Identity Boxing: A New Technique for Consistent Global Identity

Douglas Thain
University of Notre Dame
Department of Computer Science and Engineering

ABSTRACT

Today, users of the grid may easily authenticate themselves to computing resources around the world using a public key security infrastructure. However, users are forced to employ a patchwork of local identities, each assigned by a different local authority. This forces each grid system to provide a mapping from global to local identities, creating a significant administrative burden and inhibiting many possibilities of data sharing. To remedy this, we introduce the technique of identity boxing. This technique allows a high-level identity to be attached directly to each process and resource that a user employs, rendering the local account name irrelevant. This allows a grid user to be known by the same name consistently at all sites, thus reducing administrative burdens and enabling new forms of sharing. We have implemented identity boxing at the user level within a secure system-call interposition agent and applied it to a distributed storage and execution system. The performance overhead of this implementation is only 0.7 to 6.5 percent for a selection of scientific applications, but as high as 35 percent for a metadata-intensive software build. We conclude with some reflections on how the operating system might be modified to better support grid computing.

1. INTRODUCTION

Today, the GSI public key security infrastructure allows grid users to be identified with strong cryptographic credentials and and a descriptive, globally-unique name such as /O=UnivNowhere/CN=Fred. This powerful security infrastructure allows users to perform a single login and then access a variety of remote resources on the grid without further authentication steps [17].

However, once connected to a specific system, a user's grid credentials must somehow be mapped to a local namespace. There are a variety of techniques for performing this mapping. Systems today employ untrusted accounts, private accounts, group accounts, anonymous accounts, and account pools. Each of these methods presents some administrative difficulties. Most techniques must run as the super-user in order to create a new protection domain for the calling user. Many require some explicit interaction with a human administrator in order to generate a new account and update a mapping table. Most permit little or no sharing of data or resources between users on a given system. Large systems such as Grid3 have worked around these problems by employing the old insecure standby of shared user accounts [18].

Even worse, user identities are not employed consistently across the grid. A single user may be known by a different account name at every single site that he or she accesses, in addition to a variety of identity names given by certificate authorities. In order to access a resource, the user may need to have a local account generated. In order to share resources, each user must know the local identities of users that he/she wishes to share with. However, local identities are often inconsistent or transient, thus preventing any sort of sharing at all.

Ideally, a grid computing system would hide these details from the end user. A user should simply be able to log in and be identified by his or her grid identity without reference to local accounts. If several users wish to share data or resources, they ought to be able to identify each other via their grid identities rather than by arbitrary local names. This ideal is difficult to realize in today's computing systems because of the inflexible nature of the underlying account scheme. Every new user of a grid system must be entered by the administrator into the local account database. Although it is a small burden to do this for one user, it is a full-time job for systems with many thousands of users.

To attack these problems, we introduce the technique of identity boxing. This technique is similar to sandboxing: an untrusted program is run by a secure supervisor that evaluates its actions. The difference is that the identity box attaches a high-level grid identity to every process and resource in the system without regard to the local account details. This allows a user to execute programs and access data in a coordinated way using only grid identities. Further, the administrator of a resource is relieved of the obligation to create and manage accounts: an identity box can create and destroy protection domains as they are needed. A familiar access control interface allows for the controlled sharing of resources.

We have implemented an identity box using Parrot [41], an interposition agent that provides operating-system-like services at the user level. Parrot works by trapping system calls using the debugging interface, therefore it is able to perceive and contain all external effects of an application. Users can-

Account Type	Required Privilege	Protect Owner?	Allow Privacy?	Allow Sharing?	Allow Return?	Admin Burden	Example Systems
Single	-	no	no	yes	yes	-	Personal GASS [7]
Untrusted	root	yes	no	yes	yes	per user	WWW, FTP
Private	root	yes	yes	no	yes	per user	I-WAY [12]
Group	root	yes	fixed	fixed	yes	per group	Grid3 [18]
Anonymous	root	yes	yes	no	no	-	Condor on NT [42]
Pool	root	yes	yes	no	no	per pool	Globus [16] Legion [26]
Identity Box	-	yes	yes	yes	yes	-	Parrot [41]

Figure 1: Identity Mapping Methods

not escape from an identity box, so the supervisor becomes an augmented operating system for grid applications. However, because of this secure implementation, system calls are penalized by an order of magnitude in latency. This has a marginal overhead on a selection of scientific applications, which are slowed down by 0.7 - 6.5 percent in runtime. However, identity boxing is more expensive in meta-data intensive application such as a program build, which is slowed by 35 percent.

To demonstrate the expressive simplicity of identity boxing, we have employed it within the Chirp [40] storage system. The combination of identity boxing with familiar access controls creates a system in which a wide community of users can share resources with little or no intervention by a human administrator.

2. CURRENT SOLUTIONS

Figure 1 summarizes methods currently used for admitting grid users to local systems. Each system has various strengths and weaknesses that we define as follows. A method *requires privilege* if the operator of the service must be the root to employ it. It *protects the owner* if it prevents grid users from harming the service owner after they are admitted. It *allows privacy* if grid users are able to easily protect their data from other users at the same site. It *allows sharing* if grid users are able to easily share their data with others at the same site. It *allows return* if a grid user may store some data, log out, and then log in again at a later time and still be able to access that data. Finally, the *administrative burden* describes how often a human must perform some manual activity as root to admit a new user.

Single Account. The simplest method of identity mapping is to run all visiting processes in the same account. This method is easy to implement and is often a necessity because it requires no special privileges. Obviously, it does not protect the account holder from malicious users, nor does it afford visiting users any privacy from each other. However, it does allow all users admitted to the account to share data and communicate with each other, if they can be trusted to do so. This approach can be acceptable if it is expected that grid credentials will always correspond to one controlling user. For example, one might reasonably operate a personal GASS file server [7] using only a single account.

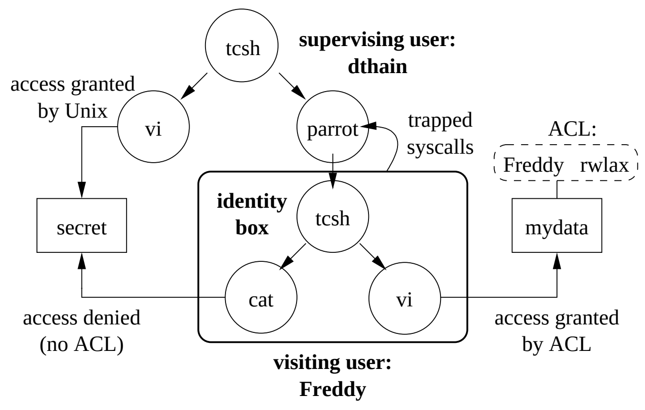
Untrusted Account. If it is desired to protect the resource owner from malicious users, a slight variation is to run all processes in a special account for unknown or untrusted users (nobody) that carries fewer privileges than an ordinary user. This approach is generally used by Web and FTP servers. The untrusted account has the same sharing properties as the single account approach, but requires privileges in order to create and use it.

