Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs*

Zhenkai Liang, V.N. Venkatakrishnan and R. Sekar
Department of Computer Science
Stony Brook University, Stony Brook, NY 11794.
{*zliang, venkat, sekar*}*@cs.sunysb.edu*

Abstract

In this paper, we present a new approach for safe execution of untrusted programs. This approach is based on isolating the effects of untrusted program execution from the rest of the system. Isolation is achieved by intercepting and redirecting file modification operations made by the untrusted process so that they access a "modification cache" invisible to other processes in the system. To ensure a consistent view of system state, the results of file read operations made by the untrusted process are modified to incorporate the contents of the modification cache. Any operation with a potential to modify a non-file resource is disallowed for untrusted processes. On termination of an untrusted process, the user is presented with a concise summary of the files modified by it. Additionally, the user can inspect these files to determine if the modifications are acceptable. The user then has the option to commit these modifications, or simply discard them. Essentially, our approach provides "play" and "rewind" buttons for running untrusted software. Key benefits of our approach are that it requires no changes to the untrusted programs (to be isolated) or the underlying operating system; it cannot be subverted by malicious programs; and it achieves these benefits with acceptable runtime overheads. We describe a prototype implementation of this system for Linux called Alcatraz and discuss its performance and effectiveness.

1. Introduction

The widespread deployment of firewalls and related solutions for network security has significantly raised the bar for remote attacks on an enterprise network. However, even the best perimeter solutions can be easily defeated by an attacker that can induce users inside the enterprise to download and execute malicious code. While virus detection and similar techniques can be deployed to detect widely prevalent instances of malicious code, such techniques are limited in theory (by the fact that detection of malicious code is undecidable in general) as well as practice (by the difficulty of object code analysis and factors such as encryption).

A more promising approach for defending against malicious code is based on *sandboxing,* wherein the resource accesses made by untrusted code are suitably restricted to ensure security. However, use of such approaches in practice has been hampered by the difficulty of *policy selection*: determining resource access rights that would allow the code to execute successfully without compromising system security. Too often, sandboxing tools incorporate highly restrictive policies that preclude execution of most useful applications. The net result is that users end up choosing functionality over security, and thus execute untrusted code outside such sandboxing tools, exposing themselves to unbounded damage if this code turned out to be malicious.

An alternative to sandboxing is *isolated execution,* wherein the actions of untrusted code are isolated from other applications and resources to be protected. Isolated execution has previously been studied by researchers [14, 6] in the context of Java applets. Such applets do not require much access to system resources, other than being able to interact with a user. Hence the implementation approach used by these works relied on executing untrusted applets on a "remote playground", i.e., an isolated computer (other than a user's desktop). However, applications that perform more useful functions will require access to resources such as the file system on the user's computer. To provide such access, the entire environment on the user's computer, including file system contents, must be duplicated on the remote playground.

Logical isolation, wherein the effects of a malicious process are logically isolated from other processes, can achieve the benefits of isolated execution without the drawback of requiring dedicated hardware or solving the difficult problem of accurate duplication of environment. It was proposed in [18] to permit continued operation of compromised processes without alerting attackers and without risking damage to the rest of the system, and in [10] in the context of databases. The theory of data isolation was further devel-

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

oped systematically in [13] in the context of databases as well as file systems, and isolation protocols that demonstrate the feasibility of the approach were presented. However, practical issues that arise in implementing this approach on contemporary operating systems were not studied. In contrast, this paper develops an application- and OS-transparent approach for isolated execution of untrusted programs, and describes a tool called *Alcatraz* that implements this approach on the Linux operating system.

Our approach is based on system-call interposition. It permits untrusted applications to access the entire file system that is accessible to the end users, thereby making it possible for most applications to carry out their tasks. Using a copy-on-write semantics, modifications to the file system that are performed by the application are hidden from the rest of the system, which ensures that malicious actions of the untrusted code will not compromise the integrity of the system. Accesses to non-file resources are restricted as needed to ensure integrity. At the completion of execution, the users can examine the accesses made by the untrusted code to see if it changed any files of interest to them, and if so, examine these changes. If the users are convinced that these changes are benign, then they can *commit* these changes, so that they become visible to the rest of the system. Otherwise, the users can *abort* these changes. The key benefits of our approach are:

- *Application and operating system transparency.* Our approach requires no changes to the underlying operating system or the untrusted application itself. Moreover, the technique can be applied regardless of whether the files accessed by the application are local, or are located on a remote file server.
- *Secure yet application-friendly.* Our approach provides security against malicious code without imposing undue restrictions on such code. This makes it possible for a large class of existing software to execute successfully, unlike sandboxing based approaches.
- *Convenient and user-friendly.* Our approach provides a compact summary of the file system resource accesses made by untrusted code at the end of its execution. This contrasts with sandboxing approaches that prompt users every time an application accesses a file that is not permitted by the sandboxing policy. In addition, the user is given the ability to examine these files to determine whether the application carried out the functionality that the user wanted.

