Name: German Florez-Larrahondo
Submission date: 14/10/2003
Practical version: 1.4
Option: 1

Anomaly Detection in High-Performance Applications Running on Linux Clusters

Abstract

Linux clusters have become widely used computational resources in security conscious environments. In these systems, MPI (Message Passing Interface) is a standard programming technique used to implement complex scientific programs, which often run for long periods of time with large amounts of sensitive data. As these programs are being executed, a number of different types of irregularities can occur, including those that result from user misbehavior, intrusions, corrupted data, deadlocks or failure of cluster components. We present a method for automatically detecting such irregularities based on the identification of anomalies in the behavior of MPI programs. The behavior is learned from the library function calls issued by the program in each node of the Linux cluster. Two different techniques are used to build the models of normal behavior and to detect deviations from normality. A prototype system has been built that exhibits a high level of detection with a low performance penalty, and it can be installed in every node of the cluster as part of a defense-in-depth strategy.

1. 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. Many of them, use system call logs produced by the operating system or a purpose-specific trace tool to detect anomalies in a UNIX host. However, as library functions are currently the preferred way to access system services, the quality of information produced by system calls traces has decreased, *i.e.* it can be possible to better understand the behavior of complex programs by looking at the function calls issued by a process.

Actual UNIX and Linux systems support collection of information in real time from different type of processes, including parallel programs implemented in C with 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.

A cluster of workstations is a special environment of interest because generally parallel programs running in such a distributed environment execute large and periodic tasks, and therefore, we can collect *accurate* patterns of activity. In contrast, in a typical LAN, a given process can perform a wide variety of tasks, and the definition of "normal" behavior becomes a hard task. Examples of applications used in a wide variety of forms include *UNIX shells* and *sendmail* among others. Others interesting aspects of a Linux cluster are related with the fact that each of the software and hardware components may have its own vulnerabilities, configuration errors can prevent components from working together correctly, and network failure and CPU misuse may lead the entire system into abnormal behavior.

1.1 Motivation for High Performance Cluster Monitoring

High Performance clusters are not new, but their use and employment is proliferating as the cost of hardware decreases and processing power and network speeds increase. It is not difficult to imagine a not too distant future where specialized clusters are employed in the dashboard of a modern automobile offering a drive-by-wire application, the nose cone of an anti-missile missile controlling targeting maneuvers, or in the cockpit of a high performance aircraft handling various avionics applications. Such systems will be characterized by a reasonably static suite of software applications whose behavior can be characterized and monitored. As with all software, maintenance will be performed over time, which has the risk of introducing malicious exploits. Detecting anomalies and reporting them quickly will be an essential requirement in these systems. Reporting might take the form of alerts to a system administrator or even, perhaps, turning on a "maintenance required" warning light on a console.

The techniques reported in the remainder of this paper are showing promise of high reliability and low overhead detection. While we continue to explore these techniques as components of an intrusion detection system, it should be apparent that there are other related applications.

1.2 Why Anomaly detection in a cluster

We assume an environment where the application base (MPI programs written in C) is well defined, but the user-base may be not. Therefore, patterns of usage are expected to emerge, and they can be used to detect irregularities in the execution of parallel programs. These irregularities include intrusions, user misbehavior, corrupted data, deadlocks and failure of cluster components among others. The proposed system is designed as a host-based anomaly detector that can be installed in the nodes of the cluster as part of a *defense-in-depth strategy*.

Current monitoring systems are able to detect some of those irregularities by testing individual services with simple message exchange among the cluster components. Other systems provide useful methods for configuration and prevention of errors in the system. However, none can find every possible error or misconfiguration. A typical example is testing a *telnet* service: Determining that the *telnet* port is open does not necessarily ensure that a user can login remotely [15]. Furthermore, theory of computation indicates that with the current computing model, the problem of determining whether or not a program (a Turing Machine) will halt with an input X is not decidable. Therefore, in the general case, we must assume that the execution of any program might fail.

Some of the advantages of deploying a host-based anomaly detection system in a cluster are:

- It can quickly determine when some types of intrusions are taking place.

- It provides another layer of security, especially in the case where the user-base is no longer controlled. While a grid cluster has to have an authorization and authentication policy, the most useful mode of operations is one in which *classes* of users are likely to be acceptable. In this way, special assumptions about users and their behaviors over time cannot be made.
- It provides a qualitative measure of the execution of a job, in an attempt to answer the question "*is the parallel program being executed correctly*''.

This paper is organized as follows. Section 2 will introduce related work. The architecture of our prototype system is described in section 3. The techniques used to model the behavior of MPI applications are introduced in section 4. Experimental results demonstrating the effectiveness and performance of our algorithms are presented in section 5. Conclusions are described in section 6.