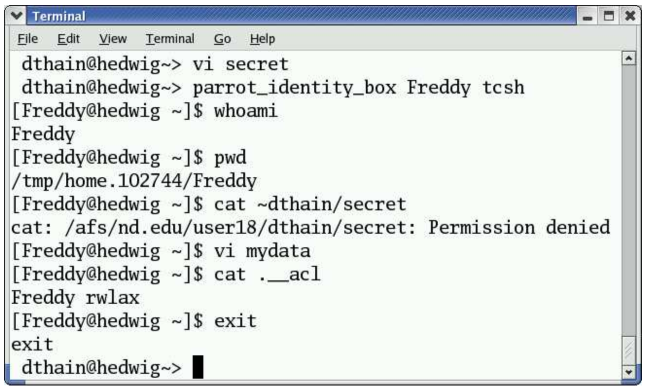
Private Accounts. In systems with distinct users that wish to be protected from one another, one may create a distinct local account for every single user. A table called a "gridmap" file is then needed to map from grid identities to local accounts. This approach was first demonstrated by I-WAY [12] and is widely used today. This approach allows each account to maintain privacy, but does not allow for sharing between accounts. Most importantly, it requires privileges to execute and requires a human administrator to be involved for each new local account creation. In this configuration, the grid credentials are used for securing the connection, but every user still bears the burden of establishing an identity at every site.

Group Accounts. Because of the high administrative burden of creating and maintaining private accounts at every grid site, some systems have turned to creating shared group accounts at every site. This approach is used by the Grid3 [18] system. In this model, there are a small number of accounts, each corresponding to a well-known experiment or collaboration. The involvement of the system administrator is necessary to create the accounts, but once established, multiple users are mapped onto those accounts. These accounts essentially enforce static privacy and sharing policies. Within one group, nothing is private, and all data is shared. Between groups, there is privacy but no sharing. As with the other approaches, privileges are required to manage group accounts.

Anonymous Accounts. As an alternative to group accounts, a system may create a temporary account that lasts only for the duration of a single job. As with private accounts, this requires special privileges, provides privacy, but does not permit sharing. However, it does not require the administrator's involvement for every user. Condor [42] uses this approach on Windows NT by taking advantage of the large numeric user ID space to create a fresh user for every single new job. The primary drawback to this method is that an ID no longer has any meaning after a job completes. Thus, this technique is not suitable for any situation where a job creates persistent data and then must return to it later.

Account Pools. A variation on anonymous accounts may be employed on Unix-like systems. The system administrator may create a pool of anonymous accounts (i.e. grid0-grid99) for use by a grid system, allowing a resource manager to assign available accounts to jobs on the fly. This approach is available in both Globus [16] and Legion [26]. Like anonymous accounts, an account pool does not allow for return: a given user might be grid9 today and grid33 tomorrow. However, it does protect the system owner from users and users from each other.





```
 dthain@hedwig~> vi secret
 dthain@hedwig~> parrot_identity_box Freddy tcsh
[Freddy@hedwig ~]$ whoami
Freddy
[Freddy@hedwig ~]$ pwd
/tmp/home.102744/Freddy
[Freddy@hedwig ~]$ cat ~dthain/secret
cat: /afs/nd.edu/user18/dthain/secret: Permission denied
[Freddy@hedwig ~]$ vi mydata
[Freddy@hedwig ~]$ cat .__acl
Freddy rwlax
[Freddy@hedwig ~]$ exit
exit
 dthain@hedwig~> 
```

Figure 2: Example of Identity Boxing in an Interactive Session
*An example of identity boxing shown as a schematic and as a shell transcript. The supervising user (*dthain*) creates a file* `secret` *in his home directory. He then creates an identity box for the visiting user* Freddy*, who is not allowed to access* `secret` *because there is no ACL present by default. However,* Freddy *can create a file* `mydata` *in his new home directory, where the ACL has been initialized to give him complete access.*

Identity Boxing. Identity boxing, as we will explain shortly, dispenses with all of the difficulties of account management that we have described. It allows named protection domains to be created on the fly without reference to any account database. Identity boxing can be employed by any user without root privileges. This allows ordinary users to create grid services without creating new security risks by becoming root. Because each visiting user runs in a secure protection domain, identity boxing protects the owner from grid users, protects grid users from each other, and allows for both sharing of data, and return to stored data. No administrator intervention is needed to create an identity box.

3. IDENTITY BOXING

An identity box is a secure execution space in which all processes and resources are associated with an external identity that need not have any relationship to the set of local accounts. That is, within an identity box, a program runs with a high-level name such as /O=UnivNowhere/CN=Fred rather than with a simple integer UID or account name.

Identity boxing makes it possible to use identities consistently throughout a grid computing system. Regardless of the machine, account, or resources in use, a program and all of its data components use and perceive the same identity everywhere. Permission checks and access control lists are based upon the high-level name rather than low-level account information. Further, identity boxing dramatically reduces the administrative burden of operating a grid computing system. Identity boxes can be created at runtime by unprivileged users without consulting or modifying local account databases. A single Unix account may be used to securely manage several identity boxes simultaneously, thus eliminating the need to services to run as root.

Ideally, identity boxing would be implemented within the operating system kernel. However, as many have observed, practical grid computing requires that we live with unmodified operating systems. Thus, we have implemented identity boxing using an *interposition agent* [29] that provides operating-system-like behavior at the user level without special privileges.

We have modified the Parrot [41] interposition agent to perform identity boxing on arbitrary processes by securely intercepting and modifying system calls through the debugging interface. Parrot may be thought of as an augmented operating system. In order to execute system calls on behalf of applications, it must track a tree of processes, keep tables of open files, and direct system calls to device drivers. Such an architecture makes it easy to attach filesystem-like services to existing applications. For example, Parrot has been used in the past to access GSI-FTP [2] sites by simply opening files under the path `/gsiftp`. Thus, it is natural to add a new operating-system-like feature such as a change to user identity and access control.