Our implementation does not require the users of our system to possess administrator privileges. It imposes modest overheads for isolation (below 20% for all the applications we have studied). However, the mechanism we have used for system call interposition poses moderate overheads, ranging from under 10% for CPU-intensive applications to nearly 100% for I/O-intensive applications.

The description in the rest of the paper is set in the context of the Linux operating systems, but the techniques are applicable to most modern operating systems. The organization of the rest of paper is as follows. We begin with two motivating examples in Section 1.1. Section 2 provides an overview of the system design, followed by more detailed descriptions of the system components. Implementation results are discussed in Section 3, followed by related work in Section 4. Finally, concluding remarks appear in Section 5.

1.1. Motivating Examples

Photo organizer. Consider an application that scans specified directories for image files and generates photo album files that are written to the same directories. It also generates thumbnail pictures from these files (for creating index files) and has the ability to modify/resize these files. Similar applications that modify images and other media such as audio files) are available as freeware on the Internet, e.g., the `picturepages` [20] package.

Safe execution of applications such as the photo organizer poses two challenges for sandboxing approaches.

- *apriori policy selection:* Users have to anticipate the resource access requirements of a program prior to its execution, which is often difficult. To overcome this problem, some sandboxing approaches allow changes to policies through runtime prompts to the user: when the sandboxed application violates the initially specified policy, the user is informed and queried whether he/she wants to permit this access. Unfortunately, such repeated prompts lead to "click-fatigue," as a result of which the user simply grants (or refuses) all subsequent prompts without reviewing them carefully.
- *policy granularity:* Development of enforceable policies [17] that permit the application to perform the file access operations that it needs to perform, while ensuring that these files are not corrupted or deleted. Such a policy would have to permit "legitimate" changes to files, as needed for resizing files or including preview images, while disallowing other changes. Development of a policy that can capture such legitimate transformations on files is likely to be hard. Even if they can be expressed, enforcement of such policies is likely to be inefficient, if not impossible.

Due to these difficulties, sandboxing policies tend to be conservative and often disallow a large class of useful programs such as the `picturepages` program. In contrast, our proposed approach will permit execution of programs as long as they don't make system changes other than file modification operations. Few applications violate this constraint, and hence a majority of applications can be run safely using isolation. Moreover, users need not develop safety policies ahead of time. Finally, they have the opportunity to examine the system state resulting due to the execution of the un-

trusted program, and then decide whether to "keep" or "roll back" these changes. They can use standard system utilities such as `find` and `diff`, as well as arbitrary helper applications such as image viewers, to examine the system state.

Software installation. Users are all too familiar with poorly packaged software that crashes during its installation, or simply does not function correctly. Even worse, the new package may "break" other applications installed on the system. In all these cases, the users are faced with the daunting task of rolling back the installation. If the package made use of standard package management utilities, this roll back is usually not burdensome. However, if the package came as a self-installing executable or as a source package, roll backs are almost always very difficult. The package may install its files into standard directories such as `/usr/local/bin` and `/lib`. It may also modify system configuration files such as `/etc/passwd`, `/etc/mime.types` or user profile files such as `~/.bashrc`. Identifying the exact set of files that were modified is cumbersome. It is also prone to errors as the user does not know the directories where the package installed files, and hence has to search the entire file system. This may result in identifying many files that may have been modified by applications other than the installer. Even if the modified files are identified correctly, roll back is still a hard problem: it is possible only if the user had backed up modified files, but unfortunately, the user doesn't know which files to back up.

Using our isolation approach, all of the above problems can be tackled easily. The user simply installs the package in isolation. Within the isolation environment, the user can then try out the package. They can then examine the files modified by the package, and see if it includes security-critical files, or files that may be used by other packages. System configuration databases, such as the Redhat Package Manager (RPM) database, can help in identifying files used by other packages. If so, they can examine these files to identify the changes made. Alternatively, they can try out the applications that depend on these modified files to ensure that they are not broken. If the user is convinced, after making all these checks, that the new packages were installed correctly and are functioning properly, he/she can commit the installation. Otherwise the user can discard the installation — at this point, the file system state will be as if the installation never took place.

