Fine–Grain Access Control for Securing Shared Resources in Computational Grids*

Ali Raza Butt† Sumalatha Adabala† Nirav H. Kapadia† Renato Figueiredo††
José A. B. Fortes‡

†School of ECE
Purdue University
W Lafayette, IN 47907
{butta, adabala, kapadia}@purdue.edu

††Dept. of ECE
Northwestern University
Evanston, IL 60208
renato@ece.nwu.edu

‡Dept. of ECE
University of Florida
Gainesville, FL 32611
fortes@ufl.edu

Abstract

Computational grids provide computing power by sharing resources across administrative domains. This sharing, coupled with the need to execute untrusted code from arbitrary users, introduces security hazards. This paper addresses the security implications of making a computing resource available to untrusted applications via computational grids. It highlights the problems and limitations of current grid environments and proposes a technique that employs runtime monitoring and a restricted shell. The technique can be used for setting–up an execution environment that supports the full legitimate use allowed by the security policy of a shared resource. Performance analysis shows up to 2.14 times execution overhead improvement for shell–based applications. The approach proves effective and provides a substrate for hybrid techniques that combine static and dynamic mechanisms to minimize monitoring overheads.

Key Phrases: access control, grid environments, grid security, Unix access model.

1. Introduction

Grid environments of the future will require an ability to provide a secure execution environment for applications. The Grid should allow arbitrary code from untrusted users to legitimately share resources, while providing an active enforcement of the security policy of the shared resources. This requirement, if not addressed, presents significant obstacles to the viability of computational grids, which typically span multiple administrative domains.

The ability to share resources is a fundamental concept for realization of grids; therefore, resource security and integrity are prime concerns. If the goal is to allow arbitrary users to submit applications to the grid, new dimensions to security issues will be introduced [2]. The issues of authorization and secure communications are addressed in great length in Globus [www.globus.org]. However, in this case, the fact that the users and resources no longer enjoy a mutual relationship complicates the problem. Two scenarios may arise, the shared resource may be malicious and affects the results of the program utilizing the resource, or the grid program may be malicious and threatens the integrity of the resource. Although the first scenario is a crucial issue [25], this paper focuses on the latter issue providing a step towards achieving better security.

A two–level approach consisting of a restricted shell and runtime monitoring is presented, to provide a secure execution environment for unmodified binaries. The second level provides active runtime monitoring to prevent any malicious use in programs. However, development environments generally require an interactive shell environment. If only runtime monitoring were to be employed, the shell itself will also have to be monitored therefore incurring extra overheads. To overcome these overheads, the first level of the approach consists of a restricted shell that is not monitored directly by the second level. The shell utilizes its own checks to control the user's ability to freely peruse resources during an interactive session. For example, the shell controls reads to otherwise world–readable re-

*This work was partially funded by the National Science Foundation under grants ECS–9809520, EIA–9872516, and EIA–9975275, by the Army Research Office Defense University Research Initiative in Nanotechnology, and by an academic reinvestment grant from Purdue University. Intel, Purdue, SRC, and HP have provided equipment grants for PUNCH compute–servers.

sources and access to arbitrary directories. The experimental measurements of execution time overheads for various shell–based applications show that the two-level technique is up to two times faster than runtime monitoring alone.

Traditional methods of controlling access to computing resources are based on choosing a principal [22], an entity responsible for usage and actions of the resources. In UNIX–based systems, this assignment is done by creation of an account for users. The process of assigning such user accounts serves two purposes: 1) it helps to ensure that users are responsible for their actions (for example, by obtaining personal and contact information for each user); and 2) it allows administrators to enforce usage policies (for example, by not giving out accounts on certain machines). The logistical overheads associated with manual account creation and maintenance become overwhelming when this approach is extended to grid environments. Dynamic and temporary usage, which will constitute a vast majority of grid users, may be precluded by these overheads. Another problem is that resource owners must implicitly rely on the accountability of the users of the resources, making it difficult and imprudent for resource owners to share resources outside their administrative domains. Another issue results from the inability of resource owners to control how users employ the resources. For example, resource owners may want some users to use only certain machines for specified applications, however given the conventional account paradigm, there is no easy way to enforce this.

