# Pastures: Towards Usable Security Policy Engineering

Sergey Bratus, Alex Ferguson, Doug McIlroy, Sean Smith
Dartmouth College

## Abstract

*Whether a particular computing installation meets its security goals depends on whether the administrators can create a policy that expresses these goals—security in practice requires effective* policy engineering. *We have found that the reigning SELinux model fares poorly in this regard, partly because typical isolation goals are not directly stated but instead are properties derivable from the type definitions by complicated analysis tools. Instead, we are experimenting with a security-policy approach based on copy-on-write "pastures", in which the sharing of resources between pastures is the fundamental security policy primitive. We argue that it has a number of properties that are better from the usability point of view. We implemented this approach as a patch for the 2.6 Linux kernel.*

## 1 Introduction

Computing systems typically depend on their operating systems to enforce trustworthy behavior. Architects and administrators describe this behavior in terms of various security goals, such as protecting the integrity and confidentiality of data. To achieve these goals, architects and administrators should be able to configure them with security policies that actually express these goals. Thus we reach a point when policy engineering becomes a critical issue.

The problem of engineering usable OS security policies has been worked on for decades. However, we claim that age does not imply maturity. The continuing trouble with securely configuring real systems for real applications in the real world demonstrates the absence of the accepted and effective best practice that would come with a solved problem. Approaches based on formal models do not appear to have gained much traction with system administrators and defenders, outside of a few small specialized communities. Satisfactory approaches to securing systems in manageable ways are still being pursued and discussed[1], and we will later review several such approaches that enjoy de facto practitioner acceptance (most notably *vserver* and the use of *BSD jails* for building virtual systems). We offer our own work as a contribution to this effort.

In recent years, older ideas have re-emerged, with NSA's *Security-Enhanced Linux (SELinux)* (e.g., [11]) considered by many to be the de-facto best-of-breed for those wanting a high-assurance but contemporary OS. Unfortunately, SELinux has a high cost from the point of view of usability: its monolithic and awkward policy structure makes it difficult for programmers to configure and maintain it for real-world applications – and difficult for stakeholders to trust that the resulting policy actually confines system behavior to "secure" operation only.

Previous studies (e.g., [4]) led us to the conclusion that the principal usability obstacles for the SELinux policy approach are:

- any reasonable degree of integrity protection requires a large and complex policy, essentially profiling all the allowed accesses for protected applications;
- by following the application execution profile, such a policy becomes as complex as the original software – without any corresponding software engineering tools to keep it manageable;
- no protection can be afforded before such profiles are compiled, a labor-intensive procedure.

All of these obstacles are ultimately due to SELinux's reliance on the fundamental and time-honored design principle of *denying access if it is not explicitly permitted by the policy*.

In addition to these obstacles, checking if a SELinux policy satisfies information flow goals is a non-trivial task, because flow properties of the SELinux model cannot be expressed explicitly in its policy language, but are instead derived from the type specifications, i.e. access profiles. Thus even though flow goals are often of primary interest, they are not first-order policy objects that can only be derived from a rather large number of access statements.

We explore an alternative approach, based on two principles:

[1] In particular, practitioner conferences, including the so-called hacker conventions like Defcon and Blackhat, often feature talks on exploratory methods for protecting common server software known to be vulnerable.

- All accesses not explicitly allowed or denied by a policy statement result in a *copy-on-write (COW)* duplication of the accessed object, rather than a denial.
- Flow properties are directly specified by the policy rather than derived from other kinds of statements. Specifically, sharing of resources between environments isolated by default becomes a basic policy primitive.

In order to implement these principles, we assign protected applications to *COW pastures* or, for short, *pastures*, where a *pasture* is a security context similar to a *vserver* context or a *BSD jail,* which contains private copies of files modified by processes assigned to it. The term, of course, is a pun on COW, the copy-on-write operation that adds new objects to a pasture. The term also suggests the permissive nature of a pasture as compared to a "jail": the point of pastures is to allow programs to proceed by creating private copies of objects modified by their writes not explicitly listed in an access profile, rather than terminating them for a violation of the profile.