2. System Description

2.1. Technical Goals and Design Approaches

The goal of logical isolation is to preserve system integrity. (Confidentiality can be preserved to the extent the untrusted application can be prevented from making network communications, but this not our main goal in this paper.) In particular, if the file system changes made by an untrusted application were not committed, then the integrity of the system must not be compromised by this application.

Our approach is based on preserving the contents of the file system. However, in order to ensure overall system integrity, we also need to consider operations other than those involving file systems. Such operations must be prevented from being executed if they can change the system state. We need to be conservative in determining whether an operation can change system state: unless we know for sure, an operation made by an untrusted process must be prevented. A simple implementation of such a conservative approach may disallow all network communications (as they can modify the state of other hosts), file operations that modify devices, `ioctl` operations, etc. A more usable approach will recognize a subset of these operations that do not change system state, and permit them. For instance, it is reasonable to consider that DNS queries do not modify system state. Similarly, sufficient intelligence may be built into the implementation to recognize and permit certain `ioctl` and device-level operations that query system state without modifying it. More generally, service-specific proxies may be built that can forward those service requests that do not change system state, while disallowing other requests. Such service-specific proxies may be built to access `X-windows`, web servers, audio devices, etc. For the rest of this paper, we do not dwell on these topics, but focus on achieving file system level isolation.

In our approach, file-level isolation is achieved using *isolation contexts.* An isolation context can be thought of as a "private copy" of the entire file system. It is implemented using a copy-on-write technique, so that its storage requirement is proportional to the changes made within the isolation context, and not on the size of the entire file system. A new isolation context is created when an untrusted process needs to be executed. If this process forks children, then all such children and their descendants are executed within the same isolation context. This ensures that the untrusted process and its descendants have an identical (and consistent) view of the file system state. Multiple untrusted applications may be executed independently, each within its own isolation context. (We note that copy-on-write provides one-way-isolation, i.e., changes made within an isolation context are shielded from the rest of the system, but the changes made outside of isolation contexts may be visible inside. This is sufficient to achieve our goal of preserving system integrity.)

To implement isolation contexts, file system changes made by an untrusted process should be redirected so that they do not change global system state. Such redirection may be built into the application itself or within the system libraries that are used to access files. Neither approach is

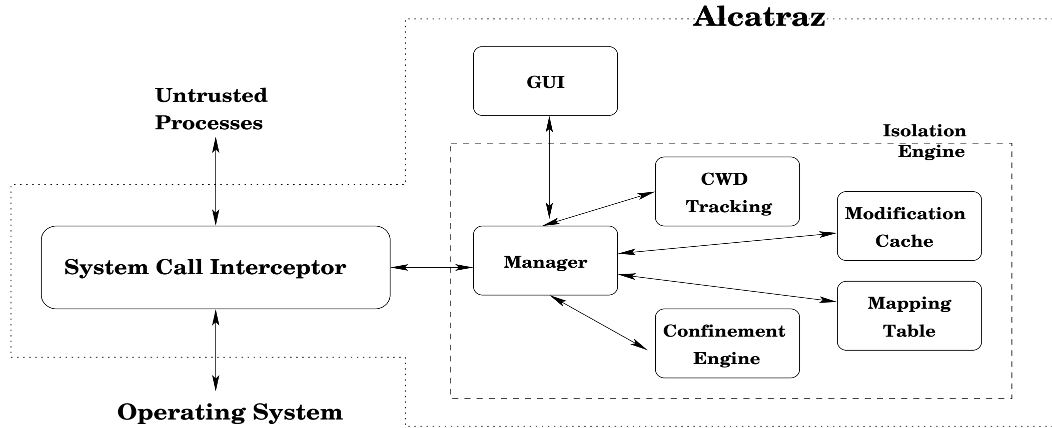


Figure 1. System Architecture

satisfactory, since they require the applications to be trusted. In particular, a malicious application can bypass such redirection, and make direct access to the system calls provided by the OS for manipulating files. We therefore rely on OS-level mechanisms that can support secure redirection. There are two main choices in this regard:

- *System-call interposition:* Since all accesses to system resources (including accesses to files, devices and the network) are effected through system calls, interposing at this level provides a secure way to achieve isolation.
- *Interposition at the VFS layer:* The Virtual File System layer provides an abstract interface within the OS kernel for accessing all file systems. One benefit of interposing at this layer is that of higher performance: only file system operations are interposed, as opposed to all system calls.

