Sub-Identities: Toward Operating System Support for Distributed System Security

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

Abstract

We propose sub-identities, a new model for protection domains in the operating system. In this model, user identities are arranged in a hierarchy, allowing each user to create arbitrarily named sub-identities at runtime without the help or approval of an administrator. This model gives users control over their own environment and also simplifies the interaction of distributed systems with local operating systems. The abstract model of sub-identity can be approximated by three implementations that vary in fidelity and complexity: user-level sandboxes, a username toolkit, and kernel modifications. We implement one method — user-level sandboxes — and demonstrate how sub-identities can be applied to the problems of secure login and untrusted web browsing.

1 Introduction

Distributed systems rely on operating systems to provide basic mechanisms of security and resource management. Without sufficient mechanisms on local systems, it is difficult or impossible for a distributed system to provide strong guarantees [28, 3]. An excellent example of this is the problem of *containment*. If a distributed system is to protect one user from another, it must have a method of containing processes on each local node. Without an effective method of containment on each node of a system, no amount of sophistication in the distributed system will protect a user's programs or data on a single node.

Unfortunately, the mechanisms available for containment in today's operating systems are limited. In both the Unix and Windows families of operating systems, there are only two levels of privilege: ordinary users and the super-user (*root* or *Administrator*.) Ordinary users are contained: they cannot affect each other's programs or data without the permission of the owner. Only the super-user has the ability to perform containment by creating processes owned by distinct users. In addition, the super-user must also maintain a list of all users authorized to access the system.

This simple model of identity is poorly suited for supporting distributed systems. A computing environment today consists of a potentially unbounded set of users. Non-technical users commonly download and run programs written by remote and possibly anonymous authors. Ordinary people communicate with hundreds of web servers, many identified by strong public key credentials. A large web site might accept thousands of new user identities in a day. A grid computing system [15] might facilitate the interaction of hundreds of scientists, along with their affiliated students, colleagues, and administrators. In all of these cases, there is no global super-user, nor can one generate a static list of users for any one machine.

Consider the user that wishes to download a program from the web and run it on his personal machine. Because the program is not fully trusted, the user would like to run within a protection domain so that it cannot read or write his personal data. Most users are well aware that operating systems support multiple identities, if only by virtue of the fact that they are required to login with a name and password. The obvious thing to do is run the program with a new user identity, perhaps simply as a user named *webapp*. This allows the operating system to accept responsibility for protecting the user from the untrusted program. The program would be unable to read the supervising user's designated data, and unable to write new data, except perhaps in a private directory. Even better, the identity of the program's creator, such as *JoeHacker* or *BigSoftwareCorp*, may already have been established via an exchange of credentials. If this identity could be attached to the program, its actions could be audited and traced back to the responsible party.

Ordinary users are unable to do this because they cannot create new identities, and thus cannot create new protection domains. Even if a user could become the super-user on his personal machine, it makes little sense to run a large, complex, and possibly untrusted program such as a web browser as the super-user. A vulnerability in the browser itself would put the entire system at risk. Either way, the user runs untrusted programs at his own peril. Thus, users of conventional operating systems are presented with a catch-22. A user cannot protect himself from an untrusted program unless he becomes the super-user first. But, running as the super-user introduces its own set of risks and complications. That is, in order to *restrict* one's own privilege, one must first be *elevated* to maximum privilege, exposing the entire system to risk. This offends common sense, as well as the principle of least privilege [36].

We propose to remedy this situation by introducing *sub-identities* into the operating system. Using sub-identities, every ordinary user can create new protection domains on the fly. Each new domain has a meaningful name and can be used to enforce access control, perform auditing, or simply isolate sub-processes from one another. Sub-identities also simplify the mapping of identities in distributed systems into protection domains on operating systems. In this paper, we present an abstract model of sub-identities as it would be implemented in a hypothetical operating system. We also describe how sub-identities can be approximated with varying degrees of fidelity and complexity by employing user-level sandboxes, a username toolkit, and a full-blown kernel implementation. We describe our experiences with the first implementation using sandboxes, and apply it to the problem of untrusted web browsing and a secure login server.

2 An Abstract Model of Sub-Identities

We begin by describing an abstract model of sub-identities, assuming that all aspects of the operating system are open to modification. We recognize that an implementation adapted to an existing system might deviate from this model in certain ways, but we defer that discussion until later. Figure 1 shows how sub-identities might be used in a Unix-like system. Each edge indicates the creation of a user, running from the *superior* user to the *inferior* user. In this example, the root of trust is the *root* user, which is superior to its three inferior users *alice*, *www*, and *kerberos*. We assume that the *root* user is responsible for accepting console logins and for starting service processes, much as in Unix.

The programming interface to sub-identities is simple. A process owned by user *x* may call *subuser(n)* in order to change the identity of the current process to *x:n*. A process may create a new user identity without abandoning its own by invoking *fork()* before *subuser(n)*. This naming convention reflects the hierarchical nature of user identities. For example, the full name of *betty* in Figure 1 is *root:alice:betty*. This full name also permits distinction between identities in different branches of the tree: *root:alice:browser* is distinct from *root:kerberos:david:browser*. A process may obtain its identity by calling *getuser()*.

