One-way Isolation: An Effective Approach for Realizing Safe Execution Environments*

Weiqing Sun Zhenkai Liang R. Sekar
Department of Computer Science,
Stony Brook University.
{wsun,zliang,sekar}@cs.sunysb.edu

V.N. Venkatakrishnan†
Department of Computer Science,
University of Illinois at Chicago.
venkat@cs.uic.edu

Abstract

In this paper, 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. Our SEE supports 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 the user does not accept these changes, 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 also develop an efficient technique for implementing the commit operation. Our implementation results show that most software, including fairly complex server and client applications, can run successfully within the SEE. The approach introduces low performance overheads, typically below 10%.*

*This research is supported in part by an ONR grant N000140110967 and an NSF grant CCR-0208877.

†Research conducted when the author was a graduate student in the Department of Computer Science, Stony Brook University.

1. Introduction

1.1. Motivating Applications

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 execute 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 a system is susceptible to the latest email virus, worm 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 are 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 shutdown 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.

1.2. SEE Requirements and the Need for New Approach

In order to support the kinds of applications mentioned above, an SEE must provide the following features:

- *Confinement without undue restrictions on functionality.* The effects of process execution within an SEE should not "escape" the SEE and become visible to normal applications running outside. Otherwise, one cannot rule out the possibility of SEE processes altering the operation of other applications running on the same system or elsewhere in the network. Such confinement can be achieved using access control restrictions, e.g., by prohibiting all operations that modify files or access the network. However, such restrictions will prevent most applications from executing successfully within an SEE.
- *Accurate environment reproduction.* For SEEs to be useful in the above applications, it is essential that the behavior of applications be identical, whether or not they operate within the SEE. 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.* In many of the above applications, including untrusted software execution and software or system updates, a user would like to retain the results of activities that were successful. Thus, the SEE must provide a mechanism to "commit" the results of activities that took place within it. 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 achieve confinement, but do so by severely restricting functionality. Virtual machines (VMs) and related approaches [3, 36] relax access restrictions, but do not offer any support for environment reproduction or committing. File versioning systems [26, 40, 39, 16, 5, 23, 25, 31, 19] can provide rollback capabilities, but they don't provide a mechanism to discriminate among changes made by different processes, and hence cannot support selective rollback of the effects of untrusted process execution.

The concept of *isolation* has been proposed as a way to address the problem of effect containment for compromised processes in [8, 13, 28]. [13] developed the concept of *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). They also develop protocols for realizing one-way isolation in the context of databases and file systems. However, they only provide a high-level treatment, and do not address practical issues that arise in implementing such an approach for COTS applications running over commodity OSes.

In our previous work [12], we addressed some of these issues and developed a user-level tool for isolating the effects of COTS applications on the Linux OS. The focus of that effort was on untrusted software execution, and on a solution that was realized entirely at the user level. Such a solution does not require OS changes or even administrative privilege to install or use the tool. However, in order to achieve a completely user-land solution, [12] compromises on performance as well as generality. In particular, the approach suffers from high overheads that can be over 100% in some instances. Moreover, isolation semantics cannot be faithfully reproduced for operations that concern file meta-data such as permissions and ownership. For directories, isolation is achieved using an ad-hoc approach that is hard to implement and provides semantics that is inconsistent with that of files. Finally, no systematic solution to the commit problem is provided. The approach developed in this paper addresses all these drawbacks. Moreover, it generalizes the approach so that isolation can be provided for non-file operations, e.g., certain classes of network accesses.

1.3. Approach Overview

The SEEs described in this paper are 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 sense, the SEE processes can (and do) see the environment of their host system, and hence accurate reproduction of environment is assured. However, the effects of SEE processes are isolated from outside applications, thereby satisfying the confinement requirement.

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, a user will start a command shell within a new SEE, and use this shell to carry out tasks such as running untrusted programs. She may also run

helper applications, such as image or document viewers, or arbitrary utility applications to examine the resulting system state. Finally, if she wants to accept the changes made within the SEE, she 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.