To implement identity boxing, we have modified Parrot to carry with each process a free-form text string indicating the user's high-level identity. The user calls `parrot_identity_box` with an identity string and a command to run. The supervising user can choose absolutely any name for the visitor. MyFriend, JohnQPublic, and Anonymous429 are all valid names. This identity is then visible to the child process through a new system call `get_user_name`. We do not expect programs to be changed to use this system call. Rather, the identity is used internally for access control, much like credentials augment identity in Kerberos [38] or AFS [24].

Within an identity box, access control to files and other objects is somewhat complicated because visiting identities are free-form strings. These new identities do not fit into the existing data structures that record integer UIDs, nor can Parrot modify objects not owned by the supervisor. Our solution to this problem is to abandon the Unix protection scheme and adopt access control lists (ACLs) instead. In each directory, Parrot looks for a file named `.__acl` that describes what actions users can perform on files in that directory. Any program run within an identity box will respect these ACLs. Each entry of an ACL lists an identity and the set of operations that can be performed. Identities may contain wildcards in order to match patterns. For example, this ACL allows /O=UnivNowhere/CN=Fred to read, write, list, execute and administer this directory. It also allows any user at /O=UnivNowhere/ to read and list it:

```
/O=UnivNowhere/CN=Fred    rwlax
/O=UnivNowhere/*          rl
```

Visiting users are given a fresh home directory with an appropriate ACL. Newly-created directories inherit the parent ACL. Of course, Parrot cannot retroactively place ACLs throughout the file system. When it encounters a directory without an ACL, Parrot enforces Unix permissions as if the visiting user was the Unix user nobody. This ensures that the supervising user's data is protected from the visiting user. A user must have the A right to modify an ACL.

Note that ACLs are only respected by processes run within an identity box. A process outside of the box owned by `dthain` would be free to modify such files directly. In this sense, the supervising user is `root` with respect to users in the identity box. A typical server application would place all visiting users in distinctly named identity boxes.

An example of an interactive identity box is shown in Figure 2. Here, the Unix user dthain has created an identity box for Freddy. Note that Freddy does not appear anywhere in the system account list. Freddy attempts to access a file `secret` owned by dthain, but is denied because that file is private to dthain. However, Freddy is given a home directory in which he can work and is allowed to write the file `mydata`.

Figure 2 also shows that the identity box causes the Unix account name to correspond to that of the identity string. This allows `whoami` and similar tools to produce sensible output. This is accomplished by creating a private copy of the `/etc/passwd` file, adding an entry at the top corresponding to the visiting identity, and then redirecting all accesses to `/etc/passwd` to that copy. In addition, a temporary home directory is created for the visiting user's startup files and private data. However, this is merely a convenience. Neither the existing user database nor the private copy play any role in access control within the identity box.

Although this paper describes mostly the semantics of file sharing, it is important to note that the external user identity is employed for all matters that requires some form of privilege check. For example, a process within an identity box may only send signals to other processes with the same identity. This is easily enforced within the supervisor, which keeps a table of processes under its care. Similar comments apply to other kernel resources.

One may easily image a variety of uses for identity boxing on a standalone system. An identity box could be used to securely loan computer access to a visitor without creating a new account. Untrusted programs downloaded from the web could be run within an identity box named by the credentials associated with the program. However, identity boxing is most useful in the context of a distributed system or a grid where there may be an unbounded number of cooperating users.

4. IDENTITY BOXING IN A DISTRIBUTED SYSTEM

Identity boxing allows a grid computing system to securely admit visiting users while retaining their high-level identities to be used for access control. It also simplifies deployment and administration by not requiring superuser privileges. We demonstrate the expressive power of this technique by applying it to the Chirp [40] distributed storage system.

A Chirp server is a personal file server for grid computing. It can be deployed by an ordinary user anywhere there is space available in a file system. A Chirp server exports the available file space using a protocol that closely resembles the Unix I/O interface. This file space can be accessed remotely like a distributed filesystem by using Parrot with ordinary applications. A collection of Chirp servers report themselves to a catalog, which then publishes the set of available servers to interested parties.

Of course, there exist a variety of systems for storing data on the grid. GridFTP [2] provides secure, high-performance access to legacy systems. SRB [4] combines databases, file systems, and other archives into a coherent system. SRM [37] defines semantics for storage allocation in time and space. IBP [33] makes storage accessible through a malloc-like interface with access control via capabilities. NeST [6] provides unified access to grid storage through a variety of protocols. However, Chirp is a particularly interesting platform in which to explore identity boxing because it has a fully virtual user space. This means that the space of local users is completely hidden from external users. All data is stored and referenced by external identities.

A Chirp server supports a variety of authentication methods, including Globus GSI [17], Kerberos [38], ordinary Unix names, and a simple hostname scheme. Upon connecting, the client and server negotiate an acceptable authentication method and then the client must prove its identity to the server. If successful, the server then knows the client by a principal name constructed from the authentication method and the proven identity. One user might be known by any of these names:

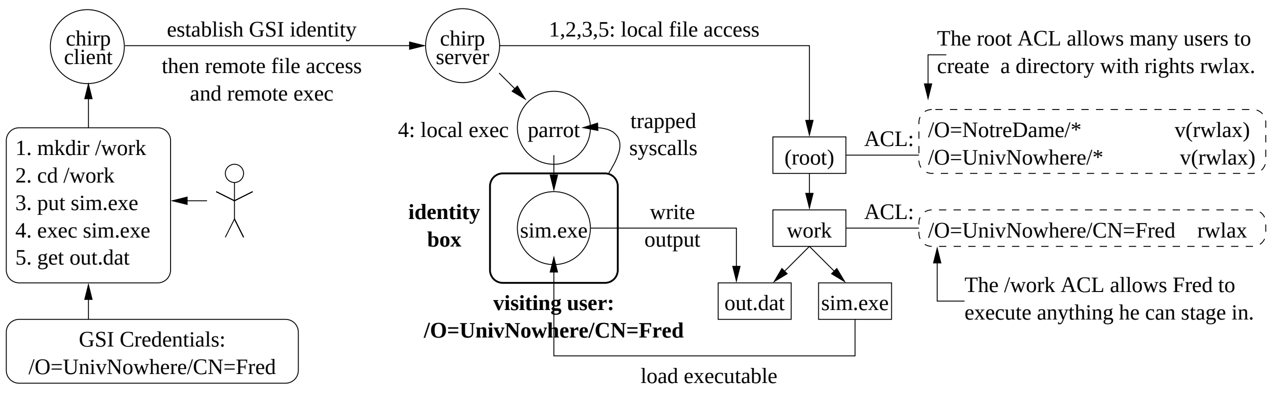
```
globus:/O=UnivNowhere/CN=Fred
kerberos:fred@nowhere.edu
hostname:laptop.cs.nowhere.edu
```

Once identified, a user may access files on the server like any other file server. Using Parrot, files on a Chirp server appear as ordinary files in the path `/chirp/server/path`. These files are protected by ACLs like those used in Parrot.

