Total Installation Awareness

Robert Grant
The University of Texas at Austin
Dept. of Electrical and Computer Engineering
bgrant@mail.utexas.edu

John Thywissen
The University of Texas at Austin
Dept. of Computer Sciences
jthywiss@cs.utexas.edu

ABSTRACT
Users often install applications from untrusted sources. Alcatraz is a system that provides isolation of a system from potentially untrustworthy user programs. However, we find performance and functionality concerns with Alcatraz. Nevertheless, Alcatraz demonstrates the feasibility of one-way isolation of user programs. During a port of Alcatraz to a modern commodity operating system, Mac OS X, we observe that the various security goals present in such a system can be at odds with each other. This conflict impedes implementation of features such as isolation. We propose an alternative that preserves system adaptability properties while still serving the trusted platform needs of DRM. A substantially different approach to implementing one-way isolation appears feasible, and is left for future work.

Categories and Subject Descriptors
D.4.6 [**Operating Systems**]: Security and Protection—*access controls, invasive software*

General Terms
Design, Performance, Security

Keywords
Alcatraz, digital rights management, etrace, Mac OS X, system call interposition

1. INTRODUCTION
In the course of normal computer usage, users often install applications from untrusted sources. Users currently have very poor control over the application installation process, and often have very little knowledge of all the side effects of running an installer. This is bad from the perspectives of both system manageability and system security: at least the user cannot know all the effects of running an installer, and at worst an installer has malicious and untracked side effects.

Course project:
C S 380L, *Advanced Operating Systems*
Fall 2008
Prof. M. Dahlin
The University of Texas at Austin

Alcatraz is a software system developed at the Secure Systems Lab of Stony Brook University for protecting the integrity of a system from untrusted programs. It does this by allowing users run suspect software in a *Secure Execution Environment* (SEE) which attempts to provide *one-way isolation* through copy-on-write semantics.

More precisely, changes made outside the SEE are visible inside, but changes made inside the SEE are not visible outside until the user explicitly commits those changes to the base file system. Modifications which cannot be easily captured with copy-on-write file semantics are generally disallowed, though a few special exceptions are made.

Alcatraz is written as a plugin to an earlier system called Etrace [19] which supports system call interposition and system call rewriting on Linux. Etrace is well designed and architected and isolates system-dependent code for portability, though Alcatraz itself does not follow suit.

We make several contributions in this paper. First, in Section 2, we empirically evaluate the functionality and performance of Alcatraz in its native Linux environment (CentOS 4.6). Then, in Section 3 we present an in-depth study of the difficulties of providing similar isolation in Mac OS X v10.5. Finally, we summarize other work related to Alcatraz in Section 6, we outline possible future work in Section 7, and we we conclude in Section 8. More detailed experimental results are placed in Appendix A for the interested reader.

2. EVALUATION OF ALCATRAZ
In this section we empirically evaluate the functionality and performance of Alcatraz in its native Linux environment. Throughout, we examine our results in the context of the stated goals and claims of Alcatraz as outlined in the ACSAC 2003 paper [15].

The stated functionality goals of Alcatraz include

- application and operating system transparency,
- security yet application-friendliness,
- convenience and user-friendliness,
- preservation of system integrity (moreso than system confidentiality), and
- one-way isolation.

Some more precise technical claims made in the paper are

that

- Etrace (system call interposition) imposes from 10-100% overhead and
- Alcatraz (secure execution environment) imposes below 20% overhead

for the applications they studied.

2.1 Experimental Design

Hardware Setup. All experiments are performed in a CentOS 4.6 VMware virtual machine allocated 512 MB of memory with an ext3 file system. The underlying system is a MacBook Pro running Mac OS X v10.5.5 with a 2.6 GHz Intel Core 2 Duo processor and 4 GB of RAM.

Functionality. To evaluate the functionality of Alcatraz we use a qualitative methodology in the form of a series of case studies. In *Case Study 1* (A.1), we attempt to answer Functionality Questions 1 and 2 from Section **??** by running small, targeted tests by hand. In *Case Study 2* (A.2), we study these same questions using the more extensive POSIX File System Test Suite [11]. Finally, in *Case Study 3* (2.3.3) we run real applications in a SEE and study how well (and how successfully) they do so. We have relegated more lengthy summaries of these case studies along with detailed results to Appendix A; however, we analyze and summarize our findings in Section 2.3.

