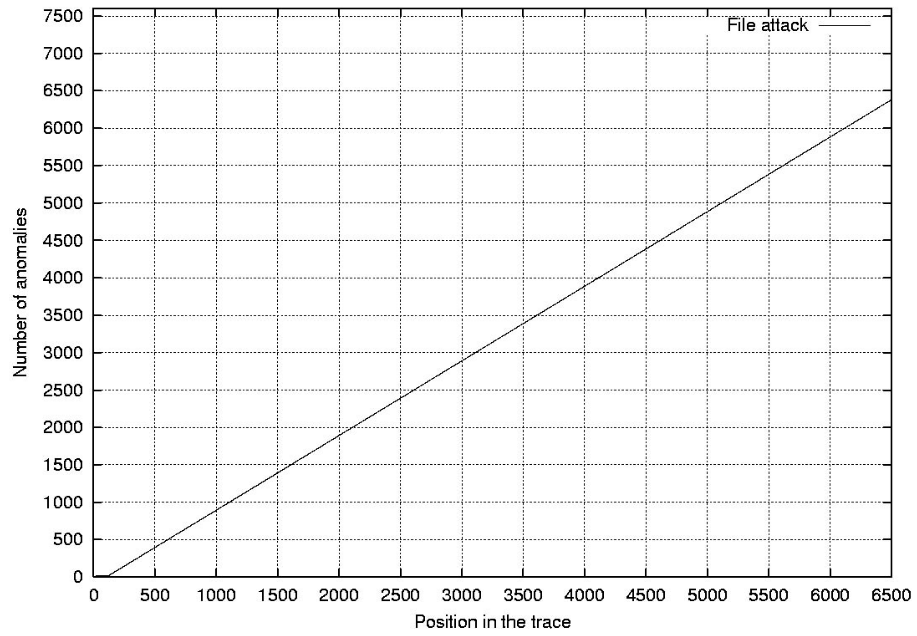

Figure 6.5 Detecting *File attack* in *ring-modified-* using a 17-state HMM

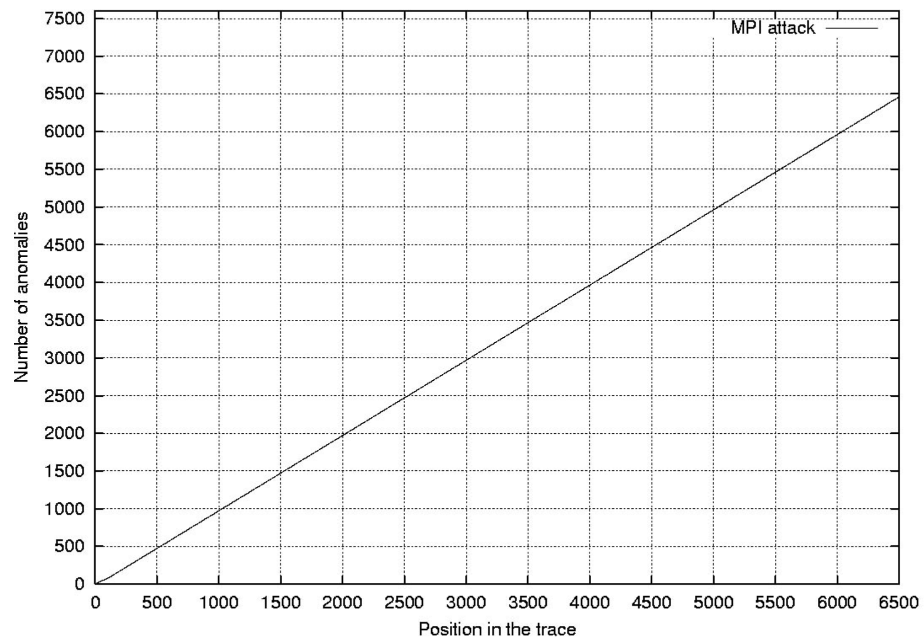

Figure 6.6 Detecting *MPI attack* in *ring-modified-* using a 17-state HMM

These experiments demonstrated that in order to achieve online detection with low false positive rate we need to improve either the HMM model or the detection algorithm. The original experiments used the following stop conditions for the Baulm-Welch algorithm:

- When the *log* of the probability of generating the sequence O with the model λ is less than 0.001.
- When the number of iterations reaches 1000.

Attempting to build more accurate models, we changed those stop conditions as follows:

- When the *log* of the probability of generating the sequence O with the model λ is less than 0.0000001 .
- When the number of iterations reaches 2000.

However, we were not able to find any λ with the Baum-Welch algorithm that satisfies those conditions. Since we could not improve the stochastic model, we tried to modify the anomaly detection algorithm itself by using a *local frame count* (LFC) [33]. The LFC is given by the expression $LFC(A) = \sum A_i$, where A is the *locality frame*. Let O_t be the current function call and n the size of the *locality frame*. A is given by $A_{t \ mod \ n} = 1$ if O_t is anomalous, and 0 otherwise. In our experiments, the total number of anomalies detected by the HMM is incremented if and only if $LFC(A) = n$, i.e. if the last n function calls were flagged as anomalous.

So the new detection algorithm can be expressed as follows: Given a new observation $\overline{O}$ (that corresponds to the trace of an unseen instance of the MPI/C program, named *UT*):

1. Using the model λ
2. Initialize A
3. For each one of the observations $\overline{O_t}$.
    - If $LFC(A) = n$ then the counter of anomalies *C* is increased.
    - For each one of the states S_i (if the state can be reached from the previous one, i.e. if the probability of moving to the current state is greater than some user threshold θ) .
        - If the probability of producing the symbol $\overline{O_t}$ in the current state $B(i, \overline{O_t})$ is less than θ then the function call in the trace *UT* is labeled as anomalous
    - If $\overline{O_t}$ could not be produced by any state (i.e. the function call in the trace was tagged as anomalous in each state S_i) then compute $A_{t\ mod\ n} = 1$ else compute $A_{t\ mod\ n} = 0$
4. If *C* exceeds some user threshold ρ then the whole trace *UT* is tagged as anomalous.

With a 17-state HMM using *LFC=9* we are able to reduce the number of false positives when comparing normal behavior on processor 3 with the *profile*. This can be seen in Figure 6.7. The HMM detects 40 anomalies at the end of the trace.

Figure 6.8 shows the online detection accuracy of the HMM when the *ring-modified* program is executed with the *File attack*. Finally, even with the improvements described above the number of false positives of the HMM for the *MPI attack* is overwhelming (see Figure 6.9).