The rest of the paper is organized as follows. Section 2 presents a description of the various requirements of grid applications and the premises that underlie this investigation. Section 3 describes the various approaches adopted by current grid systems and the shortcomings associated with providing a secure execution environment. Section 4 describes an online technique to implement access control that can support arbitrary code from untrusted users. Section 5 provides a brief discussion of other techniques that can be used to provide the necessary execution environment required for grid applications. Finally, Section 6 presents concluding remarks.

2. Background

Although this work is applicable to generic grid environments, the implementation observations were conducted in the context of Purdue University Network Computing Hubs (PUNCH) [16] that is a platform for grid computing that allows users to access and run unmodified applications on distributed resources via standard Web browsers (see www.punch.purdue.edu). As in typical grid environments, users in PUNCH are also transient and mostly utilize the system for specific projects. Usage policies associated with machines are complex and often change. The diversity of PUNCH users and applications has significant value in terms of validating research concepts and simulations. However, operating and supporting such a service in a research environment is impractical unless the administrative cost of maintaining security is kept under control.

In this context, there are two categories of grid applications. There are applications that are provided by the grid–service providers for use–only purposes. Users have very restricted access to these applications and cannot do much damage. However, a more challenging scenario is presented by research applications that are typically submitted by the users, and no safety guarantees can be provided for the behavior of these applications.

3. Grid Security

From a security standpoint, the users the applications, or the grid middleware — or some combination of the three — must be trusted. In dynamic, scalable, wide-area computing environments, it is generally impractical to expect that all users can be held accountable for their actions. Accountability comes after the damage has been done, making this a costly solution. Another option is to trust the applications. This is typically accomplished either by constraining the development environment to a point where the generated applications are guaranteed to be safe or by making sure that the applications come from a trusted source. However, limiting the functionality of applications also limits the usefulness of the computing environment. History has shown that it is too easy for applications from trusted sources to contain bugs that compromise the integrity of resources. Grid applications can be classified, as shown in Figure 1, based on the entity trusted for guaranteeing safety of grid applications, and the corresponding limitations imposed on allowed grid applications. At the origin of the plot, we have the ideal grid environment that allows the execution of arbitrary legal code from untrusted users while preventing illegal code from causing damage. Systems such as PBS [4], GLOBUS [10], and Sun Grid Engine [24] only allow accountable users. During the application generation process, safety checks can be implemented at the source code level (by using a safe language), at compile–time or at static link-time (as in Condor [18]). The space between the region where access checks are applied and execution time indicates the window of opportunity