It is important to note that the programming interface does not dictate how superior users make authentication decisions. *subuser(x)* is roughly analogous to *setuid(n)* in Unix; it simply modifies the identity of the current process. Each level of the hierarchy has its own authentication method. For example, the *root* user can employ the traditional user database in */etc/passwd* in order to validate passwords and admit new users. This database only reflects the users in the second level of the tree: it is no longer a global database of users. The local Kerberos service need not consult */etc/passwd*. It relies entirely on the remote Kerberos

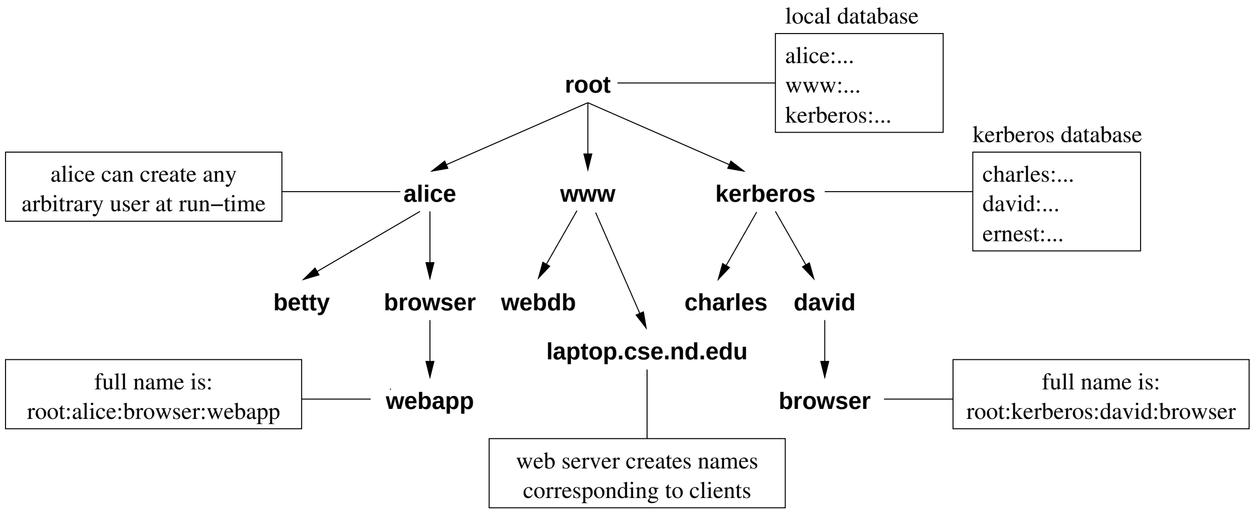

local database
alice:...
www:...
kerberos:...
root
kerberos database
charles:...
david:...
ernest:...
alice can create any arbitrary user at run−time
alice
www
kerberos
betty
browser
webdb
charles
david
laptop.cse.nd.edu
full name is: root:alice:browser:webapp
webapp
browser
full name is: root:kerberos:david:browser
web server creates names corresponding to clients

Figure 1. Example of Sub-Identities
This figure shows a variety of users that might be employed on a system with sub-identities. The `root` *user starts services and accepts ordinary logins, consulting a local user database before granting access.* `alice` *creates a variety of identities for her personal use.* `www` *is used to run the web server, and safely services each incoming request with a sub-identity.* `kerberos` *also accepts logins and creates new sub-identities corresponding to users that appear in the remote Kerberos database.*

service to decide what users to admit. Other ordinary users simply invoke *subuser(n)* as they see fit. Every inferior process retains the ability to perform *subuser(n)*, but this is safe because *subuser(n)* does not allow any process to elevate its privilege.

A superior user is effectively *root* to its inferior users. A superior user may send signals to its inferiors, debug their processes, modify their data, and perform any other activity necessary to ensure the safety and correctness of their operation. Naturally, an inferior user has no such power with respect to its superior. This means there is very little reason for any user to assume the *root* identity. Of course, there are a few cases where *root* is still needed, such as to modify kernel structures or install device drivers.

A few examples following the figure above should serve to illustrate uses of sub-identities.

Suppose that Alice attempts to log into the console. After consulting */etc/passwd*, *root* creates a new identity *alice* to run her programs. Alice then proceeds to work as normal. If she wishes to run any program that she does not fully trust, she may create a new sub-identity for that program. For example, if she has a visitor in the office, Betty, that wishes to use her machine, she may simply create a new user *betty*. This new identity protects Alice from any mishap by Betty, but it also gives Betty a clean workspace and the ability to store data under her own name and return to it later if needed. If Alice is browsing the web, she runs the risk of being attacked by malicious software. To defend herself, she may create a new user *browser* simply for the purpose of running a web browser. If the browser itself should be compromised, it will not be able to directly attack any programs owned by Alice or superior users. To go even further, the web browser itself might create an inferior user *webapp* in order to defend itself from any subprograms that are downloaded and run locally. The ability to create sub-identities allows for a multi-layered defense.

A web server can also make good use of sub-identities. Many powerful web services are implemented by running sub-programs from the web server. These programs are often hastily composed scripts and thus contain many security weaknesses [34]. A web server may defend itself by running each sub-process with a sub-identity. Each sub-identity may employ a meaningful name that allows it to access selected portions of