Analyzing the detection algorithm and the model λ we concluded that the large amount of false positives with this attack is related with the first set of function calls of the trace: The *MPI attack* tries to allocate memory 100 times when the function *MPI_Init* is called, so at least the first 100 function calls can be correctly tagged as anomalous.

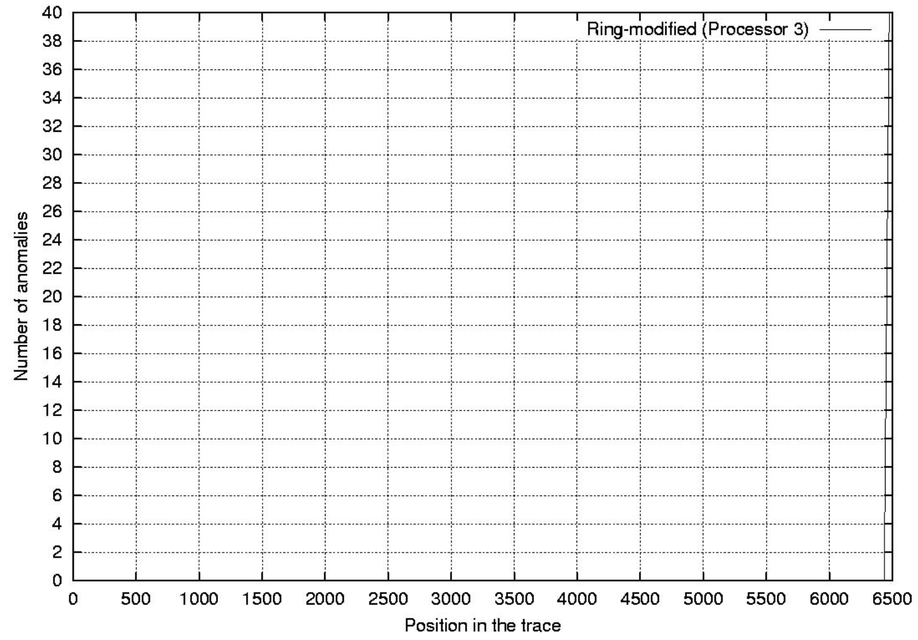

Figure 6.7 Monitoring *ring-modified-* on processor 3 using an HMM, *LFC=9*

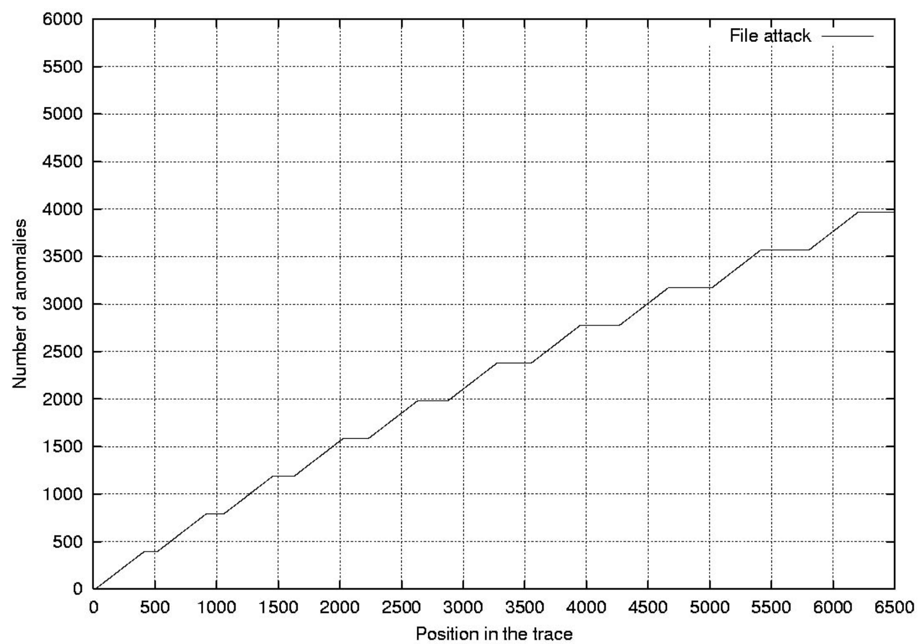

Figure 6.8 Detecting *File attack* in *ring-modified-* using a 17-state HMM, *LFC=9*

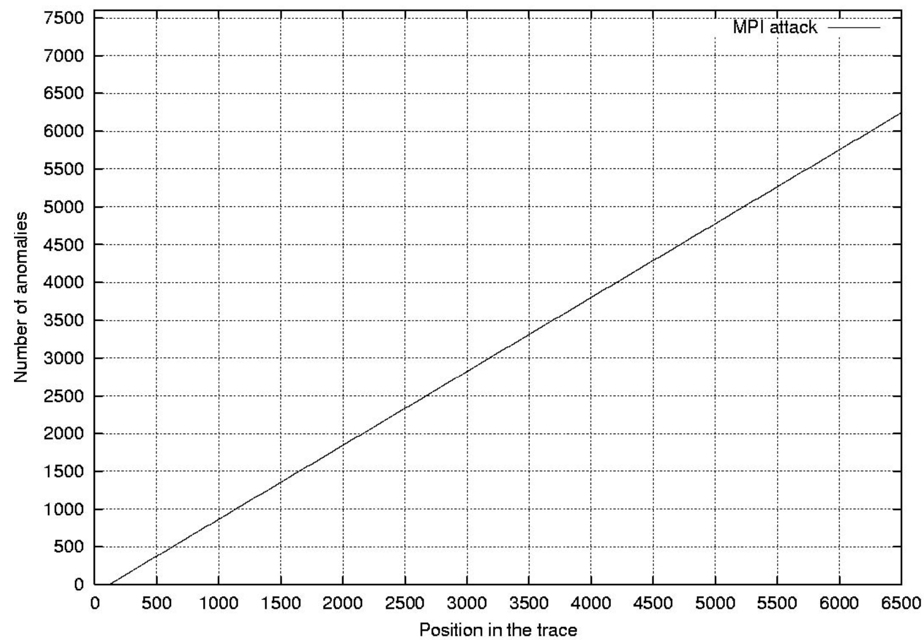

Figure 6.9 Detecting *MPI attack* in *ring-modified-* using a 17-state HMM, *LFC=9*

Since there are very few distinct sequences of length 9 produced by the attack to *MPI_Init* and the detection algorithm keeps track of the possible states that can be reached from the current one, the set of possible next states is dramatically reduced and does not change over time. When the detection algorithm tries to classify normal sequences (i.e. after the initial attack) the small subset of states of λ (the probabilities of transition and probabilities of producing outputs) that the algorithm is visiting does not contain enough information to generalize the normal behavior of the program. That is the reason why the IDS keeps labeling function calls as anomalous even when normal functions are being presented to the algorithm.

