

FILE SYSTEM ISOLATION FOR UNTRUSTED APPLICATIONS

by

Fareha Shafique

A thesis submitted in conformity with the requirements
for the degree of Master of Applied Science
Graduate Department of Electrical and Computer Engineering
University of Toronto

Copyright © 2008 by Fareha Shafique

Abstract

File System Isolation for Untrusted Applications

Fareha Shafique

Master of Applied Science

Graduate Department of Electrical and Computer Engineering

University of Toronto

2008

When computer systems are compromised by an attack, it is difficult to determine the precise extent of the damage because the state changes made by an attacker and those made by regular users can be closely intertwined. In particular, the file system provides a single namespace that, when compromised, can have cascading effects on the entire system, making intrusion analysis and recovery a difficult process.

This thesis proposes using a copy-on-write file system, called Isolation File System (IFS), to provide a transparent, restricted privilege sandboxing environment for running untrusted applications. The goal of IFS is to limit propagation of attacks by confining each application to its own complete namespace. If a sandboxed application is exploited, a coarse-grained recovery method allows completely removing the footprint of the software. Our approach supports existing applications by providing mechanisms for explicit sharing across IFS environments. Our evaluation shows that IFS is fairly easy to configure.

Acknowledgements

I would first like to thank Dr. Ashvin Goel for his thoughtful guidance, continuous encouragement and financial support. Many thanks to Shvetank Jain and Vladan Djeric who worked with me on different parts of this project and who made my experience at graduate school very enjoyable. I am grateful to the University of Toronto as well as the Department of Electrical and Computer Engineering for their financial support. I would also like to thank Dr. Angela Demke Brown, Dr. David Lie and Dr. Shahrokh Valaee for accepting to be on my defense committee. Finally, I would like to thank my husband for supporting me and having confidence in me at all times.

Contents

1	Introduction	1
1.1	Research Approach	2
1.2	Contributions	4
1.3	Thesis Structure	5
2	Related Work	6
2.1	Sandboxing and Virtualization Techniques	7
2.2	Access Control	9
2.3	Capability Systems and Restricted Privilege Systems	11
2.4	File Systems	14
3	An Overview of the Solitude System	16
3.1	IFS Isolation Environment	16
3.2	Sharing Policies	17
3.3	Taint Propagation and Recovery	17
3.4	Usage Model	18
4	Isolation File System	20
4.1	Copy-on-Write File System	21
4.2	Chroot Isolation Model	22
4.3	Capability Model	23

4.4	Structure of Policy Files	25
4.5	IPC Discussion	26
5	Implementation of IFS Isolation Environment	28
5.1	Implementation of Copy-on-Write File System	28
5.2	Implementation of Chroot Isolation Model	34
5.3	Implementation of Capability Model	35
5.4	Specifying Capabilities	36
6	Evaluation	38
6.1	Policy Files	38
6.1.1	Web Server: Apache2 and Apache2 + Gallery	39
6.1.2	MTA and MDA: Postfix and Procmail	41
6.1.3	IMAP Server: Dovecot	41
6.1.4	FTP Server: vsftpd	43
6.1.5	DHCP Server: dhcpd3	43
6.1.6	Printer Server: Cupsd	44
6.1.7	SVN Server: Svnserve	44
6.1.8	Discussion	45
6.2	Performance Overhead	46
7	Conclusion	48
7.1	Future Work	49

List of Tables

3.1	IFS sharing modes	18
4.1	IFS capability model	24
5.1	Summary of IFS databases	33

List of Figures

3.1	The Solitude architecture	16
4.1	Example policy for Apache2 web server	25
5.1	IFS Copy-on-Write	30
6.1	Policy for the Apache2 web server	39
6.2	Policy for Gallery running on Apache2	40
6.3	Policy for the Postfix MTA	40
6.4	Policy for Dovecot IMAP server	42
6.5	Policy for vsftpd FTP server	42
6.6	Policy for dhcpd3 DHCP server	43
6.7	Policy for cupsd printer server	44
6.8	Policy for svnserve SVN server	44
6.9	Performance Overhead of IFS	47

Chapter 1

Introduction

Several research efforts in recent years have focused on analysis and recovery of compromised systems [39, 25, 32]. This problem is both very real and hard: once a system is compromised, it is incredibly difficult to untangle the state changes made by an attacker, for instance the replacement of system binaries, from those made by normal users or administrators. While attempting recovery, an administrator is generally left with the choice of either confidently removing all attacker modifications or preserving all valid user activity, but not both.

Implicit sharing that exists in modern operating systems is a major cause of this problem. For example, all users and processes share a single common namespace. Compromises that manage to make unauthorized updates to this namespace, for instance by replacing the commonly used UNIX `ps` command, can have cascading effects across the entire system. While operating systems provide separate address spaces to protect physical memory, comparable protection is limited for persistent state.

This implicit file-system sharing problem is exacerbated as users increasingly download and install software from untrusted sources on the Internet. Users are faced with the choice of either not downloading and running the application, or they risk compromising the integrity and the stability of the system. For instance, a downloaded media player application can have serious vulnerabilities that can allow attackers to attach malicious code and infect computers

without the user's knowledge. Additionally, audio and video streams and downloads can be used to hijack or corrupt computers [76].

This thesis focuses on confining untrusted applications into separate isolation environments. Within each environment, an application is bound to its own complete file-system namespace, similar to process address spaces, via a copy-on-write file system that we have designed, called Isolation File System (IFS). The benefit of namespace isolation is that malicious changes made by one application cannot inadvertently effect the operation of other applications. In our design, each untrusted application within IFS is run in a sandboxed environment with restricted privileges. As a result, even if the application is compromised, escaping the sandbox is difficult and any damage that can be done to the system is limited because all-powerful super-user capabilities are disallowed within IFS.

IFS allows unshared persistent states to diverge freely across isolation environments under the presumption that file sharing across applications is rare. The challenge with such an environment is that since applications cannot share data across the isolation environments, they may not work correctly. Therefore, sharing across environments must be supported, and IFS provides explicit sharing mechanisms for this purpose. However, the detailed description of these mechanisms and the specific sharing policies needed to support IFS applications is the focus of another thesis [36].

1.1 Research Approach

The goal of IFS is to limit the propagation of attacks by running each untrusted application in a separate sandbox that confines it to its own complete file system. IFS offers a transparent view into the base (or regular) file system for reading operations, but any modifications made by the untrusted process or its children processes are confined to the separate namespace. If at any point the user decides that the software is malicious or undesirable, the entire IFS environment can be discarded without concern for the integrity of the base file system.

A typical usage scenario of this system may involve running a peer-to-peer (P2P) client program within an IFS. For example, a user may download and install a photo editing application using the P2P client. The user knows that files on P2P networks are sometimes modified to include malicious components and thus installs the application in an IFS. This may be the same IFS as the P2P program or a new IFS, but in both cases, no changes are made to the base file system. If the application exhibits unexpected or suspicious behaviour, the user can remove the program and its changes by discarding its IFS.

The basic IFS isolation model enhances `chroot` isolation with copy-on-write from the base file system. As with any isolation environment, there is a trade-off involved between the security provided by the IFS isolation environment, application-level functionality and ease-of-use. We rely on two mechanisms to resolve these concerns: support for restricted privileges for running server applications, and support for explicit file sharing policies. Running applications with restricted privileges provides increased security while supporting full application functionality. Similarly explicit file sharing, although not as secure as complete isolation, allows us to fully support *existing* applications that are based on the current access control model in Linux. Each of these mechanisms is discussed below.

There are many well-known techniques for escaping `chroot` jails. Several precautionary measures and rules can be followed to alleviate this problem. The most important rule is to disallow superuser capabilities in a jail which makes it significantly harder to escape the jail. Running programs with restricted privileges in IFS inhibits the spread and effectiveness of malware such as spyware, rootkits and memory-resident viruses that attempt privileged operations (e.g., loading kernel modules), and makes it harder to compromise the isolation mechanism. IFS restricts the privileges of root or `setuid` applications by enhancing the capability system available in Linux [43]. Each IFS environment can specify the capabilities that should be enabled in an associated policy file. For example, a web server IFS environment would allow opening privileged ports.

Support for file sharing policies enables rich system functionality and helps with ease of

use. Although by default the IFS copy-on-write mechanism shares reads with the base file system and isolates all writes, write sharing policies can be specified in the same policy file as the capabilities linked with an IFS environment. These simple, yet flexible sharing policies allow 1) reads to be isolated (directed to a snapshot of the base file) or denied, and 2) writes to be denied or shared (either immediately or at a later time). Our isolation environment is implemented as part of the Solitude application-level isolation and recovery system. Solitude consists of three major components: the IFS isolation environment, the explicit file sharing policies, and the taint propagation, logging and recovery system. IFS was briefly described above and will be discussed in more detail later in the thesis. Solitude requires that any sharing of persistent data be performed explicitly through its file sharing policies, which are supported by IFS. Finally, the recovery component allows two options: 1) coarse recovery by discarding a complete IFS environment, or 2) fine taint-based analysis and recovery derived from Taser [25].

1.2 Contributions

This thesis explores the use of separate copy-on-write based file-system namespaces for running untrusted networked applications, similar to address space separation via copy-on-write memory. Network applications pose security risks since they are increasingly being used to download and install data and code from untrusted sources. For example, audio and video downloads are often incorrectly labelled and can contain viruses that can corrupt computers [76]. If these files were downloaded in an isolated namespace and used within a restricted-privilege sandbox environment, computer systems could be protected from extensive damage.

We implement a prototype of IFS and propose running each networked application within a separate IFS environment. Our implementation provides a file-system based restricted-privilege isolation environment that is reasonably easy to specify and can be used for both client- and server-side applications. IFS enhances the Linux capability model to run each process with the minimum set of privileges required and replaces the Linux coarse-grained file

access capabilities with finer-grained per-file or directory capabilities. These capabilities are specified in a policy file associated with each IFS.

IFS limits attack propagation in a system by isolating all writes and ensures ease of configuration by sharing all reads with the base. This model provides a simple recovery solution that consists of discarding an entire compromised IFS environment without affecting the rest of the system. IFS also provides mechanisms that allow explicit file sharing to support existing applications that may require communication across IFS environments.

Our detailed evaluation based on running several server applications shows that capability specifications in policy files are short, intuitive and reasonably easy to specify. IFS, although usable on its own, is most beneficial when used with Solitude, and as such, we provide a performance evaluation of Solitude in addition to that of IFS.

1.3 Thesis Structure

The rest of the thesis describes the IFS in more detail. Chapter 2 discusses related work in this area. IFS is designed to be used as part of Solitude, a system that provides explicit file sharing mechanisms and recovery facilities in addition to the isolation properties of IFS. Chapter 3 provides an overview of Solitude. Chapter 4 presents a detailed design of the IFS isolation model and Chapter 5 describes its implementation. Chapter 6 provides an evaluation of IFS in terms of ease of configuration and performance. Finally, Chapter 7 concludes the thesis and highlights directions for future work.

