Lightweight monitoring of MPI programs in real-time

German Florez, Zhen Liu, Susan M. Bridges*,†, Anthony Skjellum‡ and Rayford B. Vaughn

Center for Computer Security Research and High Performance Computer Laboratory, Department of Computer Science and Engineering, Mississippi State University, MS, 39762-9637, U.S.A.

SUMMARY

Current technologies allow efficient data collection by several sensors to determine the overall evaluation of the status of a cluster. However, no previous work of which we are aware analyzes the behavior of the parallel programs themselves in real-time. In this paper, we perform a comparison of different artificial intelligence techniques that can be used to implement a lightweight monitoring and analysis system for parallel applications on a cluster of Linux workstations. We study the accuracy and performance of deterministic and stochastic algorithms when we observe the flow of both function library and OS system calls of parallel programs written with C and MPI. We demonstrate that monitoring of MPI programs can be achieved with high accuracy and in some cases with a 0% false positive rate in real-time, and we show that the added computational load on each node is small. As an example, the monitoring of function calls using a Hidden Markov Model generates 3.97% overhead. The proposed system is able to automatically detect deviations of a process from its expected behavior in any node of the cluster, and thus it can be used as an anomaly detector, for performance monitoring to complement other systems, or as a debugging tool. Copyright © 2000 John Wiley & Sons, Ltd.

KEY WORDS: Anomaly detection, message-passing interface, cluster monitoring, interposition library, loadable kernel module, hidden Markov model, artificial neural networks

*Correspondence to: Susan M. Bridges, Center for Computer Security Research, Mississippi State University, MS, 39762-9637, U.S.A.
†E-mail: bridges@cse.msstate.edu
‡Department of Computer and Information Sciences, University of Alabama, Birmingham
Contract/grant sponsor: National Science Foundation; contract/grant number: CCR-0098024, 98-18489, 99-88524

Copyright © 2000 John Wiley & Sons, Ltd.
1. INTRODUCTION

Linux clusters have become widely used computational resources in security conscious environments. The workstations within these clusters exchange information not only through TCP/IP networks, but also through special high-speed network fabrics that facilitate rapid communication.

As our computer systems have become increasingly complex, they have also become more unpredictable and unreliable. It is difficult, for example, to correctly configure a cluster environment that involves many workstations and COTS software packages. Furthermore, 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. Many monitoring tools have already been designed to help system administrators identify system failures. However, few of them use artificial intelligence (AI) techniques to analyze raw data and provide useful guidance to the system administrators. Similarly, many detection systems collect data from well-known sensors provided by the operating system, but they are not combined in any meaningful way.

This paper introduces a systematic way to create an anomaly and fault detection system in a cluster environment by using machine learning algorithms collecting data from new sensors.

1.1. Motivation for High Performance Cluster Monitoring

The authors of this paper have been working for more than four years on the problem of intrusion detection in high performance computing clusters. The work reported here has been an outgrowth of that research. High Performance (HP) clusters are characterized by a number of computing nodes (some may be multiple processor) jointly working on a parallelized problem. These machines communicate with each other over two network fabrics - a traditional TCP/IP based network and a high speed network operating at gigabit speed. Because speed is the essence of such architectures, security software such as intrusion detection sensors must not significantly degrade performance. Our work has resulted in the development of a prototype intelligent intrusion detection system (IIDS), which uses several novel artificial intelligence techniques to detect anomalous behavior at both host and network levels. While the details of this IIDS work are not reported in this paper, the techniques used and experimental results are.

HP 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 HP 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 (corrective, perfective, and adaptive) 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 is well defined, but the user-base may not be. 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 user misbehavior, intrusions, corrupted data, deadlocks and failure of cluster components among others.

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 [3]. Furthermore, the 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.

For these reasons, we believe that research must be conducted in the field of anomaly detection until we are able to build error-proof software and hardware. If we can achieve low overhead and high accuracy with anomaly detection systems, they can be embedded in the next generation of cluster and grid architectures.

Some others advantages of anomaly detection in the cluster are:

- 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?”;
- It supports many traditional monitoring systems. For instance, network congestion detected by a network monitoring system might be related to the misuse of an application.
- It can monitor misbehavior on several components of the cluster environment. Clusters are generally composed of multiple internal networks; therefore, by using a protocol-independent architecture the detection of anomalies can take place on any component of the cluster.
- It provides security, especially in the case where the user-base is no longer controlled. While a grid cluster has to have an authorization and authenticacon 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 can quickly determine when an intrusion is taking place.
1.3. Why monitor library function calls and operating system calls

Standard technologies create statistical behavior models of a cluster of workstations 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 or a operating system call issued by the parallel application.

1.4. Why use AI techniques

We want to demonstrate that the behavior of a parallel program can be profiled and that this profile can be used to detect anomalies, such as misuse of resources, network and hardware failures, intrusions, deadlocks, etc. This is not a trivial problem, because we are interested in the real-time behavior (dynamic analysis) of a process, and not in the details of its underlying algorithms (static analysis). As an example, in the case of randomized algorithms, we might require an infinite table to store the flow of every instance of an algorithm. Therefore, we require some alternative but powerful data models such as a Markov model or an artificial neural network to model behavior.

1.5. Why chose a host-based monitoring system

We agree with Buyya [3]: “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.
1.6. Scalability issues

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.