We have demonstrated that the HMM (with *locality frame count*) is able to distinguish between normal and abnormal behavior, but it produces too many false positives when the program's trace contains a quite large set of consecutive anomalies.

6.2 The MPI/C program's *profile*

By definition the *profile* must represent the overall behavior of a program under normal conditions and does not change over time. However, it is important to observe that when analyzing normal behavior of UNIX processes, researchers often find the problem that is very hard to define a unique flow of events since even with non-complex programs the interaction of one process with the operating system (file and network system for example) or with other user processes can cause a wide variety of interruptions, system calls and function calls. Hence it is possible for two identical programs to create different traces. SPMD applications introduce a new problem, because even with the same set of instructions on different nodes the input for the algorithms can be different.

As an example, take *ring* with 4 processors. Using MPIguard's *profiler* we obtained four different traces, three of them can be used as a *profile* (the first node acts like a *master node*, so its behavior differs from the other 3 *slave* nodes).

The following experiments use *profile_p2*, the program trace of processor 2 containing 6748 calls with 10 different functions and *profile_p3*, the program trace of processor 3 containing 6745 calls and 10 different functions. Figure 6.10 shows the detection of anomalies of a new execution of *Ring* with 4 processors. MPIguard's *Analyzer* is active in

the four nodes. As we expected, the IDS detects several anomalies when the *Analyzer* is executed in the *master node* and detects few anomalies for the others processors.

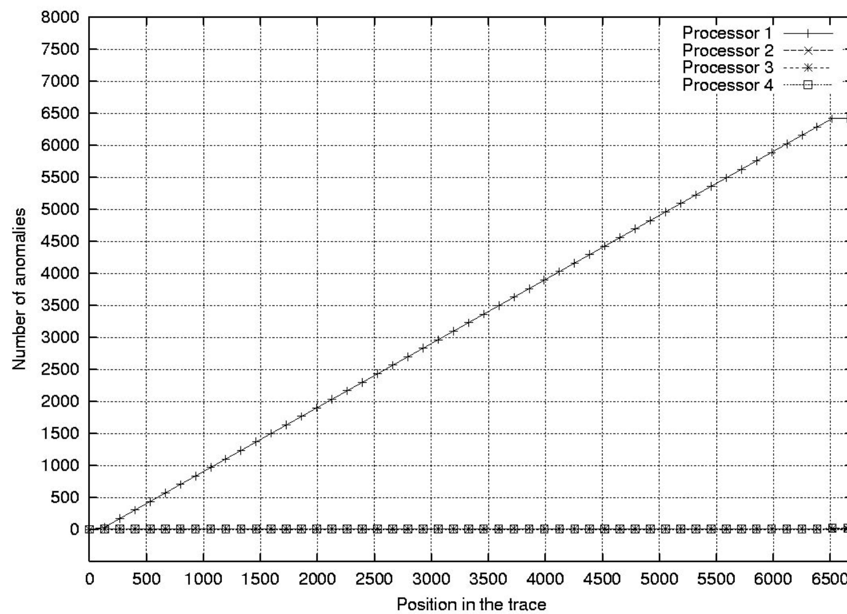

Figure 6.10 Comparing the execution of *ring* with *profile_p2*

Figure 6.11 repeats this experiment without using the *Analyzer* on processor 1, so we can see clearly the detection capability of the IDS when *ring* is executed under normal conditions in the *slave* nodes. The sequences of function calls that the IDS is labeling as anomalous correspond to the start-up and the end of the MPI communication.

Figure 6.12 depicts the behavior of the *Analyzer* using *profile_p3* and the result is very similar to the detection accuracy using *profile_p2*.

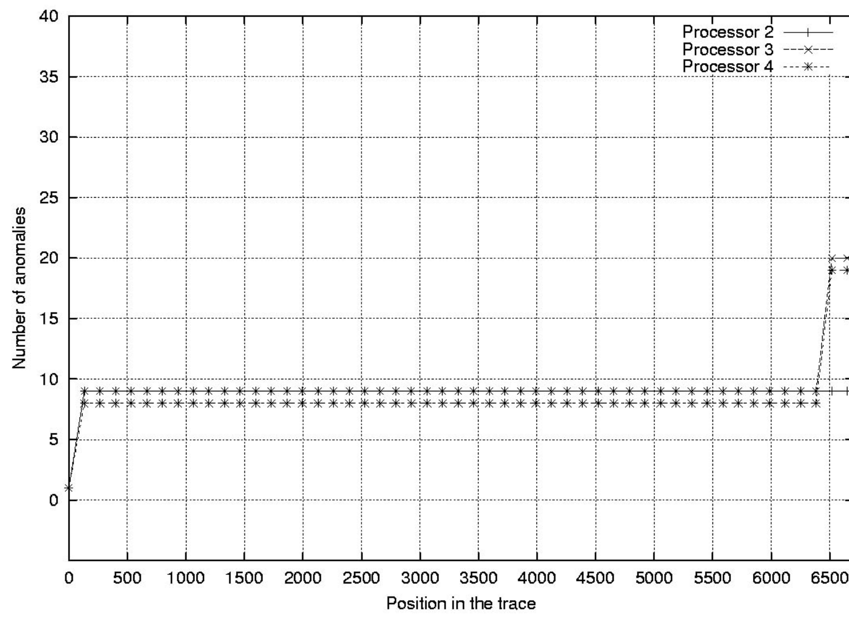

Figure 6.11 Comparing the execution of *slaves* nodes of *ring* with *profile_p2*

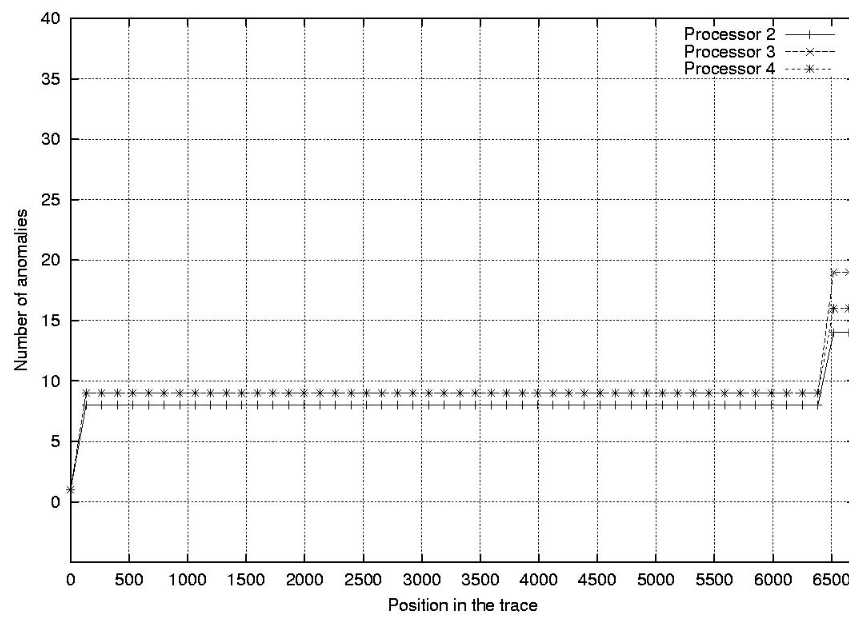

Figure 6.12 Comparing the execution of *slaves* nodes of *ring* with *profile_p3*

We conclude that the trace of any *slave* processor can be used as a *profiler* for the IDS: The only anomalies that we might expect correspond to the beginning and finish of the MPI communication tasks and specific functionality associated with a nodes as defined in the source code of the MPI/C program.

