Alcatraz: An Isolated Environment for Experimenting with Untrusted Software

ZHENKAI LIANG
National University of Singapore
WEIQING SUN
University of Toledo
V. N. VENKATAKRISHNAN
University of Illinois, Chicago
and
R. SEKAR
Stony Brook University

In this article, we present an approach for realizing a *safe execution environment (SEE)* that enables users to “try out” new software (or configuration changes to existing software) without the fear of damaging the system in any manner. A key property of our SEE is that it faithfully reproduces the behavior of applications, as if they were running natively on the underlying (host) operating system. This is accomplished via *one-way isolation:* processes running within the SEE are given read-access to the environment provided by the host OS, but their write operations are prevented from escaping outside the SEE. As a result, SEE processes cannot impact the behavior of host OS processes, or the integrity of data on the host OS. SEEs support a wide range of tasks, including: study of malicious code, controlled execution of untrusted software, experimentation with software configuration changes, testing of software patches, and so on. It provides a convenient way for users to inspect system changes made within the SEE. If these changes are not accepted, they can be rolled back at the click of a button. Otherwise, the changes can be committed so as to become visible outside the SEE. We provide consistency criteria that ensure semantic consistency of the committed results. We develop two different implementation approaches, one in *user-land* and the other in the *OS kernel*, for realizing a safe-execution environment. Our implementation results show that most software, including fairly complex server and client applications, can run successfully within our SEEs. It introduces low performance overheads, typically below 10 percent.

Categories and Subject Descriptors: D.4.6 [**Operating Systems**]: Security and Protection; H.4.0 [**Information Systems Applications**]: General

This research is supported in part by grants from ONR (000140110967, 000140710928) and NSF (CCR-0208877, CNS-0627687, CNS-0551660, and CNS-0716584).

Author’s email: V. N. Venkatakrishnan; email: venkat@cs.uic.edu.

General Terms: Systems, Security

Additional Key Words and Phrases: Isolation, one-way isolation

ACM Reference Format:
Liang, Z., Sun, W., Venkatakrishnan, V. N., and Sekar, R. 2009. Alcatraz: An isolated environment for experimenting with untrusted software. ACM Trans. Inf. Syst. Secur. 12, 3, Article 14 (January 2009), 37 pages. DOI = 10.1145/1455526.1455527.
http://doi.acm.org/10.1145/1455526.1455527.

1. INTRODUCTION

System administrators and desktop users often encounter situations where they need to experiment with potentially unsafe software or system changes. A high-fidelity safe execution environment (SEE) that can support these activities, while protecting the system from potentially harmful effects, will be of significant value to these users. Applications of such SEE include:

—*Running untrusted software*. Often, users run downloaded freeware/ shareware or mobile code. The risk of damage to the user's computer system due to untrusted code is high, yet a significant fraction of users seem to be willing to take this risk in order to benefit from the functionality offered by such code. An SEE can minimize security risks without negating the functionality benefits provided by such software.

—*Vulnerability testing*. System administrators may be interested in probing whether their computer systems, in their specific configuration, are susceptible to the latest e-mail virus or other attacks. A high-fidelity SEE can allow them to perform such testing without the risk of compromising production systems.

—*Software updates/patches*. Application of security patches is routinely delayed in large enterprises in order to allow time for compatibility and interoperability testing. Such testing is typically done after shutting down production systems for extended periods, and hence may be scheduled for weekends and holidays. In contrast, a high-fidelity SEE can allow testing of updates to be performed without having to shut down production systems. These concerns apply more generally to software upgrades or installations as well.

—*System reconfiguration*. Administrators may need to reconfigure software systems, and would ideally like to test out these changes before deploying them on production systems. This is currently accomplished manually by saving backup copies of all files that may be modified during reconfiguration. An SEE will automate this process, and moreover, avoid pitfalls such as overlooking to backup some of the modified files.

1.1 SEE Requirements and the Need for a New Approach

Consider an untrusted application that scans specified directories for image files and generates photo album files that are written to the same directories. (Several freeware programs (e.g., [Picturepages]) exist that provide this functionality.) The program also generates thumbnail pictures from these files (for creating index files) and has the ability to modify/resize these files. Additionally, the program is untrusted, and therefore may modify security-critical files of the user.[1] In order to support this application, an SEE must provide the following features:

—*Confinement without undue restrictions on functionality.* The untrusted photo album program needs to be confined. On one hand, the effects of this program running within an SEE should not escape the SEE and become visible to normal applications running outside. Otherwise, one cannot rule out the possibility of this program altering the operation of other applications running on the same system or elsewhere in the network. For instance, inserting a public key into the `authorized_keys2` file in the above example can enable an attacker (who crafted this program) to login to the user's account without requiring the user's password. The system must therefore alert the user to such security critical changes. On the other hand, we cannot disallow file system modifications by the photo album application; otherwise no album will be created.

—*Accurate environment reproduction.* For SEEs to be useful in the above application, it is essential that the behavior of applications be identical, whether or not they operate within the SEE. Specifically, the album program needs to access photos in the host system. Since the behavior of an application is determined by its environment (contents of configuration or data files, executables, libraries, etc.), it is necessary to reproduce, as accurately as possible, the same environment within the SEE as the environment that exists outside SEE.

—*Ability to commit results.* Once a photo album is successfully generated by this application, a user would like to retain it. Thus, the SEE must provide a mechanism to "commit" the results of activities that took place within it, if the user is satisfied with the results. A successful commit should have the same effect as if all of the operations carried out within the SEE actually took place outside.

Most existing approaches for safe execution do not satisfy these requirements. For instance, sandboxing techniques [Goldberg et al. 1996; Dan et al. 1997; Acharya and Raje 2000; Prevelakis and Spinellis 2001; Scott and Davidson 2002; Provos 2003] intercept security-critical operations made by a program, and disallow those operations that violate users' security policies. Sandboxing achieves confinement, but does so by severely restricting functionality of the sandboxed program.

File versioning systems [Santry et al. 1999; Zhu and Chiueh 2003; Muniswamy-Reddy et al. 2004; Chutani et al. 1992; Quinlan and Dorward 2002; Roome 1991; Soules et al. 2002; Peterson and Burns 2003] can provide rollback capabilities, but they don't provide a mechanism to discriminate among changes made by different processes, and hence cannot support

[1] /home/joe/.ssh/authorized_keys2.

selective rollback of the effects of untrusted process execution. For the same reason, it is also hard to commit the net effect of the observed program back to host environment.

Virtual machines (VMs) and related approaches [Chen and Nobl 2001; Whitaker et al. 2002; Malkhi and Reiter 2000; Chiueh et al. 2000] execute programs in environments isolated from users' host system, so that access restrictions can be relaxed. As discussed in detail in our related work section, VM approaches face difficulties in several areas. It is difficult to reproduce the exact host environment in the VM. VMs also have the difficulty to isolate changes made to external file systems (such as NFS). Furthermore, tracking changes made by untrusted processes from within is unreliable as the environment in a VM may be compromised.

The concept of *isolation* has been proposed as a way to address the problem of effect containment for compromised processes [Jajodia et al. 1998; Liu et al. 2000; Sekar et al. 1998]. Liu et al. [2000] proposed *one-way isolation* as an effective means to isolate the effects of running processes from the point they are compromised (or suspected of being compromised). But they do not consider the full range of applications of safe execution environment described above. Moreover, their work is focused on high-level protocols for realizing one-way isolation, and does not consider implementation issues that are central to our approach, such as application transparency, efficiency, and the subtleties in defining and implementing consistency criteria. We address these issues and present an efficient and easy-to-use safe-execution environment called *Alcatraz* that can support the range of applications discussed above.

1.2 Approach Overview

Our Alcatraz SEE is based on the concept of one-way isolation. Whereas VMs generally employ two-way isolation between the host environment and the environment that exists within a VM, one-way isolation makes the host environment visible within the SEE. In this way, Alcatraz processes see the environment of their host system, and hence accurate reproduction of environment is assured. However, the effects of Alcatraz processes cannot escape Alcatraz and interfere with the operation of processes outside Alcatraz.

In our approach, an SEE is created to run a process whose effects are to be shielded from the rest of the system. One or more such SEEs may be active on the host OS. Any children created by processes within an SEE will also be confined to that SEE, and will share the same consistent view of system state. Typically, users create a new SEE and carry out their tasks within it. Our SEE presents users with the changes made within the SEE. Users examine the changes from the host system, using helper applications, such as image or document viewers, or arbitrary utility applications. Users can not only access the states inside an SEE, but also the states in the host system, which is unaffected by the processes in the SEE. For example, users can compare file modified in an SEE and the same file in the host system to see the modification details. Finally, if users want to accept the changes made within the SEE, they can commit the results. The commit process causes the system state, as viewed inside the SEE, to be merged with the state of the host OS. We present consistency criteria aimed at ensuring the correctness of the results of the commit process.

Two distinct implementation approaches are described in this article. The first approach is implemented entirely at the user-level. The resulting system has several benefits from an end-user perspective. First, it empowers ordinary users (without administrative privileges) so that they can benefit from safe execution of untrusted code. Second, the absence of OS-resident components has the added benefit that it may be more readily ported, and more easily adopted by users that may be concerned about the impact of OS modifications to system stability. However, in order to achieve these benefits, the approach has to trade off performance and flexibility. In particular, it typically introduces overheads of the order of 100 percent. Moreover, a user-level implementation makes it difficult to accurately reproduce the semantics of certain operations involving directories, file permissions, and ownerships. To overcome these drawbacks, a complementary approach based on kernel-land implementation is described, allowing accurate reproduction of isolation semantics, and reducing performance overheads to under 10 percent.

1.3 Article Organization

The rest of this article is organized as follows. Section 2 presents an overview of our approach. Section 3 presents the implementation details of this approach. Specifically, Section 3.2 presents the user-land tool that implements this approach, and Section 3.3 describes our kernel-land approach. Section 4 discusses the criteria and algorithms for committing changes made to the file system. A comparison of the two implementations as well as other aspects of our approach are discussed in Section 5. Section 6 provides an evaluation of the functionality and the performance of our implementation. Related work is discussed in Section 7, followed by concluding remarks in Section 8. The Alcatraz tool is available for download in the software download section of the Web site.[2]

2. OVERVIEW OF APPROACH FOR IMPLEMENTING SAFE EXECUTION ENVIRONMENT

The two functions of our SEE are (a) to provide one-way isolation, and (b) to support commit operations. These two aspects of SEE are described in more detail below.

2.1 Achieving One-Way Isolation

Figure 1 illustrates the overview of our Alcatraz SEE. Alcatraz is a layer between the isolated program and the operating system, which is based on intercepting and manipulating the requests made by the isolated program. The primary goal of this isolation layer is *effect containment:* preventing the

[2] http://seclab.cs.sunysb.edu.