Now, imagine the user that wishes to execute a program using data stored on such a server. Traditionally, the user would have to arrange for a login on the same server and use that to access the data directly. However, the user would also have to arrange for the server to store the data under that same identity, which would require the server to run as root. If this was impossible, the user would have to extract the data from the server and run the computation on a different host entirely.

The technique of identity boxing allows to sidestep these difficulties. To demonstrate this, we have added to the Chirp protocol a simple `exec` call that invokes a remote process. This process is run within an identity box corresponding to the identity negotiated at connection. The identity box enforces access to resources as described above, allowing ordinary applications run unmodified in a remote environment. Of course, the calling user must have the execute (x) right on the program (and any sub-programs) to be executed.

The combination of file access and remote execution allows for simple but powerful controls. If the user has the write and execute (wx) rights on a directory, then he/she can stage in an executable and run it. If the user has only the read and execute (rx) rights, then he/she is limited to running programs already there. For example, this ACL would allow



chirp client
establish GSI identity
then remote file access and remote exec
chirp server
1,2,3,5: local file access
The root ACL allows many users to create a directory with rights rwlax.
4: local exec
parrot
trapped syscalls
1. mkdir /work
2. cd /work
3. put sim.exe
4. exec sim.exe
5. get out.dat
identity box
sim.exe
write output
(root)
ACL:
/O=NotreDame/* v(rwlax)
/O=UnivNowhere/* v(rwlax)
work
ACL:
/O=UnivNowhere/CN=Fred rwlax
The /work ACL allows Fred to execute anything he can stage in.
visiting user: /O=UnivNowhere/CN=Fred
out.dat
sim.exe
GSI Credentials: /O=UnivNowhere/CN=Fred
load executable

Figure 3: Example of Identity Boxing in a Distributed System
Identity boxing can be used to support visiting users in a distributed system. The Chirp file server provides remote file access and remote file execution to network users. A remote user using a Chirp client creates the `/work` *directory, stages in the* `sim.exe` *program, executes it, and then retrieves the output* `out.dat`*. The Chirp server runs* `sim.exe` *in an identity box corresponding to the remote user. The system may be run by any ordinary user and does not require the creation of any accounts before or during its operation.*

any user in `nowhere.edu` to run existing programs, while allowing any user holding a `UnivNowhere` certificate to stage in and run any program.

```
/:  hostname:*.nowhere.edu      rlx
    globus:/O=UnivNowhere/*     rwlx
```

The flexibility of identity boxing creates some new challenges. Identity boxing encourages the use of wildcards in access controls. But, a large set of users identified by a wildcard will not necessarily want to share a namespace. Imagine the chaos of allowing one hundred users using the same directory to store files and run programs! Visiting users will want a fresh namespace and the ability to adjust the ACL in order to work with collaborators. For this purpose, an ACL may also include the *reserve right* (V), which is a variation upon amplification [28]. Suppose that the remote users had been given only the reserve right:

```
/:  hostname:*.nowhere.edu      rlx
    globus:/O=UnivNowhere/*     v(rwlax)
```

When a user performs a `mkdir` in a directory in which he/she only holds the reserve right, the newly-created directory is initialized with an ACL containing the rights listed in parentheses after the V. Not only does this create a private namespace, but it also allows the user to selectively grant access to others. Suppose that the above ACL is present in the root directory when `globus:/O=UnivNowhere/CN=Fred` invokes `mkdir(/work)`. The ACL in `/work` would be:

```
/work:  globus:/O=UnivNowhere/Fred    rwlax
```

By virtue of the A right, Fred can further adjust the ACL to give access to other users. Of course, if the system owner does not want a visiting user to extend rights to others, then the A right may simply be left out of the reserve set.

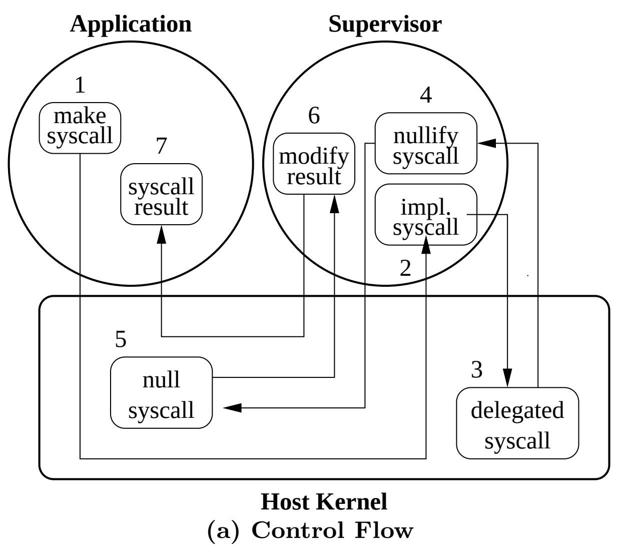
The combination of identity boxing with a virtual user space and powerful ACLs allows for a dramatically simplified user experience. Given appropriate ACLs, users may discover storage, stage data, run programs, and retrieve output without special privileges or interaction with an administrator. Further, any user is permitted to be a supervisor, deploying and administering any resource that they are able to access. Owners of resources remain in control, delegating and restricting rights as they see fit.

Figure 3 demonstrates how all this fits together. The user `Fred` wishes to run `sim.exe` on a remote machine using his grid credentials. He uses a client tool to contact a Chirp server and creates the `/work` directory using the reserve (V) right. He then stages in the input data and the executable to the remote machine. Using the `exec` call, he invokes the simulation, which is run in an identity box annotated with his name. The identity box allows his simulation to run and access his data securely, even though he does not have an account on the machine. Finally, he retrieves the output and cleans up.

At this point, it is worth pointing out an important aspect of identity boxing. The identity box simplifies the creation and management of protection domains: a system may create an identity box on the fly without regard to any external user database. However, this does *not* mean that identity boxing requires a system to admit arbitrary users. Rather, identity boxing allows a system to have complex admission policies, such as access controls with wildcards, or reference to a community authorization service [32], without the difficulty of reconciling that policy to the existing user database.