2. Related Work

Verifying a program's behavior by analyzing the processes, methods, tasks or calls that a program executes, has been an active field of research. Both static and dynamic inspections of algorithms have been proposed, including Java Virtual Machine [16], efficient certified code [17], Janus [18] and the execution monitor [19]. Forrest and Longstaff [20] reported one of the first research papers on analysis of system calls. Other algorithms include the EMERALD system [7] and Somayaji's pH [10]. Warrender, Forrest and Pearlmutter [13] performed a comparison of different algorithms for solving the problem of anomaly detection of privileged UNIX programs using system calls. In previous work, we have successfully applied different artificial neural networks and boosting algorithms in the field of intrusion detection [2,4].

Markov processes are widely used to model systems in terms of state transitions. Some detection algorithms that exploit the Markov property implement Hidden Markov Models (HMM), Markov chains, and sparse Markov trees. Lane [21] used HMMs to profile user identities for the anomaly detection task. An open problem with this profiling technique is the ability to select appropriate model parameters. Others experiments performed by Warrender, Forrest and Pearlmutter [13] 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.

Library interposition is widely used to deploy debuggers and monitoring systems in standard UNIX-like operating systems. Examples include the Curry's Shared Library Interposer [1], Kuperman and Spafford's data library [22], Jain and Sekar's system [23] and the monitoring of function calls for intrusion detection of Jones and Lin [24]. Some of these systems include an automatic generation of source code.

A number of applications have been implemented to gather data from the execution of parallel programs in a cluster of workstation, but none were developed with anomaly detection as an objective. Some examples include the automatic counter profiler [8] and the Umpire manager [12]. Massie, Chun and Culler developed Ganglia [25], a scalable distributed monitoring system for high-performance computing systems. It is able to collect between 28 and 37 different built-in and user-defined metrics ``which capture arbitrary application-specific state'' [25]. Marzolla [26] implements a performance monitoring systems for a large computing cluster. Examples of the metrics used are available space on */tmp*, */var* and */usr*, cached memory, available memory and total swap. Finally, Luecke et al. [5] improve MPI-CHECK to detect deadlocks in MPI programs written in Fortran using a handshake strategy.

Recent work in the field of cluster monitoring has shown that it is possible to efficiently combine the output of several sensors in the cluster and present an overall status to the system administrator. However, because of the large amount of information generated, the wide variety of sensors used, and the complex behavior of the parallel applications being executed in the cluster, it is difficult to determine if a parallel program is behaving as expected or not. Our research addresses this problem.

3. System Architecture

Our goal is to demonstrate that detection of anomalies during the execution of MPI applications can be conducted in (near) real-time with high accuracy and low false alarm rates. To achieve this goal, a prototype system called *MPIguard* has been implemented and its effectiveness has been demonstrated.

We agree with Buyya [15]: ``The network is just a system component, even if a critical one, but not the sole subject of monitoring in itself.'' The development of high-speed network technologies is changing the original TCP/IP based network philosophy. Myrinet, Gigabit Ethernet and Infiniband, for example, are widely used to build cluster systems, providing more than 1Gb/s bandwidth (in contrast with the 10Mb/s or 100Mb/s of the traditional technologies). To obtain such a high bandwidth, an OS kernel bypass method is used to copy data directly from the user space memory to the buffer in the physical device by using direct memory access (DMA). We believe that a traditional network based detection system does not fit well in a cluster environment, because generally detection is conducted by analyzing TCP/IP streams, and thus, only a portion of the real traffic on a high-speed cluster is being monitored. Additionally, in a fully saturated network traditional detection systems cannot handle the amount of information generated in a cluster environment and both the accuracy and the performance may degrade.

Standard technologies for monitoring nodes in the cluster create statistical behavior models based on the output of different sensors spread over the entire cluster system, measuring quantities such as network latency, CPU time, memory usage, etc. This information is quite useful for determining the overall status of the cluster, but it can be misleading when it is used to *monitor a particular parallel application*. For instance, the execution of a parallel program with different input data and parameters will result in very skewed values for CPU usage times, thus summarizing this feature using mean and variance would not be appropriate.

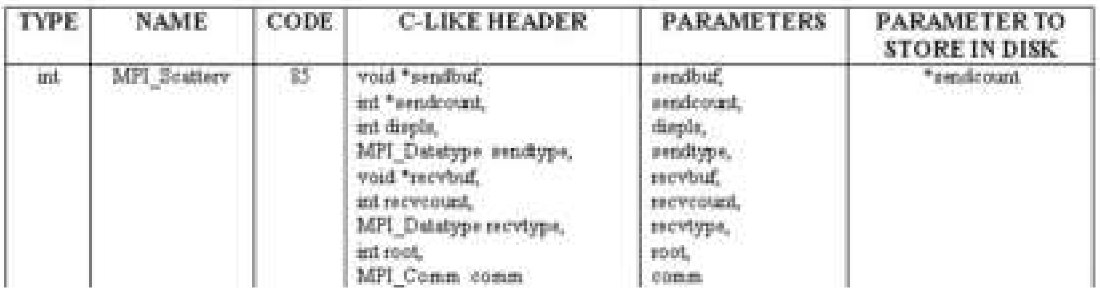
This problem is even more difficult to address when the application makes use of randomized algorithms. Since most of the sensors used in current monitoring systems are time-driven, (a snapshot of the sensor is taken at a given time period), creating a useful model of the sensor for a parallel application can be a difficult task. For these reasons, we have chosen an event-driven system, where specific sensors collect information every time an event is generated. In our research, this event corresponds to a library function call issued by the parallel application.