Of these choices, we have adopted system call interposition for two reasons. First, it can be implemented without requiring changes to the operating system. Indeed, the `ptrace` mechanism in Linux permits ordinary users to intercept system calls made by their processes, without requiring them to make any OS-level changes that need superuser privilege. Second, as discussed earlier, we need to monitor non-file operations made by the untrusted process, and hence system call interposition would be necessary even if file level isolation were implemented using VFS interposition.

2.2. System Overview

The architecture of our system, called *Alcatraz*, is shown in Figure 1. The isolation engine consists of several components. The *manager* module coordinates the operations of the isolation engine. It uses the *modification cache* as a scratch-pad area where new files (or directories) created by the untrusted process are held. The modification cache is a dedicated area within the file system, and uses a distinctive name so that multiple Alcatraz sessions can run on the same system. For files (and directories) stored in the modification cache, the mapping table provides the translation between file names used by an untrusted process and their corresponding names within the modification cache. The table also records other information pertaining to modified files, e.g., whether a file is deleted, time stamp of original file and so on.

Note that the isolation engine holds all the information about modifications to the file system, and the operating system kernel does not know about these changes. Therefore the isolation engine needs to modify the arguments and/or the return values of system calls that access files. In particular, when a system call is invoked in an isolated process, the *system call interceptor* sends a notification to the manager module. The manager module handles file system modification operations, while forwarding the rest of the system calls to the confinement engine. If the file operation refers to objects that have been modified, then the manager modifies the path name argument so that it refers to the modified object located within the modification cache. These (possibly modified) arguments are returned back to the system call interceptor. When the system call returns, the manager module is once again notified, so that it may modify the results returned by the system call as necessary.

The mapping table maps one absolute file name into another. However, not all the system calls are invoked with a absolute path names. Hence path names must be resolved into an absolute path name, with symbolic links expanded, and the "." and ".." entries resolved. The CWD Tracking module helps this process. It maintains the current working directory of each process and updates them when a process makes a system call that results in changes to that directory. The current working directory of a parent process will be inherited by its children.

After the untrusted process finishes execution, the isolation engine invokes a GUI (graphical user interface), which

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

Figure 2. Classification of system calls

presents a compact summary of the security relevant actions made by the process. If these changes are accepted by the user, then they are "copied over" so that they become visible to other processes in the system. Criteria for determining whether such copying can be done while preserving isolation semantics is described later in the paper. Below, we describe the key components of Alcatraz in further detail.

2.3. Manager

As mentioned above, the key problem in implementing the isolation engine is that of modifying file-related system calls in a manner that provides a consistent view of the system state to the isolated process. This becomes a challenging task when we consider 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). To tackle the above complexity, we make the following observations about the kinds 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. They also indicate whether the type of object referenced by the inode: whether it is a file, directory, device, etc.) File modification operations 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 the other types, such as symbolic links, the only modification operation permitted is that of deleting the file. In this sense, it is nothing more than a directory modification operation. So from the modification point of view, there are only three types of objects on the file system: regular files, directories, and inodes.

Let us now consider the system call operations on the file system. For the isolation operation, we only need to consider system calls that are pathname related. System calls that operate on file descriptors (e.g., `read`, `write` and `mmap`) can be left to the operating system to handle. These operations can be classified as shown in Figure 2 based on whether they modify the file system, and if so, the type of file system object modified. Since the manner in which "read" operations are implemented is determined by the way modifications are implemented, our description below is organized by the three categories of modification operations.

Regular file modifications. Consider a process that opens a file f for writing. The natural way to isolate the execution of the process, is to create a new copy f' of f that is stored within the modification cache. All future accesses to f, whether they be modifications or reads, will be redirected to f'. To enable this redirection, an entry that associates f with f' is inserted into the mapping table. An optimization that avoids copying of files is possible in the common case when file is truncated to zero length at the open, or immediately afterward.

Directory modifications. The above simple implementation of copy-on-write does not directly extend to directories. In particular, to implement copy-on-write for directories, we would need to copy the contents of the directory, as well all of the file system contents located below the directory. Clearly, such an approach would be far too inefficient.

To develop a more efficient approach for copy-on-write, we observe that unlike a regular files, a directories are accessed in a structured manner using specialized directory operations such as `mkdir` and `getdents`. Thus, our approach is one of modifying 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 modification cache.