Our approach is implemented using interposition at the system call and virtual file system layers, and hence does not require any changes to applications that run inside the SEE. Even complex tasks such as compilation and installation of large software packages, and execution of complex client and server applications can be carried out successfully within the SEE. This is because our approach places few restrictions on operations performed by most applications. In particular, no restrictions are placed on file accesses, except in the case of access to special devices. Network operations that correspond to "read" operations, such as querying a name server, can be permitted as well. Network accesses that correspond to "write" operations can be permitted when the target of the communication satisfies one of the following conditions:

- it is an application running within an SEE, possibly on a different host, or
- it is a special-purpose "proxy" that is layered between the application and the service accessed by it, and can buffer the write actions until commit time.

The key challenge in implementing such proxies is that even though they buffer certain operations, 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" operation, the proxy should return the result that would have been returned if the write operation had actually been carried out.

1.4. Paper Organization

The rest of this paper is organized as follows. Section 2 presents an overview of our approach. Section 3 describes our file system proxy, namely, the Isolation File System (IFS). Section 4 discusses the criteria and the procedure for committing changes made to the file system. Other aspects of our approach are discussed in Section 5. Section 6 provides an evaluation of the functionality as well as the performance of our approach. Related work is discussed in Section 7, followed by concluding remarks in Section 8.

2. Design of Secure Execution Environment

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

2.1. Achieving One-way Isolation

The primary goal of isolation in our approach is effect containment: preventing the effects of SEE processes from affecting the operation (or outcome) of processes executing outside the SEE[1]. This means that any "read" operation (i.e., one that queries the system state but does not modify it) may be performed by SEE processes. It also means that "write" operations should not be permitted to affect system state. There are two options in this context: one is to *restrict* the operation, i.e., disallow its execution. The second option is to *redirect* the operation to a different resource that is invisible outside the SEE. Once a write operation is redirected, it is important that subsequent read operations on the same resource be redirected as well.

By *restriction,* we mean that an operation is prevented from execution. An error code may be returned to the process, or the operation may be silently suppressed and a success code returned. In either case, restriction is easy to implement — we need only know the set of operations that can potentially alter system state. The main drawback of restriction is that it will likely prevent applications from executing successfully. For instance, if a program writes 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 non-trivial 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 operation that modifies some component of the host environment is instead redirected to a different component that is not accessed by the host OS processes. For instance, 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 operation redirected to this copy. Redirection is intended to provide a consistent view of system state to a process, thereby allowing it to run successfully.

Redirection can be *static* or *dynamic.* Static redirection requires the source and target objects to be specified manually. It is ideal for network operations. For instance, one may statically specify that operations to bind a socket to a port p should be redirected to an alternate port p'. Simi-

[1] Note that we are interested in confinement [11] from the point of view of system integrity, rather than confidentiality. As such, we do not deal with with issues such as covert channels.

larly, one may specify that operations 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, e.g., 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 operations on different file objects, e.g., an operation 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, won't work. 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. Specifically, in our SEE architecture, dynamic redirection is supported by *service-specific proxies.* Currently, there is a proxy for file service, and we envision proxies for other services such as WWW or email.

In our current implementation, system call interposition is used to implement restriction and static redirection. We restrict all modification operations other than those that involve the file system and the network. In the case of file operations, all accesses to normal files are permitted, but accesses to raw devices and special purpose operations 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, but our current implementation provides no mechanism to monitor and enforce policies on access to X-server.)

Dynamic redirection is currently supported in our implementation for only file system accesses. It is realized using a proxy called the Isolation File System (IFS), which is described in detail in Section 3.

2.2. Committing Changes

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. Below, we provide a high-level overview of the issues involved in commit.

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, e.g., 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 retry 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. Isolation File System (IFS)

3.1. High-Level Overview

In principle, a file system can be viewed as a tree structure. Internal nodes in this tree correspond to directories or files, whereas the leaves correspond to disk blocks holding file data. The children of directory nodes may themselves be directories or files. The children of file nodes will be disk blocks that either contain file data, or pointers to file data.