# 2 Our approach

## 2.1 Revisiting types

We consider the set of SELinux types as a set of intersecting "access jails", defining their allowed interactions. Indeed, the purpose of SELinux policy analysis tools is mostly to *derive* a description of such interactions (e.g., the information flow between types). It seems that a direct specification of allowed flows combined with the default integrity protection given by the copy-on-write namespace isolation approach, could be much more concise and intelligible, being closer to the actual security goals of many administrators. The principal reduction in complexity will come from the conceptual simplicity of namespace isolation-by-default.

## 2.2 Motivating Examples

Let us consider two motivating examples.

**Server protection**

An administrator runs a number of trusted and possibly interacting servers on a system wishes to introduce a new piece of third party software, say a license server. Not being sure of the security properties of this software, the admin would like to take steps to protect the integrity of his other servers from its possible interference, whether unintentional or due to hostile transactions.

**Client protection** A rich client such as a web browser needs to access and interact with many OS resources, and read and create multiple files, including files in the user's home directory (cache, javascript, cascading stylesheets, media files etc.). A vulnerability in one of its modules could damage the user's private files. Due to its significant footprint, it typically needs a complex security profile with many special cases.

Users nowadays urgently need to maintain separate roles (and effective namespaces) for "browsing" and "working", or face dangers to more valuable private files from the much less valuable "web" files.

Internet Explorer's reasonable idea of "zones" probably owed something to such considerations, but was ruined by the "integration" with the OS and other design decisions that the current "phishing" epidemic has taken such broad advantage of.

Hence client software is best run in an isolated compartment. However, for most user activities, full isolation is not an option (e.g., a web browser is needed to interpret the very working files that we would like to protect from being damaged by it, should it misbehave). However, we note that the number of files that actually need to be imported back to the user's working set is typically much smaller than the total number of files read or written within a session.

Adjusting the policy to fit the quickly changing working environment is hardly a usable solution when the policy is to deny by default all operations that are not explicitly allowed. A denial could well lead to an error and loss of work in the current session.

Thus using a "deny by default" policy means too many *administrative interruptions*, when the user has to switch roles to perform an administrative action, and then spend some time restoring his working context. To avoid such interruptions users often work with elevated privileges all the time. An extreme example was older MS Windows where administrative actions could not be easily taken without logging out and back in as the Administrator.

## 2.3 The proposed solution

The essence of our approach is to allow groups of related programs to run in "pastures" with the same namespaces as the original system, by default creating private copies of resources when they are modified and describing the cases when this should *not* happen in the policy that enumerates allowed sharing and communication between pastures. In each pasture, the bindings in the namespace thus point either to the original object (if the policy specifies this), or to the private object (created by default if written to by the pasture).

Pastures sacrifice some of SELinux's power for detecting misbehavior but avoid the burden of detailing "good" behavior before protection becomes effective. The effects of bad behavior are managed by keeping them private to the program's pasture.

Advantages of pastures are

- new progams can be introduced in protected manner, eliminating the need for the risky *audit2allow* profiling step;
- architects and administrators can *directly specify* information flow properties;
- the overall complexity of the policy is thereby reduced; and
- access error conditions fatal under SELinux enforcement but not critical to security goals are no longer fatal.

# 3 The implementation

## 3.1 Overview

Processes start in pasture environments specified by a policy mapping of the pair *(command, user)* $\rightarrow$ *pasture ID*. The *command* is either an absolute path to an executable or a *sudoers(5)*-style alias to a command with options[2]. Our code resolves the mapping of a process to its appropriate pasture at the point of the *exec* system calls.

The system creates new pastures as needed. Each new pasture starts out with the namespace identical to that of the base system, perhaps with some part of the namespace explicitly excluded by the policy.