Performance. To evaluate the performance of Alcatraz we use a quantitative experimental approach. Specifically, we use a 2^2 factorial design and test each independent variable for statistical significance at $\alpha = 0.05$ using Analysis of Variance (ANOVA). If ANOVA shows there to be a significant effect, we determine which levels of a factor differ significantly using the Tukey-Kramer HSD test. The factorial design allows us the increase the power of our statistical tests by blocking on confounding variables (such as file size and record size) which would otherwise affect our results for our interesting independent variable, *System Type* a.k.a. system.

In both *Experiment 1* and *Experiment 2* an important independent variable is **System Type** = {bare, etrace, alcatraz}. In these experiments, *bare* means "interposition or isolation", *etrace* means "System call interposition only", and *alcatraz* means "etrace + full copy-on-write semantics and isolation". In *Experiment 1* we have the dependent variable **Performance (MB/s)**, and in *Experiment 2* we have a similar dependant variable **Execution Time (s)**.

Additionally in *Experiment 1* we add the following independent variables:

File Size (kB) = {64, 128, 256, 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144, 524288}
Record Size (kB) = {4, 8, 16, 32, 64}
Load Type = {Write, Re-write, Read, Re-Read, Random Read, Random Write}

More concretely, for this experiment we use the Iozone Filesystem Benchmark Revision 3.311 [18] and have it generate the above Files Sizes, Record Sizes, and Load Types. We actually had it generate record sizes all the way from 4 Bytes to File Size for each file size, but for ANOVA we chose a common subset of record sizes so we would have uniform sample sizes. This experiment addresses Performance Question 1 from Section **??**.

For *Experiment 2*, we instead add the independent variable **Application** = {bzip, tar, make, make-install} to examine more "representative" load types. This experiment addresses Performance Question 2.

2.2 Validity and Reproducibility

To leave no question about the interval validity of our case studies and experiments, we have been extremely explicit about what questions we are trying to answer and which evaluations address which questions. Concerning the external validity of our tests, we rely only on the *difference* in performance of various loads and applications and try not to rely on or place too much emphasis on any particular runtimes, though we do show some for illustrative purposes.

Our evaluations should also be very repeatable. We have listed all of our "by hand" tests in full in Appendix A, and used the publicly available Iozone and POSIX File System Test Suites for our automated tests. Alcatraz 0.6.4 [20] and Etrace 0.8.2 [21] are available from their authors' web sites at Stony Brook University. Even our environment, a CentOS 4.6 VMware image, was downloaded from [9], though we have added gnome-desktop and a few packages required by Etrace and Alcatraz.

2.3 Results

2.3.1 Case Study 1

The full results from this case study can be seen in Appendix A.1. This study was very illuminating, and we have found many examples of common operations that do not work (or do not work as expected), and a few ways of violating one-way isolation and affecting the integrity of the system outside the SEE.

First, some common file modifications do not work. Many operations that affect file metadata are broken, such as `chmod, chown, chgrp`. When these are attempted on files a user does not create himself within the SEE they are either ignored or explicitly "Operation not permitted."

Most file creation and deletion operations are supported, and a file `mv` is committed as a delete and a create. However, `mv`-ing an existing directory does not work—"cannot move...: Is a directory." The authors do note in their paper that a simple copy-on-write implementation does not extend to directory, but they offer solutions. These do not seem to be in the implementation.

Hard links cause Alcatraz a great deal of difficulty. When creating a hard link to a pre-existing file from within an SEE, Alcatraz appears to let you, and even lets you commit. However, it appears the file it creates is just a copy of the "link"-ed file, as `stat` shows that neither the original nor the

"link" actually have more than 1 link, and changes to one aren't reflected in both, neither before nor after the commit.

Alcatraz leaks around the edges in even some simple cases. For example, after creating a file "foo" in the SEE, when performing an `ls -l`, an error message "ls: foo: No such file or directory" is produced immediately before listing the directory correctly. `ls` also fails to list the `/` directory and the `/dev`, the latter probably because of efforts to restrict access to devices within the SEE. Some subitems of both are accessible however. You can `stat /`, `stat /dev`, `ls /etc` and `cat /dev/urandom` just fine, but you cannot, for example, `cat /dev/random`.