This view of file system as a tree suggests an intuitive way to realize one-way isolation semantics for an entire file system: when a node in the original file system is about to be modified, a copy of this node, as well as all its ancestors, is created in a "private" area of the file system called *temporary storage*. The write operation, as well as all other subsequent operations on this node, are then redirected to this copy.

In essence, we are realizing isolation using copy-on-write. Although the copy-on-write technique has been used extensively in the context of plain files, it has not been studied in the context of directories. Realizing IFS requires us to support copy-on-write for the entire file system, including directories and plain files.

In our approach, copy-on-write on directories is supported using a *shallow-copy* operation, i.e., the directory itself is copied, but its entries continue to point to objects in the original file system. 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 copied in their entirety.

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 [37]. 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.

3.2. Design Details

The description in the previous section 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 hardlinks. In addition, file systems contain other types of objects, including symbolic links and special device files. As mentioned earlier, IFS does not support 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. As mentioned earlier, the 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.

Figure 1 illustrates the operation of IFS. The bottom layer corresponds to a host OS file system. The middle layer is the temporary storage to hold modified copies of files and directories. The top layer shows the view within IFS, which is a combination of the views in the bottom two layers. Note that the ordering of the bottom two layers in the figure is significant: the view contained in the temporary storage overrides the view provided by the main file system.

The temporary storage area is also known as "private storage area" to signify that fact that it is not to be accessed by the host OS. In order to support efficient movement of files between the two layers, which is necessary to implement the commit operation efficiently, it is preferable that the temporary storage be located on the same file system as the bottom layer. (If this is not possible, then temporary storage can be on a different file system, with the caveat that committing will require file copy operations as opposed to renames.) Henceforth, we will use the term *main file system* to denote the bottom layer and *IFS-temporary storage* (or simply "temporary storage") to refer to the middle layer.

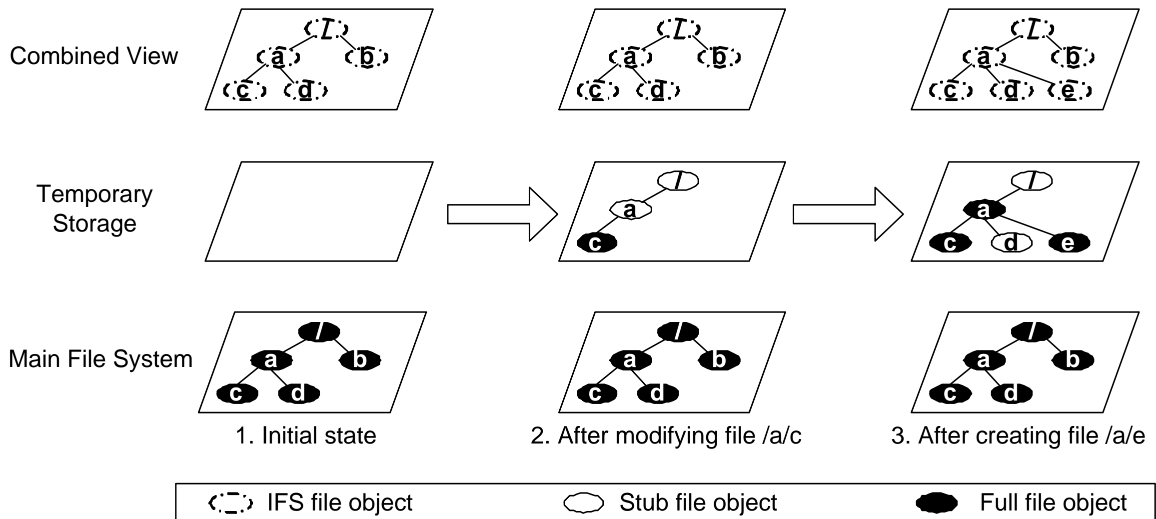


Figure 1. Illustration of IFS Layout on Modification Operations

In addition to storing private copies of files modified within the SEE in the temporary storage, the IFS layer also 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.