Chapter 2

Related Work

In current operating systems, by default, all users share the same file system. For instance, in most Unix systems, any user or application can write to the shared `/tmp` directory. This file sharing is regulated with the help of access controls. Unfortunately, with a single file-system namespace, an error in the access control configuration may allow an attacker to compromise the entire system. Now consider memory in modern operating systems. By default, memory is not shared by different processes even when the processes are run by the same user. Memory sharing is allowed only explicitly, for example, via shared memory. We propose a file system model similar to the memory model, whereby files are isolated across applications by default and can be shared only explicitly.

The goal of this research project is twofold: to limit the effects of attacks, and to simplify post-intrusion analysis and recovery. Attack propagation is limited through namespace isolation using a copy-on-write file system and restricted privileged execution of applications. Analysis and recovery is made easier by requiring explicit sharing of persistent data. In this chapter, we first describe related work in the area of sandboxing and virtualization. Then we describe related research in access control and file systems. Finally, we discuss work on restricting privileges of applications.

2.1 Sandboxing and Virtualization Techniques

Hypervisor-based virtualization machines (VMs), such as those based on full virtualization approaches like VMWare [33, 17] and para-virtualization approaches like Xen [6] and Denali [79], provide strong isolation guarantees. They have been used to protect trusted and private data from applications as well as to protect applications from one another [16, 9, 74, 23]. However, it is incredibly difficult to configure sharing in these VM environments. For example, a virtual machine can be used to run multiple versions of the Office word processor, but each machine has a separate unsynchronized desktop that leads to a confusing and error-prone user experience.

A second virtualization approach that trades security for efficiency is to use operating system-level virtualization [67]. Linux-Vservers [45], Virtuozzo [72], FreeBSD Jails [37] and Solaris Zones in Solaris10 [54] make use of this approach. This approach, while similar to our isolation environment, is still designed primarily for isolating applications run by untrusted users and thus focuses on avoiding denial-of-service attacks and provides limited sharing. For instance, in university or small corporate environments, a single machine is often able to run several server applications such as web server, mail server, print server etc. on behalf of the same set of users. With OS virtualization, by default, each of these servers would have its own list of users and user directories.

WindowBox [5] provides virtual desktops inside Windows 2000 and allows explicit sharing of data through direct point-and-click commands or warning dialogue boxes. However, the system is not very usable because users are expected to configure each desktop for a particular task and switch between the desktops as they work.

System call interposition has been used extensively for restricting a program's access to the operating system [26, 1, 55, 35, 64]. These sandboxing techniques have not been widely deployed because they are hard to configure. Janus [26] is difficult to configure because it requires per system call policies for each application. MAPBox [1] attempts to group application behaviour into classes based on the expected functionality, and then it applies the same system

call policies to all applications in a single class. If an application has not been classified, it fails to run. Systrace [55] automatically generates policies via training runs to determine the resources used by an application under normal circumstances. All these approaches confine the damage an untrusted application can cause on the system but they do not isolate applications from one another.

In Alcatraz [42], Sekar et. al. provide file system isolation through system call interposition in addition to restricting OS access. In their later work on One-Way Isolation [70], they improve the file system isolation by intercepting file operations at the Virtual File System layer rather than at the system call level. In both these approaches, untrusted processes observe the environment of their host system, but the effects of these processes are isolated from other applications. Once the code is trusted, all changes made by it can be committed to the host system. While these works propose using one-way isolation for testing and debugging, we propose to limit sharing by running applications in the long term in an isolation environment. A consequence of our approach is the need to correctly secure the isolation environment when executing privileged applications and provide support for limited file sharing. In addition, our overall goal with Solitude is to provide a specification that allows explicit sharing, the capability to commit selectively and perform recovery even after data is committed.

We envision using isolation environments for different applications run by the same user or within the same administrative domain (same set of users) and thus aim to provide better support for sharing. Since our primary focus is on limiting attack propagation to simplify analysis and recovery of persistent data, our isolation environment uses application-level virtualization. Microsoft has recently released its Softgrid/SystemGuard technology for virtualizing applications [13]. Softgrid uses a single OS, but uses the SystemGuard virtual application environment to keep application dependencies (DLLs, registry entries, fonts, etc.) separate from the rest of the system, which allows streaming and running multiple versions of an application such as Office within the same OS. SystemGuard uses a copy-on-write file system but does not allow users to explicitly share configurations or applications with the base. Greenborder is another

application virtualization technology that provides copy-on-write protection, but is tailored to provide protection for specific applications such as web browsers [31]. In Windows Vista, Microsoft has introduced a Protected Mode for Internet Explorer 7 [50, 14]. When running in this mode, the browser runs in a low integrity level with restricted privileges. This prevents hackers from taking over the browser and installing new software. It also disables write access to most of the file system including the registry. In this mode, IE is given its own low integrity copy of the cache, TEMP folder, Cookies and History, but it shares the Favorites with IE running in the high integrity level. Protected mode provides predefined isolation and sharing policies that are not easily reconfigurable.

2.2 Access Control

Access control policies restrict access to a system and its objects based on a set of discretionary, role-based or mandatory policies. Discretionary policies, specified by object owners, involve setting file permissions and ownership. They allow implicit file sharing through the creation of user groups and across independent applications run by the same user. Role-based policies are common in corporate settings where system administrators create roles according to job functions in the organization, configure permissions for these roles and then assign employees to the roles based on their job responsibilities [18, 19, 59]. Mandatory policies enforce explicit sharing and are specified by an administrator based on the principle of least privilege. For example, SELinux [46] provides a powerful mandatory access control model, but it is commonly acknowledged that designing SELinux policies is a complicated process [34]. Similar to SELinux, the RSBAC framework [52] and the Medusa DS9 security system [48] also provide flexible mandatory policies allowing system administrators to enforce any security model. These systems, like SELinux, are also complex to configure.

Domain and Type Enforcement (DTE) is also designed to provide mandatory access control to protect a system from subverted super-user processes [29, 4, 3, 78]. DTE systems partition

processes into access control domains, and read policies at boot time that define how to enter each domain, transfer between domains and what information each domain can access. Once again this allows enforcement of strong security policies for information protection, but at the cost of configuration complexity.

Access control lists, or ACLs, are also a common mechanism used to restrict access to a system. ACLs can be either discretionary or mandatory depending on whether they are determined by a user for protection of her own data or by a system administrator to be enforced system-wide. Patches implementing discretionary ACLs for the Linux kernel provide more fine-grained control over sharing compared to the default 9-bit DAC permissions [27]. A user can associate an ACL with each object that defines exactly how particular users and groups can access the object. The Multics operating system also focused on providing improved security by disallowing access to files by default and defining all access through ACLs [60]. The Windows operating system uses ACLs as its main access control mechanism and has tried to improve its model over the years [71].

Mandatory ACLs define how each program, rather than user, on a system can access files. LIDS [82] associates an ACL, implemented at the VFS layer, with each file or directory specifying default accessibility and allowing exceptions to this default for particular executables. It also implements capability ACLs discussed in Section 2.3. Similarly, PACL [80] keeps ACLs for each file containing a list of programs that can access it. Later approaches associate ACLs with programs, rather than with files, to define the list of files that a program can access. Several of the systems discussed in sandboxing above use this approach along with system call interposition to restrict access to the system and its objects [26, 1, 55, 63, 15, 7]. Janus [26], MAPBox [1] and Systrace [55] all use policies (equivalent to ACLs) to determine if each system call succeeds or fails. SubDomain [15] and TRON [7] only intercept file system calls and the `execve` call to control file access including whether a file can be executed. In his work on execution controls, Gamble [22] takes access control lists further and uses them to define a user and program combination that can access a particular file. The main problem with ACLs is that

the approach does not scale as the number of users, files and programs on a system increases because a list must be created for each file, listing all the users and programs that can access it, or for each program, listing all the files it can access.

The UMIP model, similar to Solitude, aims to preserve system integrity in the face of network-based attacks [41]. This model leverages information available in existing discretionary access control (DAC) policies to derive file labels for mandatory integrity protection. The basic UMIP policy partitions processes into low and high integrity. When a process performs an operation that potentially contaminates it, such as via reading from a network socket or communicating with another low integrity process, it drops integrity and cannot perform sensitive operations. The basic UMIP policy is enhanced with capability exceptions to support server applications. Our capability model was developed concurrently and has many similarities with UMIP capabilities. The primary difference is that in our default copy-on-write policy, reads are shared with the base file system and not denied, and hence our policy files are easier to specify because they typically do not need exceptions for reading files. More importantly, UMIP does not provide isolation to client-side applications run by the *same* user because it uses DAC policies to configure its policies. Since UMIP is an access control mechanism, it shares the limitation with SELinux that the policies must be correctly specified when files are updated. In contrast, our copy-on-write approach allows files to be in *both* low and high integrity states.

We provide a simpler, but more coarse-grained isolation model as compared to the access control approaches discussed above, in which policies are primarily needed for file sharing. However, more importantly, the above access control approaches require correct policies at updates, while with our copy-on-write policy, errors can be handled until a later time.

2.3 Capability Systems and Restricted Privilege Systems

The main goal of using restricted privileges or capabilities in a system is to enforce the principle of least privilege and provide better security guarantees. Several approaches have been

proposed over the years. This section discusses some of these approaches starting with classical capability systems, followed by limited authority in WindowsNT, and then describing systems based on the POSIX capability model that we use in IFS. Finally, privilege separation is discussed as another approach for enforcing the least privilege principle.

Traditionally, a capability is just a token used by a process to prove that it is allowed to perform an operation on an object. For instance, a file descriptor is a capability allowing either read, write, or both read and write on a file. In classical capability systems, a process carried with itself a set of access rights to particular objects. For example, in the EROS micro-kernel [66], each process has capabilities and can only perform operations that are authorized by its capabilities. Some of the systems discussed in Sections 2.1 and 2.2 also provide a form of capabilities. For example, the system call interposition systems [26, 1, 55] restrict the success of a program's system calls according to a policy linked to the program.

WindowsNT uses restricted contexts by creating a limited version of a user that can access only a subset of objects, and running programs as this user [71]. However, it has been shown that most Windows users run with administrator privileges all the time and this increases the vulnerability of the system [10].

The POSIX capability model was incorporated in the Linux kernel in version 2.2. It divides traditional superuser privileges into 30 capabilities that a process can independently enable or disable [43]. Several systems, including ours, make use of these POSIX capabilities [37, 82, 45]. FreeBSD Jails [37] limit the privileges of all processes running within the jail by reducing the capabilities to a default set. Linux-Vserver [45], in addition to limiting the capabilities within each context, extends the model to provide fine-grained capabilities. For example, it divides the `cap_sys_admin` capability needed to mount, unmount, set hostname etc., into separate capabilities for each functionality. LIDS [82] uses ACLs to limit the capabilities granted to a program. It also extends the default POSIX model and implements a fine-grained `cap_net_bind_service` to ensure that an application can only bind to a particular port, rather than any port less than 1024. SELinux [46] provides security policies to control the use of

Linux capabilities and also allows extensions to the current model, such as granting privileges based on the subject attributes and object attributes. This can be used, for example, to provide `dac_override` to a process for a particular set of files instead of the default system-wide override. IFS also uses Linux capabilities to run programs with least privileges. It ensures that even applications running as root have only the capabilities they need. We also provide fine-grained `dac_override` through our file capabilities described in Section 4.3.