To note one last inconsistency, commit does not always work predictably. The stated commit policy is to discard changes "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". However, in the "conflict1" test this holds, while in the "conflict2" test the SEE changes overwrite the conflicting external changes. In a more devious example, Alcatraz appeared to let us write to `/proc/sys/fs/file-max`, but the changes failed on commit.

In addition to these transparency/consistency/user-friendliness problems, we have also managed to break one-way isolation. Alcatraz is successful at stopping many things that would thwart this, such as writing to sockets (network or UNIX), communicating with system services such as `dbus`, and killing outside processes (such as Alcatraz itself!). However, using an existing FIFO, writing to that FIFO from within an SEE is immediately visible to a reader outside the SEE. Additionally, though a malicious user cannot kill outside processes, he can `renice` them.

That being said, Alcatraz is research software and is not expected to have a bulletproof implementation. However, the large number of special-case ways we found to break transparency and even isolation makes this solution seem very inelegant, and it leads us to believe that it would be very hard to get completely right. At least the authors have not proved through this implementation that this is the right solution to the problem.

2.3.2 Case Study 2

The full results from this case study can be seen in Appendix A.2.

The POSIX File System Test Suite is used as part of the quality assurance process of the NTFS-3G NTFS driver for Linux. As such, it tests many file system corner cases for compliance. After successfully running the tests on the bare Linux VM as a sanity check, we ran the suite of tests inside Alcatraz run as superuser. As can be seen in the results for this test, Alcatraz does not do very well.

This suite specifically tests 12 system calls: `chflags`, `chmod`, `chown`, `link`, `mkdir`, `mkfifo`, `open`, `rename`, `rmdir`, `symlink`, `truncate`, and `unlink`. As this suite is not meant specifically for Alcatraz as our tests in *Case Study 1* were, these tests tell us nothing about the commit behavior of operations, only whether they appear to be functioning correctly from *within* the SEE. The tests mostly work by running these system calls and checking their return values.

Alcatraz causes the test suite to hang for 11/12 of the system calls, and once the offending subtests were removed, Alcatraz failed 20-80% of the tests for each system call (aside from `chflags`, the only test Alcatraz completely passed).

The `chflags` tests plays to Alcatraz's strengths. Every test consists of creating a file or directory, changing some flags, and then unlinking. Since this all happens within the SEE *to* files created in the SEE, Alcatraz does well.

A random sampling of its failings shows Alcatraz failing for a very wide assortment of reasons, with some repeat offenders including incorrect behavior on really long path names, not updating ctime correctly for some calls, and a large number of `chmod` errors (since many of the other tests besides `chmod` use `chmod` in the course of their tests).

The main takeaway from this case study, and what we should have perhaps known from *Case Study 1*, is that the Alcatraz implementation is not ready for as rigorous a test suite as this, as Alcatraz fails much simpler tests. Failing the POSIX File System Test Suite is further evidence of Alcatraz not meeting its goals of transparency and application-friendliness, and indicates trouble ahead for running real applications in Alcatraz.

2.3.3 Case Study 3

A common usage of a system like Alcatraz (if it were widely used) would be to test installation of new programs. In this case study we attempt to install and run a common program (Vim 7.2) in a directory local to the user, from within Alcatraz. This requires the following:

1. `bunzip vim-7.2.tar.bz2`
2. `tar -xf vim-7.2.tar`
3. `cd vim-7.2/; ./configure -prefix=<localdir>`
4. `make`
5. `make install`

In our test, all of these were successful from within the SEE except for the `./configure ...` step. Suspiciously in the ACSAC 2003 paper [15] all of these above steps are tested except for the `./configure` step. We also have to skip the `config` step in Experiment 3 where we run similar performance tests.

After completing these steps (cheating by performing `configure` outside the SEE), we were able to run Vim 7.2 from within Alcatraz.

2.3.4 Experiment 1

The full results from this case study can be seen in Appendix A.3.

In this experiment, we examine each Load Type and see if System Type = {bare, etrace, alcatraz} is a significant factor. As you can see, for all Load Types the p-value for System Type is $< .0001$, so we can say that System Type is a significant factor for Performance with 95% confidence for all Load Types.