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.

We illustrate the operation of IFS using the example shown in Figure 1. 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` first creates a stub version of the ancestor directories, namely, `/` and `/a`. Then the file `/a/c` is copied from the main file system to the temporary storage. Subsequent accesses are redirected to this copy in temporary storage.

The third step of Figure 1 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, a shallow copy of the directory is made. Next, the file `e` is created within the directory. The combined view of IFS reflects all these changes: accesses to file `/a/c` and `/a/e` are redirected to the corresponding copies in the temporary storage, while accesses to file `/a/d` will still go to the version in the main file system.

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, e.g., 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,* i.e., 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*, i.e., 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, e.g., 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*, i.e., 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 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.

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, i.e., 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 timestamp information we have about T_i while being able to cope with the minimal timestamp 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, e.g., 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 approach in this context is to traverse the contents of the temporary storage and copy them into the main file system. However, this simple approach does not always produce expected results. Consider, for instance, a case where a file a is first renamed to b and then modified. A simple traversal and copy will leave the original version of a as is, and create a new file b whose contents are the same as in the temporary storage. The correct result, which will be obtained if we redo all the (write) operations at the point of commit, will leave the system without the file a. Thus, the simple approach for state-based commit does not work correctly.

The above example motivates a log-based solution: 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 higher than an approach that uses state-based committing. For instance, consider an application that creates and deletes many (temporary) files. The state-based approach will need to store very few files in temporary storage, but a log-based approach will need to store all the write operations that were performed, including those on files that were subsequently deleted. Moreover, redoing the log can be substantially more expensive than state-based commit, since the latter can exploit rename operations to avoid file copies altogether.

The above discussion brings forth the complementary benefits of the two approaches. The first approach makes use of the accumulated modification results on file system objects, thus avoiding the expense associated with the maintenance and redoing of logs. The second approach, by maintaining logs, is able to handle subtle cases involving file re-

names. In our implementation of the commit operation, we combine the benefits of both.

We refer to our approach as state-based commit. 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 maintaining "condensed" logs, 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 superceded by other operations, in which case only latter operations are maintained. For instance, a delete operation supercedes 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 superceded 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 is 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

In the previous two sections, we discussed aspects of IFS, our filesystem proxy. In this section, we discuss how the other components of SEE fit together, including the components that support restriction, network level redirection, and user interface.

Implementing Restriction at System Call Layer. The actions of SEE processes are regulated by a kernel-resident 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).* IPC operations are allowed among the processes within the same SEE. However, no IPC may take place between SEE and non-SEE processes. An exception to this rule is currently made for X-server access. (To be safe, we should restrict X-server accesses made by SEE applications so that they don't interfere with X-operations made by non-SEE applications. However, our implementation does not currently have the ability to enforce policies at the level of X-requests.)
- *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 [29, 34] 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 us-

age. 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 it 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 [34].

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.

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, e.g., access to domain name service and informational web sites, can be permitted by configuring the kernel 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 kernel 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 email proxy may be implemented which simply accepts email 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 email is expected.

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

User Interface. In this section, we describe the support provided in our implementation for users to make decisions regarding commits.

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.

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. In particular, we 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. 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.)

6. Evaluation

In this section, we present an evaluation of the functionality and performance of our SEE implementation.

6.1. Evaluation of Functionality

Untrusted applications. We describe two applications here: a file renaming utility freeware called `rta` [33], 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` [30]. 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 the file `.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.

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, e.g., creating a huge file in the user's home directory. These actions were aborted. Also, at the time of writing this paper, 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.