In the first stage, we collect the function call or system call traces from all the compute nodes in a central database and train the detection algorithms. Although this stage incurs substantial communication and computational costs, it is assumed to be done off-line one time to obtain a profile. We assume that the profile of an application seldom changes, therefore the overhead caused by data collection and training can be ignored.

During the second phase, 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. In Section 5 we will show that our detection system imposes a small overhead in the entire cluster system. Although our work has been tested only in a small system, scalability is not a major issue in the monitoring process of an application because there is no extra communication among the nodes. Scalability concerns do arise in the communication of the global status of the cluster. These issues are discussed later.

1.7. Contributions of this work

- This work is a first attempt to detect anomalies in the behavior of MPI parallel programs in real-time. As a result of our research, we have built a software system that is useful as an anomaly detector, a performance monitoring tool to complement other systems, or as a debugging tool.
- In our work, we compare three different data models that can be used to automatically analyze parallel programs in Linux. Previous work on analysis of process behavior [44] demonstrated that a simple deterministic data model can be used to represent normal behavior and detect anomalies with better accuracy and performance than more complex models. In contrast, our results demonstrate that simple deterministic methods perform poorly when the program flow grows in size and variety, and that more complex methods are required.
- We assume that we cannot define all the possible normal behaviors of an application since many scientific applications use probabilistic algorithms. Thus randomness is included in our training datasets and the flow of an algorithm may change every time it is executed, even if the input data is the same. Analyzing this kind of program behavior is by itself an interesting challenge.

The remainder of the paper is structured as follows. Section 2 contains a survey of related work. In Section 3 we present the methodology used to measure the accuracy and performance of our system. In Section 4 we describe three data models useful to detect anomalies. In Section 5 we present experimental results that demonstrate the efficiency of our system. Finally in Section 6 we state the most important conclusions of our research and in Section 7 we discuss planned and plausible extensions of this work.
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 [24], efficient certified code [43], Janus [15] and the execution monitor [20].

Forrest and Longstaff [12] reported one of the first research papers on analysis of system calls. An extended overview is described in [30]. Other algorithms include the EMERALD system [26], Somayaji’s pH [29] and Eschenauer’s ImSafe [9]. Warrender, Forrest and Pearlmutter [44] 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 [10, 22].

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 [31] 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 [44] 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.

Different kernel methods have been used for enhancing the security of operating systems. Some have built their function only in a loadable kernel module [37, 13, 14], some require a patch to the kernel to support the module [8, 4], and others have implemented their frameworks for the Linux kernel as kernel patches [32, 29].

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 [5], Kuperman and Spafford’s data library [21], Jain and Sekar’s system [18] and the monitoring of function calls for intrusion detection of Jones and Lin [19]. 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 [27] and the Umpire manager [42].

Massie, Chun and Culler developed Ganglia [36], a scalable distributed monitoring system for high-performance computing systems. This system is able to collect between 28 and 37 different built-in and user-defined metrics “which capture arbitrary application-specific state” [36].

Marzolla [35] 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.

Lyu and Mendiratta [34] use resource monitoring for reliability analysis of a cluster architecture in their system RCC (Reliable Clustered Computing), and present a Markov reliability model for software fault tolerance.
Augerat, Martin and Stein [1] implement a scalable monitoring tool that uses *bwatch* (a Tcl/Tk script designed to watch the load and memory usage of Beowulf clusters).

Luecke et al. [33] improve MPI-CHECK to detect deadlocks in MPI programs written in Fortran using a handshake strategy.

Finally, Baldoni, Marchetti and Verde [2] use CORBA’s portable interceptors to implement a fault-tolerant CORBA client invocation without modifications to the source code of a distributed application.

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. APPROACH

We focus our research on parallel programs based on the Message Passing Interface (MPI) protocol because it is a current popular standard [7, 17, 38, 39]. 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 two levels of profiling: library function calls and operating system calls issued by an MPI program;
- Instrument two parallel applications, one performing primarily local computations and the other performing extensive message passing;
- Develop “cluster attacks” 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 attacks 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.

3.1. Data streams

As explained above, we have chosen an event-driven architecture, collecting calls issued by a program written in C. In this paper, the sequence of calls are referred to as data streams. This section provides a technical description of those calls and how they are collected.
3.1.1. Operating system calls

System call interfaces are the APIs through which the operating system kernel provides low-level operations (such as memory allocation, file access, and network transfers) to the user space applications. For Linux, this interface is wrapped by the GNU standard C library (glibc) so that C programmers can have a common standard interface. Although user space applications usually utilize the low-level operations through C library function calls, some of applications and attacks bypass the libc interface and invoke system calls directly.

Several mechanisms can be used to obtain system call information. We use a method based on loadable kernel modules (LKMs) because of the flexibility, efficiency and compatibility this approach provides. LKM s are used by the Linux kernel and many other UNIX-like systems as well as Microsoft Windows to expand their functionality. An LKM can access all the variables in the kernel. This method is more flexible than the methods based on kernel patching because it does not require changes to the kernel source code. A privileged user can load or unload the LKM during runtime. The disadvantage is that an attacker can install malicious code into the kernel before the tool is loaded. This weakness can be resolved by adding the tool into a startup script.