Unfortunately, today server applications still mostly execute as the root user rather than use these process capabilities. In the future, the Linux kernel will introduce file capabilities to make POSIX capabilities more usable [28]. With file capabilities, a process will not have to enable or disable capabilities. Instead they can be assigned to executable files, similar to setting permissions on files such as the `setuid` bit. Any time the executable runs, it will execute with the capabilities assigned to it. Note that these file capabilities, expected to make the mainline kernel by version 2.6.24, are different from our file capabilities, which we discuss in Section 4.3. IFS file capabilities override per-file or directory permissions in the base system. The IFS isolation environment makes use of process capabilities to provide, effectively, what Linux POSIX file capabilities will allow in the future.

Privilege separation was proposed by Provos et. al. [56] to reduce the amount of code that runs with special privileges and thus limit the scope of programming bugs to a smaller and more easily secured trust base. They demonstrated that this approach prevents security vulnerabilities by separating the OpenSSH code into privileged and non-privileged code. Privman [38] provides privilege separation through a library that supplies C functions for many operations that traditionally need privileges. In addition to using this library, developers must write a configuration file expressing the security policy. Privtrans [8] attempts to automatically separate code into privileged and non-privileged programs based on annotations in the source code. Proxos [74] separates code such that system calls that access sensitive resources are executed on a private VM and all other system calls execute on a commodity OS VM. The separation is based on rules set by the developer. These approaches do not work with existing

code, and they present a security approach orthogonal to our restricted privilege sandbox.

2.4 File Systems

Many file systems [61, 53, 49, 20, 51] have been developed for creating snapshots for versioning and recovery. Others [68, 57, 11] use check-pointing to provide backups for rollback and recovery. UnionFS [81] virtually merges the view of different directories such that they appear to be one tree. It can be used for snapshotting and copy-on-write by marking directories as read-only. Then modifications are carried out in a separate directory, which is unified with the original read-only directory. Self-securing storage [69] audits operations and keeps versions for some time for intrusion detection and recovery. RFS [84] also provides comprehensive versioning along with dependency logging and dependency analysis for recovery. These file systems typically implement versioning and/or check-pointing at the block level which is simpler to implement and provides good performance. However, Solitude's goal is to enable limited sharing at the file-system level, which requires understanding the logical structure of a file system. Hence IFS uses copy-on-write at the file-system level. This method also fits well with Solitude's taint propagation and recovery model, which is performed at the level of files and directories. User-level file systems developed in the past include Wayback [12] for versioning and Ufo [2] for providing a file system that treats remote files as if they were local. Ufo uses system call interposition and Wayback uses FUSE [73]. Our IFS implementation also uses FUSE, and we describe it in Section 5.1.

Transactional file systems, for example QuickSilver [30, 62] and Vista's TxF (transactional file system) [77], allow file system operations to be handled like transactions so that all the changes within a transaction are committed to disk atomically and the intermediate states of a transaction are not visible to other applications or transactions within the same application. Both file systems require changes to applications to use a transactional interface to start, abort or commit a transaction, and they use a pessimistic locking mechanism for ensuring consis-

tency. Quicksilver holds read locks on files until the file is closed and write locks until the end of a transaction. Directories are locked when they are modified, for example when a directory is renamed, created or deleted. TxF's locking mechanism is also very similar to QuickSilver. However, a file can be read and written in two different transactions concurrently. In this case, the reads do not see the modifications made by the other transaction. To provide this isolation, TxF intercepts all file system operations and captures the state of the file or directory before carrying out the operation. These file snapshots are also used for rollback when a transaction fails or is aborted.

In contrast to QuickSilver and TxF, our IFS environment supports existing applications without requiring any changes to these applications. It provides transactional semantics at the IFS granularity and hence transactions can exist for long periods of time. To ensure availability in the face of long-running transactions, IFS uses an optimistic concurrency control method that allows the different IFS environments to concurrently access and modify files. IFS transactions can either be rolled back by discarding the entire IFS environment or IFS allows using resolution policies when conflicts occur during a commit [40, 58, 75].

Chapter 3

An Overview of the Solitude System

Solitude provides a copy-on-write, file-system based sandbox environment for running untrusted applications, and it uses an explicit file sharing mechanism that limits attack propagation without compromising system functionality. The Solitude architecture is shown in Figure 3.1. It consists of three main components, the IFS isolation environment, the sharing policies, and the taint propagation, logging and recovery system. We describe these components below.

3.1 IFS Isolation Environment

Solitude allows running an untrusted application in an isolation environment called IFS that provides the application with a transparent view of the base file system, but restricts any file-

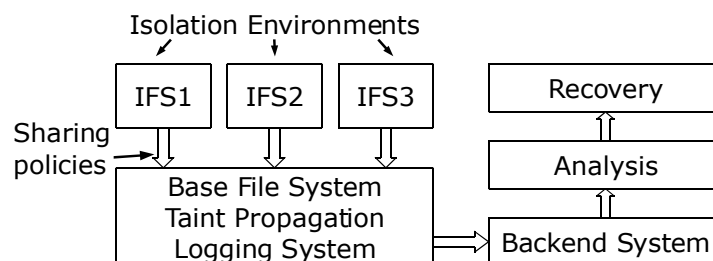


Figure 3.1: The Solitude architecture

system changes with a copy-on-write policy. In other words, an application running inside an IFS has the illusion that it is running inside the base file system, but in reality, the base file system is oblivious to its existence unless the user has configured explicit sharing policies that allow synchronizing the base file system with the IFS. An application running in an IFS executes with restricted privileges. This makes it difficult to escape the isolation environment and also limits the adverse effects a malicious application can have on the system if it escapes the sandbox. The following chapters of this thesis describe the IFS in more detail.

3.2 Sharing Policies

Solitude isolates the persistent changes made by applications by using copy-on-write as its basic isolation model, but it allows refining this model with policies that enable explicit sharing of specific files and directories between an IFS and the base system. At the time of its creation, each IFS can be associated with a policy file, stored outside the IFS in the base system, that specifies the file sharing policies. The intended authors of these policy files are application creators and system administrators although the policy language is simple and policies are easy to write. For untrusted applications, it may be safer to obtain policy files from user communities or to use the default policy.

The sharing policy language is designed for simplicity and intuitiveness. It specifies three possible sharing modes for reading and three modes for writing, although a few combinations are not meaningful. These modes are shown in Table 3.1. All the sharing modes apply to a file or directory and are subject to the access control restrictions of the base.

3.3 Taint Propagation and Recovery

The sharing policies described above enable collaboration between applications running in an IFS and base, or between different IFS contexts. Without such sharing, an increasing number of

Read/Write Mode	Description
Rshare (default)	Applications read from the base until they make an update
Rsnapshot	IFS makes a snapshot of the base immediately at startup
Rdeny	Hide the base file or directory from the IFS
Wdeny (default)	Confine all writes to the IFS permanently
Wcommit	Confine changes to the IFS, but allow delayed sharing with the base
Wshare	Immediately share writes from the IFS with the base

Table 3.1: IFS sharing modes

applications would be run in the same isolation environment, negating the benefits of isolating the applications. However, the sharing policies could be poorly designed, potentially leading to contamination of the base file system either via commit or write sharing of malicious data or applications. Solitude addresses this issue by tracking how other applications access files that are committed or write shared and then using a taint propagation method to log their resulting actions. If untrusted files reach the base, Solitude uses a modified version of the Taser system [25] that helps with analysis and fine-grained recovery of the base system.

3.4 Usage Model

Based on the notion that intrusions start with a network connection and then cascade into multiple system activities such as file accesses and outgoing connections, we envision that Solitude will be useful for various networked applications. On the client-side consider instant messaging applications to communicate and share data. The sharing policies can be used to offer the user the option to preserve, say the chat logs, that MSN writes to a certain directory in the base. Similarly, with a mail client, the local mail directory and the mail-client configuration files could be explicitly shared with the base system while any other persistent data would be unshared.

On the server side, consider a web site that provides an on-line photo album service. The web server can be run in an IFS environment while configuring only the users' photo data to be shared with the base system. In this way, the persistent data that is important to users can be shared with the base system, such as for archival or file search, but any updates made by the web server are unshared and cannot affect the rest of the site even if the web server is compromised.

Chapter 4

Isolation File System

The main goal of the IFS isolation environment is to limit the effects of attacks and simplify the post-intrusion recovery process by supporting explicit sharing of persistent data. It is specifically targeted for client-side applications run by the same user and for server-side applications running within the same administrative domain.

The IFS isolation environment allows running multiple applications within an isolation environment. For example, a user can download a file using a peer-to-peer application and save it to a standard location. The user can then run a viewer application within the same session or mark the standard download location as explicitly shared and use the viewer in the base or separate IFS environment. All other updates by the peer-to-peer application remain unshared and could be easily discarded after session termination. Note that isolation environments are persistent in the sense that the IFS state is preserved across multiple invocations of the application.

Administrators can also choose to use IFS environments for certain low-privilege users. For example, IFS can be used to isolate directories that are shared across users such as the Unix `/tmp` directory that has been the source of several exploits, and to ensure that anonymous FTP users cannot affect the base file system.

An isolation environment's design space involves making trade-offs between security, application-

level functionality and ease-of-configuration and use. The design of our isolation model was motivated by our objective to support better sharing and application-level functionality than either hypervisor or system-level virtualization, and easier configuration than a Unix chroot and BSD jail sandbox.

At a high-level, our IFS isolation model consists of running applications with restricted privileges in a chroot copy-on-write file system. This approach isolates file modifications and simplifies chroot configuration by sharing reads with the base system. This chapter will describe the copy-on-write file system (Section 4.1), chroot isolation (Section 4.2), and our capability model (Section 4.3). Section 4.4 explains the structure of the policy files that, among other things, specify the capabilities. Finally, a short discussion on limiting IPC mechanisms is also provided in Section 4.5.

4.1 Copy-on-Write File System

The copy-on-write file system gives an application running inside an IFS environment the impression that it is running in the base. The application can read files in the base, subject to the discretionary access controls of the underlying operating system. By default, all changes are redirected to the IFS layer. However, in the presence of explicit sharing policies, reads and writes are directed by the copy-on-write file system to the appropriate layer, base or IFS, depending on the specification.

There are several reasons that motivated a copy-on-write based isolation environment. First, the basic file-system recovery method for unshared persistent data is simple: if at any point the user decides that the software may be malicious, they can discard the entire IFS environment without concern for the integrity of the base file system. Second, preventing implicit sharing of file updates limits attack propagation that occurs as a result of persistent changes in the system and hence reduces the effort involved in overall post-intrusion analysis and recovery. Third, the explicit sharing mechanism is based on the hypothesis that write sharing of files across

applications is rare [36]. Finally, copy-on-write enables read sharing between the base and the IFS, and we do not require explicit read sharing because such operations are far more common and thus configuring them correctly would be challenging.

