Security Implications of Making Computing Resources Available via Computational Grids[1]

Sumalatha Adabala

Ali Raza Butt

Renato J. Figueiredo

Nirav H. Kapadia

José A. B. Fortes

School of Electrical and Computer Engineering
1285 Electrical Engineering Building
Purdue University
West Lafayette, IN 47907-1285

[1]This work was partially funded by the National Science Foundation under grants EEC-9700762, ECS-9809520, EIA-9872516, and EIA-9975275, and by an academic reinvestment grant from Purdue University. Intel, Purdue, SRC, and HP have provided equipment grants for PUNCH compute-servers.

Contents

List of Tables

List of Figures

Abstract

This report investigates the issues of securing access to computing resources in computational grids. Grid environments are built on top of platforms that control access to resources within a single administrative domain, at the granularity of a user. In wide-area multi-domain grid environments, the overhead of maintaining user accounts is prohibitive, and securing access to resources via user accountability is impractical. Typically, these issues are handled by implementing checks that guarantee the safety of applications, so that they can run in shared user accounts. This work shows that safety checks — language-based, compile-time, link-time, load-time — currently implemented in most grid environments are either inadequate or limit allowed grid users and applications. Techniques without such limitations are presented. Shadow accounts allow reuse of user accounts without administrative overheads, and run-time solutions — run-time monitoring and virtual machines — allow arbitrary code to execute while enforcing a given resource access policy.

Key Phrases: security in grid environments; safety of grid applications; access control of shared resources; sharing user accounts; shadow accounts; run-time application sandboxing; virtual machines.

1. Introduction

Access to computing resources has traditionally been controlled by assigning an "account" to each user. The process of assigning such user accounts serves two purposes: 1) it helps establish the *accountability* of users — by obtaining personal and contact information for each user, for example; and 2) it allows administrators to enforce usage policies — by not giving out accounts on certain machines, for example.

This approach has worked well within single administrative domains, but it has several limitations that present significant obstacles to the viability of computational grids, which typically span multiple administrative domains. For example, creating accounts has an administrative overhead, making it difficult to give users dynamic or temporary (in terms of minutes or hours) access to resources. Another problem is that resource owners must implicitly rely on the accountability of the users of the resources, making it difficult and imprudent for them to share resources outside of their administrative domains. Yet another issue results from the inability of resource owners to control how the resource is utilized by its users. For example, a resource owner may want some users to only use certain machines for specified applications — but has no easy way to enforce this.

This investigation addresses the security implications of making a computing resource available via computational grids. Both grid applications and shared resources must be secured from malicious actions of each other. This work focuses on the issue of protecting the shared resources from arbitrary code. It highlights the problems and limitations of current grid environments, and proposes solutions that are more robust, scalable, and secure.

The rest of the report is organized as follows. Section 2 describes the context for this work,

the PUNCH network computing environment. Section 3 outlines some of the limitations of the current approaches to sharing computing resources across administrative domains. Section 4 proposes new solutions that can be used to overcome these limitations. Finally, Section 5 presents concluding remarks.

2. Background

This work was conducted in the context of PUNCH, the Purdue University Network Computing Hubs [9]. PUNCH 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). PUNCH currently provides computing services to about 2,000 students and researchers across two dozen countries. More than 70 applications from different research institutions and vendors are available.

Operating a computing portal with a world-wide user base presents some rather interesting issues. For example, PUNCH provides access to several commercial tools; access to these tools must be restricted to Purdue students due to licensing constraints. Applications range from batch (e.g., CacheSim5 — a cache simulator) to interactive (e.g., DLX-View — a pipeline simulator) to development environments (e.g., the SimpleScalar set of tools for computer architecture simulation). Users are transient — students tend to use PUNCH a semester (or quarter) at a time, and researchers utilize the system for specific projects. Usage policies associated with machines are complex and often change — for example, many machines are only available for specific types of applications (e.g., ones that tend to run quickly), and non-Purdue users typically can only use them when they are not heavily loaded.

The diversity of PUNCH users and applications has significant value in terms of validating research concepts. However, operating and supporting such a service in a research environment is impractical unless the administrative costs can be kept under control. This work highlights the mechanisms in PUNCH that make it possible to streamline and automate many of the tasks associated with granting users access to computing systems without

violating usage policies or compromising on security.