Extensions that use kernel level interposition have a broad range of capabilities. “Extensions can provide security guarantees (for example, patching security flaws or providing access control lists), modify data (transparently compressing or encrypting files), re-route events (sending events across the network for distributed systems extensions), or inspect events and data (tracing, logging)” [37, p.1]. The major advantage of this method is that it cannot be bypassed. All programs must call the system call interface to access low level functionality implemented by the kernel. Another advantage is it does not need to change the COTS code. Only the kernel needs to be changed. The disadvantage is that a security hole in the monitor program is much more dangerous than with other methods. It can cause the system to crash or to grant root privilege.

3.1.2. Library function calls

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 \texttt{cos}, \texttt{sin} or \texttt{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 [21], but these tools only record kernel level functions and the trap mechanism produces too much overhead [5]. 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. Building “incomplete” executables, the link editor allows the incorporation of different objects in real time. 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].

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” [5]. 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. 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” [16].

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]: a subset of the library can be profiled instead of the whole library, different levels of profiling can be generated and 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) [18]. Also, as explained above, the process of searching the symbol table in the new interposition library and the allocation of memory increases the execution time of the target process.

Since we did not want to restrict the application of our framework to MPI programs, the definition of the libraries and functions to be profiled is in fact a system administrator’s task. With a configuration file, and by using a template like to one depicted in Figure 1, our system generates the source code needed to interpose function calls from any dynamic library.

3.2. System architecture

Figure 2 and Figure 3 describe our system’s architecture. It is important to account for the following characteristics:

- In order to monitor system calls, the monitoring system should be executed as a part of the kernel, and therefore, root access is needed. In contrast, function calls can be monitored in user space.
- Figure 3 shows the monitoring of two libraries, libc and libmpipro. However, as explained before, any other dynamic library can be included in the system. As a matter of fact, in the current experiments we are also intercepting the mathematical library libm.
- The ANALYZER is created by using a pattern recognition algorithm. We have implemented and tested algorithms based on sequence matching, Hidden Markov Models or artificial neural networks. Details of these data models are covered in Section 4.
3.3. Datasets and generation of anomalous behavior

3.3.1. Why we need a dataset

We want to create a dataset containing both normal and anomalous traces of parallel programs in an attempt to address the following questions:

- Can we detect anomalies of parallel programs using host-based detectors, or do we need network-based detectors?
- Which stream, function, or system calls must be used? Which stream is more efficient?
- Which detection algorithm should we implement?
- What kind of anomalies can be detected?
- Can we use the same detection framework to identify anomalies in work sessions instead of single processes?
3.3.2. 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 as indicated in Figure 4. 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.

3.3.3. Normal behavior

We decided to obtain normal traces in the cluster by simulating operations in a Linux cluster of workstations. In our first experiments, we used two small parallel programs representing different types of computation. The first one, LU-factorization, is an implementation of the
Figure 3. Architecture used to monitor function calls

Figure 4. Head node architecture of our Linux cluster
LU-factorization method [6], an algorithm widely used for solving systems of linear equations. The second, LLCBench2, is based on the LLCbench benchmark [25].

The original LLCbench application executed one of the three point-to-point benchmark routings (latency, bandwidth and bi-directional bandwidth) or one of the five broadcast benchmark routines (roundtrip, broadcast, reduce, all-reduce and all-to-all). In our new version, thousands of broadcast routines are selected based on a normally distributed random number generator, with the means and the standard deviation computed experimentally to execute the MPI functions reduce and broadcast quite often, and to execute the point-to-point benchmark routines 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 that LLCbench2.

We executed both applications with different inputs (using only 4 processors) in an attempt to generate as many normal variations as possible. 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 application is much more complicated than the analysis of standard privileged programs such as sendmail or ftp [29, 30, 44].

3.3.4. Anomalous behavior

Several attack scenarios can be implemented to generate anomalous behavior in a high performance cluster of workstations, affecting the integrity, confidentiality and availability of the system. We implemented some of the attacks described in our previous work [40] to create behavior that differs from the normal traces generated when programs such as LU-factorization or LLCbench2 are executed.

Basically, we want to simulate system failures and stealing of resources or proprietary information by a misbehaving user or attacker. Table I describes each of the attacks and Table II shows the number of traces included in the dataset. It is also important to note that the anomalies injected do not generally cause a dramatic change in the performance of the application. Occasionally, the parallel program prematurely terminates because of real-time failures caused by the some of the anomalies. However, the detection of such events is a trivial task and therefore those execution instances are not included in this work.

Figure 5 shows an equiwidth histogram of the time between consecutive events. In this particular example, an event corresponds to the execution of a library function call for the LU-factorization program. The abnormal behavior was created by executing the CopyFile attack. The x-axis shows the time required by each event (in microseconds) and the y-axis displays the normalized frequency. The width of each bucket range was 10 microseconds. Since the shape of the histogram is very similar for each execution, and there are no significant new
peaks of activity when the CopyFile attack is being executed, we conclude that the anomalies cannot be easily detected just by looking for drastic slow-downs in the parallel communication.