With copy-on-write isolation, any malware that attempts to conceal its presence by disabling security software or by installing rootkits will fail because the isolation mechanism safeguards the integrity of programs in the base system. For example, cryptoviral extortion, a kind of denial of resources attack, does not pose a problem with copy-on-write. A cryptovirus encrypts critical data on a machine making it inaccessible. The victim is compelled to make ransom payments to the virus author in exchange for the public key needed to decrypt the data [83]. Copy-on-write ensures the original data is not overwritten and the public key is never needed. Copy-on-write can also help with malware detection: changes made to files and directories inside an IFS are quickly spotted when a user examines the list of changed files within an IFS. Finally, copy-on-write isolation can also help stop the spread of worms and viruses that propagate across mounted file systems and network shares because IFS makes local copies of these files.

Our file system is mounted in a Linux `chroot` environment to further reinforce application isolation. The copy-on-write file system makes the configuration of our `chroot` jail much easier. With shared reads, we do not require the libraries and program binaries needed by applications to be copied into the `chroot` environment. Our `chroot` isolation is explained further in the following section.

4.2 Chroot Isolation Model

The `chroot` sandbox ensures that application reads and writes do not by-pass our copy-on-write file system and affect the base system. To strengthen the isolation mechanism, we have incorporated the `Vserver` secure `chroot` barrier [65] in IFS. This barrier uses a special flag on the parent directory of the isolation environment to prevent `chroot` escape (and allow nested `chroot`

jails). However, even with this mechanism, techniques for escaping chroot jails are known and have led to best practices for using them [21]. The most important of these rules is to disallow all-powerful root privileges in a jail, which makes it significantly harder to escape the jail. Unfortunately, this method limits functionality by disallowing setuid programs and server-side applications that, for example, may require access to privileged ports. Setuid programs are appealing targets for isolation. They are frequent targets of attacks because they provide a direct path to complete control over the system. Sendmail is a typical example of a setuid application - it uses its root powers to temporarily impersonate other users to deliver mail to their inboxes. Our capability model allows us to overcome this limitation and is described next.

4.3 Capability Model

In order to avoid using all-powerful superuser privileges, yet support setuid applications and server applications in IFS, we restrict the privileges of root by enhancing the capability system available in Linux [43]. Each IFS environment can specify capabilities that are then enabled in the environment. For example, a web server IFS environment would allow opening privileged ports. We chose Linux capabilities because they are relatively easy to specify. However, they are coarse grained, and in particular, file related capabilities apply to the entire file system. For instance, a program running with the `cap_dac_override` capability can override *all* file access restrictions. Instead of allowing such powerful capabilities, IFS provides per-file or directory capabilities. This approach may seem cumbersome, but our results show that in practice most systems configure the discretionary file-access control permissions “almost” correctly, that is, only a few permissions are incorrect. Hence privileged applications typically require few per-file capabilities and can be run correctly without full root privileges. For example, consider again the web server such as Apache2 running with restricted privileges in an IFS environment. The application does not run as the root user and is only given the capability to bind to a privileged port. As a result it does not have permission to access some files that are root

Capability	Parameters
Fcap	FilePath [owner.group] [perm]
CAP	[ExecPath] CAP_SET

Table 4.1: IFS capability model

owned, such as the error log and access log. Therefore, it must be given per-file capabilities for these files as shown in Figure 4.1.

Table 4.1 shows the specification of the IFS capability model. These capabilities are specified for each IFS environment in a policy file described in Section 4.4. The Fcap file capabilities apply to all programs run within an IFS, and allow overriding the file ownership or permissions on the file (or directory) specified by `FilePath`. Discretionary access controls associate particular permissions with particular users on the system, and the Fcap capabilities allow either changing the owner permissions or group permissions directly, or associating the permissions with a new owner. With these two options of Fcap, we provide a fine-grained `cap_dac_override` capability in order to support application functionality. The CAP capability applies only to the executable specified by `ExecPath`, that is, it is per-program. When `ExecPath` is not specified, it applies to the top-level application. This capability is enforced when an application starts executing, e.g., on a Unix `execve` system call, and the `CAP_SET` parameter is a list of capabilities provided to the application. IFS restricts certain capabilities such as create or remove mount points and accesses to raw devices, that may allow applications to escape its isolation environment.

When an application running within an IFS starts a new application by executing the `execve` system call, the CAP capability of the new application is *exactly* the set specified by `CAP_SET`, and as a result, capabilities are not inherited and `setuid` applications have no additional privileges in IFS. When a CAP capability is specified for an application, IFS disallows the Fcap capability for the `FilePath` associated with the application. This is to ensure that a vulnerable IFS application is not hijacked into executing a privileged application that has been

```
Application /usr/sbin/apache2 www-data.www-data
Fcap /var/run/apache2.pid www-data.www-data
Fcap /var/log/apache2/error.log www-data.www-data
Fcap /var/log/apache2/access.log www-data.www-data
CAP net_bind_service
Wcommit /var/log/apache2/error.log
Wcommit /var/log/apache2/access.log
```

Figure 4.1: Example policy for Apache2 web server

modified. Similarly, IFS ensures that files that have been copied into the IFS via copy-on-write do not run with any IFS capabilities. Furthermore, each capability is specified for a given IFS and is not system wide.

We always ensure that applications in the IFS environment execute with privileges more restricted than if the same applications were run in the base environment. For example, if an IFS environment is started by a regular user, then the environment will have no capabilities. This is not a limitation, since client-side applications generally do not require any IFS capabilities. Section 5.3 describes how our implementation enforces these capabilities.

4.4 Structure of Policy Files

Each IFS environment can have a policy file associated with it. This file is saved outside IFS in the base system and can specify 1) the principal the application should run as inside IFS, 2) the capabilities described in the previous section, and 3) any of Solitude's explicit sharing policies shown in Table 3.1.

Figure 4.1 shows an example policy file for the Apache2 web server. Apache is run as the `www-data` user in an IFS environment and the ownership of files with the `Fcap` capability is set to `www-data` in the IFS (not in the base) environment. This capability together with the `net_bind_service` capability for accessing a privileged port allows running Apache in an IFS

environment with no other additional privileges. The evaluation chapter shows examples of policy files that use the ExecPath argument.

The intended authors of policy files are application creators and system administrators. For untrusted applications, it may be safer to obtain policy files from user communities or to use the default policy. Section 6.1 shows that, for our targeted applications, it is easy to specify the capabilities in a policy file.

4.5 IPC Discussion

In addition to file sharing, inter-process communication mechanisms also allow implicit sharing between applications. Our IFS environment must prevent applications from leaking information into the base system via inter-process communication with base processes. Common IPC mechanisms in Unix systems include 1) FIFO, 2) Unix domain sockets, 3) shared memory and 4) local INET (TCP, UDP) sockets. The first three mechanisms have unnamed and named counterparts. The unnamed mechanisms only work for related programs in a process hierarchy and are, therefore, allowed within an IFS but disallowed across IFS and base by default.

Our approach towards the remaining IPC mechanisms is based on the IPC study performed on our cluster server and on my personal desktop machine [36]. The experiment has shown that common applications do not typically use named shared memory communication, and a very small set of applications use named FIFO and Unix domain sockets. In Linux, named FIFO and Unix domain sockets are represented by special files. Our IFS copy-on-write file system ensures that FIFO and Unix sockets can only be used for communication within the same IFS. We disabled these IPC mechanisms across IFS and base and different IFS environments to avoid implicit sharing.

For local INET sockets, the study showed no UDP based communication and just a small number of local TCP connections, for example for the X-server, print server and ssh-server. The X- and ssh server provide basic services (desktop and remote access) and would need to

be shared with many different IFS environments. Hence, they would be run in the base. These results are promising because they indicate that it is feasible to disable IPC mechanisms and, if needed, incorporate explicit specifications.

Chapter 5

Implementation of IFS Isolation

Environment

The IFS isolation environment consists of the copy-on-write file system, chroot isolation model, and capability model explained in the previous chapter. The following sections describe the implementation of these components. We have also implemented a tool to help determine the set of capabilities that are needed by an application and must be specified in the policy file. We describe this tool in Section 5.4.

5.1 Implementation of Copy-on-Write File System

The basic isolation mechanism in IFS is a copy-on-write file system. For ease of implementation, we have developed a user-level prototype of this file system using FUSE [73] (version 2.6.0) running on the Linux kernel (version 2.6.15). FUSE (or File system in User Space) intercepts operations at the virtual file system (VFS) layer so that applications do not have to be modified to work with FUSE file systems. For each intercepted operation, FUSE makes calls to wrapper functions in a user-level process that performs all file system operations on behalf of the applications running in each IFS environment. This process, which we call the IFS monitor

process, implements our copy-on-write file system by appropriately redirecting operations to the base or IFS layer. This implementation runs on the Linux ext3 file system but is mostly independent of the base file system .

IFS implements copy-on-write at the file-system level as shown in Figure 5.1. The implementation for files is straight-forward – files are copied from the base to the IFS whenever file data or attributes are modified. An IFS directory is an overlay that only contains files or sub-directories in IFS. It is created when 1) a base file or sub-directory within the directory is modified, or 2) an IFS file or sub-directory within the directory needs to be created. For example, when a base file is modified or a file is created, IFS directories are created for all ancestor directories of the file. In our default configuration, the base layer is the root (/) directory and the IFS overlay is in a special directory (/_*ifs*) but this configuration can be easily modified.

The wrapper functions in the IFS monitor process operate on VFS file system calls. Each wrapper function starts by looking up the path argument in the call, with the exception of the read and write functions that operate on file descriptors. At each level of the path, the lookup checks for the file (or directory) in the IFS overlay and then, if the file or directory is not found there, in the base system. If the file is found in the IFS or if it will not be modified by the operation, the monitor process executes the system call. Otherwise, the file is copied to the IFS overlay before executing the system call.

An application running inside the IFS isolation environment sees a combined view of the base and IFS. This is implemented in the wrapper function for the `readdir` library call. The IFS function returns all the directory entries from the IFS overlay and only those entries from the base that have not been copied to IFS.