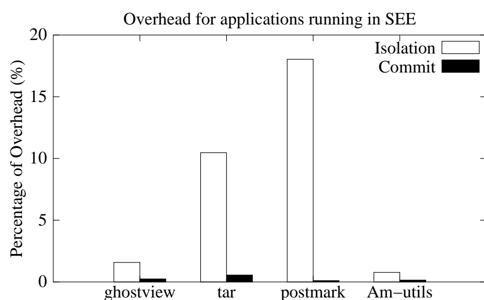
Software installation. Another experiment performed a trial installation of `mozilla` browser. During the installation, an incorrect directory name `/usr/bin` was chosen as the location for installation, instead of the default directory `/usr/local/mozilla`. Under normal circumstances, this causes Mozilla 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 non-executables) 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.

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 every one. Thus, SEE allowed the software upgrade to be tested easily and conveniently, without having to shutdown the original server.

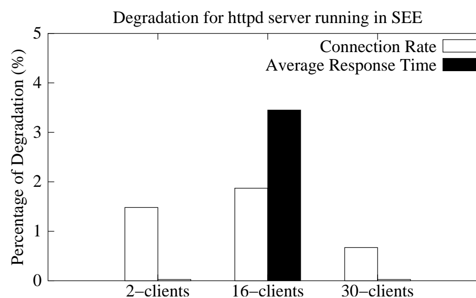
After verifying the operation of the new server, we attempted to commit the results. Unfortunately, this produced conflicts on some files such as the access and error log files used by the server. We chose to ignore updates to such output files that were made within the SEE, and commit only the rest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

6.2. Performance Evaluation

All 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. The primary metric was elapsed time.



(a) Utility applications



(b) Apache httpd server

Figure 2. Performance Results for Program Execution in SEE

For performance evaluation, we considered the following classes of examples:

- *Utility programs.* In this category, we studied `ghostview` and `tar` utilities. Specifically, we performed 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 2, we can see a 3-12% overhead incurred for such applications during isolation phase, and a negligible commit time overhead.
- *Servers.* We measured the performance overhead on the Apache web server using WebStone [35], 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, sixteen and thirty 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 2.
- *File system benchmarks.* We used `Postmark` [9] and `Am-Utils` [18] 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 email. 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 2000 file system operations were performed. In total, 1515 files were created, 1010 files read, 990 file written, and 1515 files deleted. The tests were repeated ten times. The results are as depicted in figure 2. Overall, a 18% performance degradation is observed, and commit overhead is 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 6 rename and 8 setattr operations. We ran this experiment in both original file system and IFS. The results, shown in Figure 2, indicate a low isolation overhead of under 2% and a negligible commit overhead.

	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

Figure 3. Comparison for Log-based Commit and State-based Commit. All numbers are in seconds.

In addition, we also collected results in Figure 3 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

Sandboxing. Sandboxing based approaches [7, 6, 1, 21, 27, 22] involve observing a program's behavior and blocking actions that may compromise the system's security. Janus [7] incorporates a `proc` file system based system call interposition technique for the Solaris operating system. A more recent version has been implemented on Linux, and uses a kernel module for interposition. Chakravyuha [6] uses a kernel interception mechanism. MAPbox [1] is a sandboxing mechanism where the goal is to make the sandbox more configurable and usable by providing a template for sandbox policies based on a classification of application behaviors. [21] creates the policy sandbox for programs (such as web browser) by first tracking the file requests made by the programs. This approach, however, requires a training phase, in which users need to run the programs using "normal" inputs, so that the policy sandbox can capture a complete set of files accessed by the programs. But in the case of untrusted code, the choice of such inputs may not be clear. Safe Virtual Execution (SVE) [27] uses Software Dynamic Translation, a technique for modifying binaries as they execute, is used to implement sandboxing. Systrace [22] 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.

The main drawback of sandboxing based approaches is the difficulty of *policy selection*, i.e, 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.

[28, 38] extend sandboxing by allowing operations to be disallowed silently, i.e., by returning a success code to the program. The goal of the approaches is deception, i.e., 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.