When the contents of such modified directories are read using the `getdents` operation, one way is to modify the directory entries returned by `getdents` to incorporate the information stored in the modification cache after the call returns. In particular, a directory entry f that is mapped into f' by the mapping table is replaced so as to contain the information about f'. If the file f has been deleted by the isolated process, then the entry corresponding to f is deleted from the `getdents` return value must be deleted before the results are forwarded to the isolated process. 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. (This is how `getdents` works under Linux.) To solve this problem, we retrieve all the directory entries during the first directory read operation into a buffer and apply the changes to the buffer. After the changes are applied, we append new directory entries that are recorded in the modification cache but not present in the rest of the file system. The result is returned to the isolated 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 location (i.e., just created or modified file), we can redirect this operation to its counterpart in the temporary location. If the modification is made to an unchanged regular file, we can also use the copy-on-write mechanism. However, this mechanism does not work on directories because we cannot copy a directory in a similar fashion. One possibility is to use the isolation layer to record the changed Inode information of directories and let all related system calls make use of this information. However, this solution is not very useful in all cases, as the kernel does not know about the existence of such information. For example, if the untrusted program adds write permission to an existing directory, using this approach, this change will be stored in the isolation layer, but the kernel will still not allow it to write into that directory because this changed permission information is not visible to the kernel. In our implementation, the isolation layer records an error message in such situations, and allows continued execution of the isolated process. This is a limitation of our current implementation. We note that it is abnormal for untrusted code to change (especially relax the permission) meta data associated with existing directories (i.e., those not created by untrusted software), so this limitation has not been a significant problem in practice.

Since the latest inode information is held within the isolation layer, system calls to access or manipulate meta data, such as `stat`, need to be intercepted by the manager and redirected if necessary. Moreover, since the correct permission information is not available to the file system, permission checking needs to be handled by the isolation layer. To understand the need for this, consider the case when the isolated process modifies a file that it does not own but has the write permission. The isolation engine will copy the file into the modification cache 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 will be disallowed by the kernel unless the isolation engine runs with root privileges. Since it was one of our design goals to support isolation without requiring superuser privileges, we cannot change the ownership information on the file. This means that the OS will interpret the permissions incorrectly, thus requiring the isolation engine to take over this task.

2.3.1. Confinement Engine The untrusted program may perform other operations that are unrelated to the file system. Some of these operations do not cause difficulties in preserving isolation semantics, e.g., system calls for obtaining timing information, process ownership, host attributes, etc. Others, such as those involving network communication or interaction with processes outside its isolation context, will pose a problem. It is the responsibility of the confinement engine to deal with all system calls that are unrelated to file systems. It determines which system calls can be permitted without compromising the isolation semantics.

The confinement engine is built from security policy specifications that specify which system calls can be permitted, and in what context. These policies are specified in text using a language called BMSL (Behavior Monitoring Specification Language) [19, 21]. BMSL can express describe conventional access control policies, history sensitive policies (e.g., an application cannot access the network after reading sensitive files) and resource usage policies (e.g., an application can write no more than k bytes of data). These policies are compiled using the BMSL compiler to produce the confinement engine. A detailed description of BMSL syntax, semantics, and compilation can be found in [21].

The confinement engine currently disallows networks requests such as web access, DNS queries, and X-windows operations. As outlined earlier, these limitations can be relaxed using service-specific proxies. For instance, we can have a proxy that receives DNS requests from the isolated process, and forwards them to the DNS server if it can be ascertained that this query will not change the system state. Otherwise, the proxy refuses the request.

2.4. System Call Interceptor

The system call interceptor is responsible for intercepting system calls and forwarding them to the isolation engine. The interceptor is implemented in such a way that it is portable with minimal changes to other Unix variants (that do not support `ptrace` for instance). The architecture of our interceptor is based on the design presented in [9].

The implementation of the interceptor (the *tracing process*) is based on Linux's `ptrace` system call, which allows one process, called the *monitoring process* to trace another process, called the *monitored process*. Tracing capabilities include the ability to intercept system calls made by the monitored process, and examination or modification of the virtual memory of the monitored process. When using `ptrace` for monitoring and confining untrusted processes,

we face a number of difficulties that can compromise security. Below, we summarize how our implementation tackles these difficulties.