6.3 MPIguard's *Analyzer*

Figure 3.11 depicts the architecture that we have proposed to analyze the behavior of the parallel applications in each one of the nodes of the cluster: The *Analyzer* receives function calls via shared memory from the interposition library and executes the detection algorithm. We choose the sequence matching algorithm to instrument the *Analyzer* because we have demonstrated that it can be used to detect anomalies in real-time of MPI/C programs with low false or negative rates. The set of parameters that the *Analyzer* needs are:

1. Profile of the application (an ASCII file containing the function call trace that represents normal behavior of the application).
2. Window size.
3. Output file.

6.4 Monitoring complex applications

We have demonstrated that MPIguard is able to detect anomalies in real-time for MPI/C programs running in a Linux cluster. However, we collected data from small parallel applications such as *IS*, *ring* and *ring-modified*, and we produced traces with no more

than 7000 function calls using the default configuration file of MPIguard. In practice, commercial and scientific parallel applications use algorithms with complex communication patterns that generally need several hours or even days to be completed.

6.4.1 Implementing LLCBench2

We wanted to build an MPI application complex enough to demonstrate the use of MPIguard for large parallel programs. We decided to modify the source code of LLCbench [27], an application that includes three point-to-point benchmark routines, *Latency*, *bandwidth* and *bidirectional bandwidth*, and 5 broadcast benchmark routines: *roundtrip*, *broadcast*, *reduce*, *allreduce*, and *all-to-all*. The point-to-point routines involve communication from the *master* node to the last node in the *rank* of communication, whereas the broadcast routines involve all the processors. Thus, to monitor the behavior of LLCbench in the cluster when it is executed with 4 processors we need 3 *profiles*: one for the *master* node, one for the last node (*rank_size*-1) and one for the other two processors.

This new application selects one MPI routine based on a normally distributed random number generator, with the mean and the standard deviation computed experimentally to execute *reduce* more often than the others broadcast routines, and to execute the point-to-point benchmark routines only a few times. This process is repeated 5000 times. Figure 6.13 presents the *main* function of LLCbench2. The mean of the normal distribution is 4 (i.e. *reduce*) and the standard deviation is 1.5. Figure 6.14 shows an example of the distribution of the subroutines executed by LLCbench2.

```
for (i=0; i < 5000; i++)
{

   r=(int) GetRandomNormal (4,1.5);

   switch (r)
     {
     case 0:
       loop(iterations,latency);
       break;
     case 1:
       loop(iterations,roundtrip);
       break;
     case 2:
       loop(iterations,bandwidth);
       break;
     case 3:
       loop(iterations, broadcast);
       break;
     case 4:
       loop(iterations, reduce);
       break;
     case 5:
       loop(iterations, allreduce);
       break;
     case 6:
       loop(iterations, alltoall);
       break;
     case 7:
       loop(iterations, bibandwidth);
       break;
     default: /*do the mean=reduce*/
       loop(iterations,reduce);
       break;
     }
}
```

Figure 6.13 The *main* function of LLCbench2

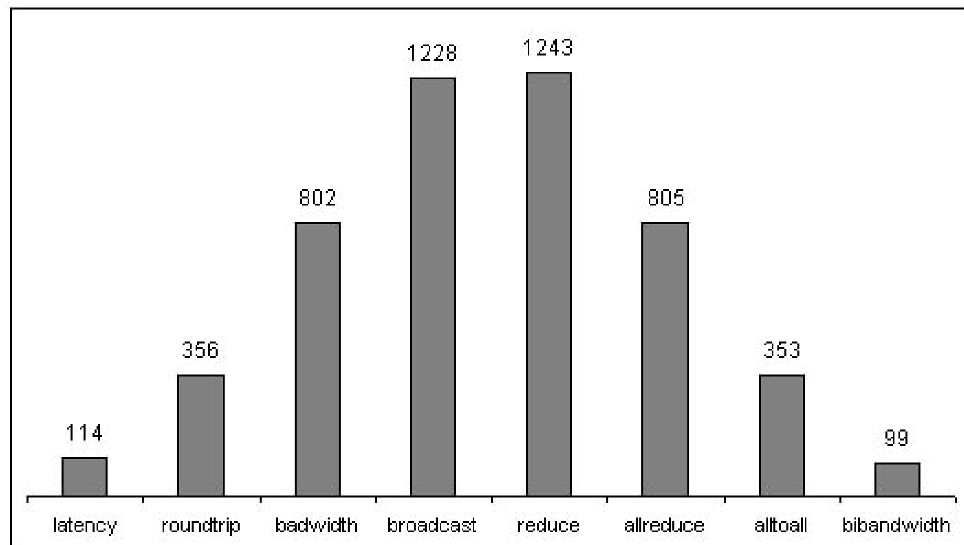

Figure 6.14 Distribution of the subroutines of LLCbench2

6.4.2 Monitoring LLCbench2

Table 6.1 shows the size of the *profile* for every processor where LLCbench2 was executed. We obtained traces with more than one million function calls, and as we expected, the traces from processor 2 and processor 3 are identical and they can be monitored using the same *profile*.

Deploying MPIGuard's *Analyzer* in the cluster and executing LLCbench2 with 4 processors produced very low false positives rates as shown in Table 6.2. It is very important to observe that the traces produced by LLCbench2 in each node change every time the program is executed. However, the *profile* can be used because the communication patterns of LLCbench2 tends to be constant as in many real world parallel applications, and because of the large sample size.