PUNCH users can request user accounts, these are logical accounts decoupled from machine accounts (Section 4.1), via its portal interface. These requests are processed automatically, and users are given access according to a default policy. Users that request additional privileges (e.g., Purdue students wanting access to commercial tools) are granted access after manual verification by an administrator. Changes are automatically propagated when machines are added or removed, or when usage policies change.

3. Securing Access to Grid Resources from Malicious Use

From a security standpoint, the *domain of trust* — the set of entities that are trusted or accountable — must lie with either the users, the applications, or the grid middleware (or some combination of the three). In dynamic, wide-area computing environments, it is generally impractical to expect that all users can be held accountable for their actions. Plus, accountability does not prevent damage from being 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, and history has shown that it is possible for applications from trusted sources to contain bugs (www.bugnet.com) that compromise their integrity.

Consequently, security is best achieved by *active enforcement* of policies within the grid middleware layers. The following discussion surveys the different approaches utilized within current grid environments and describes their limitations; the subsequent section describes new solutions that overcome some of the limitations of the current approaches.

3.1. User-based domain of trust

In systems that rely on end-user accountability, the *principal* [16] - the entity responsible for actions of a process - is the user account identifier. The process of obtaining user accounts is independent of the actual grid middleware, and is typically defined by the

owners of the computing resources. Examples in this category include Globus [4], Sun Grid Engine [17], and PBS [2].

This approach is not scalable as the administrative overhead of creation/maintenance, due to (a) the large number of accounts and (b) transient users requiring the account for short periods of time, may be inhibiting. Moreover, if a user cannot be made accountable, she/he cannot be given an account. This limits the potential users of the grid resources.

The grid process has access to all the resources of a standard UNIX user; as a consequence, if the grid user account is compromised, all other systems in the Grid on which that user has accounts are opened up for misuse. This makes proper implementation of standard Unix security on all the systems underlying the Grid more crucial.

In order to overcome the overhead of creating individual accounts, some systems rely on sharing accounts among grid users. Examples include Entropia (www.entropia.com), Distributed.Net (www.distributed.net) and United Devices (www.ud.com), where resource owners download grid applications and run them in their accounts. There are two problems with this approach. Firstly, though the grid processes may belong to different grid users, as far as the UNIX system is concerned they belong to the same user. So a grid process can misbehave and affect the other processes in the same account, for e.g., a malicious process can terminate other processes owned by the user (using 'kill -9 -1'). Secondly, since the shared account is a standard account (even user "nobody" [11] is a standard user though without a file system), it has access to local resources, and can exploit them at least to the extent allowed by the underlying UNIX access model. In addition, the grid user can misuse resources, for example, to launch distributed denial-of-service attacks.

3.2. Application-based domain of trust

One way of working around the need for end-user accountability is to ensure that applications executed in grid environments are "safe". This "safety" can be achieved in one of two ways: by constraining the application development environment, or by trusting the source of the application. Systems that rely on an application-based domain of trust typically run jobs from all users in a single, *shared account*. Examples in this category that constrain the

Point of trust	Examples	Restrictions	Issues
Entire application generation process	Entropia, Distributed.net, SETI@Home	Safe APIs; Requires application source; Trusted programmer, compiler, linker; Human intervention	High overhead of adapting application to grid; Unmodified binaries not supported
Compile-time	Static compiler analysis; proof carrying code (PCC) - proof synthesis	Analysis currently possible only for restricted subsets of languages; For PCC general verification is un-decidable	Exponential binary code bloat (PCC); Overhead of analysis may not be justified; Application can be tampered with at a later stage; Unmodified binaries not supported
Link-time	Condor; System-call wrapper approaches	Limited functionality; No dynamic linking	Application can be tampered with at a later stage; Unmodified binaries not supported
Load-time	Static analysis of machine code; PCC - proof verification	Works only for restricted subsets of languages	Overhead of analysis may not be justified; May not protect against self-modification or stack/heap execution

Table 3.1. Examples of systems that rely on application-based domain of trust, and the corresponding restrictions and limitations.