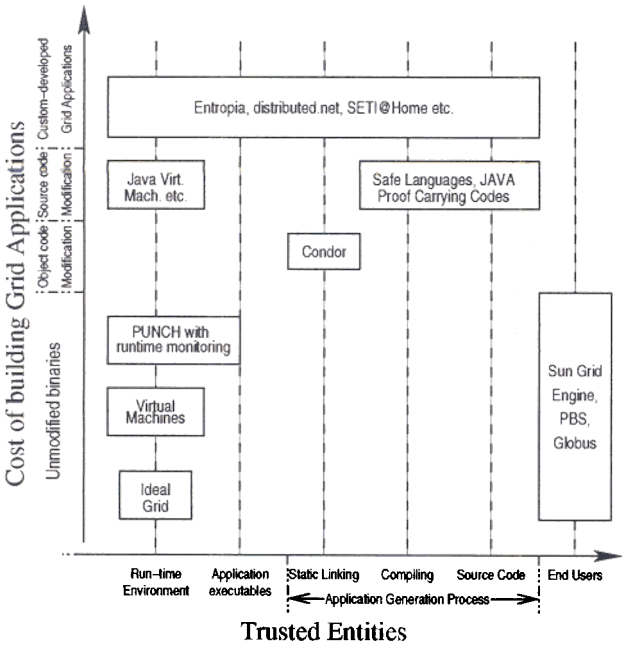

Figure 1. Classification of grid environments

Point of trust	Restrictions	Issues
Entire Process Examples: Entropia, Distributed.net, Seti@Home	Safe APIs Requires application source Trusted programmer, compiler, linker Human interaction	Overhead for adapting application to grid. Legacy binaries not supported. Problems with arbitrarily trusting a third-party.
Compile-time Examples: Static Compiler Analysis Proof Carrying Code (PCC) - proof synthesis	Analysis currently possible only for restricted subsets of languages. For PCC general verification is undecidable	Exponential binary code bloat(PCC). Overhead of analysis may not be justified. Application can be tampered with at a later stage. Legacy binaries not supported
Link-time Examples: Condor System-call wrappers Disallowing dynamic linking	Limit functionality	Application can be tampered with at a later stage. Legacy binaries not supported
Load-time Examples: Static analysis of machine code. PCC - proof verification	Works for restricted subsets of languages	Overhead of analysis may not be justified. May not protect against self-modification or stack/data execution

Table 1. Access control techniques adopted by grid environments, the corresponding restrictions on allowed code/applications and the associated issues

when an application can be tampered with to introduce malicious behavior. Grid environments, such as Distributed.Net [5], Entropia [6], and SETI@Home [23] control the entire application generation and deployment process. Though the grid users are not accountable, the grid-service providers implement or check the application on behalf of these users and are liable in their stead. The top rectangle covering many phases of application generation indicates the large amount of third-party software that needs to be trusted by the shared resources. Shadow accounts with runtime monitoring in PUNCH [15] and generic virtual machines, such as VMWare [26], can apply the access-control checks at runtime to support arbitrary code and untrusted users.

The future grid environments will be as close to the origin as possible in order to provide legitimate resource sharing to arbitrary unmodified binaries. Consequently security is best achieved by way of *active enforcement* of policies within grid middleware layers.

For an application to function properly, it should be provided with an execution environment on a remote host that grants the least-required privileges, based on the principle of least privilege [22]. However, the least required privilege of an application might require more accesses than a shared resource is willing to provide. In most cases, lack of fine grain control mechanism forces the grid-service providers to be conservative. Applications that can be run on a host with a given security policy are not permitted, only due to coarse grain of control. This limits the functionality of applications more than that could be supported on the host. The difference in what is allowed to an application and what is permitted by the host security policy is a function of the grain of control, and therefore can serve as quantifying criteria for a comparison of relative effectiveness of the various schemes.

3.1. Trusted Applications

Table 1 summarizes the techniques adopted by current grid environments for trusting applications, the corresponding restrictions on allowed code/applications, and the associated security issues.

The approach adopted by grid-service providers such as Distributed.Net [5], Entropia [6], and SETI@Home [23], is only to accept code from a trusted source and control the entire development process — generation to deployment — of a grid application. The code that actually runs on the target host is guaranteed

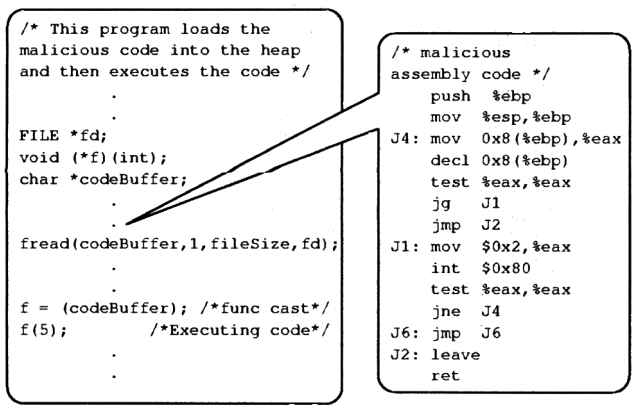

Figure 2. Sample grid application code that can invoke the malicious code at run time. It bypasses the system–call library and invokes `fork()` and `exec()` via the kernel.

to be safe by grid–service providers.