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 SIGKILL signal to the interceptor. However, this must again be done through a system call. To protect our system, the system call interceptor simply aborts those system calls that can interfere with the operation of 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 devised in the `strace` [] program. Specifically, when the monitoring process intercepts the parent's entry into `fork` system call, it replaces the the instruction in the parent's code at its instruction pointer (IP) register with a loop instruction. Note that the child will inherit this code, as well as the value of IP. This means that when control returns to the child, it will execute the loop instruction, and hence will be stuck in an infinite loop. In particular, it won't be able to make any system calls. When the `fork` system call returns to the parent, the monitoring process obtains the child PID, and issues a `ptrace` system call to attach to the child. It then restores the original instructed that was stored at the instruction pointer, so that the child process can continue with its normal execution, but now under the control of the monitoring process.

Even after the above enhancement, there still exists a possibility of a race condition: if the child process receives a signal, this will interrupt the loop and cause execution of its signal handler, which can execute system calls that may not be permitted by the monitor. Such a signal may be sent on purpose by another process in order to "free" the child and allow it to execute damaging system calls. To prevent this possibility, we note that if another intentionally cooperates with the child process to free it, then that process must itself be an untrusted process under the control of the monitoring process. For this reason, the system call used by the cooperating process to send a signal can be intercepted by the monitor and delayed until it has control of the child process.

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 [9]. 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, e.g., 10^7 or 10^8, then the likelihood of successful attacks becomes very small.

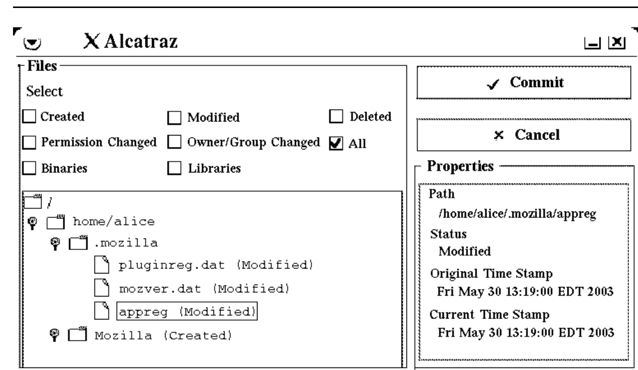


Figure 3. Graphical user interface

2.5. User Interface

After the isolated process and its children finish execution, the information maintained in the mapping table is sent to the user interface (GUI). The GUI sorts/groups file changes by path names, and then presents them to the user in a tree like representation as shown in Figure 3. The user can select the kinds of changes that they wish to see, e.g., new files created, files overwritten, etc. For modified files, users can view the difference between the original and the new version by simply clicking on the file name.

Optionally, the user can use a shell that runs in the same isolation context as the untrusted process, but has access to the original file system through the `/alcatraz` virtual directory. 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.

2.5.1. Commit Criteria. After examining the changes made by the untrusted process, a user can determine whether these changes can be committed to the system. However, it is possible that other processes, running outside of the isolation context of the untrusted pro-

cess, may have made modifications to the file system. If these changes interfere with the changes made by the untrusted process, then commitment of the changes made by the untrusted process can lead to an inconsistent system state. Hence, we adopt an approach in which the commit operation is allowed to go through only if the files modified by the isolated process were neither read nor written by outside processes since the instant the files were first accessed by the isolated process.

It may seem that this approach is too conservative and may reject results that can be consistently committed. While this is true, we observe that aborts do not cause too much difficulty in Alcatraz. In particular, the untrusted program can be executed again. Since the changes made by the untrusted program were discarded, rerunning the program will likely produce the same results. At this point, the same interference may not have taken place (assuming that such interference was a rare coincidence), and hence the results can be committed.

Our current implementation of commitment contains a race condition. In particular, interference (by processes outside of isolation) may happen during the time files are copied from the modification cache to the file system. This race condition can be avoided using file system locks. Unfortunately, mandatory locks are not supported by default on Linux due to the possibility that they may lead to deadlocks. If this were not the case, then the race condition can be avoided. In practice, however, we note that the race condition is not a significant problem in the context of untrusted program execution, as it is unlikely that the files accessed by such a program would also be concurrently accessed by other unrelated processes.

3. Implementation results

We have implemented Alcatraz on the Linux operating system [1]. The implementation has been tested on Red Hat Linux 7.3 and Red Hat Linux 8.0 distributions. The performance figures given below were obtained on a PC running Red Hat Linux 7.3 on a 1.7GHz P4 processor with 1GB memory.

3.1. Example Applications

Our implementation was tested with three applications: two freeware program that organize image/audio files, and the installation of a software package.

`Picturepages` is a photo editing program similar to the example presented in Section 1.1. We tested it with a directory of jpeg photos. Alcatraz reported the creation of a directory and changes to the picture files. We further used an image viewer to examine some of the generated pictures to make sure that they were properly modified.