Then looking at the Tukey HSD tests, we can say with 95% confidence that performance for all three System Type levels (bare, etrace, and alcatraz) are different for the `read` and `write` Load Types. For the other Load Types, the bare system performs significantly better than the other two, but etrace and alcatraz don't perform significantly differently at $\alpha = 0.05$.

When performing our analysis we ran across three significantly outlying points (out of 1260 for this test) that we retook. We believe these were anomalies of the Iozone output, however.

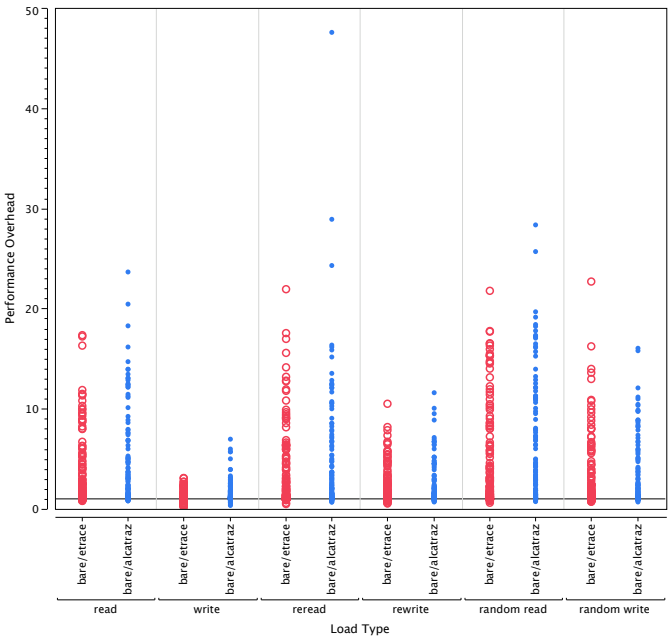


Figure 1: Performance Overhead

Figure 1 is a plot of the performance overheads we calculate in this experiment. This figure shows the performance overhead (bare performance / etrace performance, and bare performance / alcatraz peformance) for the same file system loads next to each other. The overhead for a point is normalized with an "apples to apples" normalization—for example, the alcatraz performance measurement for the "random write" load, 64 MB record size, and 512 MB file size is divided by the bare performance measurement for those same values. The grey line near the bottom demarks 1, where the etrace or alcatraz performance is exactly the same as the bare performance for the same values of the independent variables.

This side-by-side scatterplot tells us things that an average would not. It shows the wide range of overheads we see, but yet we can see that "write" and "rewrite" actually suffer the least from overhead even though "write" is the slowest performing file system load in an absolute sense. Ignoring the extremely high "reread" outlier (corresponding to a 512 kB file size and an 8 kB record size) the other file system loads appear to have similar ranges.

Our overhead values for etrace range from less than 0.2 to greater than 22, and our overhead values for alcatraz range from < 0.4 to over 47. Though the extreme values are likely anomalies and testing noise, this is still a much greater range than the (1, 2) range displayed in the ACSAC 2003 [15] paper.

2.3.5 *Experiment 2*

The full results of this experiment can be seen in Appendix A.4.

This experiment was run simultaneously with Case Study 3. Case Study 3 was intended to determine if you could run common applications in Alcatraz, while this experiment tests whether the **system** independent variable (isolation) is a significant factor in real applications. As you can see in the Appendix, for all four application types, **system** is a significant factor at $\alpha = 0.05$.

In the most computationally-bound test, `bzip`, the bare system performs significantly differently from alcatraz, but neither perform significantly differently from etrace. However, in all other tests (which are more file-system intensive) both alcatraz and etrace perform significantly differently, and in the most file-system intensive tests (`make` and `makeinstall`), all three perform significantly differently.

3. ONE-WAY ISOLATION ON A COMMODITY PLATFORM

Mac OS X is, by far, the UNIX-like OS with the most running hosts. Additionally, many customer segments that Mac OS X targets are end users that do not have the technical expertise (or motivation) to analyze the behavior of a potentially untrustworthy application. Thus, one way isolation would be a valuable addition to Mac OS X's capabilities.

Alcatraz is written on top of Etrace, so porting Alcatraz to Mac OS X implies porting Etrace to Mac OS X.

To port Etrace to a new platform, one extends the class `ArchDep` with concrete implementations of tracing functions to attach to a process, read/write process state and memory, wait for the next system call of the process, and so forth. Etrace is essentially an extended object-oriented wrapper of the UNIX `ptrace` facility.