This typically implies that applications have to be rewritten and prepared for the grid, which in turn requires that the programmer, compiler, linker, and loader must all be trusted. Such incurred adaptation overhead may be large, hence limiting the rate at which new applications are deployed. Moreover, if a trusted source is the only protection for a host system, a breach on the source side can affect all resources. This places very stringent security requirements on the source.

The compiler has access to the source code and can perform static analysis to determine code that can possibly infringe on the host security policy. However, in the absence of runtime checks, the most a compiler can do is to either allow or disallow certain actions based on this analysis. The security enforced in this manner has the drawback that if a malicious piece of code is imported into the program, the security checks can easily be avoided.

A dynamic library that allows runtime modification of a process to act maliciously is described in [19], a security breach in context of Condor [18] is provided as an example. It also provides a list of probable methods for preventing such a breach, however, no actual prevention technique is described. In the following discussion, it is shown that the process behavior can also be modified in a simple way that does not require sophisticated libraries. Moreover, a solution that overcomes the limitations of the current approaches is also proposed in the next section.

Figure 2 shows a grid application code that does a stack-smashing attack [1] on its own stack. In this case, since the grid user owns the target process, invoking the malicious code is relatively easy compared to a general case where execution requires knowledge of loopholes in the target application. The goal of this code is to access a resource that is protected by the grid security enforcement. The left panel shows a snippet of C code that is safe except for the explicit cast of a data pointer to a function pointer (also a legal C operation). It loads a file into memory and executes it. The right panel shows a piece of malicious machine code that is unavailable at compile–time, but is injected into the execution stream. The machine code, in essence, executes the `fork()` and `exec()` system–calls using system trap interrupt.

These system–calls were chosen as representative of malicious behavior, because if arbitrary access is allowed to these system–calls, all operations allowed to a local user can be done on the machine executing the code. The malicious nature of the code is not detectable; even when compiled using special libraries that do not allow the said system–calls because of the absence of the malicious part at compile–time.

The static checks can be extended by incorporating proof–carrying codes [20], enforcing security checks at link–time, or at load–time. But due to the absence of any run–time checks, these approaches remain susceptible to the malicious code of Figure 2, because the code does not rely on specific libraries and calls the kernel directly at run–time. Such codes practically render these security checks insufficient. Finally, for techniques implemented via trusting applications, the source code requirement precludes legacy binaries.

4. Proposed Approach to Grid Security

The analysis of current approaches shows that the injection of malicious code into an otherwise secure application cannot be detected offline using static methods. In order to support arbitrary user–submitted applications, irrespective of the language and compiler used to produce them, an execution environment on the host machine is necessary to meet the security requirements. Runtime monitoring of the system–calls trace of an application can provide control over the arbitrary accesses of an application. However, development environments generally require an interactive shell environment. Runtime monitoring alone of all the applications, including the shell, incurs extra overheads. To avoid these overheads, a two–level approach is presented: level–one to handle interactive shell sessions, and level–two to handle arbitrary (user–submitted) applications.

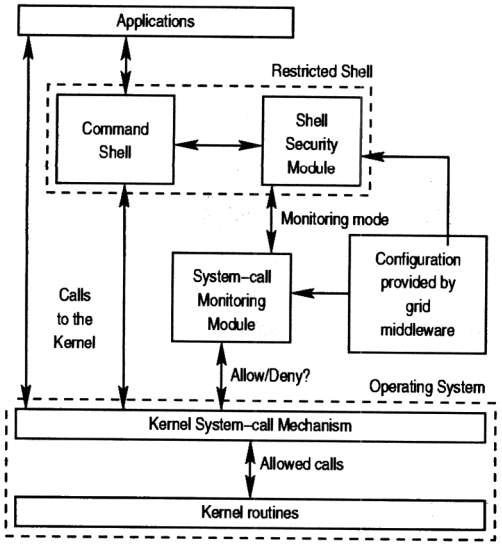

Figure 3. An illustration of modules and interactions in the two–level approach.