4. DATA MODELS

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 [29]). In our research, we wanted to analyze the accuracy and the performance of artificial neural networks, Hidden Markov Models and exact sequence matching to detect abnormal program behavior using the identifier of a function call and its relative order in a trace.
Table I. Description of the attacks used to generate anomalies

<table>
<thead>
<tr>
<th>NAME</th>
<th>TECHNIQUE</th>
<th>PURPOSE</th>
<th>DESCRIPTION</th>
</tr>
</thead>
<tbody>
<tr>
<td>daemonCopy</td>
<td>Trojan-horse</td>
<td>Steal resources</td>
<td>A daemon process is created just before the MPI program ends. Appends data to a temporal file.</td>
</tr>
<tr>
<td>daemon-Computation</td>
<td>Trojan-horse</td>
<td>Steal resources and gather information</td>
<td>A daemon process is created just before the MPI program ends. Performs math operations</td>
</tr>
<tr>
<td>nodemon-Computation</td>
<td>Trojan-horse</td>
<td>Steal resources</td>
<td>Execute random math operation just before the MPI program ends.</td>
</tr>
<tr>
<td>copyFile</td>
<td>Interposition library</td>
<td>Gather information</td>
<td>Every time a close(F) call is executed, the file F is copied</td>
</tr>
<tr>
<td>modify MPIFunctions</td>
<td>Interposition library</td>
<td>Generate failures</td>
<td>With a given probability, the behavior of some MPI functions is changed</td>
</tr>
<tr>
<td>hybrid</td>
<td>Interposition library</td>
<td>Gather information and generate failures</td>
<td>Executes ModifyMPIFunctions and CopyFile attacks</td>
</tr>
</tbody>
</table>

Table II. Summary of the dataset: Number of traces

<table>
<thead>
<tr>
<th></th>
<th>LU-factorization Function calls</th>
<th>LU-factorization System calls</th>
<th>LLCbench2 Function calls</th>
<th>LLCbench2 System calls</th>
</tr>
</thead>
<tbody>
<tr>
<td>hybrid</td>
<td>80</td>
<td>100</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>copying</td>
<td>100</td>
<td>100</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>daemoncomputation</td>
<td>80</td>
<td>80</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>daemoncopy</td>
<td>80</td>
<td>80</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>modify</td>
<td>88</td>
<td>100</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>nodemoncomputation</td>
<td>80</td>
<td>80</td>
<td>100</td>
<td>100</td>
</tr>
<tr>
<td>TOTAL ATTACKS</td>
<td>508</td>
<td>540</td>
<td>360</td>
<td>360</td>
</tr>
<tr>
<td>NORMAL(TEST)</td>
<td>220</td>
<td>220</td>
<td>200</td>
<td>200</td>
</tr>
</tbody>
</table>

We have chosen exact sequence matching as an example of a simple deterministic algorithm, backpropagation neural networks as an example of a robust classifier, and Hidden Markov Model as an example of sequence modeling.
4.1. Sequence matching

The sequence-matching algorithm is a simple approach for detecting abnormal behavior and security violations of a program.

4.1.1. Description

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:

\begin{align*}
\text{execve, brk, open, fstat, mmap, close, open, mmap, munmap}
\end{align*}

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:

<table>
<thead>
<tr>
<th>position 3</th>
<th>position 2</th>
<th>position 1</th>
<th>current</th>
</tr>
</thead>
<tbody>
<tr>
<td>execve</td>
<td>execve</td>
<td>brk</td>
<td>brk</td>
</tr>
<tr>
<td>brk</td>
<td>open</td>
<td>fstat</td>
<td>mmap</td>
</tr>
<tr>
<td>open</td>
<td>fstat</td>
<td>mmap</td>
<td>close</td>
</tr>
<tr>
<td>mmap</td>
<td>close</td>
<td>open</td>
<td>mmap</td>
</tr>
<tr>
<td>close</td>
<td>open</td>
<td>mmap</td>
<td>munmap</td>
</tr>
</tbody>
</table>

Somayaji has created a formal definition of the subsequences generated by the sliding window technique [29]. Let \( C \) be the alphabet of possible system calls, \( c = |C| \), \( T = t_1, t_2, \ldots, t_r \) \( t_i \in C \), \( \tau \) the length of \( T \), \( \omega \) the window size \( 1 \leq \omega \leq \tau \), \( P \) the profile (set of patterns associated with \( T \) and \( \omega \)). A sequence \( P_{seq} \) is defined as

\begin{equation}
P_{seq} = \{(s_i, s_{i+1}, \ldots, s_j) : s_i, s_{i+1}, \ldots, s_j \in C, 1 \leq i, j \leq \tau, j - i + 1 = \omega, \}
\end{equation}

This database can be stored as a sorted tree to perform efficient comparisons. Figure 6 shows an example of a profile with a window of length 4.

4.1.2. Anomaly detection with sequence matching

With exact matching, if a sequence \( \overrightarrow{S} = <S_i, S_{i+1}, \ldots, S_j> \) is not contained in \( P_{seq} \), the sequence \( \overrightarrow{S} \) is tagged as anomalous. We count the number of anomalous sequences within a trace and give this result to the system administrator. Research has demonstrated that this method is very efficient and it is able to detect security violations and software failures [12, 44, 29, 30].
4.2. Artificial Neural Networks

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

An artificial neural network is composed of simple processing units, or nodes, and connections between them. The connection between any two units has some weight, which is used to determine how much one unit will affect the other. In our experiments, a backpropagation neural network is used to build the model of normal program behavior. The backpropagation network consists of three types of units: input units, hidden units, and output units. By assigning a value to each input node, and allowing the activations to propagate through the network, a neural network performs a functional mapping from one set of values (assigned to the input units) to another set of values (appearing in the output units). The mapping function is stored in the weights of the network.