Table 6.1 Size of LLCbench2 traces

Node	*Number of function calls*
microcosm1	1515710
microcosm2	1050364
microcosm3	1050364
microcosm4	1413784

Table 6.2 Number of false positives for LLCbench2

Node	*False positives*
microcosm1	0
microcosm2	4
microcosm3	0
microcosm4	4

As another example of the capabilities of MPIguard, we executed LLCbench2 with the *MPI-attack* defined in Section 5.3. Figure 6.15 shows the number of anomalies reported by MPIguard in time space for the *master* node at the beginning of the execution and Figure 6.16 shows the detection at the end of the execution. As expected, MPIguard is able to detect the *MPI-attack*.

6.4.3 LU factorization

As a final example, we have chosen an application implemented by Dandass to improve the Gaussian elimination method for solving systems of linear equations such as $Ax = b$ using the LU factorization method [6]. The factorization is distributed among a 2 dimensional grid of processors and "...in order to improve efficiency, the distribution

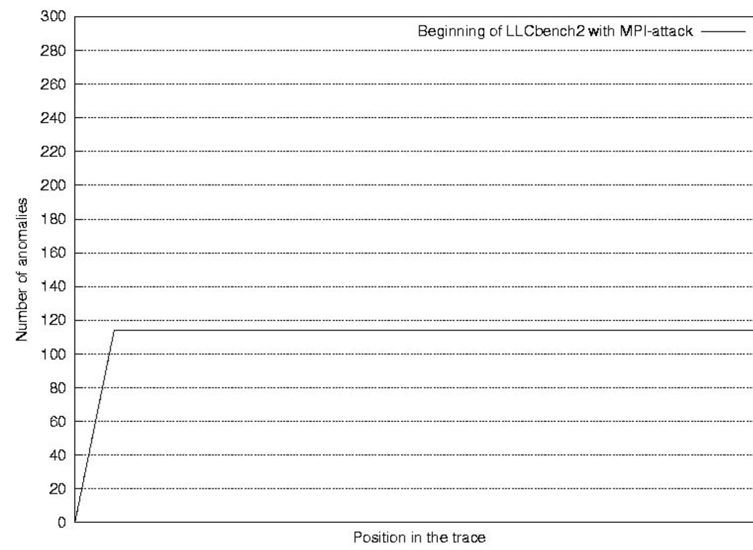

Figure 6.15 Detection *MPI-attack* at the beginning of LLCbench2

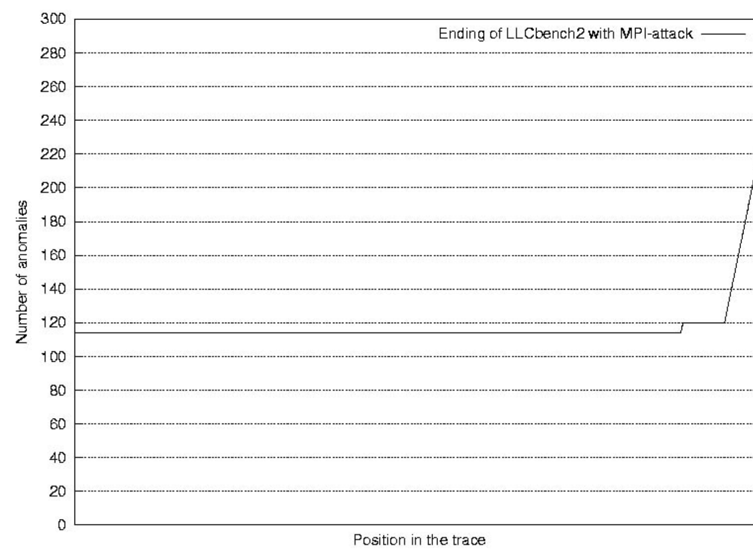

Figure 6.16 Detection *MPI-attack* at the ending of LLCbench2

scheme must be designed to minimize load imbalance by distributing the“active” elements as evenly as possible across all processors.” [6, p.4]

Figure 6.17 shows the numbers of anomalies reported by MPIguard when the LU factorization algorithm for a 1000x1000 matrix using a block size of 10 elements is executed under normal conditions with 4 processors. Once again, the anomalies detected by the *Analyzer* correspond to start-up and end of the MPI communication. The maximum number of false positives produced by MPIguard in a processor was 121. In order to improve the detection rate, we used MPIguard’s *Profiler* to generate the traces of 10 executions of the LU factorization. By doing this, MPIguard is able to learn more function call sequences that represent the normal behavior of the MPI application. The results are shown in Figure 6.18. The maximum number of anomalies generated by MPIguard is reduced to 54. This experiment demonstrates one of the most powerful features of MPIguard: the ability to learn normal behavior from several executions of an MPI application. Finally, Figure 6.19 shows the detection of the *MPI-attack* for the LU factorization application on processor 1.

The traces produced by this algorithm are smaller than the traces produced by LLCbench2, but the communication patterns of LU-factorization are much more complex, and it is a good example of the type of scientific programs that are executed on a cluster of workstations. It is important to observe that although this program was implemented in C++, MPIguard is able to collect and analyze the function call traces of *libc* and *libmpipro* without modification of its source code or configuration files.