Isolation approaches. Two-way isolation between a host and guest operating system forms the basis of security in virtual machine based approaches for realizing SEEs. The "playground" approaches developed for Java programs in [15, 4] also belong to this general category — untrusted programs are run on a physically isolated system, while their display is redirected to the user's desktop. Note that the file system on the user's computer cannot directly be accessed on the playground system, which means that there is two way isolation being employed in this case. Covirt [3] proposes that most of applications be run inside virtual machine instead of host machines. Denali [36] is another virtual machine based approach that runs untrusted distributed server applications. As outlined in the introduction, all the above approaches suffer from the difficulty of environment reproduction, and also in committing the changes back to the original system. As a result, they do not provide a helpful approach for the applications discussed in the introduction.

[13] was the first approach to present 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.

In our previous work [12], we developed a practical approach for secure execution of untrusted software based on isolation. The focus of this effort was on developing a tool that can be easily installed and used by ordinary users that may not have administrative access to a computer. It is implemented entirely at the user level, and does not require any changes to the OS kernel. In order to achieve this objective, [12] compromises on performance as well as generality. In particular, the approach suffers from high overheads that can be over 100% in some instances. Moreover, isolation semantics cannot be faithfully reproduced for certain operations that involve meta-data such as permissions and ownership. For directories, isolation is achieved using an ad-hoc approach that is hard to implement and its semantics is inconsistent with that of file updates. Finally, no systematic solution to the commit problem is provided.

The approach developed in this paper addresses all these drawbacks by implementing isolation within the kernel at the VFS layer. Moreover, it shows how the approach can be generalized so that isolation can be provided for non-file operations, e.g., certain classes of network accesses.

Recovery-oriented systems. The Recovery-Oriented Computing (ROC) project [24] is developing techniques for fast recovery from failures, focusing on failures due to operator errors. [2] presents an approach that assists recovery from operator errors in administering a network server, with the specific example of an email 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 they cannot be fully reversed. In the case of email service, they allow a certain level of inconsistency, e.g., redelivering an email 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.

File system approaches. Elephant file system [26] is equipped with file object versioning support, and supports flexible versioning policies. [5, 23, 25, 31, 19] use check pointing technique to provide data versioning. [16] 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 paper 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 [40, 39] makes use of versioning file system to bring repair facility to a compromised file server. Fastrek [20] 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, this approach 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 [14] 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 [10] 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 [32] 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. [17] 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 paper, we presented an approach 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.

Our 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%.

References

[1] A. Acharya and M. Raje. Mapbox: Using parameterized behavior classes to confine applications. In *Proceedings of USENIX Security Symposium*, 2000.

[2] A. Brown and D. Patterson. Undo for operators: Building an undoable e-mail store. In *Proceedings of USENIX Annual Technical Conference*, 2003.

[3] P. M. Chen and B. D. Nobl. When virtual is better than real. In *Proceedings of Workshop on Hot Topics in Operating Systems*, 2001.

[4] T. Chiueh, H. Sankaran, and A. Neogi. Spout: A transparent distributed execution engine for java applets. In *Proceedings of International Conference on Distributed Computing Systems*, 2000.

[5] S. Chutani, O. T. Anderson, M. L. Kazar, B. W. Leverett, W. A. Mason, and R. N. Sidebotham. The episode file system. In *Proceedings of the USENIX Winter 1992 Technical Conference*, 1992.

[6] A. Dan, A. Mohindra, R. Ramaswami, and D. Sitaram. Chakravyuha: A sandbox operating system for the controlled execution of alien code. Technical report, IBM T.J. Watson research center, 1997.

[7] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A secure environment for untrusted helper applications: confining the wily hacker. In *Proceedings of USENIX Security Symposium*, 1996.

[8] S. Jajodia, P. Liu, and C. D. McCollum. Application-level isolation to cope with malicious database users. In *Proceedings of Annual Computer Security Applications Conference*, 1998.

[9] J. Katcher. Postmark: A new file system benchmark. Technical Report TR3022, Network Applicance Inc., 1997.

[10] D. G. Korn and E. Krell. A new dimension for the unix file system. *Software: Practice & Experience*, 20(S1), 1990.

[11] B. W. Lampson. A note on the confinement problem. *Communications of the ACM*, 16(10):613–615, 1973.