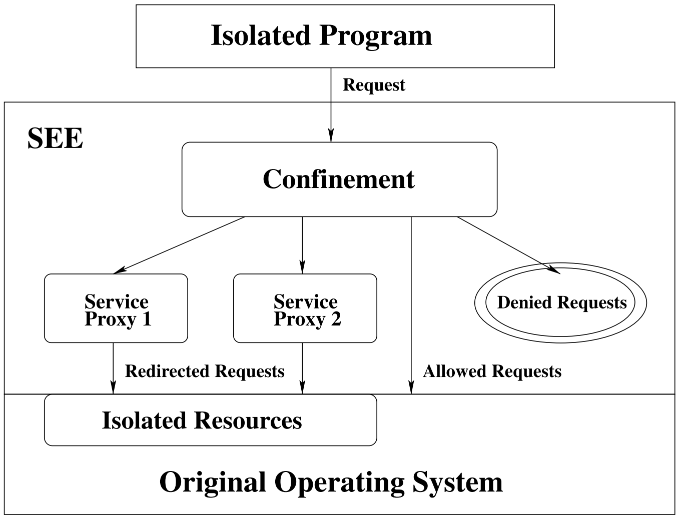


Fig. 1. Architecture of Alcatraz SEE. Alcatraz is a layer between the isolated program and operating system. It uses restriction and redirection to achieve one-way isolation.

effects processes in SEE from affecting the operation (or outcome) of processes executing outside the SEE.[3] This means that any read request (i.e., one that queries the system state but does not modify it) may be performed by SEE processes. It also means that write requests should not be permitted to keep system state from being affected. There are two options in this context: one is to *restrict* the request, that is, disallow its execution. The second option is to *redirect* the request to a different resource that is invisible outside the SEE. Once a write request is redirected, it is important that subsequent read requests on the same resource be redirected as well. This is handled by service-specific proxies.

By restriction, we mean that a request is prevented from execution. An error code may be returned to the process, or the request may be silently suppressed and a success code returned. In either case, restriction is easy to implement—we need only know the set of requests that can potentially alter system state. In Alcatraz SEE, restriction is achieved using the confinement module, as is shown in Figure 1. The main drawback of restriction is that it will likely prevent applications from executing successfully. For instance, if a program writes to a file, it expects to get back the same content at a later point in the program when the file is read. However, an approach based on restriction cannot do this, and hence most nontrivial applications will fail to run successfully under such restriction. For this reason, restriction is a choice of last resort in our approach.

By redirection, we mean that any request that modifies some component of the host environment is instead redirected to a different component that

[3] Note that we are interested in confinement [Lampson 1973] from the point of view of system integrity, rather than confidentiality. As such, we do not deal with issues such as covert channels.

is not accessed by the host OS processes. Alcatraz SEE handles redirection by service-specific proxies, which redirect all modifications to the system to components in the *isolated resources* (shown in Figure 1). For instance, in the file system proxy, when an SEE process tries to modify a file, a copy of the original file may be created in a private area of the file system, and the modification request redirected to this copy. Redirection is intended to provide a consistent view of system state to processes in SEE, thereby allowing them to complete successfully.

Redirection can be *static* or *dynamic.* Static redirection requires the source and target objects to be specified beforehand. It is ideal for network operations. For instance, one may statically specify that requests to bind a socket to a port p should be redirected to an alternate port p'. Similarly, one may specify that requests to connect to a port p on host h should be redirected to host h' (which may be the same as h) and port p'. By using such redirection, we can build *distributed SEEs*, where processes executing within SEEs on multiple hosts can communicate with each other. Such distributed SEEs are particularly useful for safe execution of a network server application, whose testing would typically require accesses by nonlocal client applications. (Note, however, that this approach for distributed SEEs works only when all cross-SEE communications take place directly between the SEE processes, and not through other means, for example, indirect communication through a shared NFS directory.)

Static redirection becomes infeasible if the number of possible targets is too large to be enumerated in advance. For instance, it is hard to predict the files that may be accessed by an arbitrary application. Moreover, there are dependencies among requests on different file objects, for example, a request to create a file has the indirect effect of changing the contents of the directory in which the file is created. Simply redirecting an access on the file, without correspondingly modifying accesses of the directory, will result in an inconsistent file system state. To handle such complexities, our approach supports dynamic redirection, where the target for redirection is determined automatically during the execution of SEE processes. However, the possibility of hidden dependencies means that the implementation of dynamic redirection may have to be different for different kinds of objects. That is why redirection is supported by service-specific proxies. The key challenge in implementing such proxies (including file system proxies and network proxies) is that, even though they buffer certain requests, they should provide a consistent view of system state to the SEE applications. Specifically, if an SEE process writes to such a proxy and subsequently performs a read request, the proxy should return the result that would have been returned if the write request had actually been carried out.

In our current implementations, system call interposition is used to implement restriction and static redirection. We restrict all modification requests other than those that involve the file system and the network. In the case of file system requests, all accesses to normal files are permitted, but accesses to raw devices and special purpose requests such as mounting file systems are disallowed. In terms of network operations, we permit any network access for which static redirection has been set up. In addition, accesses to the name server and X-server are permitted. (In reality, SEE processes should not get unrestricted access to X-server. Our current implementation solves this problem by statically redirecting X requests to a separate X-server that nested in the host X-server.)

Dynamic redirection is currently supported in our implementation only for file system accesses by a proxy layer, called the Isolation File System (IFS). In our user-land implementation, it is implemented using system call interposition (described in Section 3.2). In our kernel implementation, this proxy is implemented at the virtual file system layer, as described in detail in Section 3.3.

2.2 Committing Changes

Modifications made by Alcatraz processes are held in isolated resources. Users can check the net results of SEE processes using their security policies. Compared to traditional sandboxing approaches, Alcatraz SEE facilitates access to a richer class of information, for example, detailed list of modifications, system states before and after execution. If the modifications are desirable, they need to be committed to the original operating system, so that they are visible to other processes. There are two key challenges in committing: one is to ensure consistency of the resulting system state; the other is efficiency—to reduce the space and time overheads for logging and re-running of operations to a level that provides good performance.

Some systems expertise is required in making these committing decisions. For users with the expertise, such as system administrators, our SEE allows them to base their decisions on more details about modifications inside SEE, such as the list of modified resources and details of each modification. In addition, the system also has an option that prompts users to select a subset of files from all those that were modified in the SEE and export them to a specified directory, (e.g., a removable disk) without modifying original system files. In this way, users keep results of an SEE session without propagating these changes to the main system.

We now provide a high-level overview of the issues involved in committing results. The key problem in terms of consistency is that a resource accessed within the SEE may have been independently accessed outside of the SEE. This corresponds to concurrent access on the same resource by multiple processes, some within SEE and some outside. One possible consistency criterion is the serializability criterion used in databases. Other consistency criteria may be appropriate as well, for example, for some text files, it may be acceptable to merge the changes made within the SEE with changes made outside, as long as the changes involve disjoint portions of the file. A detailed discussion of the issues involved in defining commit criteria is presented in Section 4.1.

There may be instances where the commit criteria may not be satisfied. In this context, we make the following observations:

—There is no way to guarantee that results can be committed automatically and produce consistent system state, unless we are willing to delay or disallow execution of some applications on the host OS. Introducing restrictions or delays on host OS processes will defeat the purpose of SEE, which is to shield the host OS from the actions of SEE processes. Hence this option is not considered in our approach.

—If the results are not committed, then the system state is unchanged by tasks carried out within the SEE. This means that these tasks can be rerun, and will most likely have the same desired effect. Hopefully, the conflicts were the results of infrequent activities on the host OS, and won't be repeated this time, thus enabling the results to be committed.

—If retrying isn't an option, the user can manually resolve conflicts, deciding how the files involved in the conflict should be merged. In this case, the commit criteria identifies the files and operations where manual conflict resolution is necessary.

As a final point, we note that if a process within an SEE communicated with another process executing within a different SEE, then all such communicating SEEs need to be committed as if they were part of a single distributed transaction. Currently, our implementation does not support distributed commits. Our approach for committing the results of operations performed within a single SEE is described in Section 4.

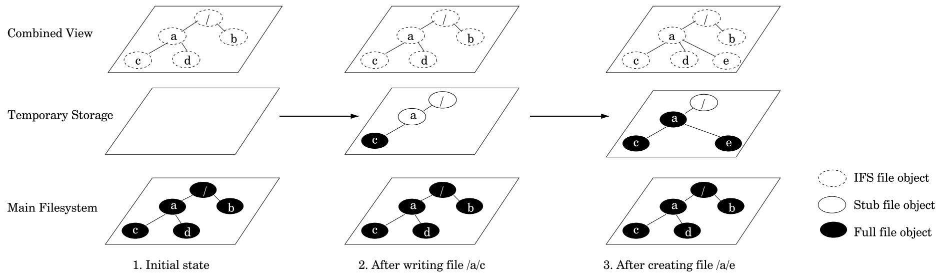
3. IMPLEMENTATION OF IFS

3.1 High-Level Overview

An intuitive way to realize dynamic redirection is to use copy-on-write: when a node in the original file system is about to be modified, a copy of this node is created in a private area of the file system, called *temporary storage*, which is part of the isolated resources in Figure 1. The write operation, as well as all other subsequent operations on this node, are then redirected to this copy. By doing so, the modification to the operating system is actually cached in the temporary storage, and the main file system remains unchanged. The isolated program's view of the file system is a combined view of the main file system and the changes in the temporary storage.

We illustrate the operation of IFS using the example shown in Figure 2. Suppose that initially (i.e., step 1 in this figure), there is a directory `a` and a file `b` under the root directory in the main file system, with files `c` and `d` within directory `a`. Step 2 of this figure illustrates the result of modifying the file `/a/c` within the SEE. The copy-on-write operation on `/a/c` copies the file `/a/c` from the main file system to the temporary storage, and remember the relationship between the two files, we call the unchanged directory in its path *stub*s. Subsequent accesses are redirected to this copy in temporary storage.

The third step of Figure 2 shows the result of an operation that creates a file `/a/e` within the SEE. Since this changes the directory `a` by adding another file to it, the directory is marked changed. Next, the file `e` is created within the temporary storage under that directory. The combined view of IFS reflects all these changes: accesses to file `/a/c` and `/a/e` are redirected to the corresponding



1. Initial state

2. After writing file /a/c

3. After creating file /a/e

Fig. 2. Illustration of IFS layout on modification operations.

copies in the temporary storage, while accesses to file `/a/d` will still go to the version in the main file system.

3.2 User-Level Implementation of IFS

3.2.1 *Underlying mechanism.* Our user-level IFS is based on a system call interceptor. The system call interceptor is designed to be easily portable to other Unix variants. The architecture of our interceptor is based on the design presented in Jain and Sekar [2000], which is implemented by Linux's `ptrace` system call. `ptrace` mechanism allows one process to monitor another process. Monitoring capabilities include the ability to intercept system calls made by the process in SEE, and examination or modification of its virtual memory.