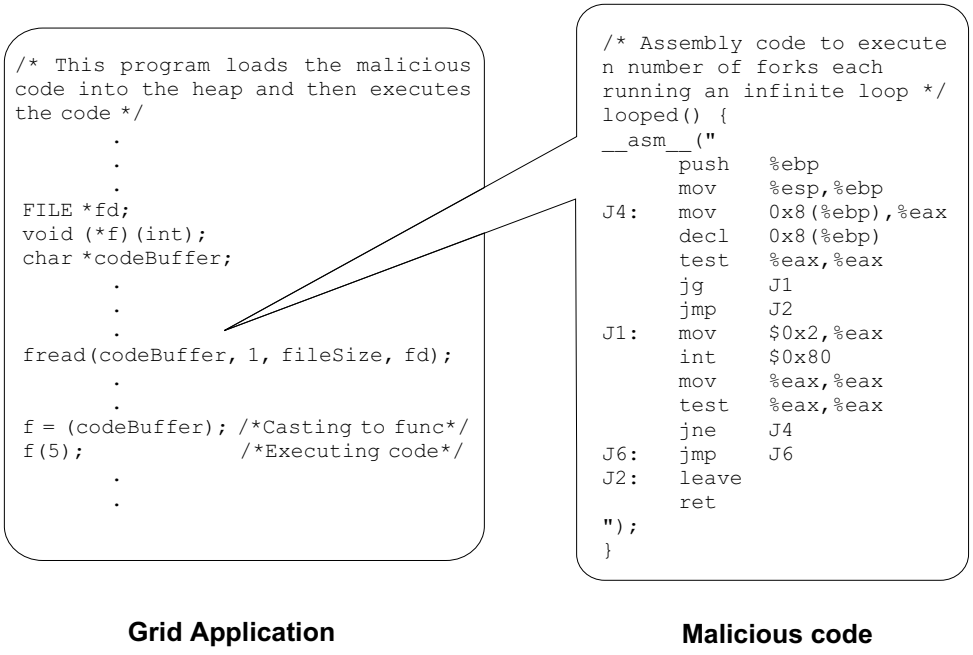

Figure 3.1. Sample grid application code that can invoke malicious code at run time. The malicious code bypasses the system call library and invokes fork() and exec() via the kernel.

application development environment include Condor [11], safe languages such as Java [6], and proof-carrying codes [12]. Examples of systems that rely on third-party trust (for the safety of applications) include Entropia, Distributed.net and United Devices.

There are several problems with these approaches (see Table 1). Entities such as Entropia protect computing resources by carefully controlling the entire process of building applications. This method, while feasible, is clearly expensive and time-consuming — particularly for large, complex applications. The use of safe languages such as Java can make this approach less complicated, but this excludes the use of unmodified binaries. Compile-time or link-time tests that attempt to verify the safety of applications are also possible (e.g., as with Condor), but such tests can be defeated.

In general, if users are allowed to modify code in any way (i.e., at the source, object, or binary levels), it is relatively simple for them to introduce trojan processes in the shared account — without being detected by any compile- or link-time tests. Consider the following example (see Figure 3.1). The grid application shown in the left half of the figure reads

a file into heap memory as a string. The string (in this case) contains machine code that invokes a `fork()` and `exec()`. Now, the grid application sets a pointer to the starting address of the string and executes it as a function (see the left half of the figure). This is possible because the stack and data segments in Unix are executable.[1]

These types of "attacks" are very difficult to detect at compile or link time. Even if one could detect and disallow explicit casting of function pointers, other approaches exist: for example, one could simply modify the executable (manually) after it has been compiled or import the malicious code into the execution stream at run-time. Also, it is not necessary to rely on the executable nature of the stack and data segments [18].

3.3. Discussion

As shown in Figure 3.2, grid environments can be classified on the basis of their domain of trust. Towards the left bottom of the figure are the ideal grid environments: they allow untrusted users to develop and execute arbitrary applications, while enforcing usage and security policies.

The X-axis represents the domain of trust. On the left, the run-time environment transparently handles trust issues; as one moves to the right, the domain of trust moves to the application, the application generation process, and finally to the end user. Observe that, in general, as one moves to the right, administrative and/or customization overheads associated with defining and enforcing security increase. The Y-axis represents the cost of building a grid application. At the bottom, the cost is negligible — grid environments in this space transparently support unmodified binaries. At the other extreme, grid environments require custom-designed applications for security.