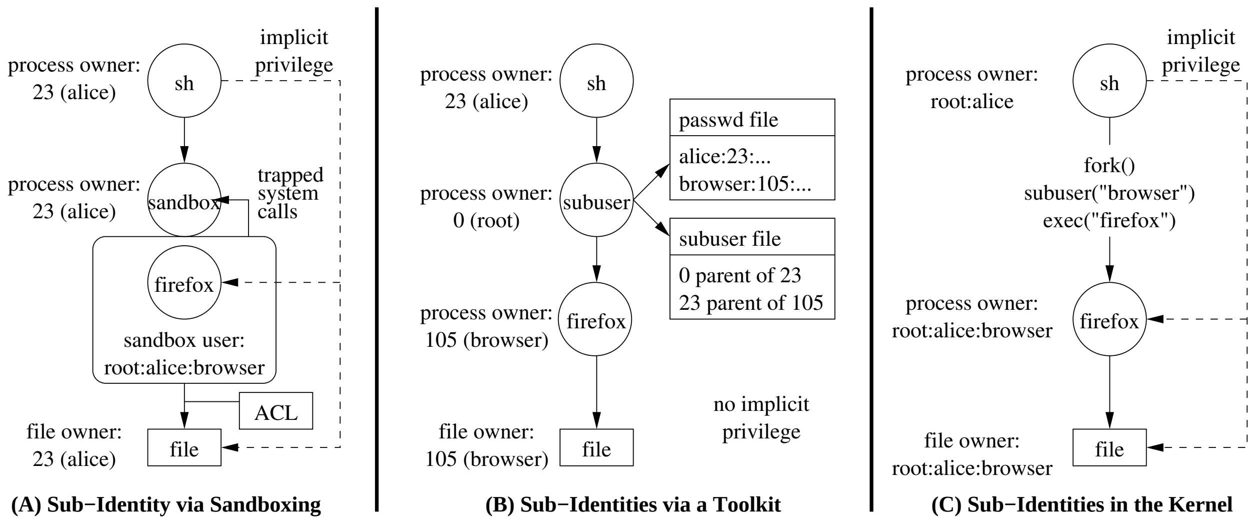

process owner: 23 (alice)
sh
implicit privilege
process owner: 23 (alice)
sandbox
trapped system calls
firefox
sandbox user: root:alice:browser
ACL
file owner: 23 (alice)
file
(A) Sub−Identity via Sandboxing
process owner: 23 (alice)
sh
passwd file
alice:23:...
browser:105:...
process owner: 0 (root)
subuser
subuser file
0 parent of 23
23 parent of 105
process owner: 105 (browser)
firefox
no implicit privilege
file owner: 105 (browser)
file
(B) Sub−Identities via a Toolkit
process owner: root:alice
sh
implicit privilege
fork()
subuser("browser")
exec("firefox")
process owner: root:alice:browser
firefox
file owner: root:alice:browser
file
(C) Sub−Identities in the Kernel

Figure 2. Three Implementations of Sub-Identity Compared
Three methods of implementing sub-identities are shown. (A) A sandbox can be used to emulate sub-identities in the kernel. More complex access controls are added to the filesystem by way of auxiliary ACL files. (B) A toolkit with setuid privileges can create and delete sub-accounts at run-time. This is more reliable than a sandbox, but has less flexible access control. (C) Kernel modifications allows for a reliable implementation with flexible access control.

the filesystem. For example, a database administrator might deploy data and make it accessible only by the *root:www:webdb* user, thus preventing access by other web applications. Or, the web server might choose sub-identities based the name of the host issuing the HTTP request, such as *root:www:laptop.cse.nd.edu*. Content developers could then use standard filesystem tools in order to control access to remote users. Most importantly, the web server no longer needs to run as the *root* user. If the web server itself is compromised, the entire system is not lost. The use of sub-identities defends servers from malicious clients, but it also defends the entire system from compromised servers.

Finally, consider how sub-identities simplifies the administration of a network authentication service such as Kerberos [39]. A traditional Kerberos installation has a globally shared user database, but it also requires the creation of local users in */etc/passwd* on each machine, corresponding in name and attributes to users in the global database. This is an enormous administration hassle for large sites. Sub-identities simplify the administration of network logins by divorcing the user database from the enforcement mechanism. Suppose that the *root* user on a workstation creates the necessary processes owned by *root:kerberos* to admit Kerberos logins. As users log in with Kerberos credentials, they are simply assigned new sub-identities such as *root:kerberos:david*. No interaction or coordination with the local user database is needed.

3 Implementation Choices

There are many possible ways of approximating this abstract model of sub-identity. In this section, we discuss how sub-identity could be implemented with a user-level sandbox, with a username toolkit, or within the kernel itself. Each of the techniques varies in semantics and implementation quality. Figure 2 shows how these three techniques differ on a common example. In each case, the user *alice* creates a sub-identity *browser* in order to run the web browser process *firefox*.

Sandbox Implementation. Sub-identities can be approximated by using user-level sandboxes. In a sandbox, an untrusted application is run under the control of a supervisor process that traps and examines all of its system calls, typically through the debugging interface. The supervisor may then accept or reject the attempted actions according to some security policy. Typically, the user must provide a mandatory access control (MAC) list that specifies the objects that the untrusted program is allowed (or not allowed) to access. By necessity, much of this specification deals with access to system files and libraries that the user may not be familiar with. This can be a significant burden for users that are not technically inclined.

A sandbox can be adapted to provide sub-identities by changing the policy controls within the supervisor process. Instead of consulting a MAC list, the sandbox is modified to carry a free-form identity string with the contained processes. This identity string is then used to enforce discretionary access control (DAC) on file system objects. This technique is known as *identity boxing*, and is described in futher detail in an earlier paper [40]. To create an identity box, the user simply invokes the sandbox with the desired identity and the program to be run. The identity may be any free-form string and need not correspond to any existing user name. Figure 2(A) shows how this works for Alice: she simply invokes *sandbox browser firefox*.

A process running inside the identity box is treated as if it possessed an identity completely distinct from the superior user. It cannot access files or manipulate processes owned by the superior user. It may only access files where given explicit permission by the filesystem. Unfortunately, Unix-like filesystems do not allow for sophisticated access control, so the identity box looks for files named *.__acl* in the filesystem to express more detailed policies on a per-directory basis. For example, Alice might put the following in *~/.firefox/.__acl* in order to allow *root:alice:browser* to read, write, and list files in the directory.

```
root:alice:browser   rwl
```

Of course, we cannot expect that the entire file system will be retrofitted with ACL files to support sub-identities. If a new directory is created within the identity box, it inherits the ACL (if any) of the parent directory. Where there is no ACL, the contained process is given access only to objects that are world-readable or world-writable. This also has the pleasant side effect that standard system files and libraries are accessible to sub-identities without any special handling.

Note that the isolation of a process in a sandbox is not symmetric. A superior user has *implicit privilege* over processes within a sandbox, and is able to send signals, modify address spaces, and directly manipulate files owned by the sub-identity. The only restriction is that an external process cannot debug a process within the identity box, as the debugging interface is controlled by the supervisor. Otherwise, the host operating system has no direct knowledge of the sub-identities and does not prevent superiors from modifying inferiors. This is consistent with the rules of the abstract model.