The implementation affects name-resolution mechanisms, such as *namei* for the filesystem. Name-resolution routines now take one extra argument, the ID of the pasture, and return the most specific binding that exists for this pasture. If a file or directory has been modified within a pasture privately, the namespace mappings of the particular pasture and the other pastures diverge at the corresponding directory level, and *namei* henceforth produces the private copy of that file, as the most specific for that pasture.

Besides the filesystem, we need consider other namespaces as well, such as those of network ports and various system and network stack constants. Our current implementation is limited to filesystems, but the same approach should be generalized to these other namespaces. To confront the practical challenges connected with virtualizing network stacks, we plan to draw on the wisdom of the *Vimage* project[3] for FreeBSD [15].

[2]The sudoer-style matching and aliasing are not yet implemented

[3]http://www.tel.fer.hr/zec/BSD/vimage/

## 3.2 Pastures policy primitives

### 3.2.1 The one-directional arrow

Suppose $G$ and $A$ are two pastures. The policy statement $G \rightarrow A$ means that *changes made to the namespace of pasture G by processes running in G are visible to processes in the pasture A, but such changes by A are not visible in G*. (If the policy also specifies the converse $A \rightarrow G$, we write $G \leftrightarrow A$; we consider this simpler case later.)

Let us return to our previous example of a sysadmin introducing a third-party application $A$ to a system that runs a number of trusted servers, which he wants to protect from possible interference by $A$. In other words, he wants his trusted processes to affect $A$ as they normally would, but he does not want $A$ to affect them back in any ways other than the few prescribed. Eventually, as the admin convinces himself that $A$ is well-behaved, he will bring $A$ into the trusted fold, but before that time he would like to be able to roll back most changes that $A$ may make to the system, except those specifically allowed by his policy.

To achieve this goal, the admin may regard his set of trusted servers as running in a global system pasture $G$, create a pasture $A$ for the new application, and specify $G \rightarrow A$.

The arrow relation between two pastures can be limited to certain files, as described below.

### 3.2.2 The bidirectional arrow

The bidirectional arrow between pastures means that references to a resource, such as a file, by name from either pasture would return the same object. The applications running in these pastures can create the same races as in classical UNIX, and these would be resolved or allowed to exist as in classical UNIX.

With a bidirectional arrow relation qualified by a file name, applications running in two separate pastures can thus share a regular file, or communicate through a socket file by that name. An unqualified bidirectional arrow between pastures would mean that they are functionally identical in their access privileges.

## 3.3 Information flow implications

The information flow meaning of the unidirectional arrow $A \rightarrow B$ is as follows: the *writes* of processes in $A$ are visible to those in $B$, but not vice versa. Changes to files made by processes in $B$ and new files created by processes in $B$ are not visible to those in $A$; $B$ can freely read information from $A$, but its writes will not affect resources in $A$, since they result in copy-on-write of the resource written to.

The bidirectional arrow $A \leftrightarrow B$ (or, equivalently, a pair $B \rightarrow A$ and $A \rightarrow B$) means that information flow via

writes can occur in both directions, since both $A$ and $B$ essentially share the same object.

We note that, in this prototype, we are mainly worried about integrity (keeping untrusted software from interfering from trusted software) rather than confidentiality. We left read access controls out of our policy language so as not to distract from the core idea of using copy-on-write as an integrity policy primitive. We plan to consider these challenges in future work.

### 3.4 Additions to the /proc filesystem

We exchange and expose the additional data needed for managing the pastures to the kernel through the */proc* filesystem. In particular, the policy is loaded by writing null-separated records to */proc/pastures* and is visible by reading it once loaded. Each process that runs in a pasture has its ID in */proc/PID/pasture* and its list of private copy-on-write files in */proc/PID/cows*. By reading this information, the administrator can examine the state of the system.