5. IMPLEMENTATION DETAILS

Ideally, identity boxing would be a service provided by the operating system kernel to all users of any privilege level. This would allow for the highest assurance in the security of its implementation, and minimize any performance overheads. However, it is not practical in the short term to ask grid computing sites to modify kernels, thus we have chosen interposition via Parrot as way of augmenting existing kernels. Parrot in particular is implemented only on the Linux operating system, but the concept of identity boxing in general is not tied to this platform. Some comments on



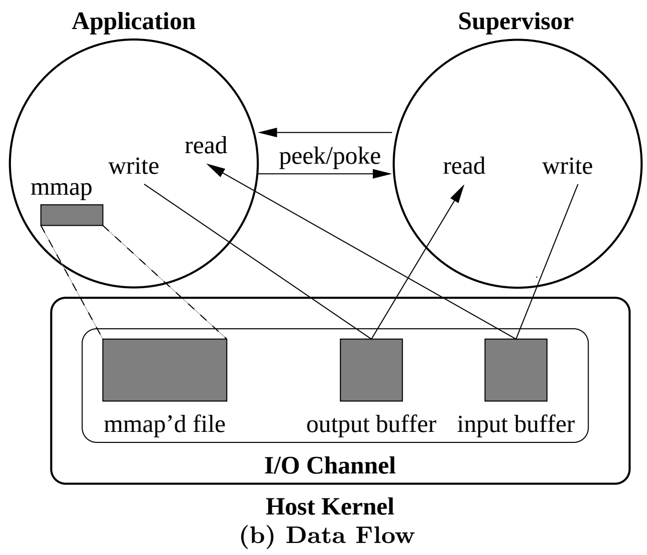


Figure 4: System Call Trapping
Identity boxing is implemented in a system-call trapping interposition agent. (a) shows the control flow. For each system call that the application attempts, (1) the supervisor gains control (2) and then implements the action by making its own system calls (3). The original system call is nullified by converting it into a `getpid` *(4). When it returns (5), the supervisor modifies the result to that of the implemented call (6), which is finally revealed to the application (7). (b) shows the data flow. Small amounts of data can be moved by peeking and poking one word at a time. Large amounts of data must be moved into the I/O channel, then the appl. must be coerced into accessing it .*

how identity boxing might be implemented in the kernel are given in the conclusion.

Parrot has been implemented as a user-level process that securely traps system calls using the `ptrace` interface on the Linux operating system. Although the Linux `ptrace` interface is often reported to be less convenient than the Solaris `proc` interface, it is sufficient for performing interposition and gives access to a more widely deployed platform for scientific computing. Readers interested in even more detail may consult an earlier paper on Parrot [41].

Figure 4 shows how the system call trapping mechanism works. The supervisor process (Parrot) runs an application as a child using the `ptrace` debugging interface. When the child attempts a system call, the kernel halts the process and notifies the supervisor. The supervisor then examines the detail of the system call, and *implements it on behalf of the child process* by either consulting its internal state and/or making one or more system calls. Thus, Parrot is a delegation architecture like Ostia [21].

Once the supervisor has computed the result of the system call and applied any necessary side effects to the child process and the surrounding system, it must return a result to the child. On most operating systems, it is not possible to abort a system call outright, so instead the supervisor modifies the child's registers to convert the system call into a fast null operation: `getpid()`. Again, the supervisor gains control when the `getpid()` call completes and updates the child's registers to reflect the desired result.

This mechanism is used for the majority of system calls that require a small amount of data to be moved in and out of the process. Modifications to registers and small amounts of memory can be performed one work at a time using the `ptrace` peek and poke operations. For system calls that require a large amount of data movement, another technique is required. Ideally, the supervisor would simply use `mmap` to directly access the memory of the child process reflected in `/proc/x/mem`. However, recent versions of the Linux kernel prevent writing to this special file, due to concerns of complexity and security.

Lacking this ability, the application must be coerced into assisting the supervisor. This is accomplished by converting many system calls into `preads` and `pwrites` on a shared buffer called the *I/O channel*. This is small in-memory file shared among all of its children. The supervisor maps the channel into memory, while all of the child processes simply maintain a file descriptor pointing to the channel.

For example, suppose that the application issues a `read` on a file. Upon trapping the system call entry, Parrot examines the parameters of `read` and retrieves the needed data. These are copied directly into a buffer in the channel. The `read` is then modified (via `poke`) to be a `pread` that accesses the I/O channel instead. The system call is resumed, and the application pulls in the data from the channel, unaware of the activity necessary to place it there. This extra data copy has some performance implications explored below.

6. SECURITY AND CORRECTNESS

System call trapping is a secure interposition method. If the mechanism is properly implemented, the child process is unable to escape the control of the supervisor. All side effects must be performed by making system calls, and each of these must pass though the supervisor for both approval and implementation. Unlike other techniques such as library interposition [42] or binary rewriting [44], no clever linking tricks nor carefully-crafted assembly code can be used to elude the trapping mechanism. Of course, an application can always attempt to trigger bugs in the supervisor by testing boundary conditions in system calls, just as in a system kernel or a server process.

Parrot supports the vast majority of Unix system calls. Process management, file access, network access, non-blocking I/O, asynchronous I/O, and many other details of the interface are working. Multi-threaded applications and inter-process communication are supported in the same way as in a real kernel. Blocking system calls place the calling thread or process into a wait state so that the supervisor can wait upon and service system calls by other threads and processes. A few system calls have not been implemented. For example, Parrot does not (yet) implement the `ptrace` interface, so processes under Parrot are not able to debug each other. In addition, a number of system calls only useful to the system administrator (such as `mount`) are also unimplemented. However, these are limitations of the implementation, not the architecture.

To give some sense of the state of implementation, here is an (incomplete) list of applications used with Parrot on a daily basis: **`mozilla, emacs, tcsh, bash, ssh, gcc, vi, make, xterm`** as well as a large number of basic utilities such as **`grep, less, cp, mv, ls,`** and **`rm`**. Also, a selection of scientific applications that work with Parrot are given below.

T. Garfinkel has noted [19] that system-call trapping is a non-trivial problem with many subtleties that can be exploited by malicious applications. We whole-heartedly agree with these observations, but modify them slightly in the context of a delegation oriented architecture such as Parrot. Here are Garfinkel's five traps and pitfalls:

Incorrectly replicating the OS. When a supervisor attempts to mirror some state that is also contained in the operating system, it is possible for the sandbox to become unsynchronized with the system. Parrot does not have this problem, because it maintains *all state* for each process within itself.