3.2.2 *Challenges and solutions.* The key challenge in implementing the IFS to maintain a consistent file system view to SEE processes after file system requests that affect other requests implicitly. This is a challenging task because of the different kinds of file system objects (regular files, directories, symbolic links, etc.) and the large number of file system related operations (34 out of the 243 system calls in Linux kernel version 2.4.18). To tackle this complexity, we aim to reduce the number of cases to be considered by classifying file system objects and related system calls. We made the following observations about the types of file system objects that need to be considered: regular files, directories, symbolic links, and Inodes. (Inodes contain meta data about files, such as permission, ownership etc.) Modification requests may be different across these file types. For example, regular files are viewed as a stream of bytes, and can be modified by seeking to any location (expressed as a byte offset) within the file, and performing a `write` system call. Directories, on the other hand, are viewed as a sequence of directory entries, which are records containing information about the files within the directory. For symbolic links, the only modification is that of file deletion, which is actually a directory modification. Thus, we need only consider three types of objects of the file system: regular files, directories, and Inodes.

Now consider the system call operations on the file system. For the isolation operation, we need to consider mostly those system calls that are related to path names. System calls that operate on file descriptors (e.g., `read`, `write` and `mmap`) can be handled automatically by the operating system once path-related

Read Only Operations	Modification Operations		
	Regular Files	Directories	Inodes
execve, chdir, access, chroot, readlink, uselib, statfs, stat, lstat, stat64, lstat64, oldstat, getdents, getdents64, readdir	open, truncate, truncate64	creat, link, unlink, mknod, rename, mkdir, rmdir, acct, symlink, open	chmod, lchown, utime, oldlstat, chown, lchown32, chown32

Fig. 3. Classification of file system related system calls.

calls are taken care of. The classification of those calls are shown in Figure 3, based on how they modify file systems. Next, we describe how IFS is achieved in each category.

Regular file modifications. Consider a process that opens a file f for writing. A natural way to isolate the execution of the process is to create a new copy f' of f that is stored in the temporary storage. All future accesses to f, whether they be modifications or reads, will be redirected to f'. To enable this redirection, a map associating f with f' is remembered by the temporary storage. As an optimization, we avoid copying of files when a file is truncated to zero length.

As a side effect, copying regular files may change its ownership. Consider the case when the isolated process modifies a file that it does not own but has the write permission. The IFS will copy the file into the temporary storage before making these changes. During copying process, the operating system will automatically set the ownership of the copy to that of the owner of the isolated process. It would be preferable to change the ownership back to the owner of the original file, but this may be disallowed by the kernel because the user may not necessarily be the superuser. Therefore, if there is a change in ownership, then the related operations, such as permission checking, need to be intercepted and performed in IFS.

Directory modifications. We observe that unlike a regular files, directories are accessed in a structured manner using specialized directory operations such as `mkdir` and `getdents`. Thus, our approach is to modify these operations in a manner that achieves copy-on-write semantics without having to perform actual copies of directory contents. In particular, modifications to directories, such as creation/deletion of new files or directories, are recorded in the temporary storage, without copying the affected directories.

When the contents of such modified directories are read using the `getdents` operation, we can apply the modification information recorded by temporary storage to the returned directory entries. For each returned directory entry, IFS checks whether it is marked as deleted in the temporary storage. If so, the entry is removed from the result. It is possible that all the entries returned by getdents may be deleted in this step. If, as a result of this, no entries are returned to the isolated process, it would conclude that the end of the directory has been reached. To solve this problem, IFS first retrieves all of the directory entries in the directory, and applies the above changes to the directory entries. We then append new directory entries that are recorded in the temporary storage but not present in the rest of the file system. The result is returned to the SEE process.

Inode modification. Modification can also be made to Inodes which store file system meta data. Inodes are associated with files and cannot be copied separately. Therefore, if the modification is made to a file that has already been copied to the temporary storage (i.e., just created or modified file), we can redirect this operation to its counterpart in the temporary storage. If the modification is made to an unchanged regular file, we can again copy the file into the temporary storage and proceed as in the previous case. If the Inode to be changed belongs to a directory, Alcatraz stores the new Inode information in the temporary storage to avoid copying the directory. One limitation of this approach is that the Inode data is not visible to the system. Therefore, even if a permission is granted to a process, such as entering a directory, the operation will still be denied as the original directory is not permitted for access. This limitation is addressed in our kernel implementation.

Since the latest Inode information is held within the temporary storage, system calls to access or manipulate meta data, such as stat, need to be intercepted to reflect the side effects of previous Inode modifications.

The focus of user-level IFS is to facilitate applicability in situations where the user does not have administrative privileges. However, as is discussed in previous section, the underlying mechanism has difficulties in maintaining consistency on access privileges of file system objects. Consequently, some tasks traditionally performed by the kernel were reimplemented in user-level IFS, and as discussed this can be prone to errors. To address these problems with the user-land implementation we discuss a kernel-level implementation of IFS, where IFS has access to internal file system objects.

3.3 Kernel Implementation of IFS

3.3.1 *Underlying mechanism.* Our kernel IFS is implemented by interposing file system operations within the OS kernel at the Virtual File System (VFS) layer. VFS is a common abstraction in Unix across different file systems, and every file system request goes through this layer. Hence extensions to functionality provided at VFS layer can be applied uniformly and transparently to all underlying file systems such as ext2, ext3, and NFS.

We realize VFS layer interposition using the stackable file system approach described in Zadok et al. [1999]. In effect, this approach allows one to realize a new file system that is layered over existing file systems. Accesses to the new file system are first directed to this top layer, which then invokes the VFS operations provided by the lower layer. In this way, the new file system extends the functionality of existing file systems without the need to deal with file-system-specific details.

3.3.2 *Challenges and solutions.* The description in Section 3.1 presented a simplified view of the file system, where the file system has a tree-structure and consists of only plain files and directories. In reality, UNIX file systems have a DAG (directed acyclic graph) structure due to the presence of hard links. In addition, file systems contain other types of objects, including symbolic links and special device files. IFS usually does not allow accesses to special device files. An exception to this rule is made for `pty`'s and `tty`'s, as well as pseudo devices like `/dev/zero`, `/dev/null`, etc. In these cases, access is redirected to the corresponding device files on the main file system. A symbolic link is simply a plain file, except that the content of the file is interpreted as the path name of another file system object. For this reason, they don't need any special treatment. Thus, we need only describe how IFS deals with hard links (and the DAG structure that can result due to their use.)

When the file system is viewed as a DAG, its internal nodes correspond to directories, and the leaves correspond to files. IFS does not look into the internal structure of files, and hence we treat them as leaf objects in the DAG. All nodes in the DAG are identified by a unique identifier called the *Inode number*. (The inode number remains unique across deletion and recreation of file objects.) The edges in the DAG are *links*, each of which is identified by a name and the Inode number of the object pointed by the link. This distinction between nodes and links in the file system plays a critical role in every aspect of IFS design and implementation, in particular, the implementation of IFS commit operation as described in Section 4.2.

IFS layer contains a table that maintains additional information necessary to correctly support IFS operation. This table, which we call as *inode table*, is indexed by the inode numbers of file system objects. It has a field indicating that whether the inode corresponds an object in temporary storage (temp) or an object the main file system (main). Further, if it is an object in the temporary storage, the flag indicates whether it is a stub object (stub). A stub object is simply a reference to the version of the same object stored in the main file system. In addition, auxiliary information needed for the commit operation is also present, as described in Section 4.2.

In our IFS implementation, copy-on-write of regular files is implemented using normal file copy operations. In particular, when a plain file f is modified for the first time within the SEE, a stub version of all its ancestor directories is created in temporary storage (if they are not already there). Then the file f is copied into temporary storage. From this point on, all references to the original file will be redirected to this copy in temporary storage.

After creating a copy of f, we create an entry in the inode table corresponding to the original version of f on the main file system. This is done so as to handle hard links correctly. In particular, consider a situation when there is a second hard link to the same file object, and this link has not yet been accessed within IFS. When this link is subsequently accessed, it will be referencing a file in the main file system. It is necessary to redirect this reference to the copy of f in temporary storage, or otherwise, the two links within IFS that originally referred to the same file object will now refer to different objects, thereby leading to inconsistencies.

The copy-on-write operation on directories is implemented in a manner similar to that of files. Specifically, a stub version of the directory's ancestor nodes are first created in temporary storage. Next, the directory itself is copied. This copy operation is a *shallow copy* operation, in that only a stub version of the objects listed in the directory are created. By performing this, the directory in temporary storage will have the same meta data and directory content as its main file system counterpart. So the redirected operation performed on this directory will exhibit the same behavior. In principle, one can use shallow-copy on files as well, thus avoiding the overhead of copying disk blocks that may not be changed within the IFS. However, the internal organization of files is specific to particular file system implementations, whereas we want to make IFS to be file-system independent. Hence files are chosen to be copied in their entirety.

4. IMPLEMENTATION OF IFS COMMIT OPERATION

At the end of SEE execution, the user may decide either to discard the results or commit them. In the former case, the contents of IFS are destroyed, which means that we simply delete the contents of temporary storage and leave the contents of the main file system as is. In the latter case, the contents of the temporary storage need to be merged into the main file system.

When merging the contents of temporary storage and main file systems, note that conflicting changes may have taken place within and outside the IFS, for example, the same file may have been modified in different ways within and outside the SEE. In such cases, it is unclear what the desired merge result should be. Thus, the first problem to be addressed in implementing the commit operation is that of identifying *commit criteria* that ensure that the commit operation can be performed fully automatically (i.e., without any user input) and is guaranteed to produce meaningful results. We describe possible commit criteria in Section 4.1. Following this, we describe an efficient algorithm for committing results in Section 4.2.

If the commit criteria is not satisfied, then manual reconciliation of conflicting actions that took place inside the SEE and outside will be needed. The commit criteria will also identify the set of conflicting files and operations. At this point, the user can decide to:

—*abort,* that is, discard the results of SEE execution. This course of action would make sense if the activities performed inside SEE are longer be relevant (or useful) in the context of changes to the main file system.

—*retry*, that is, discard the results of SEE execution, create a new SEE environment, redo the actions that were just performed within the SEE, and then try to commit again. If the conflict were due to activities on the host OS that are relatively infrequent, for example, the result of a cron job or actions of other users that are unlikely to be repeated, then the retry has a high probability of allowing a successful commit. (Note that the retry will likely start with the same system state as the first time and hence will have the same net effect as the first time.)