The implementation must handle three main issues. First, a create-delete ambiguity is introduced when a file that was copied from base is removed in IFS. We must ensure that future system calls do not access the file in the base system and re-copy it. Our implementation ensures correct copy-on-write operation by creating an empty, zero-permission file of the same name that has the sticky bit set. We can safely use this bit since it is ignored by the Linux

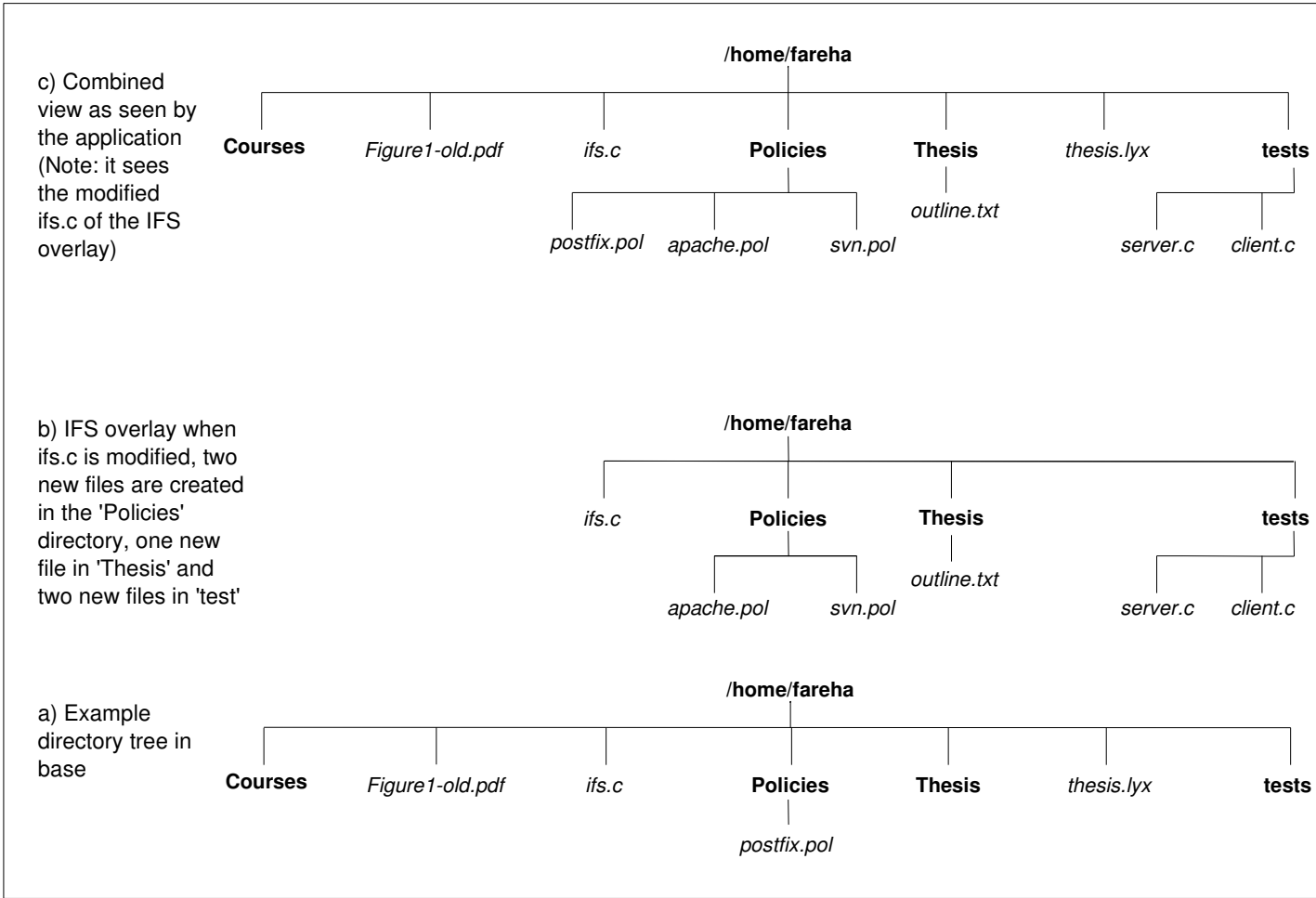


Figure 5.1: IFS Copy-on-Write

kernel¹. As a consequence, when a new file is created with the same pathname as a removed file (that was copied from base), the implementation removes the dummy file before attempting to create the new one. We could have also used the `RemovedInodeDB`, described below, which contains the inode and generation numbers of files copied from the base and removed in IFS. However, we avoid a database lookup for efficiency purposes, and use the sticky bit instead.

Second, Solitude supports a sharing policy called *commit sharing* that allows a file in the IFS overlay to be committed to the base file system. This commit may cause file conflicts when a base file and the corresponding IFS file are modified concurrently. To detect such conflicts, the implementation uses a database table, `InodeDB`, to record the time when a file is first created in IFS along with a one-to-one mapping of the base inode and generation number to IFS inode and generation number of the file. The inode and generation number uniquely identify files in a Unix file system, and thus `InodeDB` helps correlate a file in the base to the corresponding file in IFS during commit. The time-stamp is used to detect file-content conflicts, which occur when this time is earlier than the base file modification time. This same information is also stored in a `RemovedInodeDB` table, but for files that have been removed in IFS after being copied from the base. When a file is removed from the IFS overlay, its record is moved from `InodeDB` to `RemovedInodeDB`. This information is used during commit to detect if a file is modified in the base but deleted in the IFS (i.e., a remove-update conflict). Note that when an IFS application removes a base file, then the file has not been previously copied to the IFS layer, and the creation time stored in `RemovedInodeDB` is the time of deletion. It is possible to combine the two tables and use a single bit to indicate if the file exists or is removed, but we keep two separate tables because `InodeDB` is needed for hardlinks, as explained below, while `RemovedInodeDB` is not needed, and hence we wanted to keep two independent tables. When files or directories are removed in the base, the tables can become inconsistent. These consistency issues are handled at the time of commit. Also when a file is committed, Solitude's commit process updates the tables to remove records of these files.

¹On old Unix system, the sticky bit caused executable files to be hoarded in swap space [44].

Finally, hardlinks in Unix file systems, which allow a single file to have more than one name, create several complications. For example, if a file is modified and copied over to the IFS layer, all future operations to this file must be directed to the file in IFS, even if the pathname is different. The implementation uses two mapping tables for hardlinks. The `InodeDB` table, discussed above, stores the mapping from base inode (and generation number) of a file to the corresponding IFS inode (and generation number). A second database table, `ParentDB`, stores the mapping from IFS inode to the file name and the IFS inode of its parent. This latter database can have a one-to-many mapping when an IFS inode has multiple names. These two databases allow mapping a pathname in the base to a pathname in the IFS. This mapping starts by using the base inode to lookup the corresponding IFS inode from `InodeDB`, and then performs a reverse lookup using `ParentDB` to build the required pathname. Table 5.1 shows a summary of the three databases and how they are used by the IFS implementation. The databases are implemented using Berkeley DB4.

It is difficult to mimic the exact behaviour of the base file system in the presence of hardlinks. It requires an appreciable amount of bookkeeping and introduces significant complexity in the code. For example, a shortcoming of our implementation is that we do not track link counts of files. In the base, when a file with several links is removed, the link count is reduced and the pathname is deleted but the file inode and contents remain until the link count is zero. In the IFS implementation, we diverge from the expected behaviour only when a file has more than one link in the base, but only one link has been copied to IFS. In this case, if the file in IFS is deleted, we remove the file contents too. Now when another pathname of this file, which has not been copied to the overlay, is accessed, it will be copied from the base and will not have the changes that were made to the first file. A possible solution is to track link counts of a file in base. If the link count in the IFS reaches zero, but not in the base, the file should not be deleted. To work correctly, this method requires tracking link counts in the base of every file copied to IFS, garbage collecting files that are kept around due to hardlinks still present in the base, and performing additional checks during lookup. Despite this shortcoming

Database Name	Database Columns	Commit Sharing	Hardlinks
InodeDB (one-to-one mapping)	base inode, base generation→IFS inode, IFS generation, time of copy	Base inode, base generation, IFS inode and IFS generation for mapping base file to IFS file. Time of copy to detect conflicts.	Base inode, base generation, IFS inode and IFS generation for mapping one pathname in base to a corresponding pathname in IFS.
RemovedInodeDB (one-to-one mapping)	base inode, base generation→IFS inode, IFS generation, time of copy	Base inode, base generation, IFS inode and IFS generation for mapping base file to IFS file. Time of copy to detect conflicts.	N/A
ParentDB (one-to-many mapping)	IFS inode→IFS filename, IFS parent inode	N/A	IFS inode, IFS file name and IFS parent inode to form a full path by reverse lookup.

Table 5.1: Summary of IFS databases

in our implementation, all the applications we tested work as expected. As a result, in the future we plan to analyze the necessity and benefits of hardlinks compared to the complexity and overhead they introduce in systems.

5.2 Implementation of Chroot Isolation Model

Our isolation model consists of the copy-on-write file system reinforced with a Linux chroot jail. We set up an IFS isolation environment by creating a mount point in the base file system, as required by the FUSE implementation to mount the copy-on-write file system. By default, we create this mount point in the `/tmp` directory. Then we start the target application(s) within a hardened chroot jail [65]. Setting up the hardened jail consists of performing a regular chroot system call on the mount point directory, followed by an `ioctl` system call to set the special chroot barrier flag on this directory. This has been implemented on the Linux `ext3` file system and should be trivial to port to most Unix file systems.

By default, the top-level IFS application assumes the ID of the user invoking the IFS environment. However, if a user is specified in the policy file (e.g., see the first line of the Web server policy file shown in Figure 6.1), this user ID is used only if the invoker is either root or the same as the specified user and the file is owned by user root and not writable by others. Similarly, applications can only acquire the capabilities shown in Table 4.1 if the corresponding policy file is root owned and non-world writable. This makes it difficult for normal system users to tamper with policy files and damage our isolation environment. For example, if a user can easily modify a policy, she can cause the Apache2 web server to run with her own user ID. Consequently, when a system administrator attempts to start the server it would not be able to bind to port 80 and would fail to run.

5.3 Implementation of Capability Model

The capabilities in Table 4.1 are implemented using two methods: the Fcap capability is enforced by the IFS monitor process, while the per-application CAP capability is implemented by modifying Forensix [24], a kernel-level system-call interception facility. However, the IFS monitor process must also deal with certain CAP capabilities because it executes file system calls on behalf of the application running in the IFS and may need some file related capabilities to execute the calls successfully. This section first describes the implementation of the Fcap capability and then the CAP capability.

The Fcap capability applies to all programs running inside an IFS and must be enforced before the execution of any program. Therefore, this capability is implemented by the IFS monitor process during the initialization of the IFS environment. Since the IFS monitor process is setuid root, it has all powerful root privileges at start-up. Most importantly, it has the `dac_override` capability that allows the IFS monitor process to copy the file paths specified with Fcap capabilities to the IFS layer and change the ownership or permissions according to the policy file.

After handling the Fcap capability, the IFS monitor process drops all its privileges unless certain selected CAP capabilities are specified in the policy file associated with the IFS. Since the IFS monitor process acts on behalf of applications running in the isolation environment, as described in Section 5.1, it may require some capabilities that are given to programs in the IFS. These include the `setuid`, `setgid`, `chown`, and `mknod` capabilities. For example, an application that needs the `cap_setuid` and `cap_setgid` changes the id with which it runs. For the monitor process to accurately execute file system calls on behalf of this application, it needs to change its id to match that of the calling process. Additionally, in this case, the monitor process needs the `chown` capability to set the ownership of files and directories during copy-on-write to ensure that the application is subject to the same access control permissions in IFS as if it was running in the base, even when it changes its id. Similarly, if an application requires the `chown` capability, such as the FTP server, with the policy shown in Figure 6.5, the

monitor process must also have the capability to execute the chown on behalf of the application.