Overlooking indirect paths. When there are multiple links to a single object, the sandbox must be careful to check permissions on the object, rather than on the links. This problem is found in the filesystem. Parrot checks for an ACL in the directory in which a file is located before granting access. However, if the file is in fact a link elsewhere, then Parrot must follow that link and examine the target directory instead. This requires that Parrot examine each opened file; if the file is actually a symbolic link, the ACL in the target directory must be examined. No such examination can be done with hard links, therefore Parrot is obliged to prevent hard links to files that the user cannot access.

Incorrect subsetting of a complex interface. Many sandboxes attempt to outlaw a particular system call or interface entirely. This has one of two effects: either applications are rendered unusable, or the complex interface has "leaks" that allow access in other ways. This is not a problem in Parrot, as containment is achieved through access control, rather than by outlawing interfaces.

Race conditions. When a process requests a system call, a sandbox must perform one sequence of system calls to implement access control, and another sequence to implement the action. Because a sequence of system calls cannot be done atomically, it possible for the access control to be changed between the check and the access. In the context of identity boxing this is not a problem. Only the supervising user would be able to take advantage of this loophole, and the supervising user is effectively omnipotent to the visiting users already.

Side effects of denying system calls. Some operating systems do not allow a debugger to modify the return code of a system call, but only to change it to an "aborted" value or to kill the process entirely. On Linux, Parrot is able to provide any return value, including "permission denied."

From all these details, we may conclude that system call interposition as complicated as an operating system kernel. But, it can be made to work for real applications. Despite the necessary complexity, interposition is invaluable when it is simply not possible to modify the operating system. However, we also believe that identity boxing would find a better implementation in the operating system proper. We consider this in the concluding remarks.

7. APPLICATION PERFORMANCE

A user-level implementation of identity boxing has significant but not insurmountable overhead. In order for Parrot to trap and interpret the system calls of an application, at least six context switches are necessary, as shown in Figure 4(b). These extra context switches increase latency and also flush processor caches that might otherwise be preserved in an optimized system call mechanism. An additional data copy is also needed for bulk I/O operations.

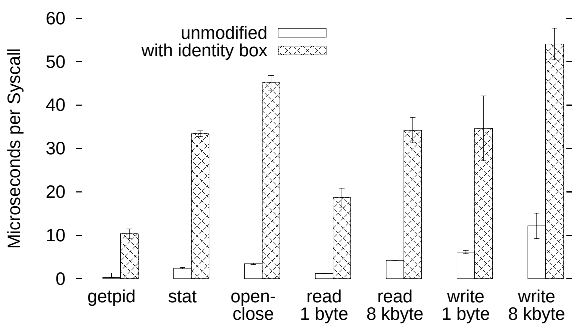
Figure 5 shows the effects of this performance overhead on individual system calls as well as real applications. Figure 5(a) shows the latency overhead of system calls handled within the identity box. Each entry was measured by a benchmark C program which timed 1000 cycles of 100,000 iterations of various system calls on a 1545 MHz Athlon XP1800 running Linux 2.4.20. Each system call was performed on an existing file in an ext3 filesystem with the file wholly in the system buffer cache. Each call is slowed down by an order of magnitude.

We also ran six real applications in order to measure the actual overhead of identity boxing amortized over application activity. Five of these were scientific applications that are candidates for execution on grid systems. AMANDA [25] is a simulation of a gamma-ray telescope. BLAST [3] searches genomic databases for matching proteins and nucleotides. CMS [23] is a simulation of a high-energy physics apparatus. HF [11] is a simulation of the nucleic and electronic interactions. IBIS [14] is a climate simulation. These applications are described in great detail in an earlier paper [39]. An additional application, make, is simply a build of the Parrot software itself.

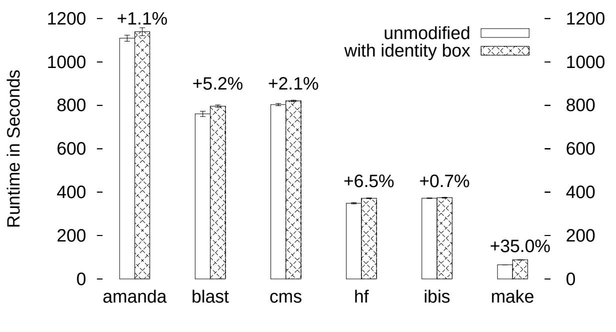
The overhead of identity boxing on these applications is shown in Figure 5(b). The five scientific applications are slowed down by only 0.7 - 6.5 percent. Although they are more data intensive than other grid applications, they perform primary large-block I/O. An interactive application such as `make` is slowed down by 35 percent because it make extensive use of small metadata operations such as `stat`. Thus, identity boxing via an interposition agent has overhead that is likely to be acceptable for scientific applications, especially if the technique empowers the user to harness a larger array of resources.

8. RELATED WORK

Sandboxing. Identity boxing is closely related to sandboxing. A sandbox runs an untrusted program underneath a supervisor process which traps its operations and checks them with a reference monitor. The mechanism can be binary rewriting, as in Shepherd [30], a kernel module, as in



5(a) - System Call Latency



5(b) - Application Runtime

Figure 5: Overhead of Identity Boxing
Within an identity box, individual system calls are slowed by an order of magnitude due to the multiple context switches between the application, the supervisor, and the host kernel. On real applications, the effective overhead varies. A selection of five scientific applications are slowed down from 0.7 to 6.5 percent, but a system-call intensive application such as `make` *is slowed down by 35 percent.*

Janus [22], or the debugging interface, as in Systrace [34]. These systems all require the user to state a list of acceptable operations. Another possibility is to associate rights with programs rather than users, as in SubDomain [10] and MAPBox [1]. Ostia [21] delegates all operations to an agent, allowing for arbitrary policies. One might also consider the Unix `chroot` mechanism to be a simplified sandbox. `chroot` creates a fresh, empty file space in which an application can work but not escape.

Traditional sandboxing requires users to provide some specification or approval of the system calls attempted by an application. This is an enormous burden because most users have no idea what happens deep within an application. For example, a user running a word processor thinks (quite logically) that the word processor only needs to read and write the file that he/she is editing. In fact, the program needs to load an executable, read a configuration file, load plugin libraries, access the dynamic linker, read the host database, create backup files, and use a whole host of other resources that the user has never heard of. In our field experience with scientific applications [41, 39, 5], even authors of technical software are surprised to learn exactly what system calls their programs attempt. Users are insulated from the system by so many layers of software that we cannot expect them to think in terms of low-level system calls. Identity boxing builds upon sandboxing by providing built-in access controls that correspond to familiar concepts. Rather than requiring the supervisor to state the access control policy in advance, identity boxing allows the visiting user to interact with others as a first class citizen.