To better understand the classification scheme, consider the following examples with respect to their positions in the figure. Systems such as Condor lie at the center of the figure — they only require applications to be relinked with special-purpose libraries. On the other hand, with languages such as Java, one would have to rewrite the application

[1] This is a fundamental characteristic of Unix and cannot be disabled without other side effects that limit functionality.

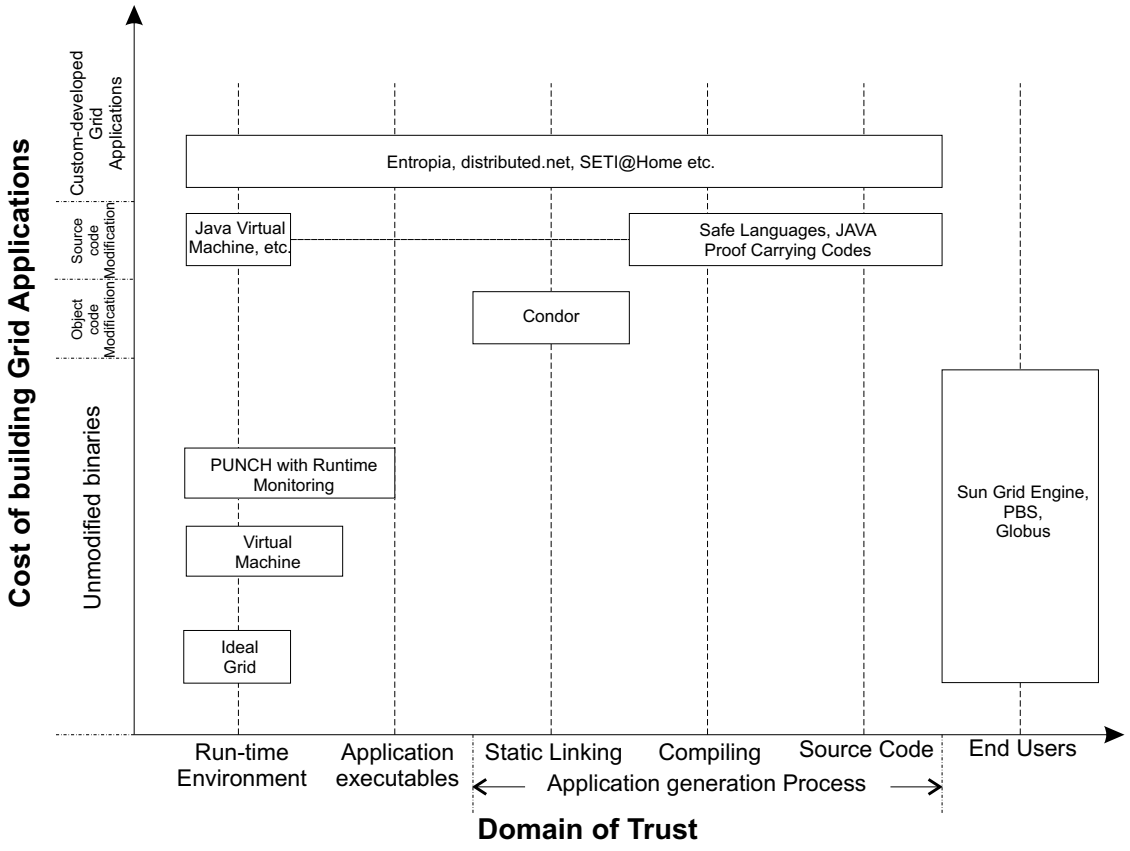

Figure 3.2. Classification of Grid Environments

completely — thus putting such approaches in the top right part of the figure. Finally, systems such as Globus and PBS require potential users to establish accountability directly with the system administrators, but allow them to run unmodified binaries — thus placing them in the right part of the classification space.

Observe that the region between the domain of trust and the run-time environment, along the X-axis, presents opportunities for intrusion, if security is not actively enforced at run-time. For example, in the case of Condor, link-time checks could be bypassed after the linking process is complete, but before the executable is submitted to Condor. Systems such as Entropia work around this problem by having a large domain of trust (by controlling application generation through execution). However, this is a manual and time-consuming approach that significantly increases the overhead (and cost) of building grid applications.

The following section describes solutions based on a combination of private, anonymous

accounts and either 1) run-time monitoring or 2) virtual machines. Such solutions lie towards the bottom left of the classification space (see figure) and can facilitate secure sharing of computing resources without compromising security.