Our attempts to port Alcatraz and Etrace to Mac OS X v10.5 were unsuccessful, because Apple has deliberately obstructed invasive system tracing on Mac OS X. This is discussed further in section 5.

4. INTERPOSING MAC OS X SYSTEM CALLS

We analyzed the path of a system call from user code through user libraries [6, 7] and the kernel [8] to locate "hooks" into this system call path. The details of the system call sequence are in appendix B.

Here, we briefly recount the various mechanisms one might use to insert an Alcatraz-like facility into a UNIX system, and how Mac OS X's current state prevents their use.

4.1 ptrace

`ptrace` is what the existing Linux variant of Etrace is based upon, so it is a natural starting point. The Linux-specific `PTRACE_SYSCALL` request is not implemented in Mac OS X, not surprisingly. However, it was a surprise to find that the read/write memory and process state requests had been removed. (Those requests have been in `ptrace` since it was added to UNIX 6th edition in 1975.) Mac OS X `ptrace` is only capable of single-stepping or resuming the traced process.

4.2 Other kernel level trace facilities

Mac OS X has two other tracing facilities that are informed of system calls (as our system call analysis determined). These are the BSD `ktrace` and the audit subsystem. Further analysis of these determined the were both unsuitable for because they were not capable of altering system call arguments. Additionally, `ktrace` was removed from Mac OS X v10.5; and the audit subsystem does not receive control directly—events are passed to it via a queue.

4.3 Kernel extension

Another possibility to gain access to the system call path would be to write a kernel extension (KEXT) [3, 1] which redirects entries in the system call dispatch table, `sysent`, to itself. This method of interposition was feasible on earlier versions of Mac OS X, but as part of a new kernel–KEXT interface introduced in Mac OS X v10.4, the `sysent` symbol was hidden from KEXTs. Unfortunately, the new kernel-KEXT interface does not provide for system call interposition [5].

There are published means of "guessing" the location of the `sysent` table based on other symbols, but we rejected use of these as too brittle and potentially quite unsafe.

4.4 DTrace

Sun's extensive DTrace facility [10] was ported to Mac OS X v10.5. Apple added an elaborate user interface, called Instruments to DTrace. DTrace and Instruments provide wide-ranging, expressive, and detailed tracing functions for user and kernel mode code, including system call tracing. However, the same obstacles that we found with `ktrace` apply to DTrace — there is no way to modify system call arguments, only read them.

Interestingly, DTrace traces system calls using the method we investigated previously: interposing by redirecting system call entries in the `sysent` table. DTrace has access to `sysent` since it is complied into the kernel before the symbol is stripped.

4.5 DYLD_INSERT_LIBRARIES

Failing to find a viable kernel-level interposition method, we investigated several possibilities on the user mode side of the system calls. This approach is only justifiable under the assumption that the program being isolated is not attempting to subvert the isolation; i.e., the suspect program has a very limited level of malevolence.

Mac OS X's dynamic linker/loader, `dyld` [4], provides a function designed to override modules in dynamic libraries, invoked by defining the environment variable `DYLD_INSERT_LIBRARIES`. Unfortunately, this has several shortcomings: 1) dyld specifically disables this for `setuid` and `setgid` programs. 2) It only works for "flat namespace" images. Modern Mac OS X images are not flat, but "two level". One can force two-level images to be treated as flat, but that causes many programs to fail. 3) It is fairly easy to defeat.

4.6 libSystem

Two options we briefly considered were supplying a replacement version of `libSystem` or `dyld`. Examination of kernel source code revealed that these are both loaded by explicit references to their file system location, not by using any search paths, so our replacement files would have to overlay the originals. Since modifying core system files is the type of behavior this project is trying to prevent, we viewed it as hypocritical to take this approach and therefore discarded it.

5. SECURITY TENSIONS IN A COMMERCIAL OS

Paradoxically, in this work, our progress was impeded by OS design decisions made to serve other security goals. Mac OS X v10.5's kernel has been secured, as described previously in the paper, to prevent "hooking" of system calls presumably for two reasons:

1. To increase the difficulty of writing malware (viruses, rootkits, etc.).
2. To increase the difficulty of subverting digital