A Framework for Application-Level Isolation and Recovery

by

Shvetank Jain

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

Abstract

A Framework for Application-Level Isolation and Recovery

Shvetank Jain

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. This problem can be especially severe for persistent state, and it occurs due to implicit sharing in operating systems. 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 time-consuming and error-prone process.

In this thesis, we propose limiting the effects of attacks and simplifying the post-intrusion recovery process by requiring explicit sharing of all persistent data. We present a system called Solitude that uses a copy-on-write filesystem to provide a transparent, restricted privilege sandboxing environment for running untrusted applications. Since file sharing across applications is relatively uncommon, Solitude uses an explicit sharing mechanism that limits attack propagation without compromising functionality.

Solitude provides two modes of recovery. If a sandboxed application proves to be untrustworthy, a course-grained recovery method allows completely removing the footprint of the software. However, if a user mistakenly moves untrusted files from the sandbox to the regular environment via the explicit sharing mechanism, then Solitude uses data dependency tracking to allow fine-grained recovery.

Acknowledgements

This thesis is the culmination of many days of hard work and dedication, and though presented as an individual work, could not have been completed without the support of the following people. First, I wholeheartedly thank Dr. Ashvin Goel for giving me the opportunity to be a part of this wonderful journey and for being a constant source of inspiration, encouragement and guidance every step of the way. Ashvin was a great mentor and friend. The long hours of hacking sessions, endless debates about design choices and countless culinary treats provided the much needed fuel essential for research. I would also like to thank Dr. David Lie and other members in the Systems group for their continuous constructive feedback and criticisms regarding the work.

Next, I'd like to thank my partners in crime, Fareha Shafique and Vladan Djeric, for keeping me company during the long working hours, and enriching my graduate school experience with their intelligence and humour.

I am honored to be a student of the University of Toronto and grateful for the financial support provided by the Department of Electrical and Computer Engineering. I would also like to thank Dr. Ravi Adve, Dr. Christiana Amza and Dr. Eyal de Lara for agreeing to be on my thesis committee despite their extremely busy schedules. Their time and expertise is greatly appreciated.

Finally, I would like to thank my family and friends for their unconditional love and for believing in me. This has truly been a memorable ride!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

There are several reasons why computer intrusions are not likely to go away. With Internet connectivity becoming indispensible for most computing tasks, computers are increasingly exposed to the perils of network attacks. Moreover, software is large and complex, often written by numerous people who may be spread across various geographical locations. Extensibility and customizability of platforms are starting to become a core requirement for wide scale adoption, taking away more control from the original developers. These challenges make it extremely difficult to protect against all possible vulnerabilities that may arise in software. At the same time, software continues to manage enormous amounts of critical personal, financial and corporate information. As a result, computer systems are an attractive target for attacks.

Detection and prevention of computer attacks have been active areas of research over the past decades. However, intrusions continue to occur causing huge losses to individuals, corporations and the government. As computer systems continue to be infiltrated and organizations lose customers and revenue due to attackers, the ability to do accurate analysis of attacks and post-intrusion recovery has become increasingly important. Currently, the analysis and recovery process is manual, time consuming and error prone. It is often the case that the operating system and the relevant applications need to be reinstalled, data retrieved from backups, vulnerability analyzed based on the attack footprint, patches installed and services restarted. In

this thesis, we attempt to improve the post-intrusion analysis and recovery process. In particular, we present a recovery system called Solitude that aims to improve the overhead, accuracy and usability of the recovery process.

1.1 Research Approach

The goal of the recovery process is to undo changes made by an attack without losing critical data. The recovery process can be thought of as a two step process. First, we need to determine the changes made by an attacker, and then second, revert those changes. However, a serious problem faced by security professionals while doing such recovery is distinguishing attack activity from legitimate user activity. For example, a malware may replace system binaries and the naive user may unknowingly continue to use the malicious binaries, entangling legitimate and malicious intent with each other.

This entanglement problem adversely affects the accuracy of the recovery process and occurs for several reasons. First, applications often have access to all user data. For example, a browser exploit can destroy the entire user home directory. Even though it may not be obvious, this is a serious problem because user data is generally not backed up by most home users, either due to limited expertise or lack of understanding of the severity of the problem. Secondly, applications run with substantial privileges. For example, most server applications run with root privileges, which when exploited can affect the entire system. Thus, attacks can affect large parts of the system making it hard to distinguish between legitimate and malicious activity.

Our approach aims to limit the damage that can be caused by the attack, which in turn helps to minimize the affect of the attack on legitimate activities. We isolate applications from each other limiting their access to only the essential data required by them for functionality. Moreover, we limit application privileges in adherence with the least privilege principle, minimizing the potential damage that can be caused by the exploited application. For isolation, we use a

copy-on-write isolation file system called IFS. [28] 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 a separate namespace.

Once an attack occurs, we aim to simplify the recovery process ensuring that critical data is safe. Our isolation approach facilitates two modes of recovery. If a sandboxed application proves to be untrustworthy, a course-grained recovery method allows a complete removal of the software's footprint. In particular, the application's IFS environment is discarded. This recovery method, being simple, is accessible to ordinary users. Moreover, it provides the user a comfortable window to evaluate the usefulness and legitimacy of the application. During this time period, if the application behaves in an unexpected or suspicious manner, the entire isolation environment can be discarded without jeopardizing the integrity of the rest of the system. For e.g., a browser exploit may delete the entire user home directory. However, since the modifications made by the browser are confined to the isolation environment, the contaminated IFS can simply be discarded without affecting the user's original home directory.

This recovery process, although simple and powerful, risks losing critical data. For e.g., the configuration settings, personal profile, downloaded content etc., of the browser would also be lost when discarding the isolation environment. Moreover, files may need to be shared across the isolation environments. For e.g., a movie downloaded via the browser may need to be shared with media player that executes in a separate isolation environment.. In order to secure critical files or share files across different isolation environments, we allow users to *commit* critical or shared data from an isolation environment to the base system. However, we realize that users rarely have perfect knowledge of a program's trustworthiness and that they may unwisely contaminate their base file system by committing malicious data or applications. If a user mistakenly moves untrusted files from the sandbox to the regular environment, then Solitude uses data dependency tracking to allow fine-grained recovery. Our tracking method uses a modified version of an intrusion recovery system called Taser. Below, we provide some background on Taser to motivate our approach.

1.2 Background and Motivation

Previously our group had designed and developed, a prototype intrusion analysis and recovery system called Taser [7]. Taser securely audits all system-level activities on a target system so that these activities can be diagnosed at a later time. After an attack, Taser provides tools for analyzing the audit log and an investigator can run taint propagation on kernel objects such as processes and files to determine the set of attack-related activities. Since Taser maintains a log of all file-system activities, it can revert the persistent changes made by an attack, e.g., removal of a trojan program.

The Taser approach is implemented within an operating system and requires *no* changes to existing applications, but it comes with a cost. First, the system needs to log all system-call activities since any activity could potentially be malicious. While such logging is not computationally intensive, it imposes heavy storage overhead and analysis can take a long time. This adversely affects the *overhead* of the recovery process. Second, and more seriously, while taint propagation is a useful tracking technique, it raises problems in current operating systems because a malicious application can affect the entire system causing widespread tainting of legitimate activities. For instance, in most Unix systems, any user or application can write to the shared /tmp directory. Similarly, suppose that an attacker is able to modify a heavily shared file, such as the password file. Any future user logins would read this file and be marked tainted, and hence taint propagation would not be able to reliably distinguish between attack and legitimate user activity. This adversely affects the *accuracy* of the recovery process.

The limitations of Taser arise primarily from the fact that it is designed purely as a post-intrusion analysis tool. Since, there is no attempt to separate legitimate and potentially malicious activity during normal operation, Taser needs to log all system operations since any activity could be malicious. Moreover, since legitimate and malicious activity can freely interact with each other, taint propagation results in false dependencies.

Our main observation is that if we can isolate applications from each other and provide limited communication paths for accessing trusted system activities, then we need to only

log system operations caused by interactions that originate at the communication paths. This improves the overhead of recovery. Also, restricted and controlled interaction between applications makes it harder for malicious applications to affect legitimate activity, thus limiting attack propagation. This improves the accuracy of recovery.

For our isolation approach to be effective, applications should be able to function in a fairly self-contained manner. If a large number of resources are shared between applications, correct configuration of the sharing paths would be challenging and could potentially reintroduce significant attack propagation. We conducted an initial study of the sharing patterns between applications. Our results show that sharing is limited, making explicit sharing of resources across applications a feasible option. We support various read-write modes of sharing to satisfy the various needs of different applications. At the same time, by controlling the interactions between the applications, we are able to successfully limit the attack footprint in the system, thereby simplifying the analysis and recovery process.

Finally, we demonstrate a method to further reduce the logging overhead by forgetting the taints in the system older than a specified time period. This helps in reducing the logging requirements of the system in the long run and simplifies the analysis process by limiting the amount of tainting.

In the following chapters, we will describe in detail our design choices and the implementation of the Solitude recovery system.

1.3 Contributions

This thesis makes three main contributions. First, it uses an explicit data sharing model between an isolation environment and the base system that limits damage from vulnerable applications and improves accountability of persistent state changes. Secondly, compared to our previous Taser system [7], Solitude can track contamination more accurately and has significantly lower logging needs for system-level intrusion analysis and file-system recovery. Finally, we

introduce two different modes of recovery, coarse-grained and fine-grained, which makes the recovery process more accessible to ordinary users. Our evaluation shows that the Solitude model can be retrofitted in existing systems.