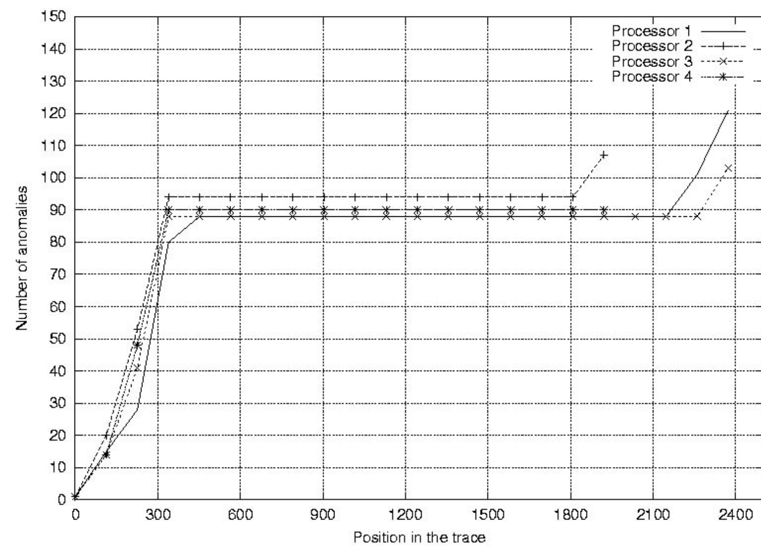

Figure 6.17 Monitoring LU-factorization on 4 processors

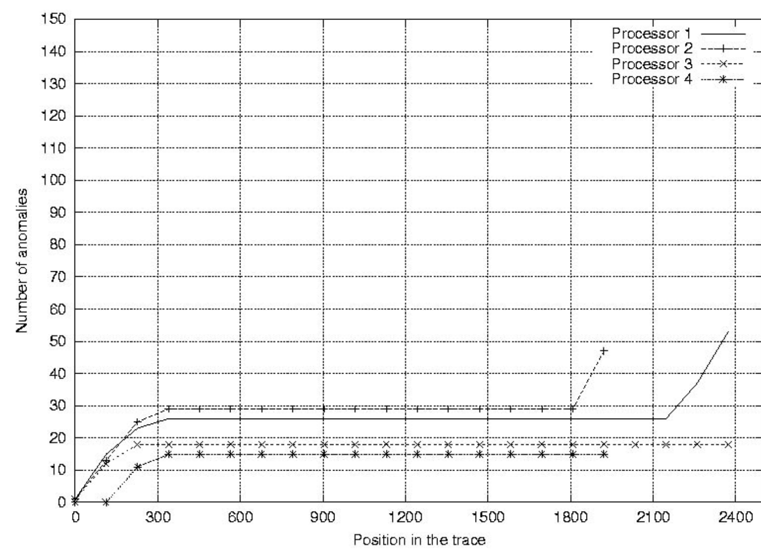

Figure 6.18 Monitoring LU-factorization using a *profile* with 10 executions

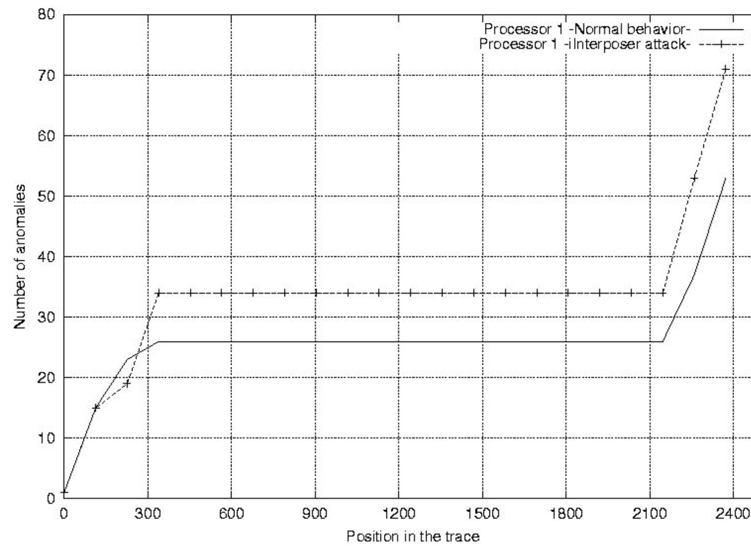

Figure 6.19 Detecting *interposer* attacks in LU-factorization

6.5 Performance

Figure 6.20 shows the latency of MPI_Send using LLCbench [27] with 4 processors and 20 iterations and Figure 6.21 shows the average bandwidth between the master and one slave node using unidirectional transmissions.

Measuring the performance of an application is a difficult task and it has been an interesting field of research. We attempted to measured the performance of MPIguard for two case scenarios: when the parallel application performs extensive computations with few messages and when the parallel application generates several messages among processors with little local computation. When the application executes local computation without the use of a library, MPIguard is idle. The application selected for the first test

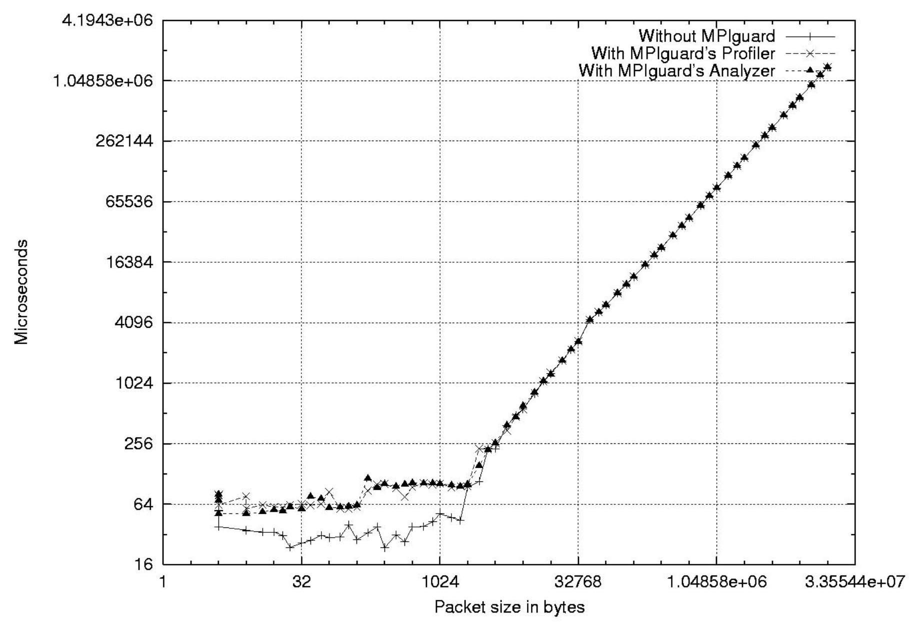

Figure 6.20 Comparison of MPI_Send latency with LLCbench using MPIguard

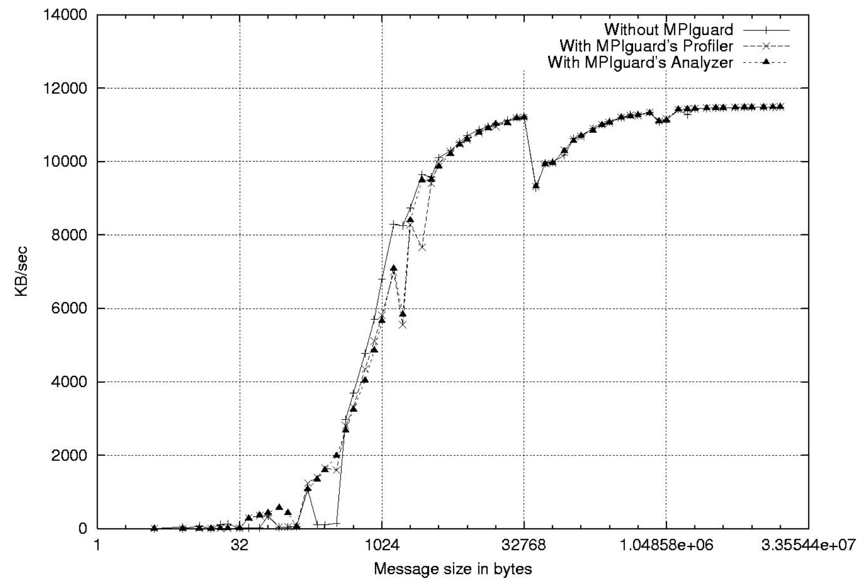

Figure 6.21 Comparison of the bandwidth with LLCbench using MPIguard

scenario was LU-factorization and for the second was LLCbench2. The startup time of the *Analyzer* on each node was ignored and the experiments were conducted 20 times.