The second program that was used is `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.

The third program we tested was the installation of `mozilla`, a freeware web browser. The installation 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 isolation system and reported through the user interface, as shown in Figure 3.

In all these examples, 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.

3.2. Performance results

We have measured the performance using two sets of applications. The first set of applications are the above examples. The second set included common UNIX utilities such as `make gcc`, `gzip`, `ghostscript`, and `tar`.

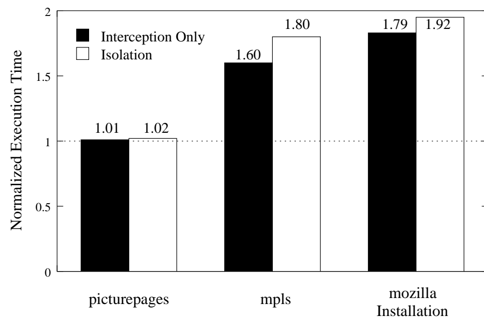
The following testing data was used:

- for `make gcc`, we compiled the openssh package version 3.7p1 under isolation. It contained 69849 lines of C code.
- for `tar`, a directory tree containing several mp3 files were used as the input for the archive operation. The size of output file was 85MB.
- for `gzip`, the output of the above tar command was used as input
- for `ghostscript`, a 10-page paper, containing 170K bytes is used as the input.

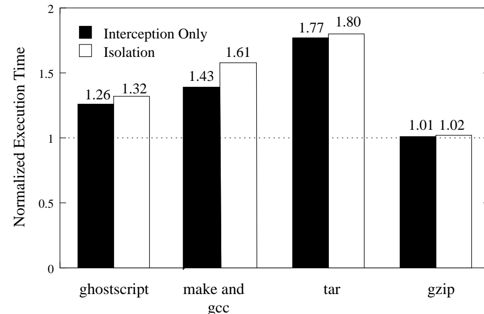
In order to know how each module in Alcatraz contributes to the overhead, we performed three time measurements of the sample application. They are the execution time without any system call interception, the execution time with only the system call interceptor, and the execution time with isolation, respectively. The normalized execution time (ratio to the execution time without isolation and without system call interposition) is shown in Figure 4.

From the performance results, we can see that the isolation mechanism itself (the difference between the overhead of "Interception Only" and the overhead of "Isolation") contributes to a modest overhead of less than 20%. However, the system call interposition mechanism contributes to a significant overhead for some programs. This overhead varies linearly with the frequency of system calls made by an application. Compute-intensive applications such as `gzip` and `picturepages` make much fewer system calls per unit time of execution, while other applications such as `tar` make system calls at a much higher rate.

A kernel interception mechanism will provide a much



(a) Downloaded program examples



(b) Common Unix applications

Figure 4. Normalized Performance Results

lower overhead due to the absence of context switches between our system process and the monitored process. However, we aimed to keep the performance overhead acceptable while keeping the system portable and completely at the user level such that it does not require kernel patches or modules. This balances our requirements of usability and performance.

An in-kernel implementation of system call interception would the interception overhead to a very small value (usually under 10%). However, if we had based Alcatraz on kernel-based interception, it would be harder to port, and moreover, cannot be downloaded, installed or run by users that do not have superuser privilege. Compared to this drawback, the additional overhead seems to be quite acceptable for the class of applications targeted by Alcatraz.

4. Related work

Sandboxing systems. Janus [8] 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 [7] is a monitoring system that uses a kernel interception mechanism to implement a sandboxing approach. MAPbox [3] is a sandboxing mechanism where the goal is to make the sandbox more configurable and usable by providing template classification of behaviors. Consh [4] provides a similar sandboxing environment while addressing transparent local and remote access to files.

SoftwarePot [11] 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 in an archive that is transferred from the code producer to the consumer. At the consumer end, a security policy determines the resources of the consumer that could be accessed by this untrusted code. Furthermore, all the operations to the files are confined to the "pot" archive. The scheme still requires apriori policy selection, that (as pointed out in the introduction) is often difficult.

Systrace [15] is a sandboxing system that notifies the user about all system calls that an application tries to execute. It then uses the response from the user to generate a policy for the application.

The disadvantages of sandboxing approaches, as compared to isolation, was discussed in Section 1.1.