4. Securing Access to Grid Resources

In this section, we present techniques to overcome the grid security shortcomings presented in Section 3. Section 4.1 describes the method used in PUNCH to overcome the administrative overhead of maintaining user accounts on grid resources. Sections 4.2 and 4.3 describe two approaches, currently being evaluated in PUNCH, to guarantee safe grid applications in the absence of user-accountability. The domain of trust in both these approaches — based on process monitoring and virtual machines — is in the run-time environment provided by the grid middleware.

4.1. Private, anonymous accounts

A standard user account provides the following functions: (a) allows data storage, (b) maintains privacy of data, (c) controls access to computing resources, and (d) ensures accountability. On standard systems a user account is associated with a unique numeric identifier (e.g. UNIX uid), which is permanently assigned to the user. Creating individual user accounts on all machines has administrative overheads because a permanent identifier must be created at the host system level for every user - temporary or permanent. However, the functions provided by the user account need not be tied to the unique identifier all the time, provided a separate mechanism is available to ensure data integrity and privacy. PUNCH maintains grid user data in *logical user accounts* [8] which are not tied to specific numeric identifiers. It also maintains a set of physical user accounts referred to as "shadow accounts". PUNCH treats the numeric identifiers of these shadow accounts as interchange-

able entities that can be recycled among grid users on demand. Hence, the logical accounts serve as capabilities to "check out" shadow accounts on the appropriate resources.

There are two advantages to this approach.

- Recycling a small number of shadow accounts among the grid users removes the problems associated with creation/maintenance of physical user accounts on the host machines in the grid.
- Since the physical accounts on the hosts are time multiplexed among the grid users, there is no need for grid processes belonging to different users to share accounts — thus implicitly providing protection between grid processes of different users on a host.

PUNCH has employed shadow accounts since early 1998.

4.2. Run-time Process Monitoring

Arbitrary applications can be sandboxed at execution time, to enforce host security policies [1, 7, 5], by monitoring the system call trace of the programs and selectively permitting or denying access to resources as specified by a policy file. This is enabled by process tracing capabilities, the `ptrace()` system-call and the `/proc` file-system, in modern UNIX systems. Using `ptrace()`, a parent process can monitor its child process, intercept system-calls, and use the `/proc` interface to modify the child's run-time environment, and thereby grant or deny access to resources at a very fine granularity. Run-time monitoring along with shadow accounts described in Section 4.1 provides a security solution that is close to the ideal grid in Figure 3.2.

When the entire application is monitored, the cost of shifting back and forth between kernel and user mode at least twice and the context switches between the application and the monitoring process for each intercepted system call can be significant. Hybrid approaches that use static analyses of source code at compile time or of machine code at load time, to determine unsafe portions of code and limit monitoring to these portions, can reduce the overhead of run-time monitoring.

When the application source is available, the compiler can perform static analyses of the untrusted program and find portions of code that cannot be guaranteed to be safe (e.g., pointer casting, memory de-referencing, etc.). Only these unsafe portions of code need to be monitored at run-time, for example using the shadow processing approach [13]. With this approach, program-slicing techniques are used to generate a customized program by deleting computations not relevant to the monitoring. The original application is run in lock-step with the customized program, the shadow process. Run-time monitoring is applied only to the shadow process, and if it fails any check, the original application is terminated. Overhead from the shadow processing can be hidden, by using idle processors in multiprocessor workstations. This approach can also be used with unmodified machine code, using the static analyses presented in [21]. This approach of run-time monitoring in shadow-accounts is being tested internally in PUNCH.

4.3. Run-time Virtual Machines

The Java Virtual Machine (JVM) [10] implements a security manager, which uses signature verification and application sandboxing, to ensure that byte code from an untrusted source cannot cause damage to the local resources of the host. There are security issues with Java implementation as discussed in [3], but even with these problems fixed, the Java approach implies grid applications are restricted to Java, and applications developed in other languages must be ported to the Java platform. Moreover, current Java applications rely on the Java Native Interface, written in languages like C, for local I/O etc, making them susceptible to the types of attacks implied by Figure 3.1.