Toolkit Implementation. Alternatively, sub-identities may be approximated within a Unix-like operating system by employing a toolkit capable of modifying the local user database in */etc/passwd*. A secondary database in */etc/subusers* would record the relationship between users. The various tools would gain *root* privilege via the *setuid* facility as needed. Key commands in the toolkit would be:

subuseradd <**name**> Create a new inferior user by modifying */etc/passwd* and */etc/subusers*.

subuserdel <**name**> If the named user is inferior to the current user and has no inferior users of its own, then delete the entries in */etc/passwd* and */etc/subuser*.

subuser <**name**> <**command**> Run a command as the named inferior user.

subown <**name**> <**file**> Change the ownership of files to and from inferior users.

Figure 2(B) shows how this would work for Alice. She simply invokes the command line *subuser browser firefox*. The *subuser* command verifies that the *browser* user exists and is inferior to *alice*. Gaining *root* privileges via the *setuid* bit, it modifies the current userid to *browser* and executes *firefox*.

	Fidelity to Abstract Model			**Implementation Quality**		
Technique	**Names Allowed**	**Access Controls in File System**	**Implicit Privilege?**	**Deployment Difficulty**	**Code Complexity**	**Perf. Penalty**
Sandbox	arbitrary	acls	yes	user applied	high	per syscall
Toolkit	limited	unix mode bits	no	root install	low	per user
Kernel	arbitrary	acls	yes	kernel changes	medium	none

Figure 3. Implementation Techniques Compared

A toolkit implementation would deviate from the abstract model in three important ways. First, the hierarchical nature of the namespace is imperfect. Although *alice* is free to create sub-identities, she must choose identities that have not already been created. If *browser* exists anywhere in the identity tree, no other sub-identity *browser* can be created. In theory, the implementation could store fully-qualified names such as *root:alice:browser* in the databases. In practice, various system tools limit user names to 8 characters. Second, a user does not have implicit privilege over her inferior users. *alice* cannot directly manipulate processes and files owned by *browser*. She must use the *subuser* and *subown* commands to manipulate inferiors, although she is free to run arbitrary commands as sub-users, including administrative tools such as *kill* or *gdb*. Third, the filesystem access controls are very limited. Due to the limited expression of Unix mode bits, Alice cannot grant *root:www:laptop.cse.nd.edu* read access to certain files without copying them or giving away ownership entirely. Even if she could, such access controls could not be expressed until *root:www* had created the user *laptop.cse.nd.edu* and assigned it an integer user ID.

Kernel Implementation. To address the limitations of the sandbox and the toolkit, we may implement sub-identities within an operating system kernel. The limitations of the toolkit are closely related the use of integers to represent identities in the kernel, relying on an external user database to map numbers to names. A kernel implementation would abandon the use of integers and instead use free-form strings to represent identity in the kernel. For kernel-level permission checks – such as before sending a signal to another process – implicit privilege would be granted if the requesting user name was equal or superior to the target user name. Thus, *root* and *root:alice* and *root:alice:browser* would be allowed to manipulate processes owned by *root:alice:browser*. *root:alice:betty* would be denied access.

Figure 2(C) again shows how this works for Alice and her web browser. To create a child process with the sub-identity *browser*, the shell must fork a new process and then call *subuser(browser)*. This causes *browser* to be appended to the identity string of the current process.

To support this, filesystems would require adjustment to store free-form strings instead of integers in file access control entries. This would be particularly disruptive to traditional Unix filesystem designs, which rely on the identity to be a small data item that can fit into each inode structure. However, several recent filesystems have added support for "extended attributes" that allow larger data structures to be attached to files and directories. These structures may be used to added long identity strings and complex access control lists. Likewise, administrative tools would require modification to support the new identity format.

Comparison. Figure 3 summarizes the the properties of each technique.

The sandbox offers a close approximation to the desired semantics because it makes all system calls available for modification. Arbitrary sub-identities may be associated with processes, and complex access controls may be placed on directories. In addition, unprivileged users may install and apply the sandbox while retaining implicit privilege over sub-processes. However, as others have observed [18], the correct implementation of a sandbox is no small matter, rivaling an operating system kernel in subtlety and complexity. Despite the best intentions of its designers, it is difficult to believe that the code quality of a sandbox will receive the same scrutiny and achieve the same quality as kernel code. In addition, the sandbox imposes a performance penalty on each system call due to the numerous added context switches.

The toolkit technique falls short of the abstract model in several ways. The namespace is restricted, users are constrained by Unix access controls, and implicit privilege is not provided on sub-processes. In addition, a toolkit must be installed by the local administrator. However, a toolkit would be the simplest of the three techniques and could be small enough to be scrutinized and widely trusted. The toolkit only interposes on user *changes* and thus does not impose the performance penalty of the sandbox.

A full-blown kernel implementation would allow precise adherence to the abstract model without the implementation difficulties or performance overhead of a sandbox. However, it would also require the greatest implementation effort and present a significant obstacle to user adoption.

Considering these tradeoffs, we have chosen to first implement and explore sub-identity using the sandbox technique because it allows for the greatest fidelity to the abstract model while preserving compatibility with existing systems. By modifying an existing sandbox, we are able to quickly provide the desired semantics and explore how real applications may take advantage of sub-identities. Implementation quality is of less concern for a research prototype. A later paper will explore a toolkit implementation, which provides lower fidelity but higher implementation quality.

4 Applications of Sub-Identity

To demonstrate the power and simplicity of the sub-identity model, we applied it to two distributed computing tools: a web browser and a secure login server. The web browser uses sub-identity to safely execute programs downloaded from untrusted sources. The secure login server uses sub-identities in order to simplify integration with a distributed authentication system. In both cases, the necessary code changes to employ sub-identity were minimal.

We have chosen the sandbox technique for these experiments for two reasons. First, we are familiar with an existing sandbox, Parrot, which we have used for a variety of purposes in grid computing [40, 25, 41]. Modifying Parrot to support identity boxing only required 407 lines of new code. Second, the sandbox technique allows us to experiment with arbitrary semantics without requiring any special privilege.