1.4 Thesis Structure

The rest of the thesis describes Solitude in more detail. Chapter 2 discusses related work in this area. Chapter 3 provides an overview of Solitude. Chapter 4 and Chapter 5 present our detailed design of the sharing policies, taint propagation and the recovery process, and Chapter 6 describes their implementation. Chapter 7 provides an evaluation of the system. Finally, Chapter 8 concludes the thesis and highlights directions for future work.

Chapter 2

Related Work

There are several areas related to this work: access control, sandboxing techniques, file systems, intrusion analysis, information flow control, and recovery. Below, I summarize related work in each of these areas.

2.1 Access Control

Mandatory policies enforce explicit sharing and are specified by an administrator based on the principle of least privilege. For example, SELinux [16] provides a powerful mandatory access control model, but it is commonly acknowledged that designing SELinux policies is a complicated process [11]. Solitude provides a simpler, but more coarse-grained isolation model in which policies are primarily needed for file sharing. However, more importantly, access control requires specifying policies correctly, while with our commit sharing policy, errors can be handled until commit is performed. Furthermore, Solitude audits and tracks commits and dependant changes so that if our sharing policies are used incorrectly and lead to attacks, it is still possible to recover the system.

The UMIP model, similar to Solitude, aims to preserve system integrity in the face of network-based attacks [15]. 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 our default policy is read sharing and not read deny, 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 same limitations as 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 with explicit commits to raise the integrity of the files.

CapDesk [37] strives to enforce the principle of least authority by offering interactive policy configuration, such as granting access to files, when running applications. When trusted policy files are not available, a similar approach could be used in Solitude.

2.2 Sandboxing techniques

Sandboxing techniques such as virtual-machine isolation and operating system-level virtualization ensure that the effects of an application are constrained within a restricted environment. Hypervisor-based virtual machines (VM) can provide strong isolation guarantees but they have limited support for sharing. For example, a virtual machine can be used to run multiple versions of the Office word processor, but each machine has its own separate desktop that may lead to a confusing and error-prone user experience.

A second virtualization approach that trades security for efficiency is to use operating system-level virtualization in which a single physical server is partitioned so that it appears

as multiple servers that can be administered independently. This approach has been implemented in several operating systems such as BSD [12], Solaris [23] and Linux [29]. While similar to our isolation environment, OS virtualization is still designed primarily for isolating applications run by untrusting users (e.g., the different customers of a service provider) 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 used to run several server applications such as a web server, mail server, print server, etc. on behalf of the same set of users. With OS virtualization, each of these server applications would require its own list of users and user directories. We envision using isolation environments for different applications run by the same user or by a group of users within the same organization and thus aim to provide better support for sharing and ease of use.

In One-way Isolation [33], 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. Our commit sharing method is motivated by this work. However, while this work proposes using one-way isolation for testing and debugging, we propose to limit sharing by running applications in the long term in this environment. As a result, our system provides support for explicit sharing across the isolation environments, an enhanced capability model for running server applications securely, the ability to commit selectively, and perform recovery even after data is committed.

Microsoft has recently released its Softgrid/SystemGuard technology for virtualizing applications [3]. 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 commit applications or their configurations to the base, and also does not allow for auditing, tracking or recovery of the base system. 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 [9].

2.3 Taint Analysis and Recovery

Information flow control systems employ the principle of least privilege to limit the impact of software vulnerabilities. For example, HiStar [40] allows users to specify precise security policies, but it often requires restructuring applications to meet its security goals. Our focus is on improving isolation by retrofitting existing applications.

Several efforts have focused on analysis and recovery of compromised systems. The Repairable File Service [41] logs file system activity and performs contamination analysis to provide system recovery after an intrusion. Backtracking [13] helps determine the source of attacks by tracking dependencies among kernel objects in reverse time order. Taser [7] determines and reverts the effects of malicious file-system activities by tracking similar dependencies in reverse and forward order. Hsu et al. [10] propose a malware removal framework that allows rolling back untrusted updates.

Many file systems have been developed for creating snapshots for versioning and recovery [27, 20, 19, 5, 17]. Others use check-pointing to provide backups for rollback and recovery [30, 24, 1]. UnionFS [39] provides a merged view of different directories so 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 [31] audits operations and keeps versions for some time for intrusion detection and recovery. RFS [41] 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.

2.4 File systems

File-system workloads have been extensively characterized for improving file system performance through prefetching and caching [36, 25]. We observe that sharing of files between different applications is relatively uncommon and thus suggest using explicit file-sharing mechanisms.

The idea of per-process namespaces first appeared in Plan 9 [22], although it was largely motivated as a method for representing various resources as file systems. Solitude uses namespaces to isolate changes made by each application. Solitude has similarities with the Ventana file system [21] that provides sharing and file-level rollback with a rich file-system level versioning scheme. However, Ventana primarily focuses on using and managing virtual disks in a virtual machine environment, while Solitude aims to maintain integrity and provide recovery facilities after an attack. Many file systems [27, 20] have been developed for creating snapshots for versioning and recovery. These file systems typically implement versioning at the block level which is simpler to implement and provides good performance. However, our goal is to enable limited sharing and selective commits at the file-system level, and hence IFS uses copy-on-write at the file-system level.

Transactional file systems, for example QuickSilver [8] and Vista's TxF (transactional file system) [35], 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 consistency. 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.

In contrast to QuickSilver and TxF, our IFS environment supports existing applications without requiring any changes to these applications. It provides transactional semantics via commit sharing 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 [34].

Chapter 3

An Overview of the Solitude System

Rather than urge users to avoid downloading applications and media, we argue that typical desktop applications that are used on an almost constant basis such as web browsers, instant messengers, word processors, e-mail readers and media players as well as most server applications should *always* be run in isolated environments to limit the impact of attacks.

Since our primary focus is on simplifying analysis and recovery of persistent data, Solitude uses a file-system based isolation environment in which untrusted applications can only compromise their own namespace. The challenge arises when applications, running in different isolation environments, need to share data such as a media file that is downloaded in a browser and used by a media player. Solitude allows explicit sharing of files between an isolation environment and the trusted base system, but it marks any such shared files as tainted. Any use of these tainted files in the base is then logged and tracked, similar to Taser. However the source of tainting is limited to the explicitly shared files, and hence as explained in more detail later, Solitude not only requires much less logging than Taser but is expected to help determine attack-related activities more accurately. Below, we provide an overview of our system followed by a discussion of the usage model and threat model of our system.

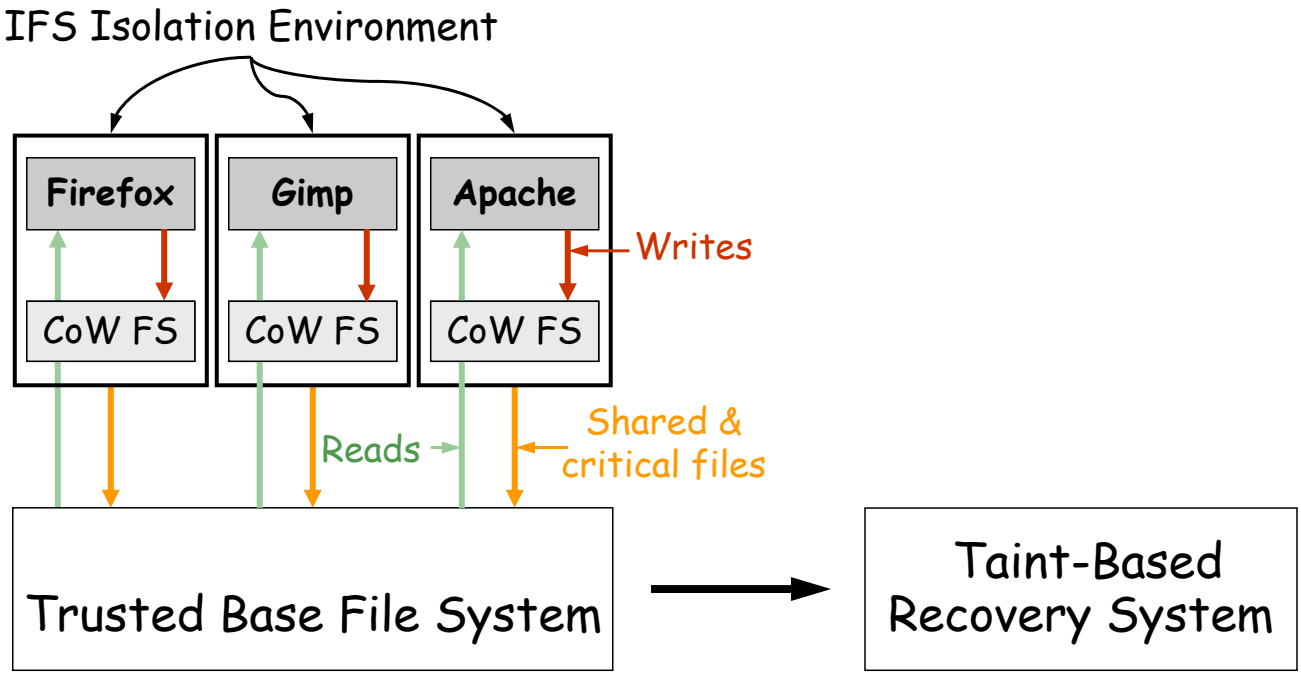

Figure 3.1: The Solitude architecture

3.1 Overview of Solitude

The figure 3.1 shows the architecture of our system called Solitude. Solitude consists of two main components, an environment for isolating applications and a recovery system. In Solitude, we isolate applications using a copy-on-write file system called IFS. For example, Figure 3.1 shows that the Firefox browser, the Gimp photo editor and the Apache web server are isolated from each other. The copy-on-write approach ensures that applications can read from the base file system but modifications are limited to IFS, thus preserving the integrity of the base system. One of the features of this model is that after an attack, an IFS can simply be discarded. While IFS isolates applications, sometimes there is a need for sharing modifications across different IFS applications or with the base system. For example, a photo downloaded with Firefox could be edited with another application such as gimp. This sharing can cause contamination of the base. We track this contamination in the base system to perform taint based recovery. Below we describe each of these components in more detail.