We use the same concept of a sliding window as described in Section 4.1. As input for the neural network, we use a binary representation of the call identifiers. Therefore, if we define a window size of 10, there are 80 inputs for the neural network. Because we seek to determine whether an input string is anomalous or normal, we use single output node to indicate the status of the input string with a value of 0 indicating normal and a value of 1 indicating anomalous. We use a single hidden layer in our network. Network weights are initialized to random values prior to training. Since the number of hidden nodes and number of epochs have
great influence on the classification performance of the learned neural network, a five-fold cross-validation was used to select the best number of hidden nodes and the most appropriate number of epochs to prevent the overfitting of the backpropagation neural network.

Training of our artificial neural networks involves the use of both normal and anomalous data. For the experiments described in this paper, we have used artificial anomalies that were randomly generated and spread throughout the training space.

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

In order to reduce computational requirements, the standard sigmoid function (Equation 2) can be modified as suggested by Tveiter [41]. This simple sigmoid function is given in Equation 3 and its derivative is presented in Equation 4. Both can be computed very quickly.

\[
y_{std} = \frac{1}{1 + e^{-x}} \tag{2}
\]

\[
y = \frac{x}{1 + |x|} + 0.5 \tag{3}
\]

\[
y' = \frac{1}{2(1 + |x|)^2} \tag{4}
\]

The purpose of our work is to classify the whole trace instead of a single short sequence. Therefore, we need a method which accumulates the effect of single anomaly sequences. Previous work has shown that the anomalous sequences usually occurred together in a short period [10, 11, 22, 23]. To reduce false positive rates, we used a method named MaxBurstCounter which takes advantage of the locality property of the anomalous sequences. In this method we define a variable that remembers the total number of anomalous sequences seen so far. We call this variable BC (burst counter). After encountering a normal sequence, we decrease this variable at a slow rate. This is similar to Ghosh et al.’s leaky bucket method [19]. Since the output of our neural network is either 0 or 1 and theirs is a continuous number, we cannot use the leaky bucket method directly. Our method has an effect similar to the LFC method of Somayaji [38] but with much less computational overhead. The assumption of Max Burst Counter is that an anomalous sequence seen long ago should only have a small effect on classification.

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

### 4.3. Hidden Markov Models

Hidden Markov Models (HMM) are used for modeling sequences of events and are widely used for speech recognition and DNA sequencing [28].
4.3.1. Description

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 [44]. The elements of an HMM are as follows [28]:

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. \( \pi \), the initial state distribution.

For convenience, an HMM model can be expressed as

\[
\lambda = (A, B, \pi) \tag{5}
\]

Rabiner [28] describes three different problems that must be solved with an HMM when we have an observation sequence \( O = O_1O_2O_3...O_T \):

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

4.3.2. The Baum-Welch algorithm

The Baum-Welch algorithm is generally used to train the transition and symbol probabilities of an HMM [28] (attempting to solve problem 3). This is an iterative algorithm that computes \( A \), \( B \) and \( \pi \) 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 corresponds to the probability of the partial observation sequence \( O \) up to time \( t \) and state \( S_i \) at time \( t \), given the model \( \lambda \). It can be defined as

\[
\alpha_t(i) = P(O_1O_2...O_t, q_t = S_i|\lambda) \tag{6}
\]

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 \( \lambda \), and it can be expressed as

\[
\beta_t(i) = P(O_{t+1}O_{t+2}...O_T|q_t = S_i, \lambda) \tag{7}
\]

The Baum-Welch algorithm updates the model \( \lambda \) 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 4.3.2 (adapted from [28]).

With \( \xi \), we can compute the probability of being in state \( S_i \) at time \( t \) given the model and observation sequence

\[
\gamma_t(i) = \sum_{j=1}^{N} \xi(i,j) \tag{8}
\]
The initial state distribution can be computed as the expected frequency (number of times) in state $S_i$ at time $t = 1$

$$\pi_i = \gamma_1(i)$$  \hspace{1cm} (9)

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 $S_i$

$$\bar{a}_{ij} = \frac{\sum_{t=1}^{\infty} \xi_t(i, j)}{\sum_{t=1}^{\infty} \gamma_t}$$  \hspace{1cm} (10)

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 $j$

$$\bar{b}_i(k) = \frac{\sum_{s.t. O_t=v_k} \gamma_t(j)}{\sum_{t=1}^{\infty} \gamma_t(j)}$$  \hspace{1cm} (11)

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

4.3.3. Anomaly detection with HMMs

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 $\lambda$, 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 program), we can apply the following algorithm [44]:

1. Using the model $\lambda$
2. For each observation \( O_t \).

- 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 \( \theta \)).
  - If the probability of producing the symbol \( O_t \) in the current state \( B(i, O_t) \) is less than \( \theta \) then the function call in the trace is labeled as anomalous.
- If \( O_t \) 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 it could be said that comparing the probability of moving to the current state \( S_i \) and the probability of producing a symbol \( O_t \) in the current state with the same user threshold \( \theta \) has no mathematical foundation, we wanted initially to include the smallest number of parameters possible for the anomaly detection task with the HMM.