4.1. The Two–level Approach

Figure 3 shows the block diagram of the two–level approach, comprising of a restricted shell and a system–call monitoring module. The shell consists of a standard command shell augmented with a security module that actively checks the commands issued by the grid user. The module, via these checks, enforces the host security policy. The policy is specified in a configuration file containing a list of options, such as allowing executables from user–directory or directory change privilege to directories outside home directory hierarchy, and a template containing allowed executables, accessible files, allowed directories outside home directory etc. These constraints can be captured either explicitly by specifying a list or implicitly by using wildcards.

The file can be configured to be either an *allow list* (default behavior) or a *deny list*, where access to specified resources is either allowed or denied respectively. Grid middleware can configure the security module according to the needs and privileges of individual grid users by providing a proper configuration file. On initialization of each session, the module reads this file for options and then bases its decisions on them.

The restricted shell works by breaking down a user issued command into two parts, the actual executable name and a list of files that are required for completion of the command. A match of the executable name to the pre–configured specifications is then searched, followed by a similar matching for the list of required files. In case access to one or more of the required files (or the executable) is not specifically granted to the user, the command is not allowed to complete. Even for allowed commands, the set of checks implemented in level–two are not completely disabled. However, before actual execution of a permitted application, the shell informs level–two to set up proper monitoring mode, based on the configuration policy for the application. In case a shell script is executed, each line is separately parsed, and the shell checks can allow or deny the commands on per line bases.

For sandboxing the user under the control of the security module, the restricted shell is made the default shell, and execution of other shell binaries is prohibited. For instance, during an interactive session, the restricted shell is capable of locking the user in a pre–specified directory and preventing reads to arbitrary world–readable files by the remote users. When arbitrary or user–developed programs are executed, the shell can no longer control the accesses, as programs can access resources maliciously via direct calls to the kernel. This is also the case when scripts written in languages such as Perl are executed. Level–two controls all the applications spawned by the shell.

The heart of level–two is the process–tracing capabilities in modern UNIX/LINUX systems provided by the `ptrace` systems-call and the `/proc` file-system. This functionality allows a parent process to keep a check on its child process and modify the behavior of the child process [8]. There have been several attempts [3, 11, 14] to intercept system calls made by an application and modify the behavior to enforce host security policies. Janus [28] is also an example of such a technique. Once the system calls trace is obtained, the techniques discussed in [9, 17, 27] can be employed to ascertain whether the application is behaving in a malicious manner. The design of level–two is based on similar methods. However, the use of level–one serves two purposes. First, for shell–based applications it avoids the extra overheads of monitoring the shell itself. Secondly, the shell provides a mechanism to switch monitoring modes of level–two on per application basis during runtime. For example, an application provided by the system may be allowed to access certain directories, whereas an application provided by a user may not.

Grid middleware initially configures level–two with two or more sets of system–calls, those which should always be monitored for proper operation of the monitoring module, for instance `fork()`, and those specified in host security policy as malicious and should be monitored for all applications other than the restricted shell, for instance `exec()`, `socket()`. At startup, the monitor defaults to monitoring the first set of calls, switch-

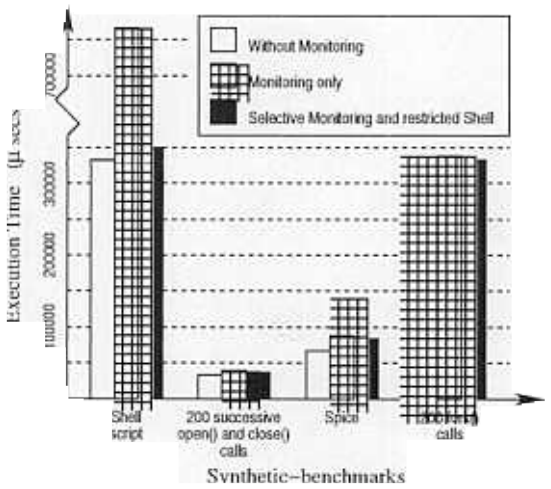

Figure 4. Performance results of the two–level approach, showing comparative execution times for various synthetic–benchmarks used for observations.