Isolation systems. [14] and [6] use physical isolation to protect against damages to the client's machine. The incoming mobile code (java applet) is sent to another set of machines, called "playground" (some machines containing no important data), to execute. As mentioned in the introduction, these two systems only target Java applets (which only constitutes a small fragment of the large body of untrusted code on the Internet), require additional resources (such as new machines), and disallow any access to the user's environment. In contrast, our approach is language independent, and requires no additional physical resources and allows safe access to the user's environment.

Logical isolation provides many benefits over physical isolation. It has been suggested before and analyzed [13, 10, 18]. Algorithms and protocols for realizing logical isolation in the context of databases as well as file systems was presented in [13]. However, practical issues that arise in implementing the approach on a modern operating system were not considered. Our work in this paper complements these works, and developing an application- and OS-transparent approach for practical approach and tool for realizing logically isolated execution of programs.

Recovery-oriented systems. The Recovery-Oriented Computing (ROC) project at Berkeley [2] is developing techniques for fast recovery from failures, focusing on failures due to operator errors. [5] presents a broad approach that assists recovery from operator errors in administering a network server, with the specific example of an email server. In spite of the apparent similarities

in the goals of this work and ours, the technical requirements are quite different. They target network-oriented applications whose actions (and their effects) needs to be visible to other processes and/or hosts. In contrast, our approach targets file-oriented applications whose actions should be invisible to the rest of the world.

[22] presents an approach for safe execution of malicious applications on Microsoft Windows by intercepting operations made by the malicious code. Their approach is to create backup copies of files before they are modified by the malicious application. A drawback of this approach, as compared to ours, is that the modifications are visible to other benign processes in the system. If a benign process modifies the system based on the files modified by the malicious process, then there may be no way to undo these effects. In contrast, our approach provides a guarantee that the actions of the isolated process(es) cannot corrupt the system.

File System approaches. The Elephant file system [16] retains all the important versions of a file, and has an interface for users to select a specific version. RFS (Repairable File Service) [23] is specifically designed to facilitate repair of a compromised network file server by maintaining previous versions of files. These approaches generally have a significant storage overhead, since storing versions can consume significant additional space. In contrast, our approach does not impose high storage overheads. More importantly, our isolation approach provides a guarantee that the effects of a malicious process can be undone. In contrast, the versioning approaches will have to undo the effects of malicious as well as benign processes.

3D file system [12] 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 uses a technique called *viewpathing* which is based on translating file names used by a process. They implement their system in an application and OS-transparent way by intercepting and modifying library calls made by an application. However, as they do not deal with untrusted code, their implementation does not ensure non-bypassability and hence applications can freely access all the files in the local filesystem. In addition, library interception cannot be used in our case since malicious applications may bypass such interception and perform sensitive operations.

5. Summary

In this paper, we presented an approach that supports safe execution of untrusted programs. Our approach uses the idea of logical program isolation, where actions of the code are invisible to the rest of the system until they are committed by a user. Before committing, the user can inspect the system state to determine if the actions of the program compromised the integrity of the system. We have presented a tool called *Alcatraz* [1] that incorporates this approach. Our approach provides security for the end-user and enjoys many benefits such as application transparency and user friendliness. We have discussed the design and implementation and presented results from an evaluation.

References

[1] Alcatraz. http://www.seclab.cs.sunysb.edu/alcatraz.

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

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

[4] A. Alexandrov, P. Kmiec, and K. Schauser. Consh: A confined execution environment for internet computations, 1998.

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

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

[7] 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.

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

[9] K. Jain and R. Sekar. User-level infrastructure for system call int erposition: A platform for intrusion detection and confinement. In *ISOC Network and Distributed System Security*, 2000.

[10] S. Jajodia, P. Liu, and C. D. McCollum. Application-level isolation to cope with malicious database users. In *ACSAC*, 1998.

[11] K. Kato and Y. Oyama. Softwarepot: An encapsulated transferable file system for secure software circulation. In *Proc. of Int. Symp. on Software Security*, 2003.

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

[13] P. Liu, S. Jajodia, and C. D. McCollum. Intrusion confinement by isolation in information systems. *Journal of Computer Security*, 8, 2000.

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

[15] N. Provos. Improving host security with system call policies, 2002.

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

[17] F. B. Schneider. Enforceable security policies. *ACM Transactions on Information and System Security*, 3(1):30–50, 2000.

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

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

[20] K. Sitaker. http://www.canonical.org/picturepages.

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

[22] J. A. Whittaker and A. D. Vivanco. Neutralizing windows-based malicious mobile code. In *Symposium on Applied Computing*, 2002.

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