A software application issues calls to the operating system to perform a wide variety of functions such as I/O access, memory requests, and network management. However, many application programmer interfaces (APIs) do not make use of system calls, mainly for performance reasons or because no privileged resources need to be manipulated. A typical example is the set of functions such as *cos*, *sin* or *tan* from the standard mathematical library of C.

Linux provides a large collection of mechanisms to trap system and function calls from any process in user-mode. For instance, in order to monitor kernel calls, the OS provides the tools *strace*, *trace* and *truss*, but these tools only record kernel level functions and the trap mechanism produces too much overhead [1]. However, another method that has been widely used for implementing performance and monitoring tools is library interposition.

The link editor (ld) in a Linux operating system builds dynamically linked executables by default. The compiler builds incomplete executables and 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" [26].

In the Linux system, the link editor uses the LD_PRELOAD environment variable to search for the user's dynamic libraries. 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" [26]. Thus, the library interposition technique allows interception of the function calls without the modification or recompilation of the dynamically linked target program. By default, C compilers in Linux use dynamic linking.



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

Figure 2 Example of an MPIguard's configuration file

MPIguard automatically generates the source code needed to gather information of any C function from dynamically linked programs. Figure 1 shows the template used. The system administrator interacts with the tool indicating which functions and parameters will be analyzed, but he/she does not need to modify any source code. Basically, a configuration file is used to describe the type, the name and the parameters of the functions (See Figure 2). *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 argument (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. In the experiments presented in this paper, such buffer size is not used. However, we expect to analyze function call arguments in our future research work.