—*resolve conflicts*, that is, the user manually examines the files involved in the conflict (and their contents) and determines if it is safe to commit; and if so, what is the merged contents of the files involved in the conflict. The commit criteria will identify the list of files involved in the conflict and the associated operations, but the rest of the steps need to be performed manually.

In addition to committing all changes made in an IFS to the host system, our approach also allows users to select a set of modified files, and export them to a specified directory (e.g., a removable disk). In this way, users can have the advantage of choosing the modifications they want without worrying about security or making system-wide changes.

4.1 Commit Criteria

The commit criteria is a set of rules which determine whether the results of changes made within an SEE can be committed automatically, and lead to a consistent file system state. Since the problem of consistency and committing has been studied extensively in the context of database transactions, it is useful to formulate the commit problem here in the terms used in databases. However, note that there is no well-defined notion of transactions in the context of IFS. We therefore identify the entire set of actions that took place within SEE in isolation as a transaction T_i and the entire set of actions that took place outside of the SEE (but limited to the actions that took place during the lifetime of the SEE) as another transaction T_h.

There are several natural choices for commit criteria:

—*Noninterference.* This requires that the actions contained in T_i be unaffected by the changes made in T_h and vice-versa. More formally, let $RS(T)$ and $WS(T)$ denote respectively the set of all filesystem objects read and written by a transaction T, respectively. Then, noninterference requires that

$$RS(T_i) \cap WS(T_h) = \phi$$

$$RS(T_h) \cap WS(T_i) = \phi$$

$$WS(T_i) \cap WS(T_h) = \phi$$

The advantage of this criteria is that it leads to very predictable and understandable results. Its drawback is that it is too restrictive. For instance, consider a conflict that arises due to a single file f that is written in T_h and read in T_i. Also suppose that f was read within the SEE after the time of the last modification operation on f in T_h. Then it is clear that T_i used the modified version of f in its computation, and hence it need not be aborted, yet the noninterference criteria will not permit T_i to be committed.

—*Serializability.* This criteria requires that the effect of concurrent transactions be the same as if they were executed in some serial order, that is, an order in which there was no interleaving of operations from different transactions. In the context of IFS, there are only two possible serial orders, namely, T_iT_h and T_hT_i. Serializability has been used very successfully in the context of database transactions, so it is a natural candidate here. However, its use in SEE can lead to unexpected results. For instance, consider a situation where a file f is modified in T_i and is deleted in T_h. At the point of commit, the user would be looking at the contents of f within the SEE and would expect this result to persist after the commit, but if the serial order T_iT_h were to be permitted, then f would no longer be available! Even worse, its contents would not be recoverable. Thus, serializability may be too general in the context of SEE: if results were committed automatically when T_i and T_h were serializable, then there is no guarantee that the resulting system state would be as expected by the user of the SEE.

—*Atomic execution of SEE activities at commit time.* If the state of main file system after the commit were as if all of the SEE activities took place atomically at the point of commit, then it leads to a very understandable behavior. This is because the contents of the main file system after the commit operation will match the contents of the IFS on every file that was read or written within the IFS. The atomic execution criteria (AEC) is a restriction of serializability criterion in that only the order T_hT_i is permitted, and the order T_iT_h, which led to unexpected results in the example above, is not permitted.

Based on the above discussion, we use AEC as the criteria for automatic commits in SEE. In all other cases, the user will be presented with a set of files and directories that violate the AEC, and the user will be asked to resolve the conflict using one of the options discussed earlier (i.e., abort, redo, or manually reconcile).

In addition to providing consistent results, a commit criteria should be amenable to efficient implementation. In this context, note that we don't have detailed information about the actions within T_h. In particular, the UNIX file system maintains only the last read time and write time for each file system object, so there is no way to obtain the list of all read and write actions that took place within T_h, or their respective timestamps. We could, of course, maintain such detailed information if we intercepted all file operations on the main file system and recorded them, but this conflicts with our design goal that operations of processes outside SEE should not be changed in any way. On the other hand, since we do intercept all file accesses within the IFS, we can (and do) maintain more detailed information about the timestamps of the read and write operations that took place within the SEE. Thus, an ideal commit criteria, from an implementation perspective, will be one that leverages the detailed time stamp information we have about T_i while being able to cope with the minimal time stamp information we have about T_h. It turns out that AEC satisfies this condition, and hence we have chosen this criteria as the basis for fully automated commits in IFS.

In order to determine whether AEC is satisfied, we need to reason about the timestamps of operations in T_h and T_i and show that their orders can be permuted so that all operations in T_h occur before the operations in T_i, and that this permutation does not change the semantics of the operations. We make the following observations in this regard:

- Any changes made within the SEE are invisible on the main file system, so the results of operations in T_h would not be changed if all T_i operations were delayed to the point of commit.
- A read operation $R(f)$ performed in T_i can be delayed to the point of commit and still be guaranteed to produce the same results, provided the target f was unchanged between the time R was executed and the time of commit. This translates to requiring that the last modification time of f in the main file system precede the timestamp of the first read operation on f in T_i.
- The results of a write operation $W(f)$ performed in T_i is unaffected by any read or write operation in T_h, and hence it can be delayed to commit time without changing its semantics.

Based on the observations, we conclude that AEC is satisfied if

> *the earliest read-time of an object within the IFS occurs after the last modification time of the same object on the main file system.*

Note that the latest modification time of an object on the main file system is given by the `mtime` and `ctime` fields associated with that object. In addition, we need to maintain the earliest read-time of every object within the IFS in order to evaluate this criteria.

A slight explanation of the above criteria is useful in the context of append operations on files. Consider a file that is appended by an SEE process is subsequently appended by an outside process. Both appends look like a write operation, and hence the above commit criteria would seem to indicate that it is safe to commit results. But if this were done, the results of the append operation performed outside IFS would be lost, which is an unexpected result. Clearly, if the SEE process were run at the time of commit, then no information would have been lost. However, this apparent problem is clarified once we realize that an append operation really involves a read and then a write. Once this is taken into account, a conflict will be detected between the time the file was read within IFS and the time it was modified outside, thereby causing the AEC criteria to be violated. More generally, whenever a file is modified within IFS without completely erasing its original contents (which is accomplished by truncating its length to zero), we treat this as a read followed by a write operation for the purposes of committing, and handle the above situation correctly.

4.1.1 *Improvements to AEC.*

The above discussion of AEC classifies operations into two kinds: read and write. The benefit of such an approach is its simplicity. Its drawback is that it can raise conflicts even when there is a meaningful way to commit. We illustrate this with two examples:

- System log files are appended by many processes. Based on earlier discussion about append operations on files, the AEC criteria won't be satisfied whenever an SEE process appends an entry e_1 to the log file and an outside process subsequently appends another entry e_2 to the same file. Yet, we see that the results can easily be merged by appending both e_1 and e_2 to the log file.

—Directories close to the root of the file system are almost always examined by SEE process as part of looking up a file name in the directory tree. Thus, if any changes were to be made in such directories by outside processes, it will lead to AEC being violated. Yet, we see that a name lookup operation does not conflict with a file creation operation unless the name being looked up is identical to the file created.

These examples suggest that AEC will permit commits more often if we distinguished among operations at a finer level of granularity, as opposed to treating them as read and write operations. However, we are constrained by the fact that we don't have a complete record of the operations executed by outside processes. Therefore, our approach is to try to infer the operations by looking at the content of the files. In particular, let f_o denote the (original) content of a file system object at the point it was copied into temporary storage, and f_h and f_i denote the content of the same file in the main file system and the IFS at the point of commit. We can then compute the difference δ_h^f between f_o and f_h, and the difference δ_i^f between f_o and f_i. From these differences, we can try to infer the changes that were made within and outside SEE. For instance, if both δ_h^f and δ_i^f consist of additions to the end of the file, we can infer that append operations took place, and we can apply these differences to f_o.

In the case of directories, the situation is a bit simpler. Due to the nature of directory operations, δ_h^f will consist of file (or subdirectory) creation and deletion operations. Let F_h denote the set of files created or deleted in δ_h^f, and let F_i be the set of names in this directory that were looked up in T_i. This information, as well as the time of first lookup on each of these names, are maintained within the IFS. Let $F_c = F_h \cap F_i$. Now, we can see that the AEC criteria will be satisfied if either one of the following conditions hold:

—$F_c = \phi$, or

—the modification time of f_o precedes all of the lookup times on any of the files in F_c.

In the first case, none of the names looked up (i.e., read) within the SEE were modified outside, thus satisfying AEC. In the second case, conflicts are again avoided since all of the lookups on conflicting files took place after any of the modification operations involving them in the main file system.

We point out that inferring operations from the state of the file system can be error-prone. For instance, it is not possible to distinguish from system state whether a file a was deleted or if it was first renamed into b and then deleted. For this reason, we restrict the use of this approach to log files and directories. In other cases, for example, updates of text files, we can use this technique with explicit user input.

4.2 Efficient Implementation of Commit

After making a decision on whether it is safe to commit, the next step is to apply the changes to the main file system. One naive solution is to maintain a complete log of all successful modifications operations that were performed within the SEE, and replay them on the main file system at the point of commit. This approach has the benefit of being simple and being correct in terms of preserving the AEC semantics. However, its drawback is that it is inefficient, both in terms of space and time. In the worst case, the storage overhead can be arbitrarily high. For instance, consider an application that creates and deletes many (temporary) files. In this case, a log-based approach will need to store all information about the write operations that were performed, including those on files that were subsequently deleted.

We notice that the desired file system state is already accumulated in the temporary storage of the SEE. It saves both time and space by simply copying them over to the host system. However, this simple solution will treat a hard link as a standalone file. Therefore, we need to treat links separately. For files, the commit action used in our approach involves simply renaming (or copying) the file into the main file system. For operations related to links, it records a minimal set of link-related operations that captures the set of links associated with each file system object. In this sense, one can think of the approach as state-based, that maintains condensed logs that were discussed above, where redundant information is pruned away. For instance, there is no need to remember operations on a file if it is subsequently deleted. Similarly, if a file is renamed twice, then it would be enough to remember the net effect of these two renames. To identify such redundancies efficiently, our approach partitions the logs based on the objects to which they apply. This log information is kept in the inode table described earlier.

Operations that modify the contents of a file or change metadata (such as permissions) on any file system object are not maintained in the logs, but simply applied to the object. In effect, the state of the object captures the net effect of all such operations, so there is no need to maintain them in a log. Thus, only information about file or directory creation and deletion, and those that concern addition or removal of links are maintained in the log. In addition, to simplify the implementation, we separate the effects of creating or deleting file system objects from the effect of adding or deleting links. This means that the creation of a file would be represented in our logs by two operations: one to create the file object and another to link it to the directory in which the object is created. Similarly, a rename operation is split into an operation to add a link, another to remove a link, and a third (if applicable) to delete the file originally referenced by the new name. As in previous sections, file objects involved in these operations are identified by inode numbers rather than path names.