ing to another set of higher monitoring, only when an application is executed as indicated by the restricted shell. The monitor works by passively waiting until an application invokes a call that is not permitted. When such a call is invoked, the kernel system–call mechanism transfers control to the security module that can then analyze the call and inform the kernel of whether or not to allow the action to complete. In case the call is permitted, it is allowed to complete, and control is finally given back to the application. Otherwise, the application is returned an abort flag, or even terminated if it continues executing the same restricted system–call. If the `fork()` system–call is allowed, it is always monitored, and a companion monitoring process has to be spawned for every child spawned by an application. This is necessary for preventing an application from spawning unmonitored children that can create security hazards.

4.2. Performance Analysis

Two aspects of the approach are discussed, the effectiveness in preventing malicious program behavior, and the execution overhead for shell–based applications as compared to runtime monitoring only.

The two–level approach avoids the ill effects of user–contributed binaries discussed earlier. For example, when the code of Figure 2 is executed on a system configured to prevent the `fork()` and `exec()` calls, level–one signals level–two that the shell is no longer in control and executes the application. Level–two then starts monitoring calls. The approach proves effective by detecting the calls and terminating the malicious process. Attacks such as unauthorized information gathering, monitoring of other users, undermining local systems, and arbitrary communication to remote resources are prevented as they require the ability to freely access the world–readable resources; an activity that can easily be restricted by specifying the allowed resources and monitoring the system–calls to enforce the policy.

Figure 4 shows performance overheads of the two–level approach, observed utilizing a set of synthetic benchmarks consisting of a program with repeated `fork()` calls, a program with successive pairs of `open()` and `close()` calls, a typical shell session emulation, and a typical run of Spice. Execution time for each set of benchmark was observed for three cases: without any monitoring, with monitoring alone, and with the proposed two–level scheme. The overhead is computed as a ratio of the execution time of two monitoring methods to the normal execution time without monitoring.

Since the `fork()` system–call has to be intercepted for each occurrence, the first set of observations shows the effects of monitoring on a program that is composed of 200 `fork()` calls. The purpose of this run is to determine a worst case bound on runtime monitoring overhead. From 1000 observations, an average overhead of 2.59 times was observed for proposed approach as compared to 2.64 times for full monitoring. The more than twice overhead is due to an extra `fork()` call made by the monitoring module for each `fork()` call executed by the program, as well as the processing time of the system–call monitoring module to determine what actions to take on specific system–calls. In typical applications the frequency of `fork()` calls is much smaller than other system–calls. The overhead of a `fork()` call has a small effect on the overall performance of the application. Moreover, the difference in the overheads of full–monitoring and the proposed scheme is due to the fact that runtime monitoring alone monitors the shell as well, where as the proposed scheme does not.

Another set of results consists of 1000 observations, obtained for 200 pairs of successive `open()` and `close()` calls on a file. Here the purpose is to determine the effect of monitoring on innocuous calls. In this scenario, because the calls are not being intercepted, the average overhead is 1.01 for both runtime monitoring and the proposed scheme. For this scenario, file caching may affect the time taken by the `open()` and `close()` calls. The relative effect remains the same on both sets of observations. This run shows that runtime monitoring alone or with in the proposed scheme in-

cur negligible overheads for unmonitored calls. Hence, if the shell can determine and change modes of monitoring on per application basis, the overall effect of monitoring can be minimized.