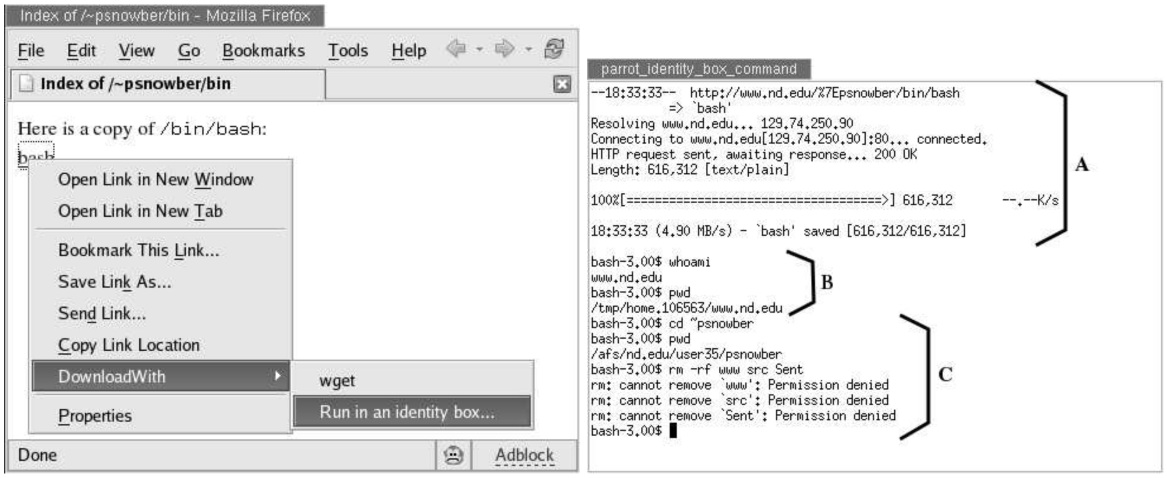
Briefly, Parrot works as follows. It runs the processes to be contained as children and traps their system calls through the *ptrace* interface. As each system call is captured, it is executed on behalf of the calling application, much as in Ostia [20]. Whenever a filesystem object is accessed by name, Parrot looks for a *.__acl* file and implements the access control described above. If a system call must be denied, the caller's registers are modified to return an error result. (Parrot is implemented on Linux, where this is possible. Some operating systems do not allow this.) To run a program in an identity box via Parrot, one simply invokes Parrot with an extra command-line argument indicating the identity to be used.

4.1 Untrusted Web Browsing

It is a common paradigm to design a computationally expensive application to be downloaded over the internet and executed on users' systems. However, because of lack of trust between the user and the host, or between the user and the medium, users want to run such applications inside a protection domain, so their systems are not compromised. Running the untrusted application as a sub-user inside an identity box provides such a protection domain.

We extended a web browser to automatically create a sub-user in an identity box when the user clicks on an executable file. The application is then downloaded from the web page and run as the subuser. Each time the application is run, if downloaded from the same server, it is run as the same sub-user. Since files owned by that user persist between invocations, the application has access to any files that it created in the previous invocation. They can also support multiple applications concurrently accessing the identity box.

We chose to extend the Firefox web browser, which is particularly well-suited to being extended, since the designers took pains to implement the entire user interface as markup that is interpreted at runtime.



```
--18:33:33--  http://www.nd.edu/%7Epsnowber/bin/bash
           => `bash'
Resolving www.nd.edu... 129.74.250.90
Connecting to www.nd.edu[129.74.250.90]:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 616,312 [text/plain]

100%[===================================>] 616,312      --.--K/s

18:33:33 (4.90 MB/s) - `bash' saved [616,312/616,312]

bash-3.00$ whoami
www.nd.edu
bash-3.00$ pwd
/tmp/home.106563/www.nd.edu
bash-3.00$ cd ~psnowber
bash-3.00$ pwd
/afs/nd.edu/user35/psnowber
bash-3.00$ rm -rf www src Sent
rm: cannot remove `www': Permission denied
rm: cannot remove `src': Permission denied
rm: cannot remove `Sent': Permission denied
bash-3.00$ █
```

Figure 4. Applying Sub-Identities to Untrusted Web Browsing
We have modified Firefox to allow the user to run untrusted programs with a sub-identity. On the left is Firefox with the DownloadWith extension enabled, and the new menu item highlighted. Clicking the menu item creates the terminal on the right, running the program which was clicked on. In callout **A***, the program is shown being downloaded.* **B** *shows that the process is running as a sub-user named* `www.nd.edu`. **C** *shows a failed attack against the superior user* `psnowber`.

Extensions are able to easily modify the user interface by adding buttons or menu items that call Javascript functions when activated. *DownloadWith*[1] is an extension that allows a clicked-on URL to be sent to an arbitrary command on the local system. Using this extension, we enabled users of Firefox to select "Run in an identity box..." from the context menu of any link. In total, this application of identity boxing required writing two dozen lines of code as a wrapper script.

We take the fully qualified domain name of the server the application was downloaded from as the name of the sub-user. However, if the application were hosted on a secure web page, the name of the sub-user should be taken from the X.509 certificate presented during the secure connection negotiation. This would be a better solution, because it would make identity spoofing attacks more difficult.

This solution gives protection from potential attacks by blocking access to resources the untrusted application shouldn't have access to. As shown in Figure 4, when a malicious application attempts to vandalize the system, the identity box it is running inside prevents the attack from succeeding. This is accomplished by checking every file access against the ACL file; if there is no ACL file in the directory containing the accessed file, then the identity box interprets the file's "other" UNIX permissions bits.