3.1.1 IFS Isolation Environment

Solitude allows running an untrusted application in acopy-on-write isolation environment called IFS. This environment allows reading the base file system and thus, helps in recreating the original file system environment for the application. Consequently, most applications can be run without modification. Copy-on-write policy also confines application updates within the isolation environment. This helps ensure that applications cannot affect each other through the file system.

The benefits of our isolation approach are lost if the isolation environment itself is compromised. To secure the IFS, an application is executed with restricted privileges inside the isolation environment. 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 all powerful root user is disabled within the IFS. Instead, applications are provided

privileges in adherence to the least privilege principle. Solitude also introduces fine-grained file-level privileges for applications to enforce least privilege principle for file system accesses. These mechanisms are required to support server side and setuid applications inside IFS in a secure manner. The details of IFS are the focus of another thesis. [28]

3.1.2 Sharing and Securing Files

Solitude isolates the persistent changes made by applications by using copy-on-write as its basic isolation model and limits application privileges. However, certain file updates may need to be shared with the base file system. The need for sharing files with the base file system arises primarily because of two reasons that are described below:

3.1.2.1 Securing critical files

The basic IFS environment isolates updates made by an application. If malicious activity is detected inside the IFS, then the isolation environment can simply be discarded. This approach provides a simple recovery mechanism but risks losing critical data. For example, critical files like configuration files, data files etc., can be lost. Solitude allows users to *commit* critical files to the base file system. The user needs to specify the set of files that are critical and need to be committed to the base system. Thus, the critical data present in the untrusted isolation environment can be secured by synchronizing with the base file system.

3.1.2.2 Sharing files across applications

Applications running in different isolation environments may need to share certain files. For example, a user may download an image using his web browser and then edit it using an image editing software that is run in a separate isolation environment. Similarly, a movie downloaded using a Bittorrent client will need to be shared with a media player.

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 IFS policy. The sharing policy language is designed for simplicity and intuitiveness. It specifies several possible sharing modes for reading and writing. These sharing modes are described in more detail in the next chapter.

3.1.3 Taint Propagation and Recovery

The key goal of Solitude is to simplify recovery after an intrustion occurs. The basic copy-on-write isolation mechanism facilitates a simple recovery model that we call *Pre-Commit Recovery*. If malicious activity is detected inside the IFS before a user has committed files to the base, the isolation environment can simply be discarded. Since the modifications made by the malicious application are isolated within the IFS, removal of the contaminated IFS recovers the system without affecting the integrity of the rest of the system. This is an easy coarse-grained recovery mechanism.

The sharing modes needed for securing and sharing files introduce the need for a more involved recovery process called *Post-Commit Recovery* or *Taint-Based Recovery.* The sharing policies enable collaboration between applications running in an IFS and base, or between different IFS environments. However, the sharing policies could be poorly designed, potentially leading to contamination of the base file system. Moreover, incorrect user judgment regarding the legitmacy of the application may cause sharing of malicious data or applications with the base file system. For example, a user may commit malware to the base via the downloads directory of the web browser. Solitude addresses this issue by marking the files shared with the base as tainted and subsequently, tracking how other applications access these tainted files in the base. Using a taint propagation method, it logs the tainted actions to a separate system. After an intrusion is detected, Solitude uses a modified version of the Taser system [7] to selectively replay legitimate events in the log to recover the base system. This allows fine-grained recovery of the base system.

3.2 Usage Model

Since most intrusions start with a network connection and then result in control of the system through multiple system activities, we believe that Solitude will be useful for various networked applications. These applications could either be client-side applications run by the same user or server-side applications (such as a web server, mail server, print server) run on a machine on behalf of the same set of users. Below, we describe some examples that represent usage models of our system.

3.2.1 Client side

Users increasingly download untrusted executables and data from the internet. These applications can be run in the IFS isolation environment to mitigate the risk and effect of an attack. Consider a user that wants to download an image from the Internet using his web browser and then edit it using an image editing software. Since both the browser and the image editing program run in separate isolation environments, only the photograph needs to be shared between them. Similarly, a user might want to preserve the chat logs archived by his instant messaging application. Even though the instant messaging application is run in IFS, the user might want to commit his chat logs to the base system. These chat logs might be shared with a text editor application for purposes of reading and searching the logs. A user might want to commit his mail-client configuration files and mail box folder from a mail client to the base for similar purpose. Consider a user that uses a peer-to-peer application to download files. After downloading the file to a standard location, the user can run a viewer application within the same session or mark the standard download location as explicitly shared and use the viewer in the base environment. All other updates by the peer-to-peer application are unshared and could be easily discarded after session termination.

3.2.2 Server side

On the server side, consider an on-line photo album service. The users can upload their photos through the web server and subsequently, only the photos can be shraed with the base file system for puposes of archival and file search. The rest of the changes made by the photo album application are isolated within the isolation environment. As a result, a vulnerable photo album application cannot easily affect the integrity of the rest of the system.

Administrators can also choose to use isolation environments for certain low-privilege users. For example, isolation environments can be used to ensure that anonymous FTP users cannot affect the base system, and to isolate directories that are shared across users such as the Unix `/tmp` directory that has been the source of several exploits. Similarly, for a database server, only the data and the configuration files can be shared with the base system for the purpose of backups. In the case of a source code versioning system server, only the repository can be shared with the base system for similar reasons.

3.3 Threat Model

A malicious application can damage the integrity, confidentiality and availability of a system. Solitude strives to preserve the integrity of the system in the presence of untrusted networked applications. The integrity of a system can be compromised by unauthorized modification of files and misuse of capabilities. For example, an application can delete important binaries or load a rootkit leaving the system in a corrupted state. Solitude attempts to contain the damage caused by such operations using two methods: 1) limiting access to the trusted file system by isolating the file modifications made by an untrusted application and providing a fine-grained protocol for sharing data with the trusted file system, and 2) limiting system-wide privileges available to an untrusted application in keeping with the least privilege principle. These methods limit attack propagation in the file system by denying access to the files that should not be modified by the malicious application, and restricting an application from causing

harm by misusing its privileges, which are far fewer than the all powerful root user in Unix systems. An untrusted application may also communicate with other processes via means other than the filesystem, such as IPC or covert channels. Solitude tracks accesses to the trusted file system and IPC communication via a taint tracking mechanism that can be used for post-intrusion analysis. However, Solitude does not track covert channels, for example, communication via the external network.

In addition, a malicious application may trick a user into granting additional privileges or may misuse its privileges. Solitude aims to counter this threat by providing a system-wide recovery mechanism, allowing users to recover their system. A malicious program can also breach confidential information on the system by leaking it to the outside world. Such techniques, e.g., employed by spyware programs, attack user privacy. Although not a fundamental objective of Solitude, these risks can be reduced by denying read access to sensitive information on the disk. A malicious application can also reduce system availability. In this case, Solitude does not provide any additional protection, but is no worse than the original system.

The Solitude architecture is shown in Figure 3.1 has three main components: the IFS isolation environment, the sharing policy and the recovery system. The IFS environment is described in a previous thesis [28]. The next two chapters describe the Solitude sharing policies and the recovery system.

Chapter 4

Sharing Policies

While Solitude, by default, isolates the persistent changes made by applications to ease post-intrusion recovery, it also allows refining the isolation model with policies that enable explicit sharing of specific files and directories between the isolation environment and the base system. Sharing between applications may be required for application functionality, for example, many Unix terminal programs write to the psuedo terminal device, or desired user behavior, for e.g., a user may want to download an image using the web browser and then edit it using an image editing software. Apart from usability reasons, explicit sharing helps in improving the accuracy of the recovery process. It provides explicit contamination entry points from the untrusted isolation environments, which can be tracked with less overhead. It also limits the interaction between applications to a few known paths, restricting the propagation of taints in the system. This explicit file sharing model is based on the idea that sharing of files across applications is rare, which makes configuring explicit sharing a relatively easy task. We evaluate this hypothesis in Chapter 7. If data sharing across applications was common, then these several applications would need to be run in the same isolation environment, reducing the benefits of isolation for the recovery process.

Solitude provides support for various read-write sharing modes to suit the requirements of various applications. Below, we describe the various modes and the policy specification

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 4.1: IFS sharing modes

associated with each of them.