Next is a shell script that emulates an interactive shell. It was executed 1000 times and measurements were taken for each case of no monitoring, runtime monitoring with a standard shell, and under the proposed scheme. The overhead of monitoring on malicious actions that can be determined via the shell is evident from the figure, 1.07 times overhead due to the proposed scheme as compared to the 2.29 times overhead of monitoring all system calls – a improvement of 2.14 times. This improvement is due to the fact that the shell employs `exec()` and `fork()` system calls for execution of each command in the script. In case of a standard shell, these calls are monitored by level–two and as measured earlier incur large overheads, where as in the restricted shell they are not, hence the gain in performance.

Another typical application on PUNCH is PROPHET, a simulator for solving systems of partial differential equations. It has a shell interface, and hence pose a problem when arbitrary users are allowed to access the shell. However, when the restricted shell replaces the default shell, and a policy of minimum system calls is employed, the risk of an arbitrary user freely accessing local resources is eliminated. In a typical run, negligible security overhead is observed, partially because the secure shell replaces the default shell and has similar runtimes, and partially due to the fact that malicious calls do not occur in typical runs, and therefore, the monitoring module remains dormant.

In the last set of observations, Spice, a commonly used application, is executed to plot voltage/current characteristics of a Nanoscale MOSFET (a typical application on PUNCH). As Spice does not make any malicious calls in its legal execution, the execution times with or without system–call monitoring, and with the proposed scheme are almost the same with only 1.06 times overhead. As Spice is not necessarily a shell–based application, the proposed scheme behaves similarly to runtime monitoring alone.

The results show that intercepting system–calls introduce a significant overhead, and there is a need for minimizing the number of calls that are intercepted. Static schemes as discussed in [21, 29] have the capability to determine unsafe portions of code by extended program analysis. Hence, some of these schemes can be leveraged in the present setting for minimizing the overhead of runtime checking, by selecting different monitoring modes for different portions of the program. The restricted shell of level–one provides a step towards this goal. Such schemes are currently being investigated to determine whether they can be employed efficiently in the grid environment.

5. Discussion

It is clear from the presented scenarios that the UNIX kernel is not designed for the tasks that are required in the execution of remote code in grid environments. The proposed solution has relied on user–mode techniques in order to avoid modifications to the kernel. Redesigning the required portions of the kernel may provide more efficient solutions to the investigated issues.

Finer–grain access–control in the form of access–control lists is evolving in the Unix kernel that allows a finer–grain user access–model. The access–control list approach is not suitable to the presented scenario because it modifies properties associated with the resource rather than the user [7]. The functionality required for setting–up an execution environment for grid applications is a means for a user to specify the list of resources to which access is allowed or denied on per–process-basis that, in essence, leads to a capability model.

Another approach for addressing the described security issues is the use of a virtual machine, such as the sandbox of Java Platform [12]. For grid environments, the need to write Java code, or port applications to Java platform can be eliminated by a virtual machine that is decoupled from applications. Examples of application–independent virtual machines include IBM's Virtual Image Facility [13] and VMware [26]. These systems support sandboxing at the level of operating systems and can provide a substrate for executing arbitrary untrusted code without compromising the host machine security.

6. Conclusions

The paper shows that the execution environment required for allowing user-developed arbitrary binaries to run on shared resources entails a more constrained view of the system than provided by the traditional Unix environment. The access–control model is not enough to handle the security hazards introduced by shared resource usage in dynamic, large–scale grid environments. The proposed approach is to employ runtime monitoring and process–tracing that allows only permitted actions. In an attempt to minimize the runtime monitoring overhead, a secure restricted shell is

utilized that avoids the process–tracing overhead of the shell environment itself. The proposed two–level approach provides a reduction of up to 2.14 times in execution–time overheads for shell–based applications as compared to monitoring alone. In the future, hybrid techniques to minimize the number of system–calls monitored, as well as virtual machines can also provide an appropriate execution environment. Virtual machines that can efficiently provide desired environments are a nascent technology and may become key players in the long run.

References

[1] N. S. A. Baratloo and T. Tsai. Transparent run-time defense against stack smashing attacks. In *Proceedings of the USENIX Annual Technical Conference*, pages 207–233 June 2000.