It is important to observe that we are creating our own profiling tool. This simple tool is, by itself, a value source of information for any other IDS or monitoring systems, since it can be personalized to collect data at different levels from a wide variety of libraries.

```
__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 1 Template used by MPIguard to collect any C function

With our framework, the monitoring of programs is a two-stage process. First, we obtain samples of the behavior of the application that are used to create a *profile*. Second, we compare the behavior of the application with this profile in real-time for each one of the nodes. In the first stage, we collect the function call traces from all the compute nodes in a central database and train the detection algorithms. This subsystem is called the *Profiler*. Although this stage incurs substantial communication and computational costs, it is assumed to be an off-line process. Even more, we assume that the profile of an application seldom changes, and therefore the overhead caused by data collection and training can be ignored. During the second phase (the *Analyzer*), the profile is loaded in memory and a detection algorithm is executed in real-time for each of the nodes where the parallel application is being executed.

Figure 3 shows the architecture used for monitoring system calls from *libc* and *libmpipro* (MPI's dynamic library). The calls collected by the interposition library are sent via shared memory either to the *Profiler* (writing the information to disk) or to the *Analyzer* (detecting anomalies in real-time).

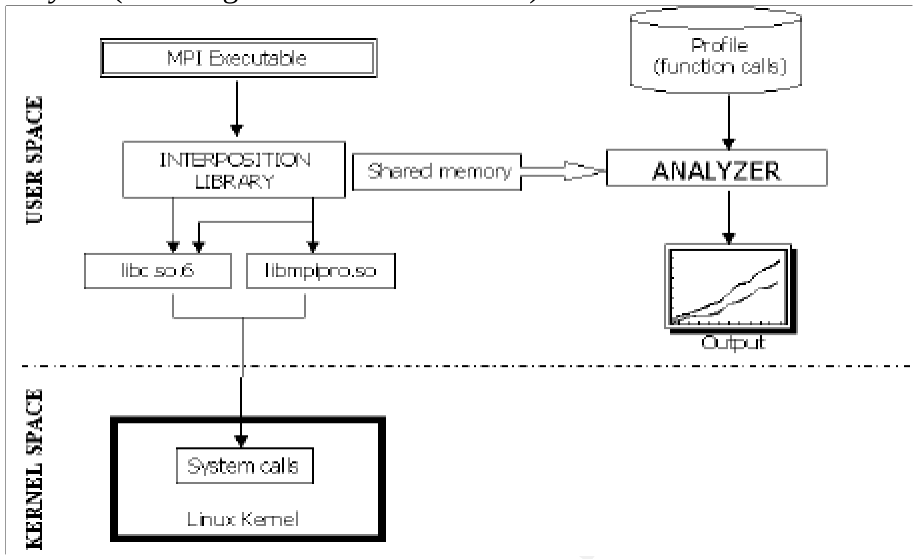


Figure 3 Architecture used to monitor function calls

4. Detection Techniques

We have used two methods to model normal behavior: exact sequence matching, and Hidden Markov Models (HMM). We have chosen exact sequence matching as an example of a simple deterministic algorithm and Hidden Markov Models as an example of sequence modeling.

4.1 Sequence matching

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

execve, brk, open, fstat, mmap, close, open, mmap, munmap

and we have defined a window size of 4. We slide this window across the sequence, and for each call we encounter, we record the calls that precede it at different positions within the window, numbering the calls from 0 to *(window_size – 1)*, with 0 being the current system call. The trace above yields the following instances:

position 3	**position 2**	**position 1**	**current**
			execve
		execve	*brk*
	execve	*brk*	*open*
execve	*brk*	*open*	*fstat*
brk	*open*	*fstat*	*mmap*
open	*fstat*	*mmap*	*close*
fstat	*mmap*	*close*	*open*
mmap	*close*	*open*	*mmap*
close	*open*	*mmap*	*munmap*

This database is stored as a sorted tree to perform efficient comparisons and correspond to the *profile* of a given MPI program. The detection of anomalies is straightforward: If a sequence gathered from the function call trace of a *new* instance of an MPI program cannot be found in the tree, an alarm is issued.

4.2 Hidden Markov Model (HMM)

Hidden Markov Models (HMM) are used for modeling sequences of events and are widely used for speech recognition and DNA sequencing. These models have the ability to capture patterns from sequences of events. A Hidden Markov Model is a doubly stochastic process, where the states represent an unobservable condition of the system. For each state, there exist two probabilities: the probability of generating any of the observable system outputs, and the probability of transition to the next state [13]. The elements of an HMM are as follows [9]:

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 symbols probability distribution, and
5. p , the initial state distribution.

For convenience, an HMM model can be expressed as:

$$l = (A, B, p)$$

The Baum-Welch algorithm is generally used to train the transition and symbol probabilities of an HMM. The detail description of the algorithm can be found in [9]. In order to train the HMM with the Baum-Welch algorithm, we should specify the observation sequence O (the trace containing the function calls from the normal execution of the parallel program). With an optimal model l , we can assume that the probabilities *A* and *B* generalize the normal behavior of the process.

Figure 4 shows an example of an ergodic HMM with two states and 3 possible symbols. In this model, the symbol *c* (e.g. the system call *close*) has a probability of being generated in the first state of 0.6, whereas it has a probability of 0.1 in the second state. Also, the probability of transition between state 1 and state 2 is very high, 0.9.

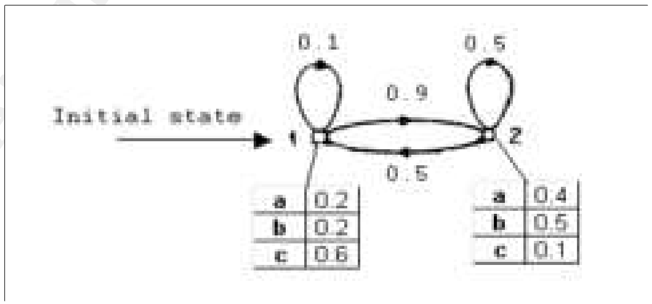


Figure 4 Example of a trained HMM model

In order to detect anomalies, we developed an algorithm that traverse all the possible transitions and emission symbols for each state. When such a probability is very low for a new stream of calls, the call is flagged as anomalous. A formal definition of this algorithm is presented below. Given a new observation $\overline{O}$, that corresponds to the trace of an unseen instance of the program, and using the model l learned from normal instances of the application.