### 3.5 Policy specifications

The policy consists of

1. statements specifying instantiation of pastures and placement of programs into a particular pasture, existing or newly created, on startup, and
2. statements describing allowed resource sharing and information flow.

**Example** Consider the policy (=> reads "place in"):

```
/usr/sbin/httpd => servers
/usr/sbin/ftpd  => servers
```

These statements specify that *httpd* and *ftpd* will get placed into the same pasture and share access to all resources. When one of them starts up, it will cause the pasture to be created. If the other program is started after that, it will be placed into the existing instance of the *servers* pasture. The identity of the user (UNIX or SELinux) who started the program is irrelevant in this example.

**Example** We can write policies that take identity into account:

```
/opt/matlab:{usr1,usr2} => shared_math
/opt/license_server     => shared_math
```

When users *usr1, usr2* start *matlab*, they will be placed in a shared environment where they could influence each other's work. Although the same data access arrangement can be described with other access control mechanisms, the choice of Matlab is not arbitrary. In order to run, *matlab*, a proprietary program, requires a separate third-party proprietary program (a "license manager") to be running on the computer (or somewhere on the same network). The user may not fully trust either of these programs and, accordingly, may want to isolate them from the rest of the system.

**Example** We can arrange per-user pastures:

```
/usr/local/bin/mozilla
   => \$user/untrusted_browser
```

This statement specifies that each user starting *mozilla* will automatically have it placed into a private pasture separate from his normal user environment.

**Example** We can use policy statements to restrict information flow via writes.

```
A -> B:
  /path/to/fileA
```

Here *A* and *B* are specifications of individual pastures or groups of pastures.

This statement (with the unidirectional arrow) allows information to flow from pasture *A* to pasture *B*. When *B* issues a read access request for *fileA*, it will receive the object bound to *fileA* in *A*'s space. When a process in *B* writes to *fileA*, a private copy of *fileA* will be created in pasture *B*'s namespace. Thus after writing to *fileA* pasture *B* will receive its own private copy of the object, in its last state.

**Example** We can also express bidirectional flow:

```
A <-> B:
  /path/to/fileA
```

Both *A* and *B* will be work on the same object *fileA*. (This will also occur if the policy contains both $A \rightarrow B$ and $B \rightarrow A$ for some resource.)

More precisely, if *fileA* is not shared with the global system pasture (in which case the system's only copy will simply be used in any pasture which which it is shared), then whichever process from *A* or *B* first writes to *fileA* will cause a new object to be created and bound to *fileA* in the respective namespace. For the other pasture, all references (read or write) to *fileA* will henceforth resolve to that new object.

### 3.6 Implementation Details

We modeled our implementation on the *vserver* approach [2] and mandate that namespace lookup routines receive the pasture ID on each system call relevant to the copy-on-write mechanism. These routines either return a

shared resource (when explicitly specified by the policy) or create a new resource via copy-on-write for the respective resource class. Both SELinux and *vserver* introduce extra "security" members into the fundamental data structures such as *task_struct*, *inode*, *file*, *superblock* and IPC descriptors. In *vserver* terms, these data structures marked with the same *context id* form a *security context*. Communication is allowed only between processes in the same context. However, *vserver* uses a different mechanism for handling file namespace isolation and copy-on-write functionality.

Our "COW pastures" occupy middle ground between SELinux types and BSD jails. They are more permissive than SELinux types and allow creation of new resources and write accesses to them as long as explicitly specified flow goals and ordinary UNIX DAC permissions are not violated. Also, they allow several isolated instantiations to take place simultaneously, which is not a feature of SELinux. Unlike the typical applications of BSD jails, they encourage the sharing of the underlying common system namespace between namespaces, when this sharing does not contradict the flow goals.