Specifically, the log contains one of the following operations:

—*create* and *delete* operations denote respectively the creation of a file or a directory, and are associated with the created file system object.

—*addlink* and *rmlink* operations denote respectively the addition and deletion of a link from a directory to a file system object. These operations are associated with the file system object that is the target of the link, and have two operands. The first is the inode number of the parent directory and the second is the name associated with the link.

The effect of some of these operations is superseded by other operations, in which case only latter operations are maintained. For instance, a delete operation supersedes a create operation. An rmlink operation cancels out a preceding addlink with the same operands.

In addition to removing redundant operations from the logs, we also reorder operations that do not interfere with each other in order to further simplify the log. In this context, note that two valid addlink operations in the log associated with any file system object are independent. Similarly, any addlink operation on the object is independent of an rmlink operation. (Both these statements are true only when we assume that operations that are superseded or canceled by others have already been removed from the log.)

Based on this discussion, we can see that a condensed log associated with a file system object can consist of operations in the following order:

—Zero or one create operation. Since the file system object does not exist before creation, this must be the first operation in the log, if it exists.

—Zero or more rmlink operations. Note that multiple rmlink operations are possible if the file system object was originally referenced by multiple links. Moreover, the parent directories corresponding to these rmlink operations must all have existed at the time of creation of SEE, or otherwise an addlink operation (to link this object to the parent directory) must have been executed before the rmlink. In that case, the addlink and rmlink operations would have cancelled each other out and hence won't be present in the condensed log.

—Zero or more addlink operations. Note that multiple addlink operations are possible if the object is being referenced by multiple links. Also, there must be at least one addlink operation if the first operation in the log is a create operation.

—Zero or one delete operation. Note that when a delete operation is present, there won't be any addlink operations, but there may be one or more rmlink operations in the log.

Given the condensed logs maintained with the objects in the inode table, it seems straightforward to carry out the commit operation. The only catch is that we only have the relative ordering of operations involving a single file system object, but lost information about the global ordering of operations across different objects. This raises the question as to whether the meanings of these operations may change as a result. In this context, we make the following observations:

—Creation and deletion operations do not have any dependencies across objects. Hence the loss of global ordering regarding these operations does not affect the semantics of these operations.

—Rmlink operation depends upon the existence of parent directory, but nothing else. This means that as long as it is performed prior to the deletion of parent directory, its meaning will be the same as it was executed in the global order in which it was executed originally.

—Addlink operation depends on the creation of the parent directory (i.e., the directory in which the link will reside) and the target object. Moreover, an addlink operation involving a given parent directory and link name has a dependency on any other rmlink operation involving the same parent directory and link names. This is because the addlink operation cannot be performed if a link with the same name is present in the parent directory, and the execution of rmlink affects whether such a link is present. Thus, the effect of addlink operations will be preserved as long as any parent directory creation, as well as relevant rmlink operations are performed before.

Among operations that have dependency, one of the two possible orders is allowable. For instance, an rmlink operation cannot precede the existence of either the parent directory or the target of the link. Similarly, an addlink operation cannot precede an rmlink operation with the same parent directory and name components. (Recall that we have decomposed a rename operation into rmlink (if needed), addlink and an object delete (if needed) operations, so it cannot happen that an addlink operation is invoked on a parent directory when there is already another link with the same name in that directory.) This means that even though the global ordering on operations has been lost, it can be reconstructed. Our approach is to traverse the file system within the temporary storage, and combine the condensed logs while respecting the above constraints, and then execute them in order to implement the commit step.

Atomic Commits. As mentioned before, the committing of modifications should be done atomically in order to guarantee file system consistency. The natural way to do atomic operations is through file-locking: to prevent access to all the file system objects that are to be modified by the committing process. We use Linux mandatory locks to achieve this. Immediately before the committing phase, a lock is applied to the list of to-be-committed files, so that other processes do not gain access to these files. Only when the committing is completely done, the locks on these files are released.

5. DISCUSSION

5.1 Implementing Restriction at System Call Layer

The actions of SEE processes are regulated by a policy enforcement engine that operates using *system call interposition*. This enforcement engine generally enforces the following policies in order to realize SEEs:

—*File accesses.* Ensure that SEE processes can access only the files within the IFS. Access to device special files are not allowed, except for harmless devices like `tty`'s and `/dev/null`.

—*Network access.* Network accesses for which an explicit (static) redirection has been set up are allowed. The redirection may be to another process that executes within a different SEE, or to an intelligent proxy for a network service. (Note that network file access operations do not fall in this category—they are treated as file operations.)

—*Interprocess communication (IPC).* IPCs are not allowed to prevent an SEE process from affecting host processes.

—*Signals and process control.* A number of operations related to process control, such as sending of signals, are restricted so that a process inside an SEE cannot interfere with the operation of outside processes.

—*Miscellaneous safe operations.* Most system calls that query system state (timers and clocks, file system statistics, memory usage, etc.) are permitted within the SEE. In addition, operations that modify process-specific resources such as timers are also permitted.

—*Privileged operations.* A number of privileged operations, such as mounting file systems, changing process scheduling algorithms, setting system time, and loading/unloading modules are not permitted within SEE.

Note that the exact set of rules mentioned above may not suit all applications. For instance, one may want to disallow all network accesses for an untrusted application, but may be willing to allow some accesses (e.g, DNS and WWW) for applications that are more trusted. To support such customization, we use a high-level, expressive policy specification language called BMSL [Sekar and Uppuluri 1999; Uppuluri 2003] in our implementation. This language enables convenient specification of policies that can be based on system call names as well as arguments. The kinds of policies that can be expressed include simple access control policies, as well as policies that depend on history of past accesses and/or resource usage. In addition, the language allows response actions to be launched when policies are violated. For instance, it can be specified that if a process tries to open a file f, then the request should be redirected to open another file f'. Efficient enforcement engines are generated by a compiler from these policy specifications. More details about this language and its compiler can be found in Uppuluri [2003].

In our experience, we have been able to specify and enforce policies that allow a range of applications to function without raising exceptions, and the experimentation section describes some of our experiences in this regard.

5.2 Support for Network Operations

Support for network access can be provided while ensuring one-way isolation semantics in the following cases:

—Access to services that only provide query (and no update) functionality, for example, access to domain name service and informational Web sites, can be permitted by configuring the enforcement engine so that it permits access to certain network ports on certain hosts.

—Communication with processes running within other SEEs can be supported by redirecting network accesses appropriately. This function is also provided by the enforcement engine.

—Accesses to any service can be allowed, if the access is made through an intelligent proxy that can provide isolation semantics.

Currently, our implementation supports the first two cases. Use of distributed SEEs provides an easy way to permit isolated process to access any local server—one can simply run the server in isolation, and redirect accesses by the isolated process to this isolated server. However, for servers that operate in a different administrative domain, or servers that in turn access several other network functions, running the server in isolation may not always be possible. In such cases, use of an intelligent proxy that partially emulates the server function may be appropriate.

Intelligent proxies may function in two ways. First, they may utilize service-specific knowledge in filtering requests to ensure that only read operations are passed on to a server. Second, they may provide some level of support for write operations, while containing the effects within themselves, and propagating the results to the real server only at the point of commit. For instance, an e-mail proxy may be implemented which simply accepts e-mail for delivery, but does not actually deliver them until commit time. Naturally, such an approach won't work in the case when a response to an e-mail is expected.

Another limitation of our current implementation is that it does not provide support for atomic commits across distributed SEEs.

5.3 User Interface

Typically, an SEE is created with an interactive shell running inside it. This shell is used by the user to carry out the tasks that he/she wishes to do inside the SEE. At this point, the user can use arbitrary helper applications to analyze, compare, or check the validity of the results of these tasks. For instance, if the application modifies just text files, utilities like `diff` can point out the differences between the old and new versions. If documents, images, video or audio files are modified, then corresponding document or multimedia viewers may be used. More generally, users can employ the full range of file and multimedia utilities or customized applications that they use everyday to examine the results of SEE execution and decide whether to commit.

Before the user makes a final decision on committing, a compact summary of files modified within the SEE is provided to the user. If the user does not accept the changes, she can just roll them back at a click of button. If she accepts the changes, then the commit criteria is checked. If it is satisfied, then the commit operation proceeds as described earlier. If not, the user may still decide to proceed to commit, but this is supported only in certain cases. For instance, if the whole structure of the file system has been changed outside the SEE during its operation, there won't be a meaningful way to commit. For this reason, overriding of commit criteria is permitted only when the conflict involves a plain file.

Optionally, the user can use a shell that has access to the same isolation context as the untrusted process, and also has access to the original file system. Moreover, the children of this shell are permitted to access X-windows, so that arbitrary helper applications (e.g., image viewers) can be launched by the user to view the modified files.

5.4 Attacks on SEEs

5.4.1 *Attacks by modifying helper application input.* Recall that SEEs may be used to run untrusted and/or malicious software. In such cases, additional precautions need to be taken to ensure that this software does not interfere with the helper applications, subverting them into providing a view of system state that looks acceptable to the user. For instance, the untrusted process may interfere with the execution of the helper application. One way for the untrusted program to accomplish this is to insert an alias into the `.bashrc` or a similar shell startup file, and have the untrusted program execute its own version of the helper application (which presumably will present false results to the user). The above situation illustrates the need to ensure that untrusted processes cannot interfere with the operation of helper application processes, or modify the executables, libraries, or configuration files used by them. To ensure this, helper applications can be run outside of the SEE, but having a read-only access to the file system view within the IFS using a special path name. This approach ensures that the helper application gets its executable, libraries, and config files from the host file system which is unaffected. Another advantage of doing this is that any modifications to the system state made by helper applications do not clutter the user interface that reports file modifications that were carried out within the SEE. (While it may seem that helper applications are unlikely to modify files, this is not true. For instance, running the bash shell causes it to update the `.bash_history` file; running a browser updates its history and cache files; and so on.)

5.4.2 *Attacks on system call interception.* System call interception techniques can be points of targets of subversion due to some of the pitfalls [Garfinkel 2003] in implementation. The user-level interposition approach is more vulnerable to attacks through race conditions, which are addressed as follows.

—Rogue processes may cause the interceptor to terminate. A malicious process may try to terminate the process that is monitoring it. For instance, it can send a kill signal to the monitoring process. However, this must again be done through a system call, which will be intercepted and aborted by the monitoring process.

—Fork/clone race condition. When a monitored process executes a `fork` system call, the child process is not traced automatically. The monitoring process must explicitly request tracing of the child process by invoking `ptrace` with the child PID (process identifier) as an argument. However, the child PID is unavailable until the `fork` system call returns to the parent. By then, it is possible that the child process may have started running, and executed system calls that the monitoring process would not permit. To solve this problem we adopt a clever trick that was originally devised in the `strace`

[Strace] program. A description of this idea can be found in Liang et al. [2003].