Table 6.3 shows the average overhead in seconds produced by MPIguard's *Profiler* and *Analyzer* for LU-factorization. An overhead of 2.48% is generated when the *Profiler* is active and of 3.37% when the *Analyzer* is executed.

As we explained before, LLCbench2 is an extreme example of message passing because it does not perform local computation and executes 5000 point-to-point and broadcast MPI functions. Also, it executes several memory and string functions from *libc*. Table 6.4 shows the average overhead produced by MPIguard for LLCbench2. The *Profiler* produces an overhead of 20.84%. This increase on the execution of the parallel applica-

Table 6.3 Performance of MPIguard for LU-factorization

Task	*Mean Duration(seconds)*	*Standard deviation*
LU-factorization	104.27	1.11
Profiling LU-factorization	106.94	2.77
Monitoring LU-factorization	107.79	2.91

tion is due to inefficient disk access, because in the current implementation, every function call that is intercepted by the *Profiler* is written on disk. Furthermore, the trace produced by LLCbench2 contains more than 1.5 million function calls. However, it is important to observe that for each MPI application the *Profiler* only needs to be executed a few times. The *Analyzer*, application that only access the disk during its start-up to create the sorted tree, produces an overhead of 15.58%.

Table 6.4 Performance of MPIguard for LLCbench2

Task	*Mean Duration(seconds)*	*Standard deviation*
LLCbench2	1083.57	48.05
Profiling LLCbench2	1309.41	67.51
Monitoring LLCbench2	1252.44	71.53

In summary, these results show that MPIguard produces an acceptable overhead of less that 5% when the MPI application performs some local computations (comparable with any other monitoring system), but there is a great impact on the performance when the number of local computations is low in each node. However, there are several factors that we have to take into account to measure the performance of MPIguard, among them the

benchmark algorithm used, the number and type of functions being profiled, the status of the network in the cluster, and the available resources in each node. An example of the impact of those factors on the 8-nodes cluster used for testing MPIguard can be seen in Table 6.4, where the standard deviation of the experiments with LLCbench2 with and without MPIguard is quite large.

CHAPTER VII

CONCLUSIONS

We have demonstrated that sequences of function calls can be used to verify the correct execution of an MPI parallel program in a cluster of Linux workstations. When the applications are executed under normal conditions, the only set of anomalies that we expect to find correspond to function calls that allow the start-up and end of MPI communication.

We have implemented MPIguard, the first distributed-IDS approach implemented for a high-performance environment, that is able to collect and analyze sequences of function calls of any C library for dynamic linked programs. The *Profiler* collects function calls for each node and creates the *profile* of the MPI application. This module can be also used for debugging or logging. The *Analyzer*, executed in each node of the cluster, reads the *profile* of the current MPI executable and executes the detection algorithm. As a result of our experiments, we concluded that the sequence matching algorithm has a better detection rate than the Hidden Markov Model . However, we believe that the data model provided by the HMM is very powerful for large parallel applications and we expect to present an empirical analysis of the detection capabilities of the HMM for the anomaly detection task using sequences of function calls.

We created artificial datasets to demonstrate the detection capabilities of the algorithms, and we implemented new attacks for MPI programs. We conducted experiments with MPIguard in off-line and on-line mode and showed that we can achieve low false positive and false negative rate.

MPIguard can be used not only to detect intrusions, but also it can be used as a fault detection mechanism. Even more, MPIguard's architecture allows the incorporation of additional sensors to monitor attributes such as memory usage or idle time of a process. With such a configuration, a new tool can be implemented on top of MPIguard to react when a anomaly has been found. Also, MPIguard's architecture provides de ability to collect and monitor function calls for more than one program, profiling a complete work session in the cluster. Experiments need to be conducted to demonstrate that MPIguard is still accurate in such a scenario.

MPIguard is highly portable to others UNIX-like systems, because the implementation of the interposition library and the shared memory communication was done with the standard C library. Furthermore, there is not need for kernel modification or *root* access.

Finally, although MPIguard's current implementation achieves low overhead when the MPI application performs local computations and its performance can be competitive with any other IDS system, we still need to investigate the performance of the *Profiler* and the *Analyzer* in a real-world cluster.

REFERENCES

[1] J. Allen, A. Chirstie, W. Fithen, J. McHug, J. Pickel, and E. Stoner, *State of the Practice of Intrusion Detection Technologies*, Technical report, Software Engineering Institute, Carnegie Mellon, Ithaca, New York, 2000, Networked Systems Survivability Program.

[2] R. Buyya, *High Performance Cluster Computing*, Prentice Hall, Leningrad, 1999, Vol. 1.

[3] W. W. Cohen, "Fast Effective Rule Induction," *Machine Learning: The 12th International Conference*, 1995, Morgan Kaufmann.

[4] R. Cunningham, R. P. Lippmann, D. J. Fried, and S. L. Garfikle, "Evaluating Intrusion Detection Systems Without Attacking Your Friends: The 1998 DARPA Intrusion Detection Evaluation," *Network Intrusion Detection*, 1999.

[5] T. W. Curry, "Profiling and Tracing Dynamic Library Usage Via Interpositon," *Proceedings of the USENIX 1994 summer conference*, 1994.

[6] Y. Dandass, *Solving Systems of Linear Equations using LU Factorization*, Tech. Rep., Mississippi State University, 1999.

[7] L. Eschenaur, "ImSafe: Immune Security Architecture," World Wide Web, 2001, http://umsafe.sourceforge.net/inside.htm.

[8] E. Eskin, W. Lee, and S. J. Stolfo, "Modeling System Calls for Intrusion Detection with Dynamic Window Sizes," *Proceedings of DARPA Information Survivabilty Conference and Exposition II (DISCEX II)*, Anaheim, CA, 2001, DARPA.

[9] W. Fan, M. Miller, S. Stolfo, W. Lee, and P. Chan, "Using Artificial Anomalies to Detect Unknown and Known Network Intrusions," *Proceedings of The First IEEE International Conference on Data Mining*, San Jose, CA, November 2001.

[10] G. Florez, "Analyzing System Call Sequences with Adaboost," *To appear in the Proceedings of the 2002 International Conference on Artificial Intelligence and Applications (AIA)*, Malaga, Spain, September 2002.