The question of garbage collection arises when considering this application: some applications' performance could benefit from cacheing downloaded data and program state locally to ease network traffic, so it makes sense to support persistent protection domains. This support exists trivially in our identity boxing solution by merely not deleting the space associated with the sub-user. The question, then, is "when should hard drive space given to a sub-user be reclaimed?" This is an open question, subject to policy based on resouce availability of the local system as well as the needs of individual applications. Our implementation does not automatically reclaim space, instead leaving the task to the user.

[1]`downloadwith.mozdev.org`

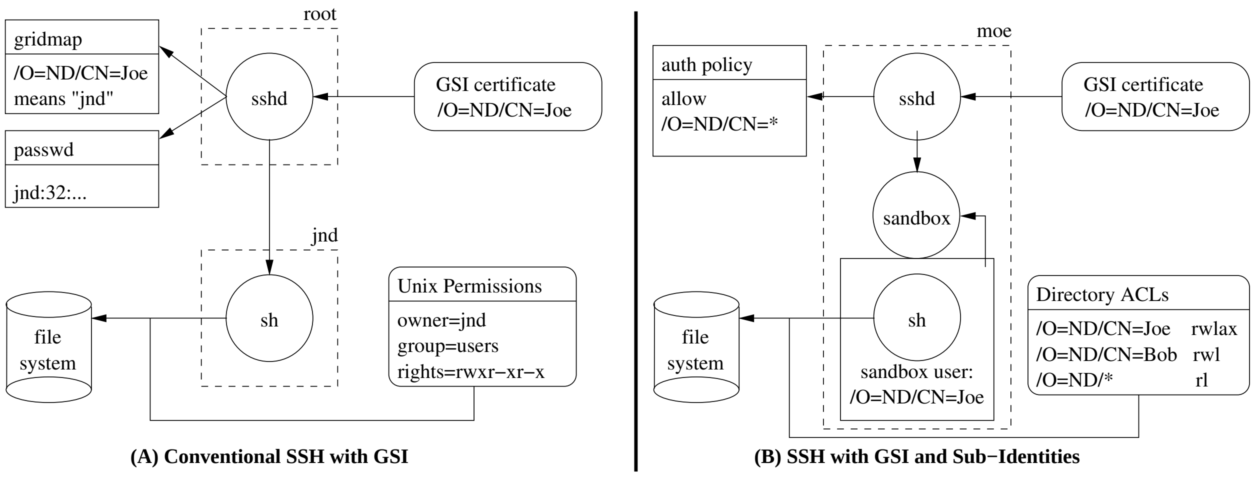


Figure 5. Comparison of Secure Login With and Without Sub-Identities
(A) Without sub-identities, a local administrator must create local accounts corresponding to all possible remote users. GSI credentials such as `/O=NotreDame/CN=Joe` *are mapped to local accounts such as* `jnd`*. The server must run as* `root` *and users are constrained to the Unix permission model. (B) With sub-identities, a simple policy states the set of users to be admitted. Sub-identities corresponding to remote username are generated on the fly. Directory ACLs with meaningful names are used to share data.*

4.2 Secure Login with GSI Credentials

Grid computing, broadly speaking, is the concept that large scale computing power should be as easy to access as the electrical grid [15]. Today, several large scale computing grids exist and provide thousands of CPUs to hundreds of scientific users [17, 5]. These systems rely upon the GSI toolkit to identify each user with a globally unique name like */O=NotreDame/CN=Joe* [16]. However, each site must maintain a local *gridmap* file that maps GSI identities to local usernames [11]. Maintaining this file for all machines and all users is an enormous hassle and has led to the old insecure standby of sharing accounts between users [17]. Sub-identities can be used to simplify the administration of such systems, as well as simplify the interactions between users. We demonstrate this by employing sub-identities with a GSI-enabled secure shell server.

Upon receiving a connection, this modified *sshd* verifies the GSI certificate, creates a sub-user inside an identity box, and then starts up a shell as that sub-user, and hands control of it to the connecting user. The name of the sub-user is derived from the subject contained in the certificate presented by the connecting party, transposing characters that aren't valid or are inconvenient in a UNIX filesystem to benign characters. (i.e. slashes or equal signs become underscores.) The potential for collisions exists, but is extremely small: a pair of Distinguished Names would have to differ only in special characters. Determining the sub-user name from the subject also allows entities to return to their sub-user, enabling them to leave temporary files and caches behind, saving network bandwidth. The problem of garbage collection of sub-users reappears here. Again, our implementation leaves the garbage collection up to the user running the daemon. In our example, Alice might decide to delete all her sub-users that have not been accessed in the last year.

This system would be little more than a curiosity if it allowed just anybody with a GSI identity to connect, since anybody can start their own certifying authority and mint certificates. However, by simply adding a mechanism to accept or deny connections based on properties of the connecting client's certificate, the user running the *sshd* is given flexible and powerful control over who may consume their resources. Using this mechanism, it would be possible to define and enforce policy as general as accepting or denying any connections from clients with certificates signed by certain certifying authorities to as specific as allowing a

(A) system call overhead			(B) application overhead		
syscall	**unmodified**	**w/sandbox**	**appl**	**unmodified**	**w/sandbox**
getpid	$0.36 \pm 0.01\ \mu s$	$13.15 \pm 0.10\ \mu s$			
stat	$2.04 \pm 0.21\ \mu s$	$49.79 \pm 1.17\ \mu s$	**gzip**	18.89 ± 0.06 s	19.33 ± 0.12 s
open/close	$4.68 \pm 0.33\ \mu s$	$68.66 \pm 1.47\ \mu s$	**tar**	0.18 ± 0.05 s	1.90 ± 0.09 s
read 1B	$1.17 \pm 0.49\ \mu s$	$23.01 \pm 0.93\ \mu s$	**make**	3.58 ± 0.01 s	5.51 ± 0.01 s
read 8KB	$2.61 \pm 0.35\ \mu s$	$25.88 \pm 0.81\ \mu s$	**encode**	17.28 ± 0.11 s	17.57 ± 0.59 s
write 1B	$6.33 \pm 1.14\ \mu s$	$32.43 \pm 1.13\ \mu s$	**acroread**	0.66 ± 0.04 s	1.25 ± 0.07 s
write 8KB	$13.28 \pm 0.57\ \mu s$	$41.59 \pm 1.14\ \mu s$			

Figure 6. Overhead of Sandbox Implementation
The table on the left show that individual system calls are slowed by an order of magnitude, due to the large number of context switches caused by the ptrace interface. The table on the right show the impact on real applications. CPU-bound codes such as gzip and oggenc are barely impacted, while system-intensive applications are more heavily affected. The toolbox and kernel implementations would have less overhead.

trusted subject to connect.