For each observation $\overline{O_t}$:

- *For each state 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 q).*
  - *If the probability of producing the symbol $\overline{O_t}$ in the current state $B(i, \overline{O_t})$ is less than q then the function call in the trace 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.*

Although there is no mathematical basis for using the same threshold θ to test both the probability of moving to the current state *Si* and the probability of producing a symbol $\overline{O_t}$ in the current state, we wanted to include the smallest number of parameters possible for the anomaly detection task with the HMM.

Finally, it is important to observe that the detection algorithms for both the exact sequence matching and the HMM can be performed online.

5. Experimental Results

5.1 Approach

As we mentioned before, we focus our research on parallel programs based on the Message Passing Interface (MPI) protocol because it is a current popular standard. Our goal is to demonstrate that detection of anomalies of MPI applications can be conducted in (near) real-time with high accuracy and low false positive rates. To achieve this goal we have followed the methodology presented below:

- Define the level of profiling: library function calls issued by an MPI program;
- Implement two parallel applications, one performing primarily local computations and the other performing extensive message passing;
- Inject ``cluster anomalies" able to produce several types of anomalies in the cluster, simulating suspicious behavior of the MPI application;
- Collect information in real-time from the two parallel applications and the anomalies in order to create a data set containing both normal and anomalous behaviors;
- Perform a comparison of different data models using the data set, and identify the advantages and drawback of the methods;
- Conduct experiments with well-know benchmarks to test our system performance; and,
- Provide analysis of the results in relation to our stated goals.

5.2 Datasets

In order to test the effectiveness of our detection algorithms and system architecture, we created a dataset containing both normal and anomalous traces of two well-known parallel programs, in an attempt to recreate large-scaled simulations and scientific programming. The first program, LU-Factorization is an implementation of the factoring method used for solving systems of linear equations. The second program, called LLCbench2 executes MPBench benchmarks [6]. Each benchmark is selected based on a normally distributed random number generator, with the means and the standard deviation computed experimentally to execute broadcast MPI functions quite often, and to execute point-to-point routings only a few times.

LLCbench2 is an example of an application that generates several messages among the processors with few local computations, whereas LU-factorization is an example of parallel applications that perform extensive computations with limited message passing. It is important to observe that the communication patterns of LU-factorization are more complex than LLCbench2.

Because non-simulated anomalous data from high-performance parallel applications is not available at the moment we wrote this document, we implemented several methods to generate anomalous behavior in MPI applications. An overview of the attacks is given in Table I, and an extended description of the techniques used can be found in our previous work [11]. We generated more than 500 anomalous instances of LU-Factorization and more that 350 anomalous instances of LLCbench2 in a cluster with 4 nodes. We also generated more than 200 normal instances of each application. An example of the daemon attack –stealing of resources using a daemon process- for an MPI program can be seen in Appendix A (Taken from Torres et al. [11]). These attacks are implemented directly on MPI programs or by using library interposition. In our current research work, we are also injecting faults in the NIC (Network Interface Card) and high-performance network libraries. It is very important to observe that both applications do use random number generators in certain steps of the algorithms, so we do not expect to generate two exact traces for same program. In practice, commercial and scientific parallel applications also make use of stochastic algorithms. The inclusion of random traces as part of our normal behavior is perhaps one of the most important characteristics of our dataset, and one of the reasons why the anomaly detection problem in this kind of applications is much more complicated than the analysis of standard privileged programs such as *sendmail* or *ftp*.

TABLE 1 Description of the "attacks" implemented to generate anomalies

NAME	TECHNIQUE	PURPOSE	DESCRIPTION
DaemonCopy	Trojan-horse	Steal resources	A daemon process is created just before the program ends. Appends data to a temporal file
DaemonComputation	Trojan-horse	Steal resources and gather information	Math operations at the end of the program, CPU exhaustion.
NoDaemonComputation	Trojan-horse	Steal resources	Random math operations. CPU exhaustion.
CopyFile	Interposition library	Gather information	Content of files is copied to unauthorized locations.