4.1 Sharing Policies

The sharing policy language is designed for simplicity and intuitiveness. In particular, there is no support for sharing between two isolation environments directly, which provides finer-grained sharing but would complicate the language design and specification. Any such sharing must be performed via the base system. The sharing policy specifies three sharing modes for reading and three modes for writing (although a few combinations are not meaningful and mentioned below). These modes are shown in Table 4.1. All the sharing modes apply to a file or directory and are subject to the access control restrictions of the base. Below we describe the different modes and show how they can be used to support the usage models described in Section 3.2.

Rshare specifies the default read policy in which the isolated applications transparently read from the base system until they make an update to the object. At this point, the data is propagated to the isolation environment and read from there (one way copy-on-write). The read-share policy protects the base system from any persistent changes made by a compromised application running in the isolation environment and also allows easily rolling back these changes.

Rsnapshot specifies copy-on-write in both directions, i.e. the isolation environment makes a snapshot of the base immediately upon start-up, rendering both the isolation environment and the base oblivious to any subsequent changes made in the other. A read snapshot ensures that software updates in the base do not affect the isolation environment. However, it also disables critical updates, such as security updates for a web server vulnerability, from becoming available in the isolation environment. The isolation environment must be shutdown and reinitialized to access these updates. At shutdown, updates in the isolation environment can be discarded or commit shared as described below.

Rdeny hides specified base files or directories from the isolation environment and therefore guards against information leaks and addresses information privacy concerns. The read-deny policy is likely to be used in systems where the isolation environments are used to run risky applications susceptible to subversion such as certain network servers. A read deny overrides any write sharing policy.

Wisolate specifies the default write-isolate policy and confines all writes permanently to the isolation environment.

Wcommit specifies that changes are confined to the isolation environment but may be eventually committed to the base system. Commit sharing delays synchronizing updates until an explicit commit is issued. During commit, updates to files and directories in the isolation environment specified via Wcommit are applied to the base system atomically and are then discarded from the isolation environment. Each commit is associated with a Commit ID that is used during recovery, as described later in Section 5. This sharing mode provides a time period (until commit) during which updates can be discarded and is suitable when sharing data loosely or occasionally with the base.

Commits are initiated by starting a commit process on a particular isolation environment. The commit process is described in more detail in Section 6.2. Commits can only be invoked

by the isolation environment owner in the base system, ensuring that malicious applications running in the isolation environment cannot commit persistent data. An alternative that is currently not supported in Solitude, is to allow commit sharing from the isolation environment after explicit user authentication similar to the use of the `sudo` program in Unix systems.

Various usage models described in Section 3.2 can benefit from commit sharing. For example, the session logs of instant messaging applications could be commit shared with the base system periodically. Similar, for the photo application described earlier, the photo album directory could be commit shared with the base system for archival or for viewing by another application such as a file viewer.

Solitude allows concurrent updates to occur in the base and the IFS and hence persistent state in the two file systems can diverge over time. An object modified in IFS can be committed successfully when the corresponding object in the base file system has not been modified after it was first copied into IFS. This criteria ensures that the commit operation is atomic and equivalent to an object being accessed and modified at commit time [33].

When the commit criteria is not met, updates can diverge and the commit is said to be conflicting. Such conflicts can be automatically resolved for directories since their semantics are known. We expect that sharing and hence conflicts across applications described in our usage models will be rare. If conflicts are common, then either the conflicting applications should be run within the same IFS, or the write sharing method described below is more appropriate.

Although Solitude is designed to run untrusted applications indefinitely within the IFS environment, it also supports committing an entire application that the user deems trustworthy after evaluation.

Wshare indicates immediate write-sharing of changes from the IFS to the base file system and is useful for files known to be of low risk. Write sharing is also used for special files such as device files that typically do not satisfy file semantics (i.e., reads provide the same data as a previous write to the file) and also do not provide persistent data. These files must be explicitly specified as shared to allow updates. For example, many Unix programs write to the

```
Application /usr/bin/firefox
Wcommit /home/shvetank/downloads/
Wcommit /home/shvetank/.mozilla/
Rdeny /home/shvetank/private_info.txt
```

Figure 4.1: An application policy file with a write-commit and a read-deny policy

`/dev/null` device file and terminal programs write to the `/dev/ptmx` pseudo terminal device. A write-shared file must be in read-shared mode. While low-risk files may be shared with the base system for ease of use, commit-based sharing is more appropriate for most files because it delays synchronization with the base and thus allows handling errors in the sharing policy until commit is performed.

Figure 4.1 shows a sample policy snippet for sharing files and directories. It specifies that any changes made by the Firefox application are permanently confined to its IFS (the default policy), with the exception of the `downloads` and the application profile directory which can be explicitly committed by the user to the base file system. Firefox is allowed to read all the files in the base except for the file `private_info.txt`, presumably because it contains sensitive information.

Policy rules are applied recursively to a directory's files and sub-directories. A more specific rule, such as one that applies to a file in a sub-directory, overrides a more "general" rule, i.e. one that applies to a higher-level directory and its sub-directories. This allows users to specify a rule covering a directory and its contents, with exceptions for some sub-directories and individual files.

The intended authors of the policy files containing the IFS capability and sharing specification are companies that provide the OS distribution or user communities. The policy language is simple and we expect that system administrators will be able to easily modify the policy files to suit their environment. For certain applications, a default read-only policy should be satisfactory. In such a scenario, the damage is restricted to information leakage. The user can

mark sensitive files with Rdeny, thereby avoiding their leakage.

Innocuous applications, when hindered, can inform and request the user to grant certain file access permissions and capabilities to function correctly. At this point, the user must judiciously grant these privileges, based on the files that can be modified as well as the capabilities that can misused. A mistake in assessment will require system-wide recovery as described in the next chapter.

Chapter 5

Taint Propagation and Recovery

The sharing policies described above enable collaboration between applications running in IFS and base, or between different IFS contexts via the base. Without support for such sharing, an increasing number of 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 our Taser system [7] for analysis and fine-grained recovery of the base system. Solitude also provides an untainting procedure where the user has the option to cease tracking certain old activities in the system. Below, we describe the Solitude taint propagation method, the logging system, our offline recovery method and the untainting mechanism.

5.1 Taint Propagation

Taint propagation tracks the dependencies between system-level objects like processes, files and sockets to determine the footprint of the attack. Since the untrusted applications are sandboxed in the isolation environment and can interact with the base file system through explicit

Operation	Propagation Rules
File and directory read operations, execute file	Tainted file → Taint process
File and directory modification operations	Tainted process → Taint file
Create child process	Tainted process→ Taint child process

Table 5.1: Taint propagation rules

sharing and commits, the sources of taints are well known. These sources of taints are then tracked to determine their affect on the activities in the base file system. An isolation environment might also be tainted by reading a tainted file from the base. Such dependencies help with the analsysis of cross-IFS dependencies during the recovery process.

Each IFS environment in Solitude conceptually has an associated *IFS monitor process* running in the base that performs all file operations on behalf of IFS processes in that environment. This process accesses a file in the base or the IFS environment based on the file sharing mode, and synchronizes files from the IFS to the base during a commit. Solitude marks this monitor process as tainted since it operates on behalf of untrusted IFS processes. Then the taint propagation algorithm tracks modifications to the base file system that depend on this process. The taint propagation rules are simple and shown in Table 5.1. These rules operate on kernel objects such as processes, files and directories that can either be untainted or tainted. The rules 1) taint a process when a process reads or executes a tainted file, 2) taint a file when a tainted process modifies the file, and 3) taint children processes of a tainted process. These rules can taint any processes running in the base environment or any files in the base file system, but they ignore IFS processes or files, because our goal is to recover the base system after an attack.

The Solitude tainting algorithm, operating at the granularity of kernel objects, is coarse grained compared to instruction-level data-flow analysis [32, 18, 4]. We choose to use a coarse tainting method for two reasons. First, data-flow techniques are vulnerable to attacks caused by implicit or control-based information flows within a program [26]. Our algorithm taints at the process level and thus does not suffer from this problem. Second, data-flow techniques gener-

ally have a large overhead and are thus not used during normal operation, while our algorithm has low overhead and can be used in real time. The main drawback of coarse-grained tainting is that it can introduce a large number of false sharing dependencies. However, we expect that file sharing will be common mainly within the isolation environments (which are ignored by the tainting algorithm), and our explicit sharing policies will limit false dependencies in the trusted base environment.

5.2 Logging System

The taint propagation algorithm uses a single bit to taint base processes or files according to the rules shown in Table 5.1. However, this approach only allows determining the entire set of processes or files that were tainted by any isolation environment. Solitude allows finer-grained recovery at the commit level by logging two kinds of operations. First, it logs all operations in which the source object in any propagation rule shown in Table 5.1 is tainted. For example, in the first rule, it logs a file read operation and the process reading the file when the file is tainted in a *tainted operation log*. Second, Solitude uses a separate privileged *commit process* to generate a *commit log* that stores an IFS ID, a commit ID (that is incremented per IFS commit), and the set of committed files on each commit. The commit process taints itself, and hence its operations are logged in the tainted operation log. This log and the commit log are sent to a backend system shown in Figure 3.1, which allows commit-level recovery as described below. The logs are stored and analyzed on a separate system so that they cannot be easily destroyed.

The previous Taser system [7] assumed that all file-system operations were untrusted and hence logged all these and related operations. In contrast, the Solitude threat model assumes that untrusted applications are executed in IFS environments and hence only operations in IFS environments and tainted operations in the base are untrusted. After an attack, copy-on-write based IFS environments can simply be discarded. As a result, Solitude does not need to log any

operations in IFS and only logs operations on tainted base objects for fine-grained recovery. This approach can reduce logging significantly compared to Taser.