The CAP capability is implemented in the kernel for two reasons. First, the default Linux kernel build does not allow one user-level process to set the capabilities of another process for security reasons. Second, the available versions of the Linux kernel clear all capabilities across the `execve` system call and the CAP capability is per executable. Therefore, to support current applications without modification, our implementation must enforce the CAP capability in the kernel. The capability is passed to the kernel at IFS start-up and stored in a per-IFS kernel data structure consisting of the application paths and corresponding Linux capabilities. During the `execve` system call, the path parameter is compared to the stored application paths. If a match is found, the capabilities are set to exactly those stored in the data structure. In the absence of a matching path, all capabilities are cleared. Unlike the current Linux security model in which applications that require any capabilities are run with all capabilities (as root), our implementation ensures that only the capabilities needed by any application are given to it, thereby restricting the Linux security model. Section 6.1 in our evaluation shows examples that illustrate the use of Fcap and CAP capabilities

5.4 Specifying Capabilities

It is important for policy file authors to specify only the minimum set of capabilities necessary to run a privileged application. If necessary capabilities are omitted from the policy file, the program may fail mysteriously or behave incorrectly because most applications are written using an all-or-nothing model of root privileges. If extraneous capabilities are allowed in the policy file, an attacker will have more tools at his disposal should a program ever be subverted. Since a process can hold any combination of 30 distinct POSIX capabilities, the task of determining the minimum set of capabilities can be challenging.

To simplify this task, we wrote a simple tool that helps with writing policy files. It uses `ptrace` functionality to profile an application's system calls across all of its processes. This

`cap_profile` tool starts by running an application with no privileges and then uses the error values in system call return codes, in particular the “permission denied” error, to re-run the application with one new capability added at a time. If the addition of the new capability results in fewer permission errors or a different set of failing system calls, the new capability is added to the set of necessary capabilities. This process is repeated until the application’s system calls no longer produce any permission errors.

Of course, some applications will fail with permission denied errors even if they are fully privileged. For example, this may occur when the application attempts to access a resource whose authentication is handled by remote systems. This is taken into account by running the application once with full privileges and white-listing any permission-denied errors. Additionally, not all permission errors are encountered during the start-up of an insufficiently privileged application. For example, a Samba server needs the SETUID privilege only after a client connects and authenticates herself as a non-root user. Hence, `cap_profile` allows user interaction during each test run to exercise the application’s functionality and reveal the full set of required privileges. This process works well because most applications require few capabilities and typically at the beginning of the run. Section 6.1 shows the capabilities determined by the tool.

Chapter 6

Evaluation

We evaluate the IFS isolation environments using two criteria: 1) the effort involved in configuring the capabilities for server-side applications run within IFS environments to determine the usability of the system and 2) the performance overhead of IFS as well as the entire Solitude system because IFS is most useful when used with Solitude.

6.1 Policy Files

In this section, we discuss the usability of our system by describing examples of capability policies for various classes of server-side applications suited for IFS environments. We wrote and tested policies for server applications like a web server (Apache2), a web server with a php-based photo application (Gallery), a mail server (Postfix and Procmail), an IMAP server (Dovecot), an ftp server (vsftpd), a DHCP server (dhcpd3), a print server (Cupsd), and an SVN server (Svnserve) based on some of the services running on our cluster server [36]. We used our `cap_profile` tool to derive the policies for these applications. The full policy files with the explicit sharing policies are shown for completeness, but this thesis focuses on the capability specifications only.

```
Application /usr/sbin/apache2 www-data.www-data
Fcap /var/run/apache2.pid www-data.www-data
Fcap /var/log/apache2/error.log www-data.www-data
Fcap /var/log/apache2/access.log www-data.www-data
CAP net_bind_service
Wcommit /var/log/apache2/error.log
Wcommit /var/log/apache2/access.log
```

Figure 6.1: Policy for the Apache2 web server

6.1.1 Web Server: Apache2 and Apache2 + Gallery

The basic web server policy, first shown in Figure 4.1, is repeated in Figure 6.1 for clarity. This policy will allow Apache2 to run only if the policy file is root owned and will ensure that Apache2 is run with the user id and group id of www-data. The three files with the Fcap capability are copied to the IFS layer and the ownership changed from root to www-data. These capabilities are needed so the web server can access these files while running as the www-data user rather than root. (Note that the files are owned, readable and write-able only by the root user in the base environment). However, we observed that these are Apache2 files that are accessed only by Apache2, and thus there is no real need for them to be root owned. The `net_bind_service` capability is needed at the beginning of execution to bind to the privileged port 80.

We also downloaded and ran the Gallery application [47] within the web server running in an IFS. We tested adding and removing users, pictures and albums, and found that Apache2 does not require any additional capabilities compared to the capabilities shown in Figure 6.2. With the IFS copy-on-write file system we observed that Gallery stores albums, pictures, album users, etc. in the `/var/www/albums` directory.

```
Application /usr/sbin/apache2 www-data.www-data
Fcap /var/run/apache2.pid www-data.www-data
Fcap /var/log/apache2/error.log www-data.www-data
Fcap /var/log/apache2/access.log www-data.www-data
CAP net_bind_service
Wcommit /var/log/apache2/error.log
Wcommit /var/log/apache2/access.log
Wcommit /var/www/albums
```

Figure 6.2: Policy for Gallery running on Apache2

```
Application /usr/sbin/postfix root.root
Wshare /home/fareha/Maildir
Fcap /var/spool/postfix/pid/ root.root
Fcap /var/spool/postfix/private perm=00750
CAP /usr/lib/postfix/master net_bind_service setgid setuid
CAP /usr/lib/postfix/pickup setgid setuid
CAP /usr/lib/postfix/qmgr setgid setuid
CAP /usr/lib/postfix/smtpd setgid setuid
CAP /usr/lib/postfix/trivial-rewrite setgid setuid
CAP /usr/lib/postfix/local setgid setuid
CAP /usr/lib/postfix/cleanup setgid setuid
CAP /usr/lib/postfix/proxymap setgid setuid
```

Figure 6.3: Policy for the Postfix MTA

6.1.2 MTA and MDA: Postfix and Procmail

The Postfix policy is shown in Figure 6.3. By default, the Postfix server runs its master process as the root user but it also has various processes running as the postfix user. The root process switches ids from root to postfix and back several times. We also run the server with the user id of root to ensure correct behaviour.

Some of the files Postfix uses are owned, readable and writable only by the postfix user in the base. However, these files are accessed by the root process. Since the root user has no special privileges in IFS, in particular no `dac_override` capability, it needs file capabilities to access these files. For example, `/var/spool/postfix/pid` is a directory owned and writable only by the postfix user in the base. Interestingly, all the files within this directory are root owned, and changing `/var/spool/postfix/pid` to be root owned in IFS solves the access permission problems. Similarly, `/var/spool/postfix/private` is a directory that is owned by the postfix user and the root group in the base. Only the postfix user has read, write and execute permissions, but the process running as root user attempts to test for the existence of a file in this directory. Giving the group execute permissions in IFS enables the file existence test to succeed.

The master process of Postfix needs `net_bind_service` to bind to privileged port 25 when it starts, and `setuid` and `setgid` to switch to the postfix user id and back to the root user id. All the other processes are forked by the master process and thus start execution as the root user. They require `setuid` and `setgid` to later switch to the postfix user. The delivery process (`local`) also needs to change its identity to each user that receives mail.

6.1.3 IMAP Server: Dovecot

Figure 6.4 shows the policy file for the Dovecot IMAP server. Dovecot's main process and authorization process need to run as the root user to access the configuration files. Each user has an `imap-login` process associated with it that runs as the dovecot user and an `imap` process

```
Application /usr/sbin/dovecot root.root
Wshare /home/fareha/Maildir
CAP /usr/sbin/dovecot net_bind_service setuid setgid chown
CAP /usr/lib/dovecot/imap-login setgid setuid sys_chroot
CAP /usr/lib/dovecot/imap setgid setuid
```

Figure 6.4: Policy for Dovecot IMAP server

```
Application /usr/sbin/vsftpd
Fcap /var/ftp/pub ftp.root
CAP net_bind_service chown setuid setgid sys_chroot
```

Figure 6.5: Policy for vsftpd FTP server

that runs with the effective id of the user that is viewing their mail. The `dovecot`, `imap-login` and `imap` processes need the `setuid` and `setgid` capabilities to switch to the necessary ids.

In the default configuration, `imap-login` requires the `sys_chroot` capability to chroot the login process to `/var/run/dovecot/login`, where all the UNIX sockets needed by this process are created.

Finally, the `cap_net_bind_service` is needed by the main `dovecot` process to bind to port 993 for IMAP with SSL and port 143 without SSL.

As mentioned in the Introduction, IFS provides mechanisms for explicit sharing to support existing applications that may require sharing across IFS environments. Postfix and Dovecot IMAP are examples of such applications. Because they both access a user's mailbox, this mailbox must be shared if the two servers are running in separate IFS environments. For example, if the servers are configured to use the Maildir mailbox format, each user's `/home/<username>/Maildir` must be shared and if using the mbox format the `/var/mail` folder must be shared.

```
Application /usr/sbin/dhcpd3
CAP net_bind_service net_raw
```

Figure 6.6: Policy for dhcpd3 DHCP server

6.1.4 FTP Server: vsftpd

Using `vsftpd` requires some initial setup of the download and upload directories before starting the server. A typical configuration consists of creating a download directory with absolutely no write permissions for security purposes. An upload directory, owned and write-able by user `ftp`, is created inside the download directory. This directory hierarchy initialization is usually done in the base by the root user, who has no special capabilities in IFS. Therefore, simple copy-on-write does not work very well and the setup must be handled as a file capability as shown in the policy file in Figure 6.5.

When `vsftpd` starts, it performs a check to ensure that the main process is run as the root user, but it has various processes that run as the `ftp` user or the `nobody` user. It needs `setuid` and `setgid` for switching to these users. On Debian-like machines, `vsftpd` runs within in a `chroot` environment and thus needs the `sys_chroot` capability. `vsftpd` requires `chown` capabilities because it can be configured to change the ownership of all uploaded files to a normal non-system user for security purposes. Finally, `net_bind_service` is needed to bind to ports 21 and 22 when the server starts.

6.1.5 DHCP Server: dhcpd3

The policy file for the `dhcpd3` server is shown in Figure 6.6. The server requires two network capabilities, `net_bind_service` to bind to port 67 for UDP communication, and `net_raw` to use the packet interface on the device level as well as raw socket communication for ICMP.

```
Application /usr/sbin/cupsd cupsys.lp
CAP net_bind_service
```

Figure 6.7: Policy for cupsd printer server

```
Application /usr/bin/svnserve fareha.solitude
Wcommit /solitude/svn/testrep
```

Figure 6.8: Policy for svnserve SVN server

6.1.6 Printer Server: Cupsd

Figure 6.7 shows the Cupsd printer server policy file for client machines that access a remote print server. When this server is run with a user id of cupsys and a group id of lp, all the file permissions in the base are correct. The only capability needed by cupsd is `net_bind_service` to bind to ipp port 631 at start-up.