ModifyMPI	Interposition library	Denial of service	Behavior of MPI function is changed

5.3 Hardware and Software Environment

The traces were collected on a Linux cluster containing one head node (able to compile and launch the parallel programs) and eight compute nodes. The head node is a four CPU SMP computer and the other nodes are dual CPU SMP computers. These machines are fully connected with Ethernet and Giganet (high-speed) switches. The operating system installed on each node is RedHat 7.1 Linux, kernel 2.4.2, and the MPI environment used in all experiments was MPI/Pro 1.5.

5.4 Experiments

The first experiments were conducted to test the ability of our methods to detect the attacks described in the previous sections. Figure 5 shows the accuracy (percentage of anomalous traces correctly classified by the detector) for each AI technique when monitoring LLCbench2. An alarm is issued when the sequence of calls deviates from the *profile* in any of the 4 nodes where the program is being executed. The accuracy of the detection for LU-Factorization is shown in Figure 6.

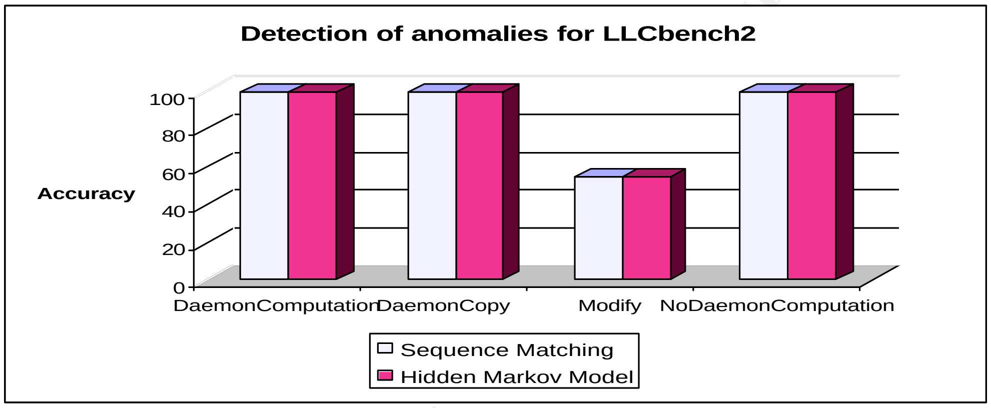


Figure 5 Detection of anomalies for LLCbench2

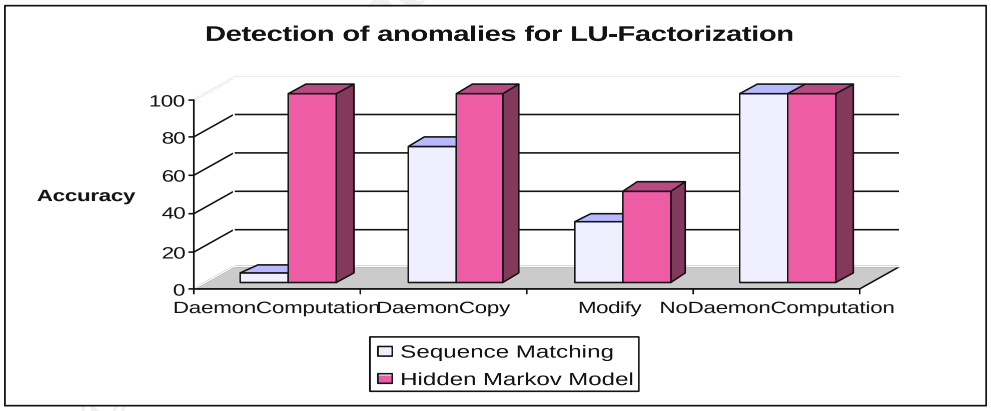


Figure 6 Detection of anomalies for LU-Factorization

One of the major conclusions of our experiments is that the exact sequence matching cannot be used to monitor applications such as LU-factorization, due to the complexity of the algorithms and the diversity of calls being generated. As mentioned before, LU-Factorization is a complex scientific program, whereas LLCbench2 is a naïve benchmark. Although other researchers have reported that sequence matching is an effective method for detecting attacks on UNIX privileged programs [13], this method performed poorly in the detection of anomalies of a scientific application. In contrast, the overall detection rates for the Hidden Markov Model were greater than 90%, with the exception of the attack called "*ModifyMPI*". In this attack, the behavior of some MPI function is changed randomly, and therefore, it is not an easy task to define whether or not the program being executed contains anomalies or not. Our detection system using the HMM generates 0% of false alarm rate, *i.e.* all normal traces were classified correctly as normal behavior. This is an encouraging result, since one of the biggest critics to anomaly detection systems is the large number of false positives generated.

Some interesting conclusions can be drawn from these experiments:

- Some anomalies cannot be detected by using function call traces. It might be interesting to combine the information produced by these traces and the streams produced by typical performance monitoring systems, *e.g.* CPU usage and memory consumption.
- The exact sequence-matching algorithm generates too many false positives (this can be seen when the detection algorithm is executed using normal data). As an example, for LU-factorization with system calls the sequence matching algorithm generates 1,987 anomalies.
- The above statement indicates that we need to specify some user threshold to be able to differentiate between normal and anomalous behavior. We have chosen this threshold empirically based upon the maximum number of false positives generated for both programs and both types of calls with normal behavior. Therefore, the threshold of LU-factorization for the sequence matching algorithm is 1,987, for the HMM is 0. It is very important to observe that we are defining this threshold to be able to compare the accuracy of our algorithms, but we believe that indicating whether or not a sequence of calls is anomalous based upon a threshold can be misleading. Instead, a measure of similarity of the behavior of the parallel application with the normal trace can be given to the system administrator

Since the prototype system conducts a real-time detection, we also conducted experiments to determine the performance penalty produced by the monitoring of calls using the HMM. We executed two well-known benchmarks, MPBench and NAS-IS [14]. Table 2 presents an average of the running time of 50 instances of the NAS parallel computational benchmark (using 4 nodes) when the detection algorithms are executed in real-time. The overhead created by the HMM is 3.97%.

TABLE 2 Overhead Analysis of NAS-IS (Seconds)

	Mean (50 runs)	Standard Deviation
Without monitoring	*30.647*	*2.647*
With monitoring	*31.863*	*3.154*

A similar result is encountered when executing the MPBench-Latency benchmark. In Figure 7, a comparison of the latency with respect to the function *MPI_Send* (send a message to one node) is presented.

6. CONCLUSIONS

We have described our effort to provide a lightweight anomaly detector of high performance parallel programs, and we have demonstrated that function calls can be used to verify the correct execution of an MPI application in a cluster of Linux workstations. As a result of our experiments, we conclude that detection Hidden Markov Models results in high accuracy and 0% false positive rate. However, traditional sequence matching algorithms perform poorly, due to the complexity of the parallel programs.