Although our current implementation is orthogonal to the Linux Security Modules architecture (LSM, the collection of system call hooks that is SELinux's mechanism for mediation of access-related systems calls to enforce its MAC policy), our assessment of LSM suggests that "COW pastures" could be implemented on top of SELinux as a secondary security module. Thus the sequence of access control decisions would be DAC $\rightarrow$ COW pasture $\rightarrow$ MAC. This would significantly simplify the MAC policy, taking away the need to describe the write flow goals in them. Thus the SELinux policies that currently combine the functions of (1) anomalous behavior detection and prevention, (2) integrity control, and (3) information flow control, would be limited to the first and partly the second of these goals.

Our existing implementation works in the *ext2* filesystem namespace, and consists of approximately 5000 lines as a kernel patch against the 2.6.12 Linux kernel.

## 4 Security and usability impact

The SELinux approach to policies is to define the full set of allowed resources and operations for each type while prohibiting all others. A look at other security setups based on by default mandatory isolation of security contexts suggests that policies aimed at describing the communication and sharing between contexts as their primary objects will be more usable and not significantly weaker in the practical integrity protections they afford.

Pastures are managed on the basis of namespace translations, creating a copy of an accessed object whenever the access is not explicitly specified in the policy. The policy specifies allowed flow relations between pastures; unless pastures are allowed to share a resource, a write-type access to an object results in the pasture getting its own private copy with the change, without affecting the other pastures' view of the object.

Thus in the most common scenario when a new untrusted application is introduced to the system and is placed into a pasture, a minimal policy already protects the integrity of the rest of the system while allowing the application to run, giving a significant savings of up-front labor as compared to SELinux.

Furthermore, the policy author directly specifies flow goals for the new application by specifying directly which objects it is allowed to share with other pastures. All other accesses result in changes limited to the application's private space and do not affect other applications and therefore the flow goals.

Let us revisit the motivating examples.

**Server protection** With an "everything not explicitly allowed is denied" profiling approach, the admin does not have many options. Most importantly, until he has compiled a full profile of the allowed accesses for the new software, the system will afford him no protection. That is, before any integrity protection can be afforded, all the effort must be invested up front. Given that the path to compiling such a profile in practice often lies through running the software with the policy engine in non-enforcing mode (e.g., *audit2allow*), we have a chicken-and-egg situation[4].

The solution we propose helps to run the software while protecting the integrity of the rest of the system from it, with the up-front effort limited to specifying the resources that must be shared with the other servers. This smaller amount of effort will allow to get the new server up and running.

**Client protection** With an "everything not explicitly allowed is denied" profiling approach, the user likely faces the necessity of multiple interruptions for policy adjustment and loss of work should his client accidentally violate the policy.

Our solution allows the client application to proceed with actions not explicitly prohibited by the policy rather than denying them and likely crashing it on the resulting error, at the same time protecting the integrity of the system. It thus makes sense to let it create private copies of the files it needs to modify or create.

[4]Given the shrinking average time between going live on the Internet and the first attack on a system, profiling an open real-world service by recording its audited accesses and assuming them to be "normal" behavior may be a really bad idea, whereas profiling a program in a limited "friendly" environment only may not provide enough diversity to cover all of its full intended functionality.

Then the few files that need to cross the private namespace boundary will be explicitly OK'ed by user, whereas the rest (cache, javascript, cascading stylesheets, media files etc.) never leave the private environment and operations on them cannot affect the user's home files or his OS.

To test the manageability of our approach, we are developing scenarios for a user study involving our lab sysadmins. The primary goal of these is to measure modification ease of Pastures policies to protect new applications.

**Merging costs.** Once the administrator has decided to eliminate a pasture, she will face the problem of merging the the changed files back into the system. We regard this as the postponed cost of the ability to start running an untrusted program while protecting the rest of the system from its changes, an exchange for the effort saved on profiling and recovering from fatal errors due to an overly tight profile.