5.3 Recovery

Recovery in Solitude can be performed in two ways. If the user realizes that the application is malicious before commiting the files to the base file system, then the entire isolation environment can be discarded without affecting the integrity of the rest of the system. However, in the event that the user shares files with the base file system, then the recovery can be performed at the backend system at a per-IFS, per-commit granularity. If a malicious file is committed to the base, the user specifies the IFS from which the file was committed and a commit ID (described earlier) as a starting point for recovery. Our analysis tools, previously developed in Taser, help determine the IFS and the rollback commit ID. For example, each commit stores a commit time and the set of committed files. If the file is known to be committed at some approximate time, then the closest previous commit ID is chosen.

The recovery process can generate the tainting dependency graph for any given commit. It starts by tainting the corresponding commit process and re-running the same tainting algorithm described earlier. This is possible because all tainted operations are logged to the backend. Our use of the IFS monitor process (see Section 5.1) ensures that if the tainted operations of an IFS are read by another IFS, then the recovery process will track the operations of the second IFS also. Furthermore, since an IFS environment will often read files after they are committed, all subsequent write-share and commit operations by the same IFS will also become tainted. As a result, recovery for the explicit shared operations in Solitude would be performed at the granularity of IFS environments. This approach represents a trade-off between the granularity and scalability of recovery. Previously, Taser was used to perform rollback at a finer process-level granularity, but that required logging all file-related operations in the system which limited the scalability of the system, and it also required more detailed analysis to determine the starting

point for the tainting algorithm. Since IFS environments are typically used to run closely-related applications, we believe that IFS-level recovery is an acceptable trade-off.

Once the set of files that depends on a particular commit ID has been generated, these files should be manually inspected to ensure the correctness of recovery. Then the files can be rolled back to an untainted state by using the unmodified selective redo algorithm in Taser [7].[1] The rollback uses a snapshot of a file taken when the file was first tainted by any commit and then replays all subsequent modifications (which were logged because the file was tainted) until the file reaches a state just before the rollback commit ID. The recovery system uses a simpler, more efficient undo algorithm for reverting tainted directory operations, and then generates a self-contained script that can be used to recover the base file system.

5.4 Untainting

Even though the logging requirements of the Solitude system are fairly acceptable, the user might want to get rid of the taints that had been generated as a result of actions in the distant past after gaining substantial confidence regarding the legitimacy of those operations over a period of time. Solitude offers a way to clean the taints that were generated as a result of the actions before a particular time. This further helps to reduce the subsequent storage requirements of the system and simplifies the analysis process in the future as there are fewer taints in the system. While the user loses the ability to do fine grained-recovery for events in the window for which the taints have been cleaned, we believe that it is unlikely that the user will remove the footprint of activities that have happened several months ago after trusting them for a long period of time. Moreover, the user can adjust the time period for cleaning the taints to suit his comfort level.

Our aim is to untaint all kernel objects such as files and processes that were tainted by activities that occurred before some time T. The system containts three different types of tainted

[1] Taser also provides tools that help a user analyze and modify the set of files that need to be reverted.

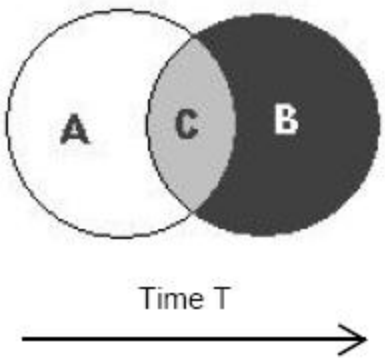

Figure 5.1: Origin of tainted objects

objects as depicted in 5.1. Some objects are tainted only by activities that occured before time T (represented by set {A}), some are tainted only by activities that occured after time T (represented by set {B}) and others by both (represented by set {C}). The untainting process has to clean the objects tainted only by activities that occured before time T {A}. Taint propagation using commits after the specified time T as a source, generates a graph of tainted objects {B+C} that need to remain tainted. The complement of this set provides the objects that need to be untainted. It is important to note that propagating taints using activities before the specified time T as a source would yield {A+C} and would not be sufficient to determine {A}.

When the user invokes the untainting operation with time parameter T, the taint dependency algorithm is run on the backend system using the commit operations that occured after time T as the sources of the taints. This generates a list of files and processes that still need to remain tainted in the frontend system as their origin of taint occured after the user specified time. This set of files and processes is added to a *new tainted log*, which is then sent to the frontend. The frontend untaints files and process that are not list in this log. The untaint process traverses the system recursively, untainting any tainted process or file not present in the new tainted log with

the help of the kernel module.

The design of the untainting process presents a few challenges. The taint information can diverge between the frontend and the backend due to propagation of taints during the backend analysis which are not accounted for during the cleaning of taints. As a result, the frontend needs to be stalled to avoid inconsistency of the taint information during the backend analysis. Although the suspend is for a short period of time, it can be optimized by differential analysis of the tainted log. In such a scheme, the operations tainted during the backend analysis can be successively analyzed, progressively reducing the analysis time and the window of spurious taint propagation. This refined approach is similar in principle to the consistency requirements of a garbage collector.

After the succesful completion of the untaint process, the user is left with a system with objects that were tainted as a consequence of operations that occured after the user specified time. Thus, the system *forgets* the taints caused by operations that have won user confidence over a long period of time.

Chapter 6

Implementation

Solitude uses several different components to provide a post-intrusion analysis and recovery tool. The architecture of Solitude is shown in Figure 3.1. In this chapter, we will describe the details and challenges of implementing those components.

6.1 Sharing Policies and Commit

The Solitude sharing policies are implemented within the IFS monitor process. The default policy is copy-on-write, i.e. read share and write isolate, with a few exceptions such as write sharing for some `/dev` devices. The IFS monitor process redirects the request to the base or the IFS in accordance with the specified policy. For read share and write share, the file operation is redirected to the base. For write isolate, the file is modified in the IFS. Currently, we are in the process of implementing the `Rsnapshot` mode in Solitude. This mode requires integration with a standard file-system versioning mechanism [38, 20, 2], and is especially useful if the user plans to run multiple versions of an application in different IFS environments. The IFS monitor process disallows the file operation for read and write deny modes.

While our sharing policy language is simple by design, we find that determining the correct policy for a large application can take time. For example, determining the full set of files used by an application and determining the correct policy for each sharing scenario can be

challenging even for individuals with intimate knowledge of the internals of an application. Even when the correct policy has been supplied externally such as by a community of users, individual system administrators may still wish to customize policy files to their specific system configurations and specific needs.

Accordingly, we have created a simple tool that profiles an application's file I/O behavior and produces a list of files that are created, deleted, read or written by the application. This output provides a good starting point for deciding the set of files that need to be shared. In Section 7.2, we describe our experiences with writing sharing policies for several popular applications.

File commit is implemented as a separate commit process, although it is logically a part of the IFS monitor process since it also enforces the sharing policies. The commit process uses redo logging to ensure atomicity and resistance to failures.

The commit process first explicitly taints itself (similar to the IFS monitor process which is always tainted), creates a commit log of operations to be performed (e.g., creating or synchronizing a file in the base), sends the commit log to the backend with a new system call called `solitude_commit`, and then performs the operations. Tainting the commit process taints all its file operations, which are then automatically appended to the tainted operation log and sent to the backend system. The solitude_commit system call helps in establishing the association between the commit id, IFS id and the commit operations performed at the backend. All IFS updates to write-shared files between two explicit commits are associated with the subsequent commit, which provides a logical grouping of the various modifications to the base file system to discrete commit points for the purposes of recovery. After the operations are performed, the commit log is removed. The existence of a previously created commit log indicates an incomplete commit operation. At this moment, the commit process will redo the incomplete operations in the commit log to ensure atomicity.

During the commit process, conflicts may arise if the file has been modified both in the IFS as well as the base. Such a conflict needs to be dealt with manually for files and can be resolved

for directories as the semantics of the directory are known. Currently, our system handles three types of conflicts. A remove conflict occurs when a file is modified in the base but removed from IFS or vice-versa. A suffix is added to the IFS file to indicate the appropriate case. A name conflict occurs when a file with the same name is created independently in the base and the IFS, which is resolved by adding a suffix to the IFS file. Finally, if the path of the file does not exist in the base, the user is prompted for permission to recreate the missing directories.

6.2 Taint Propagation and Recovery

We have implemented taint propagation and the logging system (for recovery) in the kernel of the target system with Forensix.[6]

The taint propagation algorithm starts when a user commits files to the base file system or when files are write shared. Forensix provides a system call interception layer through a kernel module. At each system call, the kernel level objects involved in the system call are analyzed for the purpose of taint propagation and logging. The logging system initially sends a snapshot of the base file system namespace to the backend. After that, it logs all base namespace modifications to the backend so that the backend recovery process can recreate a consistent view of the base file system. The taint propagation algorithm maintains the taint status of base processes and files by simply following the rules shown in Table 5.1. In case any of the taint propagation rule applies, the affected process or file gets tainted. The taint status is stored in an unused bit in the *task_struct* data structure for processes and an unused bit in the *inode* data structure for files. The modified inode data structure is marked dirty and subsequently the taint status gets saved on the disk inode data structure. Thus, the file taints are maintained persistently across reboot.