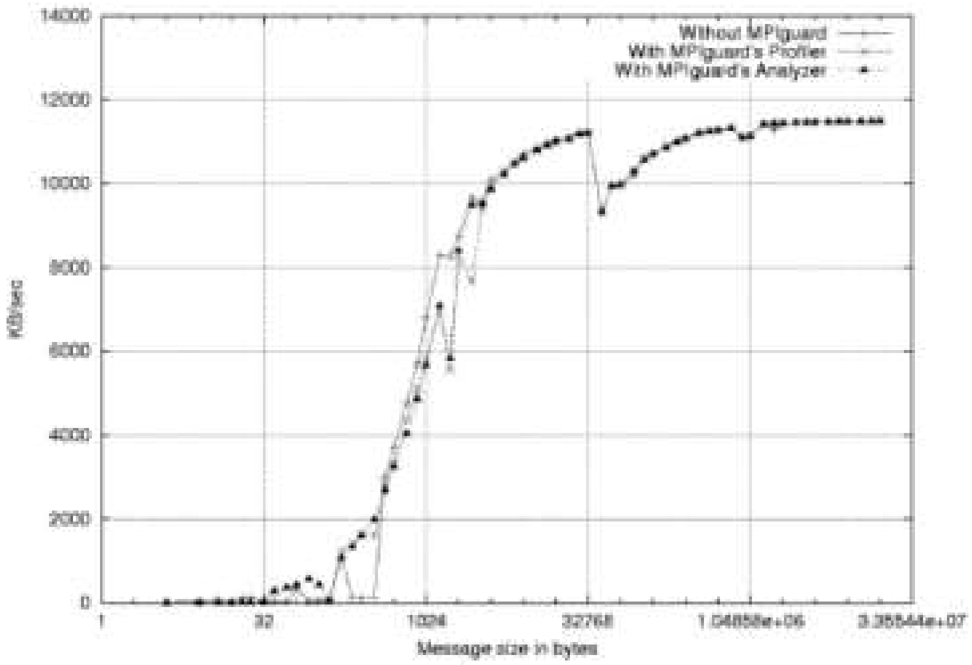


Figure 7. Comparison of MPI_Send latency with MPBench

ACKNOWLEDGMENTS

This work was supported by the Naval Research, contract N00014-01-0678; the Army Research Laboratory, contract DAAD17-01-C-0011; and the National Science Foundation, contract CCR-0098024, 98-1849 and 99-88524.

REFERENCES

[1] Curry, T. W. Profiling and tracing dynamic library usage via interposition. In Proceedings of the USENIX 1994 Summer Conference, 1994. http://citeseer.nj.nec.com/curry94profiling.html

[2] Florez, G. Analyzing system call sequences with Adaboost. In Proceedings of the 2002 International Conference on Artificial Intelligence and Applications (AIA), Malaga, Spain, September 2002.

[3] Forrest, S., S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff_. A sense of self for UNIX processes. In Proceedings of the 1996 IEEE Symposium on Security and Privacy, Los Alamitos, CA, 1996, IEEE Computer Society Press, pp. 120–128.
http://gfl.homedns.org/PublishPapers/A4_GermanFlorez_Adaboost_362_AIA.pdf

[4] Liu, Z., G. Florez, and S. Bridges. A comparison of input representations in neural networks: a case study in intrusion detection. In Proceedings of the 2002 International Joint Conference on Neural Networks (ICJNN'02), Honolulu, Hawaii, May 2002. http://gfl.homedns.org/PublishPapers/AComparisonofNNSystemCallsIJCNN02.pdf

[5] Luecke, G.R., Y. Zou, J. Coyle, J. Hoekstra, and M. Kraeva. Deadlock detection in MPI programs. Concurrency and Computation: Practice and Experience 2003; 14(11): 911–932.

[6] Mucci, P., LLCbench (Low-Level Characterization Benchmarks). http://icl.cs.utk.edu/projects/llcbench/ [July 2000].

[7] Porras, P. A., and P. G. Neumman. Emerald: event monitoring enabling responses to anomalous live disturbances. In Proceedings of the 20th National Information Systems Security Conference, 1997.

[8] Rabenseifner, R. Automatic MPI counting profiling. In Proceedings of the 42nd Cray User Group Conference, 2000. http://citeseer.nj.nec.com/rabenseifner00automatic.html

[9] Rabiner, L. A tutorial on Hidden Markov Models and selected applications in speech recognition. In Proceedings of the IEEE, February 1989, vol. 77 of 2, pp. 257–286.

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

[11] Torres, M., R. Vaughn, S. Bridges, G. Florez and Z. Liu. Attacking a high performance computer cluster. In Proceedings of the 15th Annual Canadian Information Technology Security Symposium, Ottawa, Canada, 2003.

[12] Vetter J. S. and B. R. de Supinski. Dynamic software testing of MPI applications with umpire. In Proceedings of SC2000: High Performance Networking and Computing Conference, ACM/IEEE, 2000. http://citeseer.nj.nec.com/395759.html

[13] Warrender C., S. Forrest, and B. A. Pearlmutter. Detecting intrusions using system calls: alternative data models. In Proceedings of the IEEE Symposium on Security and Privacy, 1999, pp. 133–145.

[14] Yarrow, M.. NAS Parallel Benchmarks Suite 2.3 - IS, Technical report, NASA Ames Research Center, Moffett Field, CA, 1995.

[15] Buyya. R. High Performance Cluster Computing, vol. 1. Prentice Hall. Leningrad, 1999.

[16] Sun Microsystems. Secure computing with Java: Now and the future. http://java.sun.com/marketing/collateral/security.html

[17] Walker. D. A type system for expressive security policies. In *Proceedings of the 27th ACM SIGPLAN Symposium*, Boston, 2000. ACM.

[18] Goldberg. I, D. Wagner, R. Thomas and E.A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the 1996 USEXIN Security Symposium*, 1996.

[19] C. Ko, G. Fink, and K. Levitt. Automated detection of vulnerabilities in privileged programs by execution monitoring. In *Proceedings of the 10th Annual Computer Security Applications Conference*, Los Alamitos, CA, 1994, IEEE Computer Society Press, pp. 134–144.

[20] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for UNIX processes. In *Proceedings of the 1996 IEEE Symposium on Security and Privacy*, Los Alamitos, CA, 1996, IEEE Computer Society Press, pp. 120–128.

[21] T. Lane. Hidden Markov models for human/computer interface modeling. In *Proceedings of IJCAI-99 Workshop on Learning About Users*, 1999, pp. 35–44.

[22] B. Kuperman and E. Spafford. Generation of application level audit data via library interposition, Technical Report TR 99-11, COAST laboratory, Purdue University, 1998.

[23] K. Jain and R. Sekar. User-level infrastructure for system call interposition: a platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed Systems Security*, 2000, pp. 19–34.

[24] A. Jones and Y. Lin. Application intrusion detection using language library calls. In *Proceedings of the 17th Annual Computer Security Applications Conference*, New Orleans, LA, December 2001.

[25] M.L. Massie, B.N. Chun, and D.E. Culler. The ganglia distributed monitoring system: Design, implementation, and experience. http://ganglia.sourceforge.net/talks/parallelcomputing/ganglia-twocol.pdf

[26] M. Marzolla. A performance monitoring system for large computing clusters. In *Proceedings of the Eleventh Euromicro Conference on Distributed and Network-Based Processing*, Genova, Italy, 2003, pp. 393–400.

Appendix A
Daemon Process Attack (Taken from Torres et. al [11])