Figure 8 shows an example of an ergodic HMM with two states and 3 possible symbols. Using the algorithm described before, with \( \theta = 0.3 \), figure 9 shows that the trace \( CAAC \) could be produced by that HMM, since there is at least one possible state producing every symbol \( O_t \). For instance, in step \( \text{iii.} \), \( A \) could not be produced in state 1, but it could be produced in state 2 with a probability of 0.4.

5. Experimental Results

When training each of our data models we used a five-fold cross-validation to select the best parameters, that is, the parameters that produce the smallest number of false positives. For the artificial neural network, we have selected 32 hidden nodes and a window size of 6; for the Hidden Markov Model, we have selected five states with \( \theta = 0 \), and for the exact sequence matching, we have selected a window size of 10.
5.1. Comments on function calls vs. system calls

In our current architecture, the system administrator specifies a subset of the function calls available in one or more libraries, such as the C standard library, the C mathematical library and the MPI/Pro library. In contrast, all the operating system calls issued by the application are being monitored. Therefore, by using system calls we expected to observe interesting behavior that cannot be recognized with the function call approach. As an example, take the interposition of a shared library. The process of interposing a library does generate overhead because the operating system needs to read the shared objects and allocate memory. When the parallel application is executed and a shared object is interposed, every anomaly detector recognizes suspicious events when the dataset used is composed of system calls, but none are detected based on function call logs. However, such anomalies could be detected if the system administrator indicates that C functions such as `dlsym` or `dlopen` should be profiled.

It is important to observe that the total number of system calls captured by the loadable kernel module is much larger than the function calls captured by the interposition library. This can be seen in Figure 10 and Figure 11, where the average number of calls generated by LLCbench2 and LU-factorization is depicted.

In the same way, we expected to observe interesting behavior with the function call trace that cannot be identified with system calls. This is due to the fact that not all library function calls will make use of system calls. This interesting property will be analyzed in more detail in the next section.
Figure 10. Number of calls for LLCbench2

Figure 11. Number of calls for LU-factorization
5.2. Overall detection

Table III shows the detection of anomalies for each data model. Some interesting conclusions can be drawn when analyzing this table:

- Some anomalies cannot be detected using the system call trace. Take for example, the nodemoncomputation attack. In this attack, some mathematical functions are executed in order to steal resources in each processor. We expected to notice this kind of behavior looking at the function call trace, because standard mathematical functions in C do not issue system calls \(^1\).

- 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 it is 0, and so on. 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.

5.3. Testing normal behavior

The first set of experiments were conducted to observe the behavior of the detectors for normal instances of both LU-factorization and LLCbench2 that were not used for training. As we expected, the accuracy (percentage of traces classified correctly) by the HMM and the ANNs is 100%. However, the sequence matching analysis of system calls performed poorly for LU-factorization, with an accuracy of only 20%. Its accuracy for LLCBench2 was 96%. This leads us to conclude that the LU-factorization dataset contains a more complex pattern of calls than the LLCbench2. As we explained before, LU-factorization is a commonly used scientific application, whereas LLCbench2 is a synthetic benchmark.

5.4. Testing anomalous behavior

Figure 12 shows the accuracy for each of the algorithms with anomalous data. The generation of anomalies was done with daemoncomputation, daemoncopy, modify, and nodemoncomputation.\(^1\)

\(^1\)However, as explained before, since the parallel program is dynamically linked, some system calls are generated for the allocation of the mathematical library libm of C
Table III. Average number of anomalies detected in the dataset

<table>
<thead>
<tr>
<th>Sequence Matching</th>
<th>Hidden Markov Model</th>
<th>Neural Network: Burst Counter</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>LU-factorization</strong> (Function calls)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DaemonComputation</td>
<td>86</td>
<td>23</td>
</tr>
<tr>
<td>DaemonCopy</td>
<td>5989</td>
<td>2969</td>
</tr>
<tr>
<td>Modify</td>
<td>66</td>
<td>11</td>
</tr>
<tr>
<td>NoDaemonComputation</td>
<td>3273</td>
<td>3282</td>
</tr>
<tr>
<td>Normal(Test)</td>
<td>19</td>
<td>0</td>
</tr>
<tr>
<td><strong>LU-factorization</strong> (System calls)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DaemonComputation</td>
<td>351</td>
<td>32</td>
</tr>
<tr>
<td>DaemonCopy</td>
<td>245</td>
<td>28</td>
</tr>
<tr>
<td>Modify</td>
<td>17815</td>
<td>2221</td>
</tr>
<tr>
<td>NoDaemonComputation</td>
<td>150</td>
<td>4</td>
</tr>
<tr>
<td>Normal(Test)</td>
<td>1987</td>
<td>0</td>
</tr>
<tr>
<td><strong>LLCbench2</strong> (Function calls)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DaemonComputation</td>
<td>92</td>
<td>22</td>
</tr>
<tr>
<td>DaemonCopy</td>
<td>1729</td>
<td>3835</td>
</tr>
<tr>
<td>Modify</td>
<td>173</td>
<td>14</td>
</tr>
<tr>
<td>NoDaemonComputation</td>
<td>3536</td>
<td>2647</td>
</tr>
<tr>
<td>Normal(Test)</td>
<td>10</td>
<td>0</td>
</tr>
<tr>
<td><strong>LLCbench2</strong> (System calls)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DaemonComputation</td>
<td>817</td>
<td>27</td>
</tr>
<tr>
<td>DaemonCopy</td>
<td>792</td>
<td>27</td>
</tr>
<tr>
<td>Modify</td>
<td>792859</td>
<td>73815</td>
</tr>
<tr>
<td>NoDaemonComputation</td>
<td>332</td>
<td>0</td>
</tr>
<tr>
<td>Normal(Test)</td>
<td>434</td>
<td>1</td>
</tr>
</tbody>
</table>
One of the major conclusions of our experiments is that a deterministic algorithm (the exact sequence matching) cannot be used for monitoring this type of parallel application, due to the complexity of the flow of the algorithms and the variety of calls that are being generated. A detailed analysis of the detection rate for our algorithms is presented in Figure 13 (LU-factorization monitoring system calls), Figure 14 (LU-factorization monitoring function calls), Figure 15 (LLCbench2 monitoring system calls), and Figure 16 (LLCbench2 monitoring function calls).