6.1.7 SVN Server: Svnserve

When setting up an SVN repository, all the users that will be accessing it are added to a group usually created for work on the project. The repository is also owned by this group so that the users have the necessary permissions to execute all SVN commands. As seen from the policy in Figure 6.8, an SVN server can run as a normal user, but it must run as the group that owns the repository to give it correct access to all repository files. The Svnserve server binds to a non-privileged port. To run several servers, Svnserve can be configured to bind to different port numbers. In this case, it is possible to associate a different policy file with each server running in a separate IFS. However, the standard IFS paths for the overlay directory and mount point will also need to be reconfigured to ensure that each server is running in isolation from the other.

6.1.8 Discussion

The example policy files in the previous section show that our policies are short (no longer than 15 lines) and intuitive. We found that the CAP capabilities needed by applications are easy to configure because without the required privileges, applications fail due to denied permissions and give meaningful error messages that help determine the required CAP capabilities. Furthermore, since they are usually needed at the beginning of execution, finding the necessary capabilities typically takes just a few minutes to a few hours. Only the `vsftpd` policy took a few hours because the capability it required depended on a distribution-specific configuration (`vsftpd` always runs in a `chroot` environment in Debian-like machines) and also full testing required uploading and downloading files. The challenge here is to find the *minimum* set of capabilities. Our `cap_profile` tool, discussed in Section 5.4, makes use of the “permission denied” error messages and simplifies this task.

File capabilities, on the other hand, are more time consuming to determine. The first few policy files we wrote, including `Apache`, `vsftpd` and `Postfix`, took up to a few weeks because we were still developing the appropriate model for specifying file capabilities. After writing the `vsftpd` and `Postfix` policies, we had a clear, well-defined model for specifying and implementing file capabilities. The later policies then just took on the order of hours to write correctly.

Unlike Linux capabilities, file capabilities may manifest themselves as more subtle differences in behaviour well into the execution of the application. For example, although a file system call may fail, no error is displayed and the action being performed just fails silently. Determining the file capabilities may then require looking at long system call traces of the application running inside and outside an IFS environment, as obtained from the Linux `strace` command, and finding differences and denied accesses to files that may occur well before the application output diverges from the expected. We plan to extend our `cap_profile` tool in the future to aid in determining file capabilities. In particular, any application requiring a file-related Linux capability, such as `cap_dac_override` should be further analyzed to find out the

exact failure and the permission or ownership change that would fix the problem.

6.2 Performance Overhead

We measured the overhead introduced by IFS and also Solitude by running a set of benchmarks representing different client or server workloads. We ran two client workloads within an IFS: 1) `untar` of a Linux kernel source tarball, representing a file-system-intensive workload, and 2) kernel build of the Linux sources, which is mainly CPU bound and determines the overhead imposed when running similar CPU bound applications in a regular desktop environment. We ran three server workloads in an IFS: 1) a large 230 MB file download, which stresses the file-system read performance and represents a media streaming server, 2) a large 230 MB file upload, which stresses the file-system write performance and represents an FTP or a video blogging site, and 3) the Apache `ab` benchmark, which stresses a standard Apache web server by issuing back-to-back requests with four concurrent processes running 20 clients that request files ranging from 1KB to 15KB, and is representative of a loaded server environment.

We ran the tests on a Solitude-enabled Ubuntu Linux 6.06 machine with four Intel(R) Xeon(TM) CPU 3.00GHz processors, 2GB of RAM and a local `ext3` hard disk. The client machine for the server experiments is connected to the target machine with a Gigabit network. We repeated each test at least 5 times and our results are averaged over these tests.

Figure 6.9 shows the performance overhead of FUSE, IFS and Solitude for the five benchmarks compared to a regular Linux system. The overhead is in terms of running time for the first four experiments and in terms of network throughput for the CPU-saturated web server benchmark. Each segment of the bar shows the overhead introduced by the various components of Solitude. We obtained these results by starting with the base Linux system and then running experiments that progressively added these components one at a time. The components include 1) the pass-through user-level file system built on FUSE, 2) the basic copy-on-write IFS environment, 3) IFS sharing and capability policy module, and 4) the Solitude kernel-level tainting



figures/ifs_performance.pdf

Figure 6.9: Performance Overhead of IFS

module. The tainting module is run with no tainted files or processes to isolate the overhead introduced by logging.

The Untar test creates a large number files and directories, stressing the IFS file system. The FUSE overhead is largely a consequence of file system operations being redirected into user-space code which then makes more system calls into the kernel. In particular, each create system called is translated into a `getattr`, `create`, `getattr` by FUSE. As a result, the user-level IFS code also incurs significant overhead. We expect both these overheads to decrease dramatically with a kernel-level implementation. The Solitude overhead occurs almost entirely due to hard links. Solitude, in addition to tainting, provides a file generation number for uniquely identifying files to the IFS code. As described in Section 5.1, the code stores the inode and generation number in a persistent mapping table for correctly handling the multiple names of a file due to hard links. In the future, we plan to assess whether hardlinks are sufficiently useful for IFS applications to justify the implementation complexity and overhead.

The Build and the Apache benchmarks have smaller overhead than Untar in IFS because they are comparatively less file system intensive. The Upload benchmark stresses the FUSE code in IFS since the large file is written in 4KB chunks due to a limitation in the FUSE write implementation. The Download benchmark has no overhead because it involves simply opening the file and performing reads on the file handle.

Chapter 7

Conclusion

Current operating systems provide a single common namespace that is shared by all users and processes. This implicit file sharing can lead to attacks from a single application compromising the entire system. This problem is only becoming worse as users increasingly download and install software from untrusted sources on the Internet. The IFS isolation environment addresses this shortcoming of modern operating systems. The key problem that this thesis solves is to limit the effects of attacks on systems by providing a file-system based, restricted privilege isolation environment, that is designed for *existing* applications. Applications are given their own separate namespace through a chroot-ed copy-on-write file system. The IFS capability restrictions ensure that even if malware compromises a legitimate program running with certain privileges in its isolation environment, then it would be unable to embed itself deep into the system (e.g. by loading a kernel module) because the host application would likely possess only a few capabilities.

We evaluated running several server-side applications inside our IFS isolation environment. Our evaluation shows that while finding the Linux capabilities to be specified in a policy is fairly easy, determining the file capabilities can be more time consuming and requires careful inspection of `strace` outputs. Our experience shows that file capabilities are rare and would be unnecessary if developers program with a restricted privilege model in mind rather than the

more common all-powerful root. We believe that typical network applications can and should always be run inside an IFS isolation environment since it provides a good balance between security and usability.

7.1 Future Work

There are several directions of future work related to this project. We describe improvements to IFS as well as ideas for future directions for IFS.

Our evaluation showed that finding file capabilities was the more difficult part of writing policy files. As future work, we plan to enhance our `cap_profile` tool for determining Linux capabilities to find file capabilities as well. This will allow a policy file author to more accurately and quickly write correct policies.

Currently an application running outside an IFS environment cannot be switched automatically and dynamically to run inside an IFS and also one IFS environment cannot be embedded inside another. We are exploring if these features are feasible and beneficial in further limiting the effect of attacks on the system. Furthermore, our current user-level prototype has poor performance due to FUSE. We would like to investigate if a kernel-level implementation would be able to provide the dynamic switching, as well as offer better performance.

While auditing the IFS source code, and testing and evaluating its implementation, we made two important observations. First, although our hardlink model does not exactly mirror that of Linux, all the applications we run in the IFS environment behave as expected. As a result, we believe that as future work we need to study the necessity and benefits of hardlinks versus the complexity and performance overhead they cause in the implementation of systems.

Second, all of the file capabilities specified in policy files seem to be due to imperfect DAC permissions on the files accessed by the application, or more generally, misconfiguration. We plan to explore the use of the IFS infrastructure as a debugging environment for configuration management making use of the fact that IFS provides a view of how an application accesses

and modifies every file. We believe this information can also be used to, for example, analyze malware that use various methods to deliver payloads that modify executables, delete files, install backdoors, Trojans or rootkits, and encrypt files in extortion attacks, etc.

Bibliography

- [1] Anurag Acharya and Mandar Raje. MAPbox: Using parameterized behaviour classes to confine untrusted applications. In *Proceedings of the USENIX Security Symposium*, August 2000.
- [2] Albert D. Alexandrov, Maximilan Ibel, Klaus E. Schauser, and Chris J. Scheiman. Extending the operating system at the user level: the Ufo global file system. In *Proceedings of the USENIX Technical Conference*, 1997.
- [3] Lee Badger, Daniel F. Sterne, David L. Sherman, and Kenneth M. Walker. A domain and type enforcement UNIX prototype. *Computing Systems*, 9(1):47–83, Winter 1996.
- [4] Lee Badger, Daniel F. Sterne, David L. Sherman, Kenneth M. Walker, and Sheila A. Haghghat. Practical domain and type enforcement for UNIX. In *Proceedings of the IEEE Symposium on Security and Privacy*, 1995.
- [5] Dirk Balfanz and Daniel R. Simon. WindowBox: A simple security model for the connected desktop. In *Proceedings of the USENIX Windows Systems Symposium*, August 2000.
- [6] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. Xen and the art of virtualization. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 164–177, October 2003.

- [7] A. Berman, V. Bourassa, and E. Selberg. TRON: Process-specific file protection for the UNIX operating system. In *Proceedings of the USENIX Technical Conference*, pages 165–175, New Orleans, LA, USA, 1995.
- [8] David Brumley and Dawn Song. Privtrans: Automatically partitioning programs for privilege separation. In *Proceedings of the USENIX Security Symposium*, pages 57–72, August 2004.
- [9] Peter M. Chen and Brian D. Noble. When virtual is better than real. In *Proceedings of the USENIX Workshop on Hot topics in Operating Systems*, 2001.
- [10] Shuo Chen, John Dunagan, Chad Verbowski, and Yi-Min Wang. A black-box tracing technique to identify causes of least-privilege incompatibilities. In *Proceedings of the Network and Distributed System Security Symposium*, 2005.
- [11] Sailesh Chutani, Owen T. Anderson, Michael L. Kazar, Bruce W. Leverett, W. Anthony Mason, and Robert N. Sidebotham. The Episode file system. In *Proceedings of the USENIX Technical Conference*, 1992.
- [12] Brian Cornell, Peter Dinda, and Fabián Bustamante. Wayback: A user-level versioning file system for linux. In *Proceedings of the USENIX Technical Conference*, pages 19–28, June 2004.
- [13] Microsoft Corporatin. Microsoft softgrid. <http://www.microsoft.com/windows/products/windowsvista/enterprise/softgrid.aspx>, 2007.
- [14] Microsoft Corporatin. Windows Vista: Features explained: Internet Explorer protected mode. <http://www.microsoft.com/systemcenter/softgrid/evaluation/virtualization.aspx>, 2007.
- [15] Crispin Cowan, Steve Beattie, Greg Kroah-Hartman, Calton Pu, Perry Wagle, and Vir-

- gil D. Gligor. SubDomain: Parsimonious server security. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, pages 355 – 368, December 2000.
- [16] Richard S. Cox, Jacob Gorm Hansen, Steven D. Gribble, and Henry M. Levy. A safety-oriented platform for web applications. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 2006.
- [17] Scott W. Devine, Edouard Bugnion, and Mendel Rosenblum. Virtualization system including a virtual machine monitor for a computer with a segmented architecture. US patent, 6397242, October 1998.
- [18] David Ferraiolo and Richard Kuhn. Role-based access control. In *Proceedings of the National Computer Security Conference*, 1992.
- [19] David F. Ferraiolo, Janet A. Cigini, and D. Richard Kuhn. Role-based access control (RBAC): Features and motivations. In *Proceedings of the Annual Computer Security Applications Conference*, 1992.
- [20] Michail D. Flouris and Angelos Bilas. Clotho: Transparent data versioning and the block I/O level. In *Proceedings of the IEEE Symposium on Mass Storage Systems*, April 2004.
- [21] Steve Friedl. Best practices for UNIX chroot() operations. <http://www.unixwiz.net/techtips/chroot-practices.html>, January 2002.
- [22] Todd Gamble. Implementing execution controls in UNIX. In *Proceedings of the 7th System Administration Conference*, 1993.
- [23] Tal Garfinkel, Ben Pfaff, Jim Chow, Mendel Rosenblum, and Dan Boneh. Terra: A virtual machine-based platform for trusted computing. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 193–206, October 2003.