```
#include <stdlib.h>
#include <unistd.h>
#include <signal.h>
#include <fcntl.h>
#include <errno.h>
#include <stdio.h>
#define SIZE__ 1000

void main(){
        int pid;
        void *pointerMemory;
        /* The first Child fork */
        pid = fork();
        if (pid < 0) {
                exit(1); /* error encountered, no child has been created!*/
        }
        if (pid != 0) {
                exit(0); /* this is the parent, and hence should be terminated*/
        }
        /* make the process a group leader, session leader, and lose control tty */
        setsid();
        /* close STDOUT, STDIN, STDERR */
        close(STDIN_FILENO);
        close(STDOUT_FILENO);
        close(STDERR_FILENO);
        /* close STDOUT, STDIN, STDERR */
        /* ignore SIGHUP that will be sent to a child of the process */
        signal(SIGHUP, SIG_IGN);
        umask(0); /* lose file creation mode mask inherited by parent */
        chdir("/"); /* change to working dir */
        pid = fork();
        if (pid < 0) {
                exit(1); /* fork() failed, no child process was created! */
        }
        if (pid != 0) {
                exit(0); /* this is the parent, hence should exit */
        }
        /* this is the child process of the child process of the actual calling process */
        /* and can safely be called a grandchild of the original process */
        signal(SIGPIPE, SIG_IGN);
        /* ignore SIGPIPE, for reading, writing to non-opened pipes
        every program using pipes should ignore this signal for
        being on the safe side */
        /* this is the main daemon process, also the grand child process */
        while(1){
                sleep(DURATION_SEC);
        PointerMemory=(char*)malloc(SIZE__);
        }
}
```