An interesting result is observed when the system attempts to detect the anomalies created with `modifyMPI`. This attack modifies the behavior of some MPI functions, by choosing random values for the parameters of the function. Whether or not a function is modified is based upon a uniform distribution, and there is a 50% chance of modification. It is possible that by changing the parameters of a function (e.g. MPI_Send) the flow of the application changes. This is reflected in both Figure 14 and 16, where the accuracy of the algorithms is about 50%: It seems that for some traces, the changes in the MPI function parameters do not generate behavior that deviates sufficiently from the profile and therefore they do not generate many anomalies. Therefore, the detection algorithms are not able to classify half of the traces as anomalous behavior.
5.5. Performance analysis

As explained in Section 4, our detection algorithms need to learn the normal behavior of a parallel application in order to perform the detection of anomalies. In our dataset, the trace of both system and function calls for 30 different instances of LU-factorization were collected. To estimate the size of the training data, we averaged the length of traces for the LU-factorization and we obtained an average trace length of 3,771 function calls and 9,581 system calls. Table IV shows the time required by the data models to learn the behavior of LU-factorization (please note the difference in time units). This training step needs to be done only once: when the administrator is interested in monitoring a new parallel application, or when for some reason the behavior of the application has changed over time.

We also conducted several experiments to analyze the performance of our algorithms when they are executed in real-time and in an offline fashion.
Table IV. Training times for LU-factorization

<table>
<thead>
<tr>
<th>Type of Attack</th>
<th>Sequence Matching</th>
<th>HMM</th>
<th>NN</th>
</tr>
</thead>
<tbody>
<tr>
<td>With Function Calls</td>
<td>0.0656 seconds</td>
<td>2.3140 minutes</td>
<td>1.1369 minutes</td>
</tr>
<tr>
<td>With System Calls</td>
<td>0.244 seconds</td>
<td>12.064 minutes</td>
<td>2.6094 minutes</td>
</tr>
</tbody>
</table>

5.5.1. Real-time detection

Figure 17 shows the latency of the function MPI_Send for 4 processors using LLCbench [25]. In this figure, the latency of the functions is presented when the benchmark is executed without our monitoring, when the benchmark is executed and the loadable kernel module or the interposition library is allocated in memory (but no detection is performed), and when the real-time monitoring is performed. We can observe that the overhead produced by the loadable kernel module is smaller than the overhead produced by the interposition library.
We also monitored function calls using the HMM and system calls using the ANN. In the figure, we can see that the overhead produced by the monitoring of system calls is greater than the overhead produced when monitoring function calls, due to the fact that the number of system calls is much greater than the number of function calls. These results are also confirmed when we executed the NAS benchmarks. To show the overhead in computation, we conducted experiments with the class A NAS IS benchmark [45] in 4 processors. Average results of 50 executions are presented in Table V. The overhead produced by the inclusion of the interposition library is 3.42%, whereas the overhead produced by the inclusion of the LKM is just 2.53%. The overhead created by the HMM model when monitoring function calls is 3.97% (that is, it just spends 0.55% in the detection algorithm itself). In contrast, when monitoring system calls we face a total overhead of 5.28% (the detection algorithm with system calls creates an overhead of 2.75%). We believe this overhead is reasonable and acceptable for most applications.
Figure 16. Detection of anomalies for the LLCbench2 monitoring function calls

Table V. Performance of the monitoring system using NAS IS (in seconds)

<table>
<thead>
<tr>
<th>Parallel Program</th>
<th>Mean</th>
<th>Standard Deviation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NAS-IS (CLASS A, 4 processors, TCP communication)</td>
<td>30.647</td>
<td>2.647</td>
</tr>
<tr>
<td>FUNCTION CALLS (Using a HMM)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS-IS with the interposition library in memory</td>
<td>31.098</td>
<td>2.938</td>
</tr>
<tr>
<td>Real-time monitoring of function calls for NAS-IS</td>
<td>31.863</td>
<td>3.194</td>
</tr>
<tr>
<td>SYSTEM CALLS (Using a NN)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>NAS-IS with the loadable kernel module in memory</td>
<td>31.424</td>
<td>3.235</td>
</tr>
<tr>
<td>Real-time monitoring of system calls for NAS-IS</td>
<td>32.265</td>
<td>4.105</td>
</tr>
</tbody>
</table>