—Argument race condition. There is a delay between the time when the arguments of a system call is checked by the monitoring process and the time when the arguments are actually read by the kernel. If the arguments are stored in a memory region shared by several processes or threads, it is possible for these processes/threads to modify the arguments during that time delay. We address this problem by moving security-critical arguments to a random location on the stack [Jain and Sekar 2000]. In order for the attack to succeed in spite of this change, collaborating threads (or processes) need to scan the entire stack to find the location where the argument is stored, and this scan must be completed within the short interval between the time when arguments are checked by the monitoring process and the time they are used by the kernel. If the random number is chosen over a reasonably large range, for example, 10^7 or 10^8, then the likelihood of successful attacks becomes very small.

A completely in-kernel based approach does prevent some of these vulnerabilities from arising in the first place (such as argument copying related race conditions), and that is being used in our kernel land approach.

5.4.3 *Attacks through resource exhaustion.* Another point of attack may be through exhaustion of resources used by the SEE. For instance, SEEs make use of temporary storage to save the modified/created version of files, directories, etc. Since this temporary storage is itself a part of the main file system, there is a potential chance for attacks to intentionally exhaust the disk space resources on a system. In general, such resource exhaustion attacks are usually dealt with resource usage control or resource accounting. In the particular instance of the above attack, a quota can be allocated for temporary storage and whenever disk space overuse occurs, the user will be issued a warning. Our policy specification language [Sekar and Uppuluri 1999; Uppuluri 2003] is capable of specifying such resource usage policies.

5.4.4 *Attacks through kernel vulnerability.* Alcatraz relies on the underlying operating system to serve redirected requests, and assumes its interface to the operating system is robust. If the operating system kernel that Alcatraz runs on has a vulnerability, a malicious program can exploit it to escape the SEE environment. This is true for all isolation approaches, which assume their lower layer services to be robust. If the code implementing devices of virtual machines contains vulnerabilities, they can be exploited to escape the isolation environment [Ormandy]. Similarly, when using our approach, no guarantees about integrity can be made when the lower layer is already compromised [King et al. 2006]. In this case, the host system cannot detect malicious actions of the layer beneath it. Therefore, the host system running Alcatraz SEE relies on a clean lower layer kernel with the latest patches that address known vulnerabilities.

5.4.5 *Detecting SEE Environment.* Our SEE is not designed to be undetectable, that is, it is possible for an untrusted program to detect that it is running in an SEE. However, this doesn't affect our goal of protecting system integrity. If a malicious program detects the SEE and don't show its malicious behavior, it cannot harm the host system even after its results are committed. However, users should never trust a program based on its behavior in an SEE, which is not designed to certify untrusted programs. An untrusted program should never be executed outside an SEE.

6. EVALUATION

6.1 Implementation and Evaluation Environments

The user-land version of Alcatraz was implemented on the Linux operating system [Alcatraz]. The implementation has been tested on Red Hat Linux 7.2 and Red Hat Linux 8.0 distributions. The performance figures given below were obtained on a PC running Red Hat Linux 7.2 on a 1.7GHz P4 processor with 1GB memory.

The in-kernel version of Alcatraz was implemented in the Linux operating system kernel version 2.4.18-3. Performance results reported in this paper were obtained from a laptop running Red Hat Linux 7.3 with a 1.0GHz AMD Athlon4 processor, 512MB memory and a 20GB, 4200rpm IDE hard disk.

6.2 Evaluation of Functionality

6.2.1 *Untrusted applications.* We describe two applications here: a file renaming utility freeware called `rta` [Tiilikainen], which traverses a directory tree and renames a large number of files based on rules specified on the command line, and a photo album organizer freeware called `picturepages` [Picturepages]. These applications ran successfully within our SEE. Our implementation includes a GUI that summarizes files modified in the SEE so as to simplify user's task of deciding whether the changes made by the application are acceptable. Using this GUI, we checked that the modifications made by these applications were as intended: renaming of many files, and creation of several files and/or directories. We were then able to commit the results successfully.

To simulate the possibility that these programs could be malicious, we inserted an attack into `picturepages` that causes it to append a new public key to `.ssh/authorized_keys`. (This attack would enable the author of the code to later log into the system on which `picturepages` was run.) Using our GUI, it was easy to spot the change to this file. The run was aborted, leaving the file system in its original state.

Our user-level implementation was tested with Picturepages and we observed similar results. Another application that we tested was `mpls`, which takes a list of mp3 files and creates a playlist sorted by artist, album, track, or title on the standard output. A directory containing various mp3 files was used as the input. After the program finished execution, the user-interface presented a report that summarized that no changes were made to the file system.

6.2.2 *Malicious code.* Email attachments and WWW links are a common source of viruses and other malware. We used an SEE to protect systems from such malware. Specifically, we modified the MIME type handler configuration file used by Mozilla so that executables, as well as viewers launched to process documents (e.g., `ghostscript` and `xpdf`) fetched over the Internet, were run within SEE. We fetched sample malicious PostScript and Perl code over the network using this approach. This code was executed inside the SEE. Using our GUI, we were able to see that these programs were performing unexpected actions, for example, creating a huge file in the user's home directory. These actions were aborted. Also, recently, there are several image flaw exploits (JPEG virus) that have captured the attention of many researchers. Running such image viewers inside an SEE will help eliminate this potential danger, because any malicious activity from the exploits will be isolated from affecting the main system.

Some kinds of malicious code are written to recognize typical sandbox environments, and if so, not display their malicious behavior. This can cause a user to develop trust in the code and then execute it outside of sandbox, when the malcode will deliver its payload. With our approach, we point out that running the code inside SEE does not incur significant inconvenience for the user, thereby making it easy for the user to always use it. In this case, the code will always display benign behavior.

6.2.3 *Software installation.* Another experiment performed a trial installation of Mozilla browser. During the installation, an incorrect directory name `/usr/bin` was chosen as the location for installation, instead of the default directory `/usr/local/mozilla`. Under normal circumstances, this causes Mozilla to copy a number of files into `/usr/bin`, thereby polluting the directory. After running the program in an SEE, the user interface indicated that a large number of files (some are nonexecutables) were added to `/usr/bin`, which was not desirable. Aborting this installation, we ran the installation program a second time, this time with `/usr/local/mozilla` as the location for installation. At the end of installation, we restarted the browser, and visited several sites to make sure that the program worked as expected. (For this experiment, the system call restriction layer was modified to allow all WWW accesses.) Finally, we committed the installation, and from that point on, we were able to use the new installation of the browser successfully, outside of SEE.

We also tested the user-land implementation with the same browser installation. The program modified three configuration files of a previous version of Mozilla and installed all files into a new directory. All these changes were captured by our tool and reported through the user interface.

6.2.4 *Upgrading and testing a server.* Specifically, we wanted to upgrade our Web server so that it can support SSL. We started a command shell under SEE, and used it to upgrade the apache software installation. We then ran the new server. To enable it to run, we used static redirection for network operations, so that a bind operation to port 80 was redirected to port 3080. We then ran a browser that accessed this server by connecting to this port. We verified that the new server worked correctly. Meanwhile, the original server was still accessible to everyone. Thus, SEE allowed the software upgrade to be tested easily and conveniently, without having to shutdown the original server.

After verifying the operation of the new server, we attempted to commit the results. Unfortunately, this produced conflicts on some files such as the access and error log files used by the server. We chose to ignore updates to such output files that were made within the SEE, and commit only the rest of the files, which could be done successfully.

In all examples in the above categories, the isolation operation guaranteed the safety of the user's resources, as well as provided the convenience of concise summaries on the outputs of these executions.

6.3 Implementation Performance Results

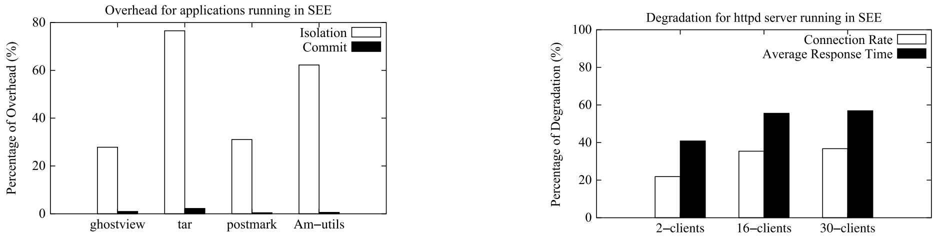
In the results reported below, the primary metric was elapsed time.

For the user-land and in-kernel system performance evaluations, we considered the following common classes of examples:

—*Utility programs*. In this category, we studied `ghostview` and `tar` utilities. Specifically, we ran ghostview on a 31M file, with no file modification operations; and `tar` to generate a tarball from a 26M directory, and the only modification operations involved was the creation of this archive. From Figure 5, we can see a 3-12 percent overhead incurred for such applications for in-kernel implementation, while higher overhead (30-80 percent) for user-land implementaion from Figure 4.

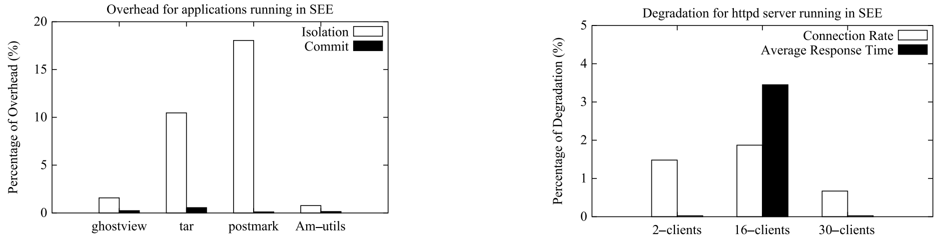
—*Servers*. We measured the performance overhead for the two implementations on the Apache Web server using WebStone [Webstone], a standard Web server benchmark. We used version 2.5 of this benchmark, and ran it on a separate computer that is connected to the server through a 100Mbps network. We ran the benchmark with two, 16 and 30 clients. In the experiments, the clients were simulated to access the Web server concurrently. They randomly fetch html files whose size is from 500 bytes to 5M. The benchmark was run for a duration of 30 minutes, and the results were averaged across ten such runs. The results are shown in Figure 4 and Figure 5 for two implementations. On average, in-kernel implementation incurred a two percent degradation and the degradation for user-land implementation is around 40 percent.

—*File system benchmarks*. We used `Postmark` [Katcher 1997] and `Am-Utils` [Pendry et al.] benchmarks to get the benchmark data for IFS. Postmark is a file system benchmark to measure the performance for file system used by Internet applications, such as e-mail. In this experiment, we configured `Postmark` to create 500 files in a file pool, with file sizes ranging from 500 bytes to 500KB. A total of 2,000 file system operations were performed. In total, 1,515 files were created, 1,010 files read, 990 file written, and 1,515 files deleted. The tests were repeated 10 times. Overall, an 18 percent



(a) Utility applications and filesystem benchmarks. (b) Apache httpd server.