A virtual machine that is decoupled from applications can also be conceived. Examples of application-independent virtual machines for UNIX-based systems include IBM's Virtual Image Facility [15] and VMware [19]. These systems support sandboxing at the level of operating systems: multiple copies of a "guest" O/S (a common case to both approaches is LINUX) may co-exist at run-time, sharing a machine's hardware through a "host" O/S (e.g. LINUX or VM/ESA).

Application-independent virtual machines provide a substrate for executing arbitrary

un-trusted code without compromising the host machine security. Furthermore, un-trusted users can be supported in a secure manner via a single-user assignment to guest O/S sessions. As long as the interface between the host and guest operating systems is secure, untrusted users can execute arbitrary applications without compromising either the host or any of the other guest operating systems (i.e. those belonging to other users).

While virtual machines support a secure environment for supporting un-trusted users and applications, there are performance and portability issues that hinder their application to existing systems. Today's virtual machines are customized to specific hardware platforms (e.g. IBM S/390 and Intel x86) and incur run-time overheads for O/S requests. Specialized support for virtual machines can mitigate run-time overheads, but may require hardware assistance not available in commodity processors (e.g. multiple levels of address translation in the S/390 [20]).

Future work will investigate resource management mechanisms that will allow virtual machines to be allocated efficiently on-demand and user-transparently, as computing units for PUNCH.

5. Conclusions and Outlook

The issues with securing access to shared resources in grid environments are the result of a mismatch between the characteristics of grid users and traditional users of the computing resources that underly the grid. The access mechanisms of the underlying computing platforms, which are at the granularity of a user, are thus not restrictive enough in grid enviroments. The limitations of techniques currently used to overcome this issue are presented. Solutions that can be implemented in the grid middleware are described — shadow accounts to overcome the administrative overhead of maintaining user accounts, and techniques that secure access to grid resources, such as run-time monitoring and virtual machines. Given the wide-scale deployment of grid environments, a long-term solution to the issues presented may involve modifying the underlying operating systems on computing resource, to support grid users, for e.g. a capability-based access model to resources.

Acknowledgements

This work was partially funded by the National Science Foundation under grants EEC-9700762, ECS-9809520, EIA-9872516, and EIA-9975275, and by an academic reinvestment grant from Purdue University. Intel, Purdue, SRC, and HP have provided equipment grants for PUNCH compute-servers.

Bibliography

[1] 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. ACM Transactions on Computer Systems. 16(3), pages 207–233, August 1998.

[2] 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.

[3] D. Dean, E. W. Felten, and D. S. Wallach. Java security: from HotJava to Netscape and beyond. In IEEE, editor, *1996 IEEE Symposium on Security and Privacy: May 6–8, 1996, Oakland, California*, pages 190–200, 1996.

[4] 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.

[5] 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.

[6] J. Gosling, and H. McGilton. *The Java Language Environment.* Sun Microsystems Computer Company, 1995.

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

[8] 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 *Proceedings of the Heterogeneous Computing Workshop (HCW) at the International Parallel and Distributed Processing Symposium(IPDPS)*, San Francisco, California, April 2001.

[9] N. H. Kapadia, R. J. Figueiredo, and J. A. B. Fortes. Punch: Web portal for running tools. *IEEE Micro*, pages 38–47, May–June 2000.

[10] T. Lindholm and F. Yellin. The Java Virtual Machine Specification. Addison-Wesley, Reading, MA, 1997.

[11] 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. IEEE Computer Society.

[12] G. C. Necula. Proof-Carrying Code. Conference Record of POPL '97: The 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, pages 106-119, Paris, France, 1997.

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

[14] C. Price. Mips IV instruction set, revision 3.1. Technical report, January 1995.

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

[16] J. Saltzer and M. Schroeder. The Protection of Information in Computer Systems. *Proceedings of the IEEE*, 63(9):1278–1308, 1975.

[17] Sun^{TM} Grid Engine Software Detailed View. http://www.sun.com/software/gridware/details.html

[18] L. Torvalds. Posting to linux kernel mailing list http://www.lwn.net/980806/a/linus-noexec.html. Technical report, August 1998.

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

[20] C. F. Webb. S/390 microprocessor design. In *IBM Journal of Research and Development*, pages 899–906, December 2000.

[21] Z. Xu, B. P. Miller, and T. W. Reps. Safety checking of machine code. In *SIGPLAN Conference on Programming Language Design and Implementation*, pages 70–82, 2000.