Privilege Separation [35] attacks the same problem in a different way. Many programs, such as login servers, only need some subset of the super-user's capabilities. A common subset is simply the ability to call `setuid()`. However, the sheer complexity of a login server makes it difficult to trust the entire program. Thus, the server itself can be run in an untrusted mode. When it requires a privileged operation, it must explicitly request it from a small kernel of privileged code, which checks the intended operation and then performs it on behalf of the server. This technique is powerful and effective, but still requires a small amount of privileged code and perhaps some code transformation [8]. Identity boxing provides the same power as privilege separation, but requires no privileged code at all.

Virtual Machines. The virtual machine has been proposed as the solution to a variety of problems in distributed computing [36, 43], grid computing [13, 9], operating system composition [15, 27], and security [20, 31]. A virtual machine can completely isolate a service provider from the contained user. This provides both security and an unrestricted workspace for the contained user, who can safely be an administrator in the virtual environment. This is enormously useful ability, particularly when developing a new operating system or performing whole-system simulation.

A virtual machine provides some of the benefits of identity boxing. However, it is less practical in two respects. First, creating a virtual machine is a non-trivial administrative activity: one must generate disk images, setup user databases, and install software within the virtual machine itself. Effectively, the creation and management of virtual machines is an activity only accessible to those already skilled in system administration. This also may come at a significant performance cost to move data in and out of the virtual machine. Second, the virtual machine inhibits sharing where it is most needed. Users that run untrusted programs generally *want* those programs to interact with the existing system in a limited way. They want to retain access to local files, to interact with existing processes, to communicate over the existing network. Virtual machines isolate visiting users, while the identity box encourages controlled sharing.

9. CONCLUSION AND FUTURE WORK

Identity boxing addresses two distinct limitations of traditional operating systems with respect to distributed computing.

First, the traditional operating system does not allow ordinary users to create new protection domains. The creation of a new account is an activity that only the superuser can perform. As a result of this, users are forced to choose between obtaining superuser privileges (if this is even possible), or running multiple untrusted programs within one account. The identity box allows users to defend themselves without obtaining maximum privilege. This permits the ordinary user to operate a secure grid service.

Second, the traditional operating system does not allow high-level names to be associated with low level names. This causes difficulty in the realm of grid computing, where the system operator is obliged to maintain some mapping between global and local usernames. Further, without the high-level name, it is virtually impossible for users to engage in data-sharing on the local system. The identity box allows for the consistent use of identities globally, allowing the user to completely ignore the local account name.

One application of identity boxing outside of the grid computing domain might be for untrusted web browsing. Many programs downloaded from the web are associated with credentials that identify the owner or creator. Yet, credentials alone do not imply that the program is trusted. Using an identity box, an ordinary user may run an untrusted program using a credentialed name such as *JoeHacker* or *BigSoftwareCorp*. In addition to protecting the supervising user, the identity box could be used for forensic purposes, recording the objects accessed and the activities taken by the untrusted user. A suitable graphical interface to identity boxing would allow the non-technical user to distinguish between trusted and contained processes.

As we have observed, the implementation of an identity box using system-call trapping is convenient, but complex and perhaps too expensive for some applications. We propose that future operating systems should include the capability for ordinary users to create new protection domains with high-level names on the fly. If each user is capable of creating arbitrary names, then a hierarchical namespace is necessary to prevent conflicts, much as in the domain name system. Figure 6 shows an example of this. An ordinary user might be known as `root:dthain`, and a new protection domain for a visitor might be `root:dthain:visitor`. In such a system, a web server could create identities for service processes, and a grid server could create identities corresponding to grid identities.

Naturally, a change to the namespace would introduce some complexities into the implementation. For example, user names would no longer be stored as integer indexes, but as full text strings. The hierarchy of users would result in new management relationships between processes. The filesystem would require some modification in order to store long names of file owners. In turn, this would require richer access controls on files (such as the ACLs shown above) in order to accommodate new patterns of sharing between users. These issues we leave open for future work.

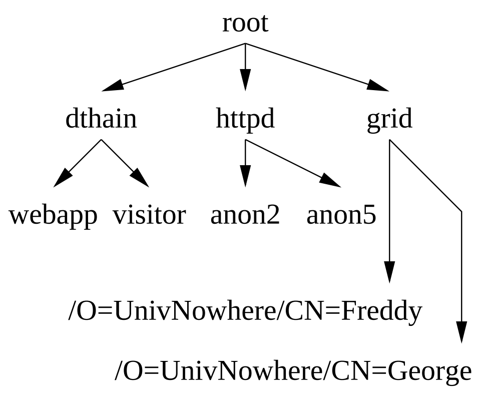


Figure 6: Hierarchical User Identity
An operating system with a hierarchical user namespace would provide the benefits of identity boxing with the performance and assurance of an operating system. A tree of identities allows every user to create protection domains as needed.

10. REFERENCES

[1] A. Acharya and M. Raje. MAPbox: Using parameterized behavior classes to confine applications. Technical Report UCSB TRCS99-15, University of California at Santa Barbara, Computer Science Department, 1999.

[2] W. Allcock, A. Chervenak, I. Foster, C. Kesselman, and S. Tuecke. Protocols and services for distributed data-intensive science. In *Proceedings of Advanced Computing and Analysis Techniques in Physics Research*, pages 161–163, 2000.

[3] S. Altschul, W. Gish, W. Miller, E. Myers, and D. Lipman. Basic local alignment search tool. *Journal of Molecular Biology*, 3(215):403–410, Oct 1990.

[4] C. Baru, R. Moore, A. Rajasekar, and M. Wan. The SDSC storage resource broker. In *Proceedings of CASCON*, Toronto, Canada, 1998.

[5] J. Bent, D. Thain, A. C. Arpaci-Dusseau, R. H. Arpaci-Dusseau, and M. Livny. Explicit control in a batch-aware distributed file system. In *USENIX Networked Systems Design and Implementation*, 2004.

[6] J. Bent, V. Venkataramani, N. LeRoy, A. Roy, J. Stanley, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Flexibility, manageability, and performance in a grid storage appliance. In *Proceedings of the Eleventh IEEE Symposium on High Performance Distributed Computing*, Edinburgh, Scotland, July 2002.

[7] J. Bester, I. Foster, C. Kesselman, J. Tedesco, and S. Tuecke. GASS: A data movement and access service for wide area computing systems. In *6th Workshop on I/O in Parallel and Distributed Systems*, May 1999.