For text or database files in amenable formats, merging techniques may help. Protocols and formats that allow merging have been discussed in a substantial body of publications; we cannot cover these due to lack of space. We need to remark, however, that no generic solution exists. For binary files, the decision comes down to replacing the system's copy of a changed with the private copy, or discarding the copy. Another possible solution is to change the policy to limit the copy-on-write behavior only to those files already changed, and share the rest of the filesystem.

**Effect on intrusion detection.** Our approach does have another downside: the loss of the de facto host intrusion detection functionality that systems such as SELinux provide. A close-fitting application profile under a fine-grained MAC mechanism will cause all accesses outside of an application's normal functionality to be logged. Log records of denied accesses can play the role of intrusion alerts. Indeed, the LIDS[1] access control system, which provided a much less granular partition of root privileges than SELinux, was originally described as an "intrusion detection system" by its authors. Still, it is debatable whether a policy system should double as a host intrusion detection system (HIDS), or whether this need is better served by introducing a HIDS proper (without the above mentioned limitations in its means of behavior description).

## 5 Performance impact

Copying incurs a noticeable performance cost. In our current implementation, this cost is paid for files whenever an existing file is modified, and for directories whenever a pasture's process first creates a new file in a directory. In current implementation we experienced 20% to 50% increase in the time for operations causing the first copy to happen, which was then amortized when more files were created in the same copied directory.

## 6 Related work

Our work was motivated by the challenges of using SELinux, and drew inspiration from the *BSD jails* and *vserver* designs. We note that although the idea of using copy-on-write for integrity protection is an old one – we traced it to Boebert's *LOCK*-[5] – it has not been, to the best of our knowledge, used as a "first-class" security policy primitive.

Several approaches to problematic server isolation are based on namespace separation and virtualization: BSD *jail* facility [7] (the so-called "chroot on steroids"[5] with a dedicated virtual network interface), the Linux *vserver* kernel patch[2], and methods based on virtual environments like the User Mode Linux (UML, [6]). In all these methods the basic integrity protection mechanism is very simple to describe – by default, an isolated process does not affect the OS namespace, and any changes remain local to the virtual environment. A number of tutorials (e.g., [9, 8]) describe this approach to setting up a system, stressing the simplicity of the model as a a key usability feature.

Any exceptions to this rule, i.e. the necessary sharing of resources for the program's functionality, need to be explicitly specified (and are discouraged as contrary to the basic separation model). We note that these exceptions would form an important part of the de facto policy for resource sharing. For example, some UML filesystems can be shared (possibly as read-only) with the host system, some files in the jail namespace can be managed by the host, and so on. An interesting example is the *vserver* setup, in which each file in the virtual environment starts out as a special type of hard link to the corresponding host file, with copy-on-write behavior when it changes. Although not implemented in *vserver*, we can imagine a policy that waives this copy-on-write behavior for specified files, thus describing the allowed communication between programs. *Vserver* is realized by assigning processes to "security contexts" and prohibiting communication between different contexts by way of kernel hooks (without an extra virtualization layer) and by namespace separation for contexts. Both BSD jails and *vservers* installations tend to start with separate filesystems for every virtual server/context[6].

While our design is largely informed by *vserver* and *BSD jails*, we take an entirely different view of sharing access between resources in different compartments. Such sharing

[5] E.g., http://www.usenix.org/publications/login/2005-08/openpdfs/musings.pdf

[6] In *vserver* the filesystem is composed of special types of hard links whose stated purpose is to save space. We argue that this trick can be generalized to be the basic mechanism of a policy.

is considered undesirable and warned against; one can say that the only security policy considered in both setups is full isolation of contexts. We, on the other hand, see it as a fundamental primitive of which flexible security policies can be built.

Solaris zones (e.g., [13, 12]) constitute a similar approach. Their containers are separated by default, being complete virtualized environments; information flows between them are by default absent. Privileges are managed at much higher levels of granularity and security labels are applied on zone level rather than on individual resource level as in SELinux, with no concept of label transitions within a zone. The designers apparently eschewed the flexibility of the LSM/SELinux approach [14, 3] in favor of a simpler conceptual model that is easier to use.