Fig. 4. Performance results for user-land implementation.



(a) Utility applications and filesystem benchmarks. (b) Apache httpd server.

Fig. 5. Performance results for in-kernel implementation.

	Log-Based Commit	State-Based Commit	
	Time	Time	Speedup
ghostview	0.03	0.03	1
tar	0.14	0.03	4.7
postmark	225	0.07	3214.3
Am-utils	16.9	0.35	48.3

Fig. 6. Comparison for log-based commit and state-based commit. All numbers are in seconds.

performance degradation is observed for in-kernel implementation, while 34 percent degradation on user-land implementation, and commit overheads for both are near zero. `Am-Utils` is a CPU-intensive benchmark result by building the Am-Utils package, which contains 7.6M lines of C code and scripts. The building process creates 152 files and 19 directories, as well as six rename and eight setattr operations. We ran this experiment in both original file system and IFS. The results, shown in Figure 4 and Figure 5, indicate a low isolation overhead of under two percent for in-kernel implementation and around 60 percent overhead for the user-land counterpart, and they both incurred a negligible commit overhead.

In addition, we also collected results in Figure 6 to show the efficiency of our state-based commit approach. An implementation that used log based committing was compared with our state based committing implementation, and the performance of both of the approaches were compared for applications such as `tar`, `postmark` and `Am-utils`. The results project the advantage of using a state based commit approach, particularly illustrating the advantage of having accumulative effects for file objects. For instance, the large number of temporary files created then deleted in Am-utils compilation and all the files created then deleted in Postmark execution, are not considered in the committing stage as candidates, while log-based commit still needs to perform the whole set of operations (e.g. write) to all these files, so there is a significant difference between the two approaches in terms of commit time.

7. RELATED WORK

This article is a combined and revised version of citations [Liang et al. 2003], which described the user-land approach, and Sun et al. [2005], which describes a kernel-land approach. In addition to revisions to these articles, Section 4 provided a significantly more refined and detailed treatment of commit criteria. Additional experimental study has been performed, and the results were included in this article.

7.1 Sandboxing Approaches

Sandboxing based approaches [Goldberg et al. 1996; Dan et al. 1997; Acharya and Raje 2000; Prevelakis and Spinellis 2001; Scott and Davidson 2002; Provos 2003] involve observing a program's behavior and blocking actions that may compromise the system's security. Safe Virtual Execution (SVE) [Scott and Davidson 2002] uses Software Dynamic Translation, a technique for modifying binaries as they execute, is used to implement sandboxing. Systrace [Provos 2003] is a sandboxing system that notifies the user about all system calls that an application tries to execute and then uses the response from the user to generate a policy for the application. SoftwarePot [Kato and Oyama 2003] incorporates a secure software circulation model that confines the behavior of the untrusted program. In this case, the software to be run is encapsulated with a file system. The user must encapsulate the complete list of the file system resources needed by the program in order to make it execute successfully. Furthermore, all the operations to the files are confined to the pot archive.

The main drawback of sandboxing based approaches is the difficulty of policy selection, that is, determining what actions are permissible for a given piece of software. Note that malicious behavior may not only involve accessing unauthorized resources, but also accessing authorized resources in unauthorized ways. For instance, a program that creates a compressed version of a file may instead create a file that contains no useful data, which is equivalent to deleting the original file. It is unlikely that a practical system can be developed that can allow users to conveniently state policies that allow write access to the file while ensuring that the file is replaced with its compressed version. In contrast, an SEE permits manual inspection, aided by helper applications, to be used to determine if a program behaved as expected. This approach is much more flexible. Indeed, it is hard to imagine that tasks such as verifying whether a software package has been installed properly can even be formally specified using any sandbox-type policy.

Another technique is to extend sandboxing by allowing operations to be disallowed silently, that is, by returning a success code to the program [Sekar et al. 1998; Fakebust]. The goal here is deception, that is, making a malicious program believe that it is succeeding in its actions so as to observe its behavior. In our terminology, these approaches use restriction rather than redirection. As we observed earlier, use of restriction is likely to break many benign applications, as well as alert malicious applications very quickly to the effect that their actions are not succeeding. For instance, if a write operation is silently suppressed, the application can easily detect this by reading back the contents.

7.2 Two-Way Isolation Approaches

Several approaches including those that involve virtual machines employ the idea of two-way isolation for security. The playground approaches developed for Java programs in [Malkhi and Reiter 2000; Chiueh et al. 2000] realize two way isolation by running untrusted programs on a physically isolated system, while their display is redirected to the user's desktop. An important point to note here is that the file system on the user's computer cannot directly be accessed on the playground system. Covirt [Chen and Nobl 2001], Denali [Whitaker et al. 2002] and Bochs [Bochs] are research efforts that support running applications inside two-way isolated virtual machines. Commercial virtual machine software such as VMWare [VMWare] and VirtualPC [VirtualPC] are convenient mechanisms to realize two-way isolation.

Two-way isolation systems suffer from three main problems in realizing a SEE:

—*Accurate environment reproduction.* For applications like system reconfiguration testing, it is necessary to duplicate the exact host environment inside a VM. This introduces problems in usability and performance. Although recent tools, such as VMware Converter [VMware], have been created to help user to duplicate host system into a VM, the task still takes a nontrivial amount of time for duplication, in addition to the time required to boot the duplicated VM for use. Furthermore, to keep the software in the VM system in sync with the host system, this step needs to be done for each change (such as software updates) in the host environment.

—*Difficulty in examining guest OS state.* Furthermore, examining changes made in a VM is not straightforward: on the one hand, it is not reliable to inspect the changes from inside of a VM because the VM may already be compromised. The other option is to mount the file system from the VM in a trusted system and examine it. However, this process of examining guest OS state from outside will display the changes made by all processes running in the system (including system processes), not just the untrusted process, and therefore may result in loss of precision in identifying changes made by a piece of untrusted code.

—*Difficulty in dealing with external file systems.* External file systems (such as a user's home directory on a NFS mount volume) cannot be used in conjunction with a two-way isolation approach. If used, changes made to an external file system will escape the boundary of the two-way isolation approach. Therefore the approach taken by these systems is to disallow use of external file systems. However, a one-way isolation approach is extremely useful here – changes made to files in an external file system can be examined (and committed) with ease.

7.3 Application Virtualization Approaches

Zap [Osman et al. 2002] creates a virtualization layer between processes and their operating system to support process migration. File system virtualization in Zap is similar to IFS in SEE in that both intercept system calls and map logical path names to physical ones. In general, Zap is more comprehensive than our SEE in that it virtualizes a broader range of system resources. However, SEE and Zap have different focus on supporting file system operations. SEE prevents effects of SEE processes from being visible to other processes, and allows modifications in SEE to be committed to host system. In contrast, Zap's file system virtualization aims to provide the same file system view and state on a different physical machine. It is implemented by static redirection and doesn't prevent Zap session and the host system (or two Zap sessions) from modifying the same file or directory.

There is also a recent line of products using software virtualization to reduce conflicts among installed software, such as Altiris's Software Virtualization Solution [SVS] and Microsoft's SoftGrid [SoftGrid]. The focus of these approaches is pure isolation. Also, the documentation on these tools do not suggest that these approaches allow users to commit changes back to the host system.

7.4 One-Way Isolation Approaches

Liu et al. [2000] presented a systematic development of the concept of one-way isolation as an effective means to isolate the effects of running processes from the point they are compromised. They developed protocols for realizing one-way isolation in the context of databases and file systems. However, they do not present an implementation of their approach. As a result, they do not consider the research challenges that arise due to the nature of COTS applications and commodity OSes. Moreover, they do not provide a systematic treatment of issues related to consistency of committed results. FVM [Yu et al. 2006] virtualizes resources of Windows operating system. The file system resources are virtualized using copy-on-write, similar to our file system proxy. But FVM aims to duplicate resources without affecting the host operating system, instead of containing all effects of a program. For example, network operations are redirected without confinement. In contrast, our SEE ensures that the actions of the isolated program cannot affect local or remote systems.

7.5 Recovery-Oriented Systems

The Recovery-Oriented Computing (ROC) project [ROC] is developing techniques for fast recovery from failures, focusing on failures due to operator errors. Brown and Patterson [2003] presents an approach that assists recovery from operator errors in administering a network server, with the specific example of an e-mail server. The recovery capabilities provided by their approach are more general than those provided by ours. The price to be paid for achieving more general recovery capabilities is that their implementation needs to be application specific, and hence will have to be tailored for each specific application/service. In contrast, we provide an application-independent approach. Another important distinction is that with our approach, consistency of system state can be assured whenever the commit proceeds successfully. With the ROC approach, which does not restrict network operations, there is no way to prevent the effects of network operations from becoming so widely distributed in the network that they cannot be fully reversed. In the case of e-mail service, they allow a certain level of inconsistency, for example, redelivering an e-mail that was previously read and deleted by a client, and expect the user to manually resolve this inconsistency. This potential for inconsistency is traded in favor of eliminating the risk of commit failures.

7.6 File System Approaches

Elephant file system [Santry et al. 1999] is equipped with file object versioning support, and supports flexible versioning policies. Several other approaches [Chutani et al. 1992; Quinlan and Dorward 2002; Roome 1991; Soules et al. 2002; Peterson and Burns 2003] use check pointing technique to provide data versioning. Muniswamy-Reddy et al. [2004] implements VersionFS, a versatile versioning file system. They use a stackable template file system as ours, and use a sparse file technique to reduce storage requirements for storing versions of large files. While all of these approaches provide the basic capability to rollback system state to a previous time, such a rollback will discard all changes made since that time, regardless of whether they were done by a malicious or benign process. In contrast, the one-way isolation approach implemented in this article guarantees selective rollback of the actions of processes run within the SEE without losing the changes made by benign processes executing outside of the SEE.

Repairable File System [Zhu and Chiueh 2003; Zhu 2003] makes use of versioning file system to bring repair facility to a compromised file server. The Taser intrusion recovery system [Goel et al. 2005] also has a similar objective and uses audit analysis techniques for recovery of a filesystem that is damaged due to an intrusion. Fastrek [Pilania and Chiueh 2003] applies the similar approach to protect databases. These approaches can attribute changes to malicious or benign process executions, and allow a user to rollback changes selectively. However, since the changes made by (potentially) compromised processes are not contained within any environment, cascading aborts can become a problem. Specifically, a benign process may access the data produced by a compromised process, in which case the actions of the benign process may have to be rolled back, as well as the actions of processes that used the results of such a benign process and so on. The risk of such cascaded aborts should be weighed against the risk of not being able to commit in our approach. Thus, these approaches as well as the ROC approach mentioned above are more suitable when the likelihood of rollbacks is low, and commit failures cannot be tolerated.