This version of the SSH daemon can and should be run without privilege; only a normal user account is required to run it. The ability to run it without privilege comes directly from giving normal users the power to create sub-users. If two users on a system were to run this modified *sshd*, the sub-users' storage would be set up in independent namespaces, as subdirectories of */tmp* with names based on their user ids. Because of this, two users can run the daemon on separate ports, and if the same client connects to each, the client is given access to two distinct sub-users, for instance *root:userA:Name* and *root:userB:Name*. In essence, all the connecting and authorized clients are granted access to sub-users that are subordinate to the user running the daemon.

This approach eliminates administrator involvement in the creation of users. Holders of GSI credentials are able to log in even without knowledge of the administrator. To the nervous system administrator, this may seem like it allows for abuse of system resources. However, any user can run their own telnet server on any UNIX system and invite their colleagues to use their account. It is up to the user running the modified daemon to enact policy to determine who is to be granted access. This allows the user control over the safety of his own account.

Because the name of the sub-user is derived from the subject of the connecting party, as long as every installation of the modified *sshd* agrees on how they derive the name, then the holder of a certificate can be guaranteed that he will have the same user name on every server he connects to, rather than potentially having a different user name everywhere and being forced to remember them.

4.3 Performance

Overhead is expected in any sandboxing system. In the case of identity boxing, the additional security gained by inspection of system calls is earned with a performance compromise. Each system call inspected incurs at least six context switches. This overhead adds up across many system calls, reducing the performance of certain applications.

In order to determine the overhead incurred on a real system by using identity boxing for various applications, we used a benchmark that timed 100 cycles of 10,000 iterations of various system calls on a 2.8 GHz Intel Pentium 4 running Linux 2.6.9. To ensure that disk latency is not a factor, the files touched by each system call are read from the disk in their entirety so that they are cached in main memory. As we can see from Figure 6(A), overhead slows system calls down by an order of magnitude or worse.

To get a better idea of whether this system call overhead disqualifies identity boxing as a protection mechanism in real systems, we ran some applications inside and outside an identity box, and collated the results into Figure 6(B). The tests ran were: **gzip**, which compressed a 200 MB file using GNU gzip; **tar**, which built a tape archive of the Linux 2.6.12 kernel tree; **encode**, which encoded a WAV file into Ogg Vorbis; **acroread**, which measured the time necessary to start the Acrobat Reader on [40]; and **make**, which built GNU gzip from source. Our results show that there is minimal performance impact on CPU-bound workloads. There is, however, a marked increase in the time it takes to run the **make** benchmark, because of the large number of system calls GNU make uses to decide whether to rebuild a target. Identity boxing can therefore be seen as providing excellent protection at a measureable cost for varied workloads. For instance, Adobe Acrobat Reader, despite being noticeably slower at startup, is perfectly useable while running inside an identity box.

5 Related Work

The concept of sub-identities is inspired by the use of hierarchical name spaces in many systems, including the domain name system [30], Lampson's authentication framework [26], and in public key infrastructures [7], just to name a few. But to our knowledge, this concept has not been proposed as an *enforcement mechanism* within an operating system. The closest system might be SubOS [22], which allows for a single level of sub-identity to be attached to objects; processes then run with the minimum privilege of the objects they access. However, a variety of other work is relevant.

Operating Systems. Two mechanisms in MULTICS [35] inspire this work. First, the GE-645 provided *protection rings* allowing each program to manipulate data in high numbered rings at will and manipulate lower numbered rings only through well-defined call gates. Sub-identities could be thought of as protection rings with names and branches. Second, MULTICS has tripartite user names in the form of *user.project.compartment* that would be called user, group, and role in today's terminology. The freedom to change roles and compartments could be used to approximate a 3-level hierarchy.

Unix has been the target of several efforts to integrate new access control methods while retaining the existing user names. This generally involves adding a new reference monitor into the kernel and providing a sufficient database of access controls. For example, TRON [4] adds capabilities to Unix, Flask [38] and REMUS [6] add mandatory access control, DTE-Unix [2] adds domain and type enforcement, and SELinux [31] adds both mandatory and role-based access control. In all of these cases, the existing notion of user identity is preserved, while the method of access control changes. The result is that the system administrator is given powerful new ways of controlling users, but the users themselves gain no ability to manage their own security environment.

Privilege Separation [33, 8] is a technique for giving a process the minimum administrative rights necessary to run. A common example is the need for a login server to call *setuid*. With privilege separation, the server process runs as an untrusted user, calling out to a security kernel when a privileged operation is required. The security kernel performs it on behalf of the server, if allowed. The toolkit implementation of sub-identities is an example of privilege separation.

Sandboxing is a common research technique for running untrusted programs. A supervisor process is responsible for running an untrusted program while checking its external operations via a reference monitor. The trapping technique may be the debugging interface [32, 20, 41], a kernel module [21], system-call reflection [23], or binary rewriting [24]. In addition to exploring trapping methods, various sandboxes have explored containment policies, such as associating rights with programs [10, 1], with data [22], or by deferring writes into a transaction which can be audited after execution [29].