The allowed intersections between "jails" are the primary object of such setups/policies, whereas profiling of the access patterns of isolated programs themselves is secondary. This approach allows to quickly add a new service to the existing set on a machine while reducing the likelihood that some unexpected feature interaction unexpectedly breaks existing functionality.

Of a number of other sandboxing and isolation systems that focus on isolation of untrusted applications and share some behaviors with our system, the *Alcatraz* system described in [10] comes the closest to our approach. It isolates untrusted programs, keeping the changes made by them in a modification cache, and giving the user the option of either accepting or discarding them when the program finishes. The scheme is described by the authors as providing "play" and "rewind" buttons for untrusted software. The implementation is based on the Linux *ptrace* mechanism. The modification cache is maintained by a user-level process using *strace* to trace the execution of the isolated process and its children.

Most importantly from our point of view, *Alcatraz* recognizes the value of letting an untrusted (and even potentially compromized) process continue running while protecting the rest of the system from it. We proceed from the same premise, but make the default copy-on-write versus explicitly allowed sharing approach into a fundamental policy primitive, and place the enforcement mechanism in the OS kernel.

## 7 Conclusion

We propose a new approach to engineering security policies that is based on the principle that can be expressed as follows: *when an operation is not explicitly permitted, it is allowed to proceed by creating a copy of the affected resource in its private namespace*. This principle allows introduction of new programs to an already functioning system as a much smaller up-front cost than traditional ones based on full access profiles, while still providing integrity protection for the rest of the system. Also, it has simpler information flow properties. We believe that this approach deserves further investigation.

## Thanks

We would like to thank George Bakos for many useful discussions.

This research program is a part of the Institute for Security Technology Studies, supported by Grant number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. The Bureau of Justice Assistance is a component of the Office of Justice Programs, which also includes the Bureau of Justice Statistics, the National Institute of Justice, the Office of Juvenile Justice and Delinquency Prevention, and the Office for Victims of Crime. Points of view or opinions in this document are those of the author(s) and do not represent the official position or policies of the United States Department of Justice.

## References

[1] LIDS: The linux intrusion detection system, http://www.lids.org/.

[2] Linux vserver project, http://linux-vserver.org/paper.

[3] 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.

[4] K.-H. Baek and S. W. Smith. Preventing Theft of Quality of Service on Open Platforms. In *IEEE/CREATE-NET SecQoS 2005*, September 2005.

[5] W. Boebert. The lock demonstration. In *Proceedings of the 11th National Computer Security Conference*, 1988.

[6] J. Dike. User-mode linux.

[7] P.-H. Kamp and R. N. M. Watson. Jails: Confining the omnipotent root.

[8] P.-H. Kamp and R. N. M. Watson. Building systems to be shared securely. *ACM Queue*, 2(5), July/August 2004.

[9] D. Langille. Virtualization with freebsd jails, 2006.

[10] Z. Liang, V. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC)*, December 2003.

[11] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference (FREENIX '01)*. The USENIX Association, 2001.

[12] D. Price and A. Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the 18th Large Installation Systems Administration Conference (USENIX LISA '04)*. The USENIX Association, 2004.

[13] A. Tucker and D. Comay. Solaris zones: Operating system support for server consolidation. In *USENIX 3rd Virtual Machine Research and Technology Symposium*, 2004.

[14] C. Wright, C. Cowan, S. Smalley, J. Morris, and G. Kroah-Hartman. Linux security modules: General security support for the linux kernel. In *Proceedings of the 11th USENIX Security Symposium*, pages 17–31, Berkeley, CA, USA, 2002. USENIX Association.

[15] M. Zec. Implementing a clonable network stack in the freebsd kernel. In *Proceedings of the USENIX 2003 Annual Technical Conference, FREENIX Track*, pages 137–150. The USENIX Association, 2003.