When a file is first tainted, a snapshot of its pre-tainted contents are sent to the backend so that, if needed, file contents can be restored to an untainted state. This snapshot is implemented using the *sendfile* function in the kernel for efficient transfer of the file. The system call that

resulted in the tainting of the file, blocks until the original contents of the file have been successfully stored at the backend. The Forensix module had been originally designed to provide buffering for system call logging to facilitate asynchronous transfer to the backend. However, in this case, pre-tainted contents had to be synchronously transferred to the backend system to enable recovery. After that, all content updates to the tainted file are logged to the backend. The backend can then rollback a file to its pre-tainted state and apply updates until the time associated with a specific rollback commit ID using the selective redo algorithm of Taser.

The recovery process in Taser has been modified to account for the commit operation. As the sources of taint are already known to be commit points and write sharing instances, the Tracing phase that was used to determine the sources of the taint is not required. The Propagation phase takes the commit id and IFS id pair as input and generates a list of affected files and processes as a result of those commit operations. The recovery scripts generate a list of operations that need to be performed on the frontend in order to restore the system to the clean state.

6.3 Untainting

The untainting algorithm has been implemented with the help of the dependency analsyis tool used in recovery. The user specifies the oldest time for which he wishes to retain taints. The backend system aggregates all the commit operations that happened after the specified time, and using dependency analsysis, generates the list of processes and files that still need to remain tainted. A file containing the new list of tainted files and processes is sent to the frontend for cleaning the residual taints. The new tainted log is structured in tuples of device, inode number, generation number for files and process id, process start time for processes for unique identification of the system level objects.

At the frontend, the processes are temporarily stalled to avoid spurious propagation of taints. The files and processes on the system are traversed and checked for taint status. In

case an object is found to be tainted, a lookup is performed in the new tainted log. If the lookup fails, the object is untainted. Consequently, taint tracking is no longer done for the newly untainted objects, thereby, reducing the analysis and logging overhead. However, the user loses the ability to track any further changes caused by activities that occured before time T.

The backend data is retained for consistency of the recovery process, which requires the namespace structure of the base file system and the untainted content of tainted files. The namespace operations that occured before time T are required to recreate the namespace structure of the base file system at time T. When a file is tainted for the first time, the untainted contents of the file are sent to the backend. However, certain files are tainted by activities that occured before time T as well as after time T (represented by set {C} in Figure 5.1). In such cases, to recreate the state of the file at the latter taint event, the original snapshot of the untainted data and the subsequent modifications are preserved. Since the user may choose any arbitrary time T, creating snapshots apriori is not possible. Solitude currently retains the entire backend database. However, an optimization that recreates namespace at time T and also generates the required state of tainted files after time T could help in further reducing the database size.

Chapter 7

Evaluation

Our evaluation of Solitude focuses on the effort involved in configuring sharing policies and how well the system limits the spread of contamination from the untrusted IFS environments. We start by evaluating sharing patterns among existing applications to gauge the complexity of isolating various classes of applications. Second, we describe the effort involved in configuring the sharing policies and capabilities for applications run within IFS environments to determine the usability of the system. Third, we measure the contamination that occurs due to explicitly shared files and the storage requirements of tracking and logging the actions of contaminated processes. Finally, we evaluate the performance overhead of Solitude.

7.1 Measurement of Sharing

Our first study is designed to evaluate file- and IPC-based sharing patterns among existing applications run in Linux environments. Our hypothesis is that *both* file and IPC-based communication is limited among applications that are targeted for IFS environments (e.g., network applications) and hence configuring explicit sharing for these applications is a viable option. We test this hypothesis for both client and server environments by using Forensix [6], a system that logs all system calls and provides MySQL-based tools that help with analysis of past system behavior. The client system is one of the authors' machines and it runs Ubuntu Linux

	Client	**Server**
Experiment dates	Jul 26-Aug 23	Jul 25-Aug 13
Experiment time	29 days	20 days
Files written	30353	151856
Write shared files	173	88154

Table 7.1: File sharing statistics

2.6.15. The server also runs the same OS and provides web server (with a php/mysql backend), imap, webmail, postfix, NFS, VNC, dhcp, tftp and sshd services to a cluster of 128 machines with roughly 10-15 active users. We present results for any potential sharing that occurs among applications based on file and IPC-based communication.

Table 7.1 shows the results for write-write sharing of files by more than one program in both a client and a server environment. This table indicates that sharing is relatively uncommon compared to the total number of file accesses on the client side. On the server side, the majority of the sharing occurs due to a set of related mail programs shown in the first row of Table 7.2. This table shows the shared files, the programs that accessed them and the type of sharing needed. Each row shows the number of shared files in the client and the server study. The Unshared files can be accessed in an IFS environment with no additional policies. The Write-shared files require a corresponding write-shared policy. Some files, such as log files, may either be in Unshared mode when accessed from IFS, or could be directly accessed by programs in the base environment. Also, certain programs, typically used for system administration, will mainly be run in the base.

We also performed an IPC study using the same experimental data to determine whether an explicit IPC specification is reasonable across IFS and base environments. Common IPC mechanisms in Unix systems include FIFO, Unix domain sockets, shared memory and local INET (TCP, UDP) sockets. The first three mechanisms have unnamed and named counterparts. The unnamed mechanisms work for related programs in a process hierarchy and are allowed

Shared files (client+server)	Programs	Type of sharing
Files in /var/spool/postfix (0+85927)	Postfix, smtp, local, cleanup	Unshared
Files in home directories (30+1639)	Compilation, etc.	Unshared
Files in /tmp (122+553)	Cron, tex, other programs	Unshared
Files in /var/mail, /var/mail/$USER.lock (0+10)	Procmail, imap, mail clients	Write shared
Device files, /dev/null, /dev/ptmx, /dev/ttyXX, /dev/pts/0 (5+3)	Numerous programs	Write Shared
Log files, /var/log/wtmp, /var/run/utmp, /var/log/lastlog, .xsession-errors (5+6)	gnome-pty-helper, xterm, sshd, sessreg	Unshared, In base
Libraries, /usr/local/lib/libfuse.a, /usr/local/lib/libulockmgr.a (2+0)	install, ranlib	In base
Files in /var/lib/belocs, /var/lib/texmf/ls-R, /var/lib/dpkg/lock, /var/cache/debconf/ (9+16)	locale-gen, synaptic, apt-get, gnome-session, gnome-panel	In base

Table 7.2: File sharing on a client and a server system

within an IFS but disallowed across IFS and base in Solitude. For named communication, our study showed that there was no shared memory communication, and a very small set of applications used FIFO and Unix domain sockets during the course of the experiment.

Based on this initial result, we disabled these IPC mechanisms across IFS and base and different IFS environments to avoid implicit sharing, and re-ran the few applications using the IPC mechanisms. Surprisingly, we found that they still worked correctly. For example, Gnome applications use Unix sockets to communicate with the Gnome application-configuration registry. Our experiments show that disabling Unix domain sockets has no effect on these applications running in an IFS because they start another configuration daemon in the IFS. The configuration data is stored in a file hierarchy, and hence we were able to use commit sharing to synchronize the application configuration data in the IFS with the base (if desired by the user).

For local INET sockets, we saw no UDP based communication during the entire experiment. We saw local TCP connections to three services, the printing server, X server and the ssh server. The printing server should be run in an IFS, but the other servers provide basic services (desktop environment and remote access) and would need to be run in the base. Services like ssh server can be run inside an IFS in principle, but as more and more applications spawned from the ssh terminal would be run within the same IFS, the benefits of isolating applications would be minimized. The recovery process would be too coarse-grained and not as beneficial. Thus, we propose to run ssh service inside the base. Moreover, in all these cases, these services would need to be shared with many IFS environments. Even so, these results are promising because the total number of such services is small, and hence we plan to incorporate explicit IPC specification in Solitude.

7.2 Sharing Policies

In this section, we discuss the usability of our system by describing examples of sharing and capability policies for various classes of client and server-side applications suited for IFS environments.

7.2.1 Client Applications

We wrote and tested policies for a web browser (`firefox`), instant messenger (`gaim`), mail client (`thunderbird`), a file-sharing client (`limewire`) and an audio player (`xmms`). These applications are representative of a large class of networked applications used for downloading data and executables on the client side. We used a combination of the copy-on-write IFS environment and our `io_profile` tool to determine the files accessed by these applications.

These applications do not require any write-shared files except some device files. In all cases, the default specification for all these applications allows committing the application profile directory in the user's home directory to ensure that the application profile is safe even if the application is subverted and its IFS environment needs to be discarded. Conflicts during commit are unlikely, since we do not expect that an application's profile directory will be modified by other base applications. The user may also specify that the downloads directory of some of these applications (e.g., `~/Share` for `limewire`) can be committed. The mail client policy is similar when using the POP or IMAP protocol. However, when mail is delivered locally, the mail INBOX folder must be write shared to enable sharing with the mail transfer agent (e.g., `postfix`) IFS and all mail folders must be write shared when using a mail delivery agent (e.g, `procmail`). Figure 4.1 showed a sample policy for `firefox`. The other applications have similar policy files.