Sandboxing is an excellent technique for prototyping and developing new concepts in access control. However, for production use, it is no replacement for a containment facility within the operating system. As we (and many others) have shown, there is a significant performance penalty to sandboxing by trapping system calls. More importantly (and less well known,) is that sandboxing mechanisms rarely achieve the completeness and reliability of an operating system kernel. As Garfinkel has noted [18], the *ptrace* interface between the supervisor and trapped process is extraordinarily complex and subtle. For these reasons, few sandboxes support the full range of system calls, and many do not support multiple processes, multiple threads, or other complex interfaces. Thus, we consider sandboxing a valuable research technique, but not a substitute for an operating system facility.

Virtual Machines. Various applications of virtual machines have been used for distributed computing, including process migration [37, 42], service construction [13, 9], service composition [14, 22, 12], and isolation for security[19, 27]. The virtual machine is a valuable technique to apply when one wishes to run a program (or an entire system) in complete isolation from the calling process.

One could approximate sub-identities with a hierarchy of nested virtual machines. This technique is suggested by Ford et al [14]. Although this would provide containment, it would not be very usable. 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 prevents our target audience — the ordinary user — from creating new protection domains for their own purposes. Second, the virtual machine inhibits sharing entirely: users that run untrusted programs generally *want* those programs to interact with the system in a limited way, by manipulating the filesystem, communicating with other processes, or using the network. A kernel implementation of sub-identities would combine the assurance of the virtual machine model with the usability of a simple process model.

6 Conclusion

Only by permitting the individual user some control of his own administrative environment can one insist that he take responsibility for his work. - J. Saltzer [35]

Sub-identity allows the ordinary user to take charge of his security environment. We have presented an abstract model of sub-identity and described three possible implementations of that model, each varying in fidelity and complexity. We have also demonstrated that one implementation of sub-identity using user-level sandboxes is feasible, although it comes at some cost in performance.

Our experience in implementing sub-identity in concert with a web browser and a secure login system demonstrates several things. First, sub-identities are easy to use! A simple user interface added to a web browser is accessible to nearly any user, unlike many other complex security technologies. Second, very few code changes were necessary to implement these facilities. By experiment, we have demonstrated that our implementation of sub-identities protects superior users from undesired actions by inferior users, such as unauthorized deleting of files. In a distributed system, sub-identity allows both users and administrators to express complex sharing policies. In fact, users can now also be administrators! Finally, sub-identity simplifies distributed systems by allowing users to employ their global identities as local identifiers.

Our next step is to explore alternate implementations of the sub-identity concept. We intend to build a username toolkit as described above. This will provide a simpler, less powerful model of sharing, but will be an implementation more agreeable to traditional Unix and will have little or no performance overhead. We will deploy this toolkit with a variety of applications and users and instrument it to understand how real users take advantage of this facility in practice. Armed with this experience, we will proceed toward a full blown kernel implementation.

Ultimately, sub-identity allows users to protect themselves in a potentially dangerous environment. In conventional systems, the user is an innocent participant that relies on the administrator for protection. In a networked environment, this is no longer acceptable. Users must have tools that allow them to take responsibility for their own safety. Sub-identity is step towards providing this capability within the operating system.

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] L. Badger, D. F. Sterne, D. L. Sherman, and K. M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1):47–83, 1996.

[3] A. Bavier, S. Karlin, S. Muir, L. Peterson, T. Spalink, M. Wawrzoniak, M. Bowman, B. Chun, T. Roscoe, and D. Culler. Operating system support for planetary scale network services. In *Networked Systems Design and Implementation*, 2004.

[4] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the unix operating system. In *USENIX Technical Conference*, 1995.

[5] F. Berman. From TeraGrid to knowledge grid. *Communications of the ACM*, 44(11):27–28, 2001.

[6] M. Bernaschi, E. Grabrielli, and L. Mancini. REMUS: A security-enhanced operating system. *ACM Transactions on Information and System Security*, 5(1):36–61, February 2002.

[7] M. Blaze, J. Feigenbaum, and J. Lacy. Decentralized trust management. In *IEEE Symposium on Security and Privacy*, May 1996.

[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] 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.

[12] B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, I. Pratt, A. Warfield, P. Barham, and R. Neugebauer. Xen and the art of virtualization. In *Symposium on Operating Systems Principles*, 2003.

[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] 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.

[15] I. Foster and C. Kesselman, editors. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann, 1998.

[16] 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.

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

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

[19] 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.

[20] 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.

[21] 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.

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

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

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

[25] S. Klous, J. Frey, S.-C. Son, D. Thain, A. Roy, M. Livny, and J. van den Brand. Transparent access to grid resources for user software. *Concurrency and Computation: Practice and Experience*, to appear.

[26] B. Lampson, M. Abadi, M. Burrows, and E. Wobber. Authentication in distributed systems: Theory and practice. *ACM Transactions on Computer Systems*, 10(4), November 1992.

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

[28] J. Lepreau, B. Ford, and M. Hibler. The persistent relevance of the local operating system to global applications. In *SIGOPS European Workshop*, 1996.

[29] Z. Liang, V. Venkatakrishnan, and R. Sekar. One-way isolation: An effective approach for realizing safe execution environments. In *ISOC Network and Distributed System Security*, 2005.

[30] P. Mockapetris and K. Dunlap. Development of the domain name system. In *Proceedings of SIGCOMM*, volume 18, pages 123–133, April 1988.

[31] National Security Agency. Security enhanced linux. http://www.nsa.gov/selinux, 2005.

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

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

[34] A. Rubin, R. Geer, and M. Ranum. *Web Security Sourcebook*. John Wiley and Sons, 1997.

[35] J. H. Saltzer. Protection and the control of information sharing in multics. *Communications of the ACM*, 17(7):388–402, July 1974.

[36] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proc of IEEE*, 69(9):1278–1308, September 1975.

[37] 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.

[38] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The Flask security architecture: System support for diverse security policies. In *USENIX Security Symposium*, August 1999.

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

[40] D. Thain. Identity boxing: A new technique for consistent global identity. In *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 *Workshop on Adaptive Grid Middleware*, New Orleans, September 2003.

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