Prepared using cpeauth.cls
5.6. Offline detection

Table VI shows the average time required to analyze one trace of LU-factorization with the nodemoncomputation attack (the experiments were executed 10 times). This particular trace contains 25,382 function calls and 34,897 system calls. The table shows a comparison of the 3 data models using the two streams. It is important to observe that the most time-efficient algorithm is the sequence matching, although as we have shown before, its accuracy is very low. Also, the amount of time needed to complete the detection with the artificial neural network is greater than the time needed for the HMM (for both function and system calls). With the current parameters, the neural network uses 32 hidden neurons, and therefore the number of computations needed to propagate the sequence of calls and obtain an answer is larger than the number of computations needed to find the sequence on a sorted tree (using the exact sequence matching) or to find an unlikely transition on a five-state HMM.
Table VI. Testing times for one instance of LU-factorization

<table>
<thead>
<tr>
<th>FUNCTION CALLS</th>
<th>SYSTEM CALLS</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
</tr>
<tr>
<td>Sequence Matching</td>
<td>0.0100 secs</td>
</tr>
<tr>
<td>HMM</td>
<td>0.1003 secs</td>
</tr>
<tr>
<td>NN (Max Consecutive)</td>
<td>0.5860 secs</td>
</tr>
</tbody>
</table>

6. Conclusions

We have described our effort to provide lightweight monitoring of parallel programs in real-time in order to automatically detect anomalies in the cluster system. Two main techniques were used to collect events from an MPI program: interposition library and loadable kernel module. Executing the class A NAS IS benchmark, the overhead of interposing the shared object needed for the collection of function calls is 3.42%, while the overhead with the kernel module loaded that is needed for collecting system calls is 2.53%.

We have demonstrated that both data streams can be used to verify the correct execution of an MPI parallel program in a cluster of Linux workstations. As a result of our experiments, we conclude that both the artificial neural network or the Hidden Markov Model result in high detection accuracy and a 0% false positive rate. This demonstrates that we were able to model the behavior of an MPI program. In contrast, the exact sequence matching algorithm generates many false positives. This is due to the fact that in order to recognize a sequence of calls as normal behavior, such a sequence needs to be stored in the database of known activity. Since many of the parallel applications include randomness, it is possible that the flow of the algorithm changes every time the program is executed, and therefore, we would require a huge database of normal activity to recognize all possible patterns.

We have also shown the efficiency of the detection algorithms when monitoring function calls is better than when monitoring system calls, because the total number of calls is much smaller than the number of calls generated from a operating system call trace. As an example, we have achieved a total overhead of 3.97% when the HMM is used to monitor function calls, of which just 0.55% corresponds to the detection algorithm itself. In contrast, system call analysis generates more overhead, but unexpected behavior is more easily detected.

We have conducted experiments with the standard C library, the mathematical library, and the MPI/Pro library, but other interesting experiments can be performed. For example, on a distributed system, the behavior of the X-window system can be monitored, or in a larger-scale system, the behavior of an intranet can be analyzed.

Furthermore, we believe that our framework can be suitable for on-demand computing, because the provider of the service can use our system to give the subscriber tangible proof of the correctness of the execution of the jobs in the cluster.
More importantly, our results challenge conventional thought that sequence matching is an appropriate technique for anomaly detection in complex systems. Our experiments have shown that HMM and NN approaches perform better with fewer false positives. We also find evidence that the anomaly detection approach of choice may possibly be strongly related to the type and purpose of the software being profiled. Further work is warranted in this regard.

7. Future Work

We are conducting experiments to test the capability of our detection systems to monitor a complete work session in the cluster, that is, monitor the behavior of possibly more than one application. For this work we also are planning to simulate new types of anomalies, such as operating system and network failures. Fault injection mechanisms need to be investigated.

A deeper investigation of scalability issues also needs to be conducted. In the proposed agent-based method, a central point is used to collect the results of the detection systems in each node and display the status of the entire cluster system. When the number of nodes in the cluster becomes large, this central host may become the bottleneck of our system. An effective approach may be to have agents report their status to other agents periodically. For example, when an agent detects abnormal behavior in some MPI program, the agent-based software could be used to react to the incident (each agent will kill the intrusive process, and send an alarm to other agents). Thus, we could make use of current failure detection systems, but we will be interested in not only exchange of “liveness” information, but also exchange of relevant program status information.

Although the focus of the work reported in this paper is on anomaly detection in dedicated high performance clusters, many of the issues addressed are also relevant to grid computing. In the grid environment, the user still writes applications using MPI and detection of anomalies at this level of abstraction appears to remain appropriate. In a grid computing environment, additional issues such as network connectivity must also be addressed for anomaly detection. One possibility is to interface host anomaly detection components like the ones described in this paper with tools for monitoring network activities on the grid such as Netlogger and NWS.

Finally, in our research we train the system in one node using the information collected from all the nodes in an off-line fashion. For more complex systems, it might be necessary to also train the system in real-time. Therefore, we believe that incremental learning algorithms, ensemble architectures or distributed data mining can be included in our framework.

ACKNOWLEDGEMENTS

We would like to thank the members of the Center for Computer Security Research and the High Performance Computer Laboratory for their support. Appreciation is also due to Dr. Yoginder Dandass for providing us with the source code of the LU-factorization method and Dr. Purushotham Bangalore for his comments concerning grid issues.
REFERENCES