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

[13] P. Liu, S. Jajodia, and C. D. McCollum. Intrusion confinement by isolation in information systems. In *Proceedings of IFIP Workshop on Database Security*, 1999.

[14] Loop back file system. Unix man page.

[15] D. Malkhi and M. K. Reiter. Secure execution of java applets using a remote playground. *Software Engineering*, 26(12), 2000.

[16] K.-K. Muniswamy-Reddy, C. P. Wright, A. P. Himmer, and E. Zadok. A versatile and user-oriented versioning file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2004.

[17] J.-S. Pendry and M. K. McKusick. Union mounts in 4.4bsd-lite. In *Proceedings of 1995 USENIX Technical Conference on UNIX and Advanced Computing Systems*, 1995.

[18] J. S. Pendry, N. Williams, and E. Zadok. Am-utils user manual, 6.1b3 edition, july 2003. http://www.am-utils.org.

[19] Z. Peterson and R. Burns. Ext3cow: The design, implementation, and analysis of metadata for a time-shifting file system. Technical Report. HSSL-2003-03, Hopkins Storage Systems Lab, Department of Computer Science, Johns Hopkins University, 2003.

[20] D. Pilania and T. Chiueh. Design, implementation, and evaluation of an intrusion resilient database system. In *Proceedings of International Conference on Dependable Systems and Networks*, 2003.

[21] V. Prevelakis and D. Spinellis. Sandboxing applications. In *Proceedings of Usenix Annual Technical Conference: FREENIX Track*, 2001.

[22] N. Provos. Improving host security with system call policies. In *Proceedings of USENIX Security Symposium*, 2003.

[23] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proceedings of USENIX Conference on File and Storage Technologies*.

[24] Recovery-oriented computing. http://roc.cs.berkeley.edu.

[25] W. D. Roome. 3dfs: A time-oriented file server. In *Proceedings of the USENIX Winter 1992 Technical Conference*, 1991.

[26] D. J. Santry, M. J. Feeley, N. C. Hutchinson, and A. C. Veitch. Elephant: The file system that never forgets. In *Proceedings of Workshop on Hot Topics in Operating Systems*, 1999.

[27] K. Scott and J. Davidson. Safe virtual execution using software dynamic translation. In *Proceedings of Annual Computer Security Applications Conference*, 2002.

[28] R. Sekar, Y. Cai, and M. Segal. A specification-based approach for building survivable systems. In *Proceedings of National Information Systems Security Conference*, Oct 1998.

[29] R. Sekar and P. Uppuluri. Synthesizing fast intrusion prevention/detection systems from high-level specifications. In *Proceedings of USENIX Security Symposium*, 1999.

[30] K. Sitaker. Picturepages software. http://www.canonical.org/picturepages/.

[31] C. Soules, G. Goodson, J. Strunk, and G. Ganger. Metadata efficiency in a comprehensive versioning file system. In *Proceedings of USENIX Conference on File and Storage Technologies*, 2002.

[32] Translucent file system, 1990. SunOS Reference Manual, Sun Microsystems.

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

[34] P. Uppuluri. *Intrusion Detection/Prevention Using Behavior Specifications*. PhD thesis, Stony Brook University, 2003.

[35] Webstone, the benchmark for web servers. http://www.mindcraft.com/webstone.

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

[37] E. Zadok, I. Badulescu, and A. Shender. Extending file systems using stackable templates. In *Proceedings of USENIX Annual Technical Conference*, 1999.

[38] M. Zalewski. Fakebust, a malicious code analyzer. http://www.derkeiler.com/Mailing-Lists/securityfocus/bugtraq/2004-09/0251.html.

[39] N. Zhu. Data versioning systems. Technical report, Stony Brook University, http://www.ecsl.cs.sunysb.edu/tech_reports.html.

[40] N. Zhu and T. Chiueh. Design, implementation, and evaluation of repairable file service. In *Proceedings of International Conference on Dependable Systems and Networks*, 2003.