[8] D. Brumley and D. Song. Privtrans: Automatically partitioning programs for privilege separation. In *USENIX Security Symposium*, August 2004.

[9] J. Chase, L. Grit, D. Irwin, J. Moore, and S. Sprenkle. Dynamic virtual clusters in a grid computing environment. In *High Performance Distributed Computing*, June 2003.

[10] C. Cowan, S. Beattie, G. Kroah-Hartman, C. Pu, P. Wagle, and V. Gligor. Subdomain: Parsimonious server security. In *USENIX Systems Administration Conference*, 2000.

[11] P. E. Crandall, R. A. Aydt, A. A. Chien, and D. A. Reed. Input/output characteristics of scalable parallel applications. In *Proceedings of the IEEE/ACM Conference on Supercomputing*, San Diego, California, 1995.

[12] T. A. DeFanti, I. Foster, M. E. Papka, and R. Stevens. Overview of the I-WAY: Wide area visual supercomputing. *International Journal of Supercomputer Applications*, 10(2/3):121–131, 1996.

[13] R. J. Figueiredo, P. A. Dinda, and J. A. B. Fortes. A case for grid computing on virtual machines. In *International Conference on Distributed Computing*

Systems, May 2003.

[14] J. Foley. An integrated biosphere model of land surface processes, terrestrial carbon balance, and vegetation dynamics. *Global Biogeochemical Cycles*, 10(4):603–628, 1996.

[15] B. Ford, M. Hibler, J. Lepreau, P. Tullmann, G. Back, and S. Clawson. Microkernels meet recursive virtual machines. In *Operating Systems Design and Implementation*, 1996.

[16] I. Foster and C. Kesselman. Globus: A metacomputing intrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[17] I. Foster, C. Kesselman, G. Tsudik, and S. Tuecke. A security architecture for computational grids. In *ACM Conference on Computer and Communications Security Conference*, 1998.

[18] R. Gardner and et al. The Grid2003 production grid: Principles and practice. In *IEEE Symposium on High Performance Distributed Computing*, 2004.

[19] T. Garfinkel. Traps and pitfalls: Practical problems in in system call interposition based security tools. In *Network and Distributed Systems Security Symposium*, February 2003.

[20] T. Garfinkel, B. Pfaff, J. Chow, M. Rosenblum, and D. Boneh. Terra: A virtual machine-based platform for trusted computing. In *Symposium on Operating Systems Principles*, 2003.

[21] T. Garfinkel, B. Pfaff, and M. Rosenblum. Ostia: A delegating architecture for secure system call interposition. In *Symposium on Network and Distributed System Security*, 2004.

[22] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications. In *USENIX Security Symposium*, San Jose, CA, 1996.

[23] K. Holtman. CMS data grid system overview and requirements. CMS Note 2001/037, CERN, July 2001.

[24] J. Howard, M. Kazar, S. Menees, D. Nichols, M. Satyanarayanan, R. Sidebotham, and M. West. Scale and performance in a distributed file system. *ACM Trans. on Comp. Sys.*, 6(1):51–81, February 1988.

[25] P. Hulith. The AMANDA experiment. In *Proceedings of the XVII International Conference on Neutrino Physics and Astrophysics*, Helsinki, Finland, June 1996.

[26] M. Humphrey, F. Knabe, A. Ferrari, and A. Grimshaw. Accountability and control of process creation in metasystems. In *Network and Distributed System Security Symposium*, February 2000.

[27] S. Ioannidis and S. M. Bellovin. Sub-operating systems: A new approach to application security. In *SIGOPS European Workshop*, February 2000.

[28] A. K. Jones and W. A. Wulf. Towards the design of secure systems. *Software - Practice and Experience*, 5(4):321–336, 1975.

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

[30] V. L. Kiriansky. Secure execution environment via program shepherding. In *USENIX Security Symposium*, August 2002.

[31] M. Laureano, C. Maziero, and E. Jamhour. Intrusion detection in virtual machine environments. In *EUROMICRO Conference*, September 2004.

[32] L. Pearlman, V. Welch, I. Foster, C. Kesselman, and S. Tuecke. A community authorization service for group collaboration. In *IEEE Workshop on Policies for Distributed Systems and Networks*, 2002.

[33] J. Plank, M. Beck, W. Elwasif, T. Moore, M. Swany, and R. Wolski. The Internet Backplane Protocol: Storage in the network. In *Proceedings of the Network Storage Symposium*, 1999.

[34] N. Provos. Improving host security with system call policies. In *USENIX Security Symposium*, August 2004.

[35] N. Provos and M. Friedl. Preventing privilege escalation. In *USENIX Security Symposium*, August 2003.

[36] C. P. Sapuntzakis, R. Chandra, B. Pfaff, J. Chow, M. S. Lam, and M. Rosenblum. Optimizing the migration of virtual computers. In *Symposium on Operating Systems Design and Implementation*, 2002.

[37] A. Shoshani, A. Sim, and J. Gu. Storage resource managers: Middleware components for grid storage. In *Proceedings of the Nineteenth IEEE Symposium on Mass Storage Systems*, 2002.

[38] J. Steiner, C. Neuman, and J. I. Schiller. Kerberos: An authentication service for open network systems. In *Proceedings of the USENIX Winter Technical Conference*, pages 191–200, 1988.

[39] D. Thain, J. Bent, A. Arpaci-Dusseau, R. Arpaci-Dusseau, and M. Livny. Pipeline and batch sharing in grid workloads. In *Proceedings of the Twelfth IEEE Symposium on High Performance Distributed Computing*, Seattle, WA, June 2003.

[40] D. Thain, S. Klous, J. Wozniak, P. Brenner, A. Striegel, and J. Izaguirre. Separating abstractions from resources in a tactical storage system. In *Proceedings of the International Conference for High Performance Computing and Communications (Supercomputing)*, November 2005.

[41] D. Thain and M. Livny. Parrot: Transparent user-level middleware for data-intensive computing. In *Proceedings of the Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.

[42] D. Thain, T. Tannenbaum, and M. Livny. Condor and the grid. In F. Berman, G. Fox, and T. Hey, editors, *Grid Computing: Making the Global Infrastructure a Reality*. John Wiley, 2003.

[43] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. In *USENIX Annual Technical Conference*, June 2002.

[44] V. Zandy, B. Miller, and M. Livny. Process hijacking. In *Proceedings of the Eighth IEEE International Symposium on High Performance Distributed Computing*, 1999.