- [24] Ashvin Goel, Wu chang Feng, Wu chi Feng, David Maier, and Jim Snow. Automatic high-performance reconstruction and recovery. *Journal of Computer Networks*, 51(5):1361–1377, April 2007. Special issue on “From Intrusion Detection to Self-Protection”.
- [25] Ashvin Goel, Kenneth Po, Kamran Farhadi, Zheng Li, and Eyal de Lara. The Taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 163–176, October 2005.
- [26] Ian Goldberg, David Wagner, Randi Thomas, and Eric A. Brewer. A secure environment for untrusted helper applications. In *Proceedings of the USENIX Security Symposium*, 1996.
- [27] Andrew Grunbacher. Linux extended attributes and ACLs. <http://acl.bestbits.at>, February 2005.
- [28] Serge E. Hallyn. POSIX file capabilities: Parceling the power of root. <http://www.ibm.com/developerworks/linux/library/l-posixcap.html>, May 2007.
- [29] Serge E. Hallyn and Phil Kearns. Domain and type enforcement for linux. In *Proceedings of the Linux Showcase and Conference*, October 2000.
- [30] Roger Haskin, Yoni Malachi, Wayne Sawdon, and Gregory Chan. Recovery management in QuickSilver. *ACM Transactions on Computer Systems*, 6(1):82 – 108, 1988.
- [31] Matt Hines. Google buys into security, acquires GreenBorder. http://www.infoworld.com/article/07/05/29/Google-buys-into-AV_1.html, May 2007.
- [32] Francis Hsu, Hao Chen, Thomas Ristenpart, Jason Li, and Zhendong Su. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the Annual Computer Security Applications Conference*, December 2006.
- [33] VMWare Inc. VMware virtual machine technology. <http://www.vmware.com/>.

- [34] Trent Jaeger, Reiner Sailer, and Xiaolan Zhang. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the USENIX Security Symposium*, pages 59–74, August 2003.
- [35] K. Jain and R. Sekar. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Network and Distributed System Security Symposium*, 2000.
- [36] Shvetank Jain. A framework for application-level isolation and recovery. Master’s thesis, University of Toronto, Toronto. In preparation.
- [37] Poul-Henning Kamp and R.N.M. Watson. Jails: Confining the omnipotent root. In *Proceedings of the Second International SANE Conference*, 2002.
- [38] Douglas Kilpatrick. Privman: A library for partitioning applications. In *Proceedings of the Freenix Track of USENIX Technical Conference*, 2003.
- [39] Samuel T. King and Peter M. Chen. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 223–236, October 2003.
- [40] Puneet Kumar and Mahadev Satyanarayanan. Flexible and safe resolution of file conflicts. In *Proceedings of the USENIX Technical Conference*, pages 95–106. USENIX, January 1995.
- [41] Ninghui Li, Ziqing Mao, and Hong Chen. Usable mandatory integrity protection for operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*, pages 164–178, 2007.
- [42] Zhenki Liang, V.N. Venkatakrishnan, and R. Sekar. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the Annual Computer Security Applications Conference*, 2003.
- [43] Linux. Man capabilities(7) in Linux man page. Confirming to POSIX.1e.

- [44] Linux. Man chmod(1) in Linux man page. Confirming to POSIX.1e.
- [45] Linux-VServer. <http://www.linux-vserver.org>, 2006.
- [46] Peter Loscocco and Stephen Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the Freenix Track of USENIX Technical Conference*, June 2001.
- [47] Bharat Mediratta. Gallery photo album organizer. <http://gallery.menalto.com/>, 2004.
- [48] Medusa DS9 security system. <http://medusa.terminus.sk>, viewed in November 2007.
- [49] Sun Microsystems. Zfs. <http://opensolaris.org/os/community/zfs>.
- [50] Microsoft Corporation Mike Friedman. IEBlog: Protected mode in Vista IE7. <http://blogs.msdn.com/ie/archive/2006/02/09/528963.aspx>, September 2006.
- [51] Kiran-Kumar Muniswamy-Reddy, Charles P. Wright, Andrew Himmer, and Erex Zadok. A versatile and user-oriented versioning file system. In *USENIX Conference on File and Storage Technologies*, 2004.
- [52] Amon Ott. Rule set based access control as proposed in the generalized framework for access control approach in linux. Master's thesis, University of Hamburg, November 1997. <http://www.rsbac.org/papers.htm>.
- [53] Zachary N.J. Peterson and Randal Burns. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage*, 1(2):190–212, May 2005.
- [54] Daniel Price and Andrew Tucker. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the USENIX Large Installation Systems Administration Conference*, 2004.

- [55] Neil Provos. Improving host security with system call policies. In *Proceedings of the USENIX Security Symposium*, pages 257–272, August 2003.
- [56] Neil Provos, Markus Friedl, and Peter Honeyman. Preventing privilege escalation. In *Proceedings of the USENIX Security Symposium*, pages 231 – 242, August 2003.
- [57] Sean Quinlan and Sean Dorward. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*, January 2002.
- [58] Peter Reiher, John S. Heidemann, David Ratner, Gregory Skinner, and Gerald J. Popek. Resolving file conflicts in the Ficus file system. In *USENIX Technical Conference*, pages 183–195. USENIX, June 1994.
- [59] Ravi S. Sadhu, Edward J. Coyne, Hal L. Feinstein, and Charles E. Youman. Role-based access control models. *IEEE Computer*, 29(2):38–4, February 1993.
- [60] Jerome H. Saltzer. Protection and the control of information in multics. *Communications of the ACM*, 17(7):338 – 402, July 1974.
- [61] Douglas S. Santry, Michael J. Feeley, Norman C. Hutchinson, Alistair C. Veitch, Ross W. Carton, and Jacob Ofir. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 110–123, December 1999.
- [62] Frank Schmuck and Jim Wylie. Experience with transactions in QuickSilver. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 239–253, 1991.
- [63] Fred B. Schneider. Enforceable security policies. *ACM Transactions on Information System Security (TISSEC)*, 3(1):30–50, 2000.
- [64] Kevin Scott and Jack Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of the Annual Computer Security Applications Conference*, pages 209–218, 2002.

- [65] Secure chroot barrier - Linux-Vserver. http://linux-vserver.org/Secure_chroot_Barrier, viewed in Aug 2007.
- [66] Jonathan S. Shapiro, Jonathan M. Smith, and David J. Farber. EROS: A fast capability system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*, pages 170–185, 1999.
- [67] Stephen Soltesz, Herbert Pötzl, Marc E. Fiuczynski, Andy Bavier, and Larry Peterson. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the EuroSys conference*, pages 275–287, 2007.
- [68] Craig A. N. Soules, Garth R. Goodson, John D. Strunk, and Gregory R. Ganger. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*, pages 43–58, 2003.
- [69] John D. Strunk, Garth R. Goodson, Michael L. Scheinholtz, Craig A. N. Soules, and Gregory R. Ganger. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 165–180, 2000.
- [70] Weiqing Sun, Zhenkai Liang, R. Sekar, and V.N. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*, February 2005.
- [71] Michael M. Swift, Peter Brundett, Cliff Van Dyke, Praerit Garg, Anne Hopkins, Shannon Chan, Mario Goertzel, and Gregory Jensenworth. Improving the granularity of access control in Windows NT. In *ACM Symposium on Access Control Models and Technologies*, May 2001.
- [72] SWSOft. Virtuozzo linux virtualization. <http://www.virtuozzo.com>.
- [73] Miklos Szeredi. File system in user space (FUSE). <http://fuse.sourceforge.net>.

- [74] Richard Ta-Min, Lionel Litty, and David Lie. Splitting interfaces: Making trust between applications and operating systems configurable. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*, pages 279–292, November 2006.
- [75] Douglas B. Terry, Marvin M. Theimer, Karin Petersen, Alan J. Demers, Mike J. Spreitzer, and Carl H. Hauser. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)*, pages 172–183, December 1995.
- [76] David Thiel. Exposing vulnerabilities in media software. Black Hat USA 2007, <http://www.blackhat.com/html/bh-usa-07/bh-usa-07-speakers.html#thiel>, August 2007.
- [77] Surendra Verma and Charles Torre. Vista transactional file system, December 2005. <http://channel9.msdn.com/Showpost.aspx?postid=142120>.
- [78] Kenneth M. Walker, Daniel F. Sterne, M. Lee Badger, Michael J. Petkac, David L. Sherman, and Karn A. Oostendorp. Confining root programs with domain and type enforcement. In *Proceedings of the USENIX Security Symposium*, 1996.
- [79] Andrew Whitaker, Marianne Shaw, and Steven D. Gribble. Denali: Lightweight virtual machines for distributed and networked applications. Technical report, University of Washington, 2002. 02-02-01.
- [80] D. R. Wichers, D. M. Cook, R. A. Olsson, J. Crossley, P. Kerchen, K. N. Levitt, and R. Lo. PACL’s: An access control list approach to anti-viral security. In *Proceedings of the National Computer Security Conference*, pages 340–349, October 1990.
- [81] Charles P. Wright, Jay Dave, Puja Gupta, Harikesavan Krishnan, David P. Quigley, Erez Zadok, and Mohammad Nayyar Zubair. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage*, 2(1):74–105, March 2006.

- [82] Huagang Xie and et al. Linux intrusion detection system (LIDS) project. <http://www.lids.org/>.
- [83] Adam Young and Moti Yung. Cryptovirology: Extortion-based security threats and countermeasures. In *Proceedings of the IEEE Symposium on Security and Privacy*, May 1996.
- [84] Ningning Zhu and Tzi-Cker Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*, pages 217–226, June 2003.