[2] S. Adabala, A. R. Butt, R. J. Figueiredo, N. H. Kapadia, and J. A. B. Fortes. Security implications of making computing resources available via computational grids. Technical Report TR-ECE01-2, Purdue University, West Lafayette, IN, September 2001.

[3] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Ufo: A personal global file system based on user-level extensions to the operating system. volume 16, pages 207–233, August 1998.

[4] A. Bayucan, R. L. Henderson, C. Lesiak, B. Mann, T. Proett, and D. Tweten. Portable batch system: External reference specification. Technical report, MRJ Technology Solutions, November 1999.

[5] Distributed Computing Technologies Inc. http://www.distributed.net

[6] Entropia Inc. http://www.entropia.com

[7] EROS-OS. Comparing ACLs and Capabilities http://www.eros-os.org/essays/aclsvcaps.html

[8] S. E. Fagan. Tracing bsd system calls. *Dr. Dobb's Journal*, pages 38–43, 03 1998.

[9] S. Forrest, S. A. Hofmeyr, A. Somayaji, and T. A. Longstaff. A sense of self for Unix processes. In *Proceedinges of the 1996 IEEE Symposium on Research in Security and Privacy*, pages 120–128, 1996.

[10] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(2):115–128, Summer 1997.

[11] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the 6th Usenix Security Symposium*, San Jose, Ca., 1996.

[12] L. Gong, M. Mueller, H. Prafullchandra, and R. Schemers. Going beyond the sandbox: An overview of the new security architecture in the java development kit 1.2. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[13] IBM Corporation. White Paper: S/390 Virtual Image Facility for Linux, Guide and Reference. GC24-5930-03, February 2001.

[14] M. B. Jones. Interposition agents: Transparently interposing user code at the system interface. In *Symposium on Operating Systems Principles*, pages 80–93, 1993.

[15] N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Enhancing the scalability and usability of computational grids via logical user accounts and virtual file systems. In *Heterogeneous Computing Workshop at IPDPS*, April 2001.

[16] N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Punch: Web portal for running tools. In *IEEE Micro*, May–June 2000.

[17] C. Ko, M. Ruschitzka, and K. Levitt. Execution monitoring of security-critical programs in a distributed system: A specification-based approach. In *Proceedings of the 1997 IEEE Symposium on Security and Privacy, Oakland, California*, 1997.

[18] M. J. M. J. Litzkow, M. Livny, and M. W. Mutka. Condor - A hunter of idle workstations. In *Proceedings of the 8th International Conference on Distributed Computing Systems (ICDCS)*, pages 104–111, Washington, DC, 1988.

[19] B. P. Miller, M. Christodorescu, R. Iverson, T. Kosar, A. Mirgorodskii, and F. Popovici. Playing inside the black box: Using dynamic instrumentation to create security holes. *Parallel Processing Letters*, 11(2,3):267–280, 2001.

[20] G. C. Necula and P. Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, 1996.

[21] H. Patil and C. N. Fischer. Efficient run-time monitoring using shadow processing. In *Automated and Algorithmic Debugging*, pages 119–132, 1995.

[22] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[23] SETI Institute Online http://www.setiinst.edu/science/setiathome.html

[24] Sun (tm) Microsystems Sun Grid Engine http://www.sun.ca/software/gridware/

[25] G. Vigna, editor. *Mobile Agents and Security*, volume 1419 of *LNCS*. Springer-Verlag, June 1998.

[26] VMware Incorporated. VMware GSX Server http://www.vmware.com, 2000.

[27] D. Wagner and D. Dean. Intrusion detection via static analysis. In *Proceedings of the 2001 IEEE Symposium on Security and Privacy*, May 2001.

[28] D. A. Wagner. Janus: an approach for confinement of untrusted applications. Technical Report CSD-99-1056, 12, 1999.

[29] R. Wahbe, S. Lucco, T. E. Anderson, and S. L. Graham. Software fault isolation. In *Fourteenth ACM Symposium on Operating System Principles*, volume 27, pages 203–216, December 1993.