[11] G. Florez and L. Boggess, "Using Adaboost with Different Network Topologies for System Call Analysis," *To appear in the Proceedings of the 2002 ANNIE Conference*, Rolla, MO, November 2002.

[12] S. Forrest and T. A. Longstaff, "A Sense of Self for UNIX Processes," *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, 1996, IEEE Computer Society Press, pp. 120–128.

[13] R. A. gingell, M. Lee, X. T. Dang, and M. S. Weeks, "Shared Libraries in Sun(OS)," *Proceedings of the USENIX 1987 Summer Conference*, Phoenix, Arizona, 1987, pp. 131–145.

[14] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer, "A Secure Environment for Untrusted Helper Applications: Confining the Wily Hacker," *Proceedings of the 1996 USENIX security symposium*, 1996.

[15] S. Goldt, S. van der Meer, S. Burkett, and M. Welsh, "The Linux Programmer's Guide," World Wide Web, 1995, http://ibiblio.org/mdw/LDP/lpg/node5.html.

[16] G. G. Helmer, J. S. K. Wong, V. Honavar, and L. Miller, "Intelligent Agents for Intrusion Detection," *Proceedings of the 1998 IEEE Information Technology Conference*, Syracuse,NY, 1998.

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

[18] A. Jones and Y. Lin, "Application Intrusion Detection Using Language Library Calls," *Proceedings of thhe 17th Annual Computer Security Applications Conference*, New Orleans, LA, December 2001.

[19] C. Ko, G. Fink, and K. Levitt, "Automated Detection of Vulnerabilities in Privileged Programs by Execution Monitoring," *Proceedings of the 10th Annual Computer Security Applications Conference*, Los Alamitos, CA, 1994, IEEE Computer Society Press, pp. 134–144.

[20] S. Kumar and E. Spafford, *An Application of Pattern Matching in Intrusion Detection*, Technical Report 94-013, Department of Computer Sciences, Purdue University, 1994.

[21] B. Kuperman and E. Spafford, *Generation of Application Level Audit Data Via Library Interposition*, Technical Report TR 99-11, COAST laboratory, Purdue University, 1998.

[22] W. Lee, S. J. Stolfo, P. K. Chan, E. Eskin, W. Fan, M. Miller, S. Hershkop, and J. Zhang, "Real Time Data Mining-based Intrusion Detection," *Proceedings of DARPA Information Survivabilty Conference and Exposition II (DISCEX II))*, Anaheim, CA, June 2001.

[23] W. Lee, S. L. Stolfo, and P. K. Chan, "Learning Patterns from UNIX Process Execution Traces for Intrusion Detection," *AAAI Workshop on AI Approaches to Fraud Detection and Risk Management*. AAAI, 1997, pp. 50–56.

[24] Z. Liu, G. Florez, and S. Bridges, "A Comparison of Input Representations in Neural Networks: A Case Study in Intrusion Detection," *Proceedings of the 2002 International Joint Conference on Neural Networks (ICJNN'02)*, Honolulu, Hawaii, May 2002.

[25] G. Marsaglia and A. Zaman, *Toward a Universal Random Number Generator*, Tech. Rep., Florida State University, 1987, FSU-SCRI-87-50.

[26] S. Microsystems, "Secure Computing with Java: Now and the Future," World Wide Web, 2001, http://java.sun.com/marketing/collateral/security.html.

[27] P. Mucci, "LLCbench (Low-Level Characterization Benchmarks) Home Page," World Wide Web, July 2000, http://icl.cs.utk.edu/projects/llcbench/.

[28] D. Oppenheimer and M. R. Martonosi, "Performance Signatures: A Mechanism for Intrusion Detection," *Proceedings of the 1997 Information Survivability Workshop*, San Diego, CA, 1997, Software Engineering Institute, Sponsored by the IEEE Computer Society.

[29] P. A. Porras and P. G. Neumman, "Emerald: Event Monitoring Enabling Responses to Anomalouls Live Disturbances," *Proceedings of the 20th National Information Systems Security Conference*, 1997.

[30] R. Rabenseifner, "Automatic MPI Counting Profiling," *Proceedings of the 42nd Cray User Group Conference, CUG SUMMIT 2000*, 2000.

[31] L. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, February 1989, vol. 77 of 2, pp. 257–286.

[32] B. Rhodes, J. Mahaffey, and J. Cannady, "Multiple Self-Organizing Maps for Intrusion Detection," *Proceedings of the 23rd National Information Systems Security Conference*, Baltimore, MD, October 2000.

[33] A. Somayaji, *Operating system stability and security through process homeostasis*, doctoral dissertation, The University of New Mexico, Albuquerque, New Mexico, July 2002.

[34] E. Spafford and D. Zamboni, *Data Collection Mechanisms for Intrusion Detection Systems*, Technical Report 2000-08, CERIAS, Purdue University, 2000.

[35] H. Steven A, S. Forrest, and A. Somayaji, "Intrusion Detection Using Sequences of System Calls," *Journal Of Computer Security*, vol. 6, no. 3, 1998, pp. 151–180.

[36] M. Stillerman, C. Marceau, and M. Stillman, "Intrusion Detection for Distributed Applications," *Communications of the ACM*, July 1999, vol. 42, pp. 62–69.

[37] A. Sundaram, "An Introduction to Intrusion Detection," World Wide Web, 1996, http://www.cs.purdue.edu/coast/archive/data/categ24.html.

[38] T.Lane, "Hidden Markov Models for Human/Computer Interface Modeling," *IJCAI-99 Workshop on Learning About Users*, 1999, pp. 35–44.

[39] M. Torres, "Algorithms for artificial anomaly data generation," Mississippi State University, 2002, Internal report.

[40] M. Torres, "Simple resource attack in C for a UNIX-based operating system and MPI library," Mississippi State University, 2002.

[41] J. S. Vetter and B. R. de Supinski, "Dynamic Software Testing of MPI Applications with Umpire," *SC2000: High Performance Networking and Computing Conference*. ACM/IEEE, 2000.

[42] D. Walker, "A Type System for Expressive Security Policies," *Proceedings of the 27th ACM SIGPLAN Symposium on Principles of Programming Languages*, Boston, 2000, ACM.

[43] C. Warrender, S. Forrest, and B. A. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *Proceedings of the IEEE Symposium on Security and Privacy*, 1999, pp. 133–145.

[44] A. Wespi, H. Debar, and M. Nassehi, "Fixed vs. Variable-length Patterns for Detecting Suspicious Process Behavior," 2000, J. Computer security.

[45] M. Yarrow, *NAS Parallel Benchmarks Suite 2.3 - IS*, Tech. Rep., NASA Ames Research Center, Moffett Field, CA, 1995.