Loopback file system [Lofs] can create a virtual file system from existing file system and allow access to existing files using alternative path name. But this approach provides no support for versioning or isolation.

3D file system [Korn and Krell 1990] provides a convenient way for software developers to work with different versions of a software package. In this sense, it is like a versioning file system. It also introduces a technique called *transparent viewpathing*, which is based on translating file names used by a process. It gives a union view of several directory structures thus allowing an application to transparently access one directory through another's path. As it is not designed to deal with untrusted applications, it needs the cooperation from the application for this mechanism to work. TFS [TFS] is a file system in earlier distributions of Sun's operating system (SunOS), which allowed mounting of a writable file system on top of a read-only file system. TFS also has a view similar to 3DFS, where the modifiable layer sits on top of the read only layers. Pendry and McKusick [1995] describes a union file system for BSD, that allows merging of several directories into one, with the mounted file system hiding the contents of the original directories. The union mount will show the merger of the directories and only the upper layer can be modified. All these file systems are intended for software development, with the UnionFS providing additional facilities for patching read only systems. However, they do not address the problem of securing the original file system from untrusted/faulty programs; nor do they consider problems such as data consistency and commit criteria.

8. SUMMARY

In this article, we presented an approach and tool called Alcatraz for realizing safe execution environments. We showed that the approach is versatile enough to support a wide range of applications. A key benefit of our approach is that it provides strong consistency. In particular, if the results of isolated execution are not acceptable to a user, then the resulting system state is as if the execution never took place. On the other hand, if the results are accepted, then the user is guaranteed that the effect of isolated execution will be identical to that of atomically executing the same program at the point of commit. We also discussed alternative commit criteria that exploit file semantics to reduce commit failures.

We presented two different implementations of Alcatraz: a *user-land* solution and the other employing an in-kernel approach. The user-land approach is a trade-off of stronger commit guarantees and performance for the sake of usability, and is implemented as a tool that can be used by desktop users without requiring administrator rights. Our in-kernel approach makes minimal modifications to the kernel in the form of modules that provide file system isolation and policy enforcement. It requires no changes to applications themselves. Our functional evaluation illustrates the usefulness of the approach, while the performance evaluation shows that the approach is efficient, and incurs overheads typically less than 10 percent for kernel implementation.

REFERENCES

ACHARYA, A. AND RAJE, M. 2000. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of the USENIX Security Symposium (SECURITY'00)*.

ALCATRAZ. http://www.seclab.cs.sunysb.edu/alcatraz.

BOCHS. http://bochs.sourceforge.net.

BROWN, A. AND PATTERSON, D. 2003. Undo for operators: Building an undoable e-mail store. In *Proceedings of the USENIX Annual Technical Conference (USENIX'03)*.

CHEN, P. M. AND NOBL, B. D. 2001. When virtual is better than real. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'01)*.

CHIUEH, T., SANKARAN, H., AND NEOGI, A. 2000. Spout: A transparent distributed execution engine for Java applets. In *Proceedings of the International Conference on Distributed Computing Systems (ICDCS'00)*.

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 (USENIX'92)*.

DAN, A., MOHINDRA, A., RAMASWAMI, R., AND SITARAM, D. 1997. Chakravyuha: A sandbox operating system for the controlled execution of alien code. Tech. rep., IBM T.J. Watson Research Center.

FAKEBUST. Fakebust, a malicious code analyzer.
http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2004-09/0251.html.

GARFINKEL, T. 2003. Traps and pitfalls: Practical problems in system call interposition based security tools. In *Proceedings of the Annual Network & Distributed Systems Security Conference (NDSS'03)*.

GOEL, A., PO, K., FARHADI, K., LI, Z., AND DE LARA, E. 2005. The taser intrusion recovery system. *SIGOPS Oper. Syst. Rev. 39*, 5, 163–176.

GOLDBERG, I., WAGNER, D., THOMAS, R., AND BREWER, E. A. 1996. A secure environment for untrusted helper applications: Confining the wily hacker. In *Proceedings of the USENIX Security Symposium (SECURITY'96)*.

JAIN, K. AND SEKAR, R. 2000. User-level infrastructure for system call interposition: A platform for intrusion detection and confinement. In *Proceedings of the Annual Network and Distributed System Security (NDSS'00)*.

JAJODIA, S., LIU, P., AND MCCOLLUM, C. D. 1998. Application-level isolation to cope with malicious database users. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'98)*.

KATCHER, J. 1997. Postmark: A new file system benchmark. Tech. rep. TR3022, Network Applicance Inc.

KATO, K. AND OYAMA, Y. 2003. Softwarepot: An encapsulated transferable file system for secure software circulation. In *Proceedings of the International Symposium on Software Security (ISSS'03)*.

KING, S. T., CHEN, P. M., WANG, Y.-M., VERBOWSKI, C., WANG, H. J., AND LORCH, J. R. 2006. Subvirt: Implementing malware with virtual machines. In *Proceedings of the IEEE Symposium on Security and Privacy (SP'06)*.

KORN, D. G. AND KRELL, E. 1990. A new dimension for the UNIX file system. *Software: Practice & Experience 20*, S1.

LAMPSON, B. W. 1973. A note on the confinement problem. *Comm. ACM 16*, 10, 613–615.

LIANG, Z., VENKATAKRISHNAN, V., AND SEKAR, R. 2003. Isolated program execution: An application transparent approach for executing untrusted programs. In *Proceedings of the Annual Computer Security Applications Conference (ACSAC'03)*.

LIU, P., JAJODIA, S., AND MCCOLLUM, C. D. 2000. Intrusion confinement by isolation in information systems. *J. Comput. Secur. 8*.

LOFS. Loop back file system. Unix man page.

MALKHI, D. AND REITER, M. K. 2000. Secure execution of Java applets using a remote playground. *Softw. Engin. 26*, 12.

MUNISWAMY-REDDY, K.-K., WRIGHT, C. P., HIMMER, A. P., AND ZADOK, E. 2004. A versatile and user-oriented versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'04)*.

ORMANDY, T. An empirical study into the security exposure to hosts of hostile virtualized environments. http://taviso.decsystem.org/virtsec.pdf.

OSMAN, S., SUBHRAVETI, D., SU, G., AND NIEH, J. 2002. The design and implementation of zap: A system for migrating computing environments. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation (OSDI'02)*.

PENDRY, J.-S. AND MCKUSICK, M. K. 1995. Union mounts in 4.4bsd-lite. In *Proceedings of the USENIX Technical Conference on UNIX and Advanced Computing Systems (ACS'95)*.

PENDRY, J. S., WILLIAMS, N., AND ZADOK, E. Am-utils user manual, 6.1b3 edition, July 2003. http://www.am-utils.org.

PETERSON, Z. AND BURNS, R. 2003. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system. Tech. rep. HSSL-2003-03, Hopkins Storage Systems Lab, Department of Computer Science, Johns Hopkins University.

PICTUREPAGES. Picturepages software. http://www.canonical.org/picturepages.

PILANIA, D. AND CHIUEH, T. 2003. Design, implementation, and evaluation of an intrusion resilient database system. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*.

PREVELAKIS, V. AND SPINELLIS, D. 2001. Sandboxing applications. In *Proceedings of the USENIX Annual Technical Conference: FREENIX Track (USENIX'01)*.

PROVOS, N. 2003. Improving host security with system call policies. In *Proceedings of the 11th USENIX Security Symposium (SECURITY'03)*. 257–272.

QUINLAN, S. AND DORWARD, S. 2002. Venti: A new approach to archival storage. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'02)*. Monterey, CA.

ROC. Recovery-oriented computing. http://roc.cs.berkeley.edu.

ROOME, W. D. 1991. 3DFS: A time-oriented file server. In *Proceedings of the USENIX Technical Conference (USENIX'91)*.

SANTRY, D. J., FEELEY, M. J., HUTCHINSON, N. C., AND VEITCH, A. C. 1999. Elephant: The file system that never forgets. In *Proceedings of the Workshop on Hot Topics in Operating Systems (HotOS'99)*.

SCOTT, K. AND DAVIDSON, J. 2002. Safe virtual execution using software dynamic translation. In *Proceedings of the Annual Computer Security Applications Conference (CSAC'02)*.

SEKAR, R., CAI, Y., AND SEGAL, M. 1998. A specification-based approach for building survivable systems. In *Proceedings of the National Information Systems Security Conference (NISSC'98)*.

SEKAR, R. AND UPPULURI, P. 1999. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of the USENIX Security Symposium (SECURITY'99)*.

SOFTGRID. http://www.microsoft.com/systemcenter/softgrid/default.mspx.

SOULES, C., GOODSON, G., STRUNK, J., AND GANGER, G. 2002. Metadata efficiency in a comprehensive versioning file system. In *Proceedings of the USENIX Conference on File and Storage Technologies (FAST'02)*.

STRACE. http://www.liacs.nl/∼wichert/strace.

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 ISOC Network and Distributed Systems Symposium (NDSS'05)*.

SVS. Software virtualization solution.
http://www.altiris.com/Products/SoftwareVirtualizationSolution.aspx.

TFS. Translucent file system. SunOS Reference Manual, Sun Microsystems.

TIILIKAINEN, T. Rename-them-all, linux freeware version.
http://linux.iconet.com.br/system/preview/8622.html.

UPPULURI, P. 2003. Intrusion detection/prevention using behavior specifications. Ph.D. thesis, Stony Brook University.

VIRTUALPC. http://www.microsoft.com/windows/products/winfamily/virtualpc/default.mspx.

VMWARE. http://www.vmware.com.

VMWARE. VMware Converter. http://www.vmware.com/products/converter.

WEBSTONE. http://www.mindcraft.com/webstone.

WHITAKER, A., SHAW, M., AND GRIBBLE, S. 2002. Denali: Lightweight virtual machines for distributed and networked applications. In *Proceedings of the USENIX Annual Technical Conference (USENIX'02)*.

YU, Y., GUO, F., NANDA, S., CHUNG LAM, L., AND CKER CHIUEH, T. 2006. A feather-weight virtual machine for windows applications. In *Proceedings of the 2nd ACM/USENIX Conference on Virtual Execution Environments (VEE'06)*.

ZADOK, E., BADULESCU, I., AND SHENDER, A. 1999. Extending file systems using stackable templates. In *Proceedings of the USENIX Annual Technical Conference (UBENIX'99)*.

ZHU, N. 2003. Data versioning systems. Tech. rep., Stony Brook University.

ZHU, N. AND CHIUEH, T. 2003. Design, implementation, and evaluation of repairable file service. In *Proceedings of the International Conference on Dependable Systems and Networks (DSN'03)*.

Received March 2006; revised August 2007; accepted May 2008