While the default policy for all these applications requires three or fewer lines, the user may choose to use a more fine-grained specification. For example, the user may choose to commit only the bookmarks file in `firefox` and specific extensions that are known to not be

```
Application /usr/bin/firefox
Wcommit /home/shvetank/.mozilla
Wcommit /home/shvetank/downloads
```

Figure 7.1: Policy for Firefox

```
Application /usr/bin/gimp
Wcommit /home/shvetank/images
```

Figure 7.2: Policy for Gimp

malicious [14]. All policies could be further refined by users to protect the privacy of specific files and directories using the read-deny policy. Finally, if these applications were IFS aware, these policies could be set up based on user input when the user runs the application for the first time.

Below, we present relevant examples of policy files for various client applications. Consider the photo editing scenario. The user downloads an image using his web browser (e.g., `firefox`) and then edits the image using a photo editing software (e.g. `gimp`). The policy files for these applications are shown in Figures 7.1 and 7.2 . The user will require to commit the image to the base for sharing the file with the photo editing software. (e.g. `.downloads` directory) The user may also choose to secure his profile settings of each of the applications. (e.g. `.mozilla` directory). Once committed, the photo editing software can read the image from the base and after having modified the image can secure the image to the base by committing the `images` folder to the base.

Gaim is an instant messaging software which has access to configuration files as well as the personal chat logs of the user. The policy file for Gaim is shown in Figure 7.3.The user may choose to commit his profile directory to the base in order to secure his configuration setttings. However, the user may write- or commit-share only the `logs` directory of the `gaim` application instead of the entire application profile directory.

Limewire is a peer-to-peer client for file sharing. The policy file for Limewire is shown

```
Application /usr/bin/gaim
Wcommit /home/shvetank/.gaim
```

Figure 7.3: Policy for Gaim

```
Application /usr/bin/limewire
Wcommit /home/shvetank/.limewire
Wcommit /home/shvetank/Share
```

Figure 7.4: Policy for Limewire

in Figure 7.4.The user may choose to commit his profile directory to the base for saving his configuration settings of the application. Apart from configuration, the user may want to share the downloaded content with other applications. For example, the user may download an mp3 file and commit it to the base in his Share folder. This mp3 file can then be played by a media player like xmms (Figure 7.5) from this folder once it has been committed to the base. In addition, the user may want to commit the configuration files from xmms to the base as well.

Gconf is a system used by the gnome desktop environment for storing configuration settings for the desktop and applications. Changes to this system are controlled by Gconfd, a daemon. Gconfd watches out for changes to the configuration files, and when they are changed, it applies the new settings to applications using it. For example, the email client Mozilla-Thunderbird uses Gconf system to maintain configuration settings. The user can commit his profile settings and the additional Gconfd state to the base. The policy file for Thunderbird is shown in Figure 7.6.

```
Application /usr/bin/xmms
Wcommit /home/shvetank/.xmms
```

Figure 7.5: Policy for Xmms

```
Application /usr/bin/mozilla-thunderbird
Wcommit /home/shvetank/.gconfd/saved_state
Wcommit /home/shvetank/.mozilla
Wcommit /home/shvetank/.mozilla-thunderbird
```

Figure 7.6: Policy for Thunderbird

7.3 Taint Propagation and Logging

In this section, we measure the contamination that can occur due to explicitly shared or committed files and the storage requirements of tracking and logging the actions of processes contaminated by the taint propagation rules shown in Table 5.1. This data is hard to collect because it requires attacks on real user systems. A honeypot can be used to detect attack activity, but it may cause little contamination because it has no real users. Instead, we use the data collected in the user study described in Section 7.1 to provide an estimate of the level of contamination that may occur and the logging requirements during normal system activity.

For this experiment, we tainted all invocations of an application and measured the number of tainted files in the system and the amount of logging that would have occurred over the course of 10-14 days. This experiment was performed entirely in the backend system, and simulates the case of an application being tested in IFS and then being committed to the base. In general, we expect users to run network applications in IFS while mainly committing data files, so this is a worst case scenario for tainting and logging during normal user activity.

Tables 7.3 and 7.4 shows the tainting and logging results. Table 7.3 shows the number of days over which taint propagation was performed, the total amount of data and the number of system call events logged by Forensix (in million), the number of file namespace related system calls, and the total number of files that either existed on the system or were created during the 10 or 14 day tainting period. Table 7.4 shows several different client or server applications that we tainted (one at a time), the number of existing tainted files at the end of tainting period, and the number of tainted events that would have been logged by Solitude

	Client	Server
Experiment time	10 days	14 days
Total log size	30.8GB	26.1GB
Total # of events	454.4 M	151.8 M
Namespace events	0.2 M (.04%)	3.5 M (2.3%)
Total files	276,118	3,315,437

Table 7.3: Forensix logging statistics

	Tainted files	Logged Events
Client		
acroread	2	0.3 million (.07%)
firefox	107	0.4 million (.09%)
amsn	134	0.4 million (.09%)
thunderbird	174	1.7 million (.37%)
nautilus	198	1.4 million (.31%)
gedit	222	10.0 million (2.2%)
Server		
svnserve	65	3.6 million (2.4%)
apache2	5	3.7 million (2.4%)
dovecot/imap	35	4.7 million (3.1%)
mysqld	15	3.7 million (2.4%)
pine	20	4.3 million (2.8%)
procmail	38	4.8 million (3.2%)

Table 7.4: Taint propagation and logging

(also shown as a percentage of the total number of events logged by Forensix). These tainted events include the namespace events (tainted or otherwise) shown in Table 7.3 that are always logged in Solitude. Table 7.4 shows that the number of tainted files is relatively small for all applications, including applications like firefox that was run over 100 times during the tainting period, and hence post-intrusion base file-system recovery should be a feasible option.

This table also shows that the total amount of logging in Solitude should be much smaller than our original Taser/Forensix system. This is primarily because Solitude only logs tainted system call events as opposed to *all* system call events in the original Taser/Forensix system.

7.4 Performance Overhead

We measured the overhead introduced by Solitude by running a set of benchmarks representing different client or server workloads. We ran two client workloads: 1) untar of a Linux kernel source tarball, representing a filesystem-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: 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 7.7 shows the performance overhead of a base process running on our system for

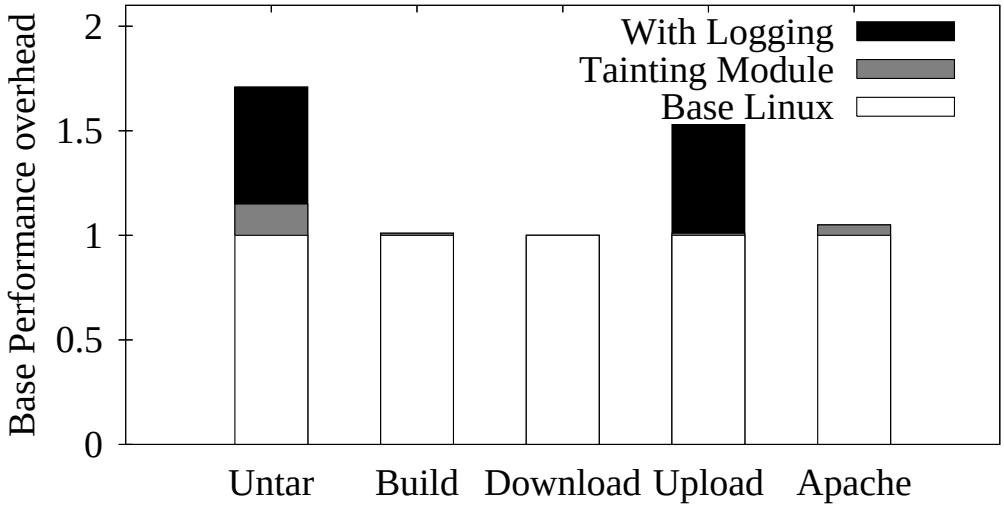

Figure 7.7: Performance overhead in base

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 (the unit value on the Y axis shows the performance of the application on a regular Linux system). The Solitude overhead consists of two components: tainting and logging. The first overhead, shown as "Tainting Module" in Figure 7.7, measures the overhead of running taint analysis without any logging. This overhead is measured by disabling logging to the backend. The second overhead, shown as "With Logging" in Figure 7.7, is measured by tainting the entire application to measure the overhead of running an application that was downloaded in an IFS and committed to the base.

The tainting module introduces the overhead of tracking taints in the frontend and performing system call level taint analysis. The taints need to be persistently stored in inode data structure of the base file system by marking the inodes dirty. The tainting module also needs to perform path checks to ignore files present in IFS. For example, Untar creates a large number of files and directories and hence, the tainting overhead is more pronounced.

Logging introduces significant overhead because all file and directory updates are logged to the backend system. This represents the worst case scenario when the entire `tar` application

is tainted and run in the base. The Build and the Apache benchmarks have smaller overhead than Untar. The Upload benchmark stresses the logging code in base since the tainted file is logged to the backend. The Download benchmark has no overhead.

Chapter 8

Conclusions

We have described Solitude, an application-level isolation and recovery system that is designed to both limit the effects of attacks and simplify the post-intrusion recovery process. At its core, Solitude provides a file-system based isolation environment for running untrusted applications. Each isolation environment is bound to its own file-system namespace, via a copy-on-write isolation file system called IFS. 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 the user decides that the application is malicious or undesirable, the entire compromised IFS environment can be discarded without concern for the integrity of the base file system. This recovery method, being simple, is accessible to ordinary users. However, certain files may need to be shared with the base file system, for securing or sharing data across different applications, leading to potential contamintation of the base. In such a case, Solitude provides a taint tracking layer to allow fine-grained recovery of the base.

8.1 Future Work

There are several directions of future work related to this project. Some of them are described below. A very nice benefit of our approach is that the system possesses additional tainting

information during the course of its operation. As a result, the system can make intelligent decisions regarding disallowing certain operations that may result in spread of false taints. For example, a tainted process may be disallowed writing to a *popular* file such as the password file. Careful monitoring of access patterns of popular objects in the system which are shared between various applications can further reduce the spread of taints in the system and improve recovery. A fair bit of research has been done in anomaly detection systems over the years. Anomaly detection along with taint information may prove beneficial in curbing anomalous behavior in the system.

We have successfully isolated the sources of taint to the commit points. A more detailed and thorough analysis of the committed files at the time of the commit would help in early detection of the problem. Analysis may include scanning with a virus checker, spyware and malware detectors, as well as verification of the file format. This may help in catching instances where, for example, a malicious file is disguised as an mp3 file that can subvert a media player application.. Such an approach would minimize the spread of contamination in the base by prevention of the commit itself.

The majority of users use Windows operating system. Hence, we want to apply our work to it. The sharing patterns would be slightly different between the applications in the Windows world which might pose new challenges. For example, in Windows, the registry would be a source of increased sharing between applications. One possible solution might be to isolate registry modifications on a per-application basis. However, the problems that may surface as a result would require a better understanding of the application behavior in the Windows Operating System.

Finally, we plan to explore the use of the Solitude infrastructure as a debugging environment and for configuration management.

Bibliography

[1] CHUTANI, S., ANDERSON, O. T., KAZAR, M. L., LEVERETT, B. W., MASON, W. A., AND SIDEBOTHAM, R. N. 1992. The Episode file system. In *Proceedings of the USENIX Technical Conference*.

[2] CORNELL, B., DINDA, P., AND BUSTAMANTE, F. 2004. Wayback: A user-level versioning file system for linux. In *Proceedings of the USENIX Technical Conference*. 19–28.

[3] CORPORATIN, M. 2007. Microsoft SoftGrid. `http://www.microsoft.com/systemcenter/softgrid/evaluation/virtualization.mspx`.

[4] COSTA, M., CROWCROFT, J., CASTRO, M., ROWSTRON, A., ZHOU, L., ZHANG, L., AND BARHAM, P. 2005. Vigilante: end-to-end containment of internet worms. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 133–147.

[5] FLOURIS, M. D. AND BILAS, A. 2004. Clotho: Transparent data versioning and the block I/O level. In *Proceedings of the IEEE Symposium on Mass Storage Systems*.

[6] GOEL, A., CHANG FENG, W., CHI FENG, W., MAIER, D., AND SNOW, J. 2007. Automatic high-performance reconstruction and recovery. *Journal of Computer Networks 51,* 5 (Apr.), 1361–1377. From Intrusion Detection to Self-Protection.

[7] GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. 2005. The Taser intrusion recovery system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.

[8] HASKIN, R., MALACHI, Y., SAWDON, W., AND CHAN, G. 1988. Recovery management in QuickSilver. *ACM Transactions on Computer Systems 6,* 1, 82 – 108.

[9] HINES, M. 2007. Google buys into security, acquires GreenBorder. `http://www.infoworld.com/article/07/05/29/Google-buys-into-AV\_1.html`.

[10] HSU, F., CHEN, H., RISTENPART, T., LI, J., AND SU, Z. 2006. Back to the future: A framework for automatic malware removal and system repair. In *Proceedings of the Annual Computer Security Applications Conference*.

[11] JAEGER, T., SAILER, R., AND ZHANG, X. 2003. Analyzing integrity protection in the SELinux example policy. In *Proceedings of the USENIX Security Symposium*. 59–74.

[12] KAMP, P.-H. AND WATSON, R. 2002. Jails: Confining the omnipotent root. In *Proceedings of the Second International SANE Conference*.

[13] KING, S. T. AND CHEN, P. M. 2003. Backtracking intrusions. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 223–236.

[14] LEYDEN, J. 2006. Spyware poses as Firefox extension. url-http://www.theregister.co.uk/2006/07/26/ firefox_malware_extension.

[15] LI, N., MAO, Z., AND CHEN, H. 2007. Usable mandatory integrity protection for operating systems. In *Proceedings of the IEEE Symposium on Security and Privacy*. 164–178.

[16] LOSCOCCO, P. AND SMALLEY, S. 2001. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the Freenix Track of USENIX Technical Conference*.

[17] MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A., AND ZADOK, E. 2004. A versatile and user-oriented versioning file system. In *USENIX Conference on File and Storage Technologies*.

[18] NEWSOME, J. AND SONG, D. 2005. Dynamic taint analysis for automatic detection, analysis, and signature generation of exploits on commodity software. In *Proceedings of the Network and Distributed System Security Symposium*.

[19] PERRIN, N. V. AND ET AL. Zfs. `http://www.opensolaris.org/os/community/zfs`.

[20] PETERSON, Z. N. AND BURNS, R. 2005. Ext3cow: A time-shifting file system for regulatory compliance. *ACM Transactions on Storage 1,* 2 (May), 190–212.

[21] PFAFF, B., GARFINKEL, T., AND ROSENBLUM, M. 2006. Virtualization aware file systems: Getting beyond the limitations of virtual disks. In *Proceedings of the Networked Systems Design and Implementation (NSDI)*.

[22] PIKE, R., PRESOTTO, D., THOMPSON, K., TRICKEY, H., AND WINTERBOTTOM, P. 1993. The use of name spaces in Plan 9. *ACM Operating Systems Review 27,* 2, 72–76.

[23] PRICE, D. AND TUCKER, A. 2004. Solaris zones: Operating system support for consolidating commercial workloads. In *Proceedings of the USENIX Large Installation Systems Administration Conference*.

[24] QUINLAN, S. AND DORWARD, S. 2002. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies*.

[25] ROSELLI, D., LORCH, J. R., AND ANDERSON, T. E. 2000. A comparison of file system workloads. In *Proceedings of the USENIX Technical Conference*.

[26] SABELFELD, A. AND MYERS, A. 2003. Language-based information-flow security. *IEEE Journal on Selected Areas in Communications 21,* 1 (Jan.), 5–19.

[27] SANTRY, D. S., FEELEY, M. J., HUTCHINSON, N. C., VEITCH, A. C., CARTON, R. W., AND OFIR, J. 1999. Deciding when to forget in the Elephant file system. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*. 110–123.

[28] SHAFIQUE, F. 2007. Application-level file system isolation. M.S. thesis, University of Toronto.

[29] SOLTESZ, S., PÖTZL, H., FIUCZYNSKI, M. E., BAVIER, A., AND PETERSON, L. 2007. Container-based operating system virtualization: A scalable, high-performance alternative to hypervisors. In *Proceedings of the EuroSys conference*. 275–287.

[30] SOULES, C. A. N., GOODSON, G. R., STRUNK, J. D., AND GANGER, G. R. 2003. Metadata efficiency in versioning file systems. In *Proceedings of the USENIX Conference on File and Storage Technologies*. 43–58.

[31] STRUNK, J. D., GOODSON, G. R., SCHEINHOLTZ, M. L., SOULES, C. A. N., AND GANGER, G. R. 2000. Self-securing storage: Protecting data in compromised systems. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*. 165–180.

[32] SUH, G. E., LEE, J. W., ZHANG, D., AND DEVADAS, S. 2004. Secure program execution via dynamic information flow tracking. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*. 85–96.

[33] SUN, W., LIANG, Z., SEKAR, R., AND VENKATAKRISHNAN, V. 2005. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the Network and Distributed System Security Symposium*.

[34] TERRY, D. B., THEIMER, M. M., PETERSEN, K., DEMERS, A. J., SPREITZER, M. J., AND HAUSER, C. H. 1995. Managing update conflicts in Bayou, a weakly connected replicated storage system. In *Proceedings of the 15th Symposium on Operating Systems Principles (SOSP)*. 172–183.

[35] VERMA, S. AND TORRE, C. 2005. Vista transactional file system. `http://channel9.msdn.com/Showpost.aspx?postid=142120`.

[36] VOGELS, W. 1999. File system usage in Windows NT 4.0. In *Proceedings of the Symposium on Operating Systems Principles (SOSP)*.

[37] WAGNER, D. AND TRIBBLE, D. 2002. A security architecture of the combex darpabrowser architecture. `http://www.combex.com/papers/darpa-review/security-review.pdf`.

[38] WATSON, A. AND BENN, P. 1999. Multiprotocol Data Access: NFS, CIFS, and HTTP. Tech. Rep. TR3014, Network Appliance, Inc. `http://www.netapp.com/tech_library/3014.html`.

[39] WRIGHT, C. P., DAVE, J., GUPTA, P., KRISHNAN, H., QUIGLEY, D. P., ZADOK, E., AND ZUBAIR, M. N. 2006. Versatility and unix semantics in namespace unification. *ACM Transactions on Storage 2,* 1 (Mar.), 74–105.

[40] ZELDOVICH, N., BOYD-WICKIZER, S., KOHLER, E., AND MAZIÈRES, D. 2006. Making information flow explicit in HiStar. In *Proceedings of the Operating Systems Design and Implementation (OSDI)*.

[41] ZHU, N. AND CHIUEH, T.-C. 2003. Design, implementation, and evaluation of repairable file service. In *Proceedings of the IEEE Dependable Systems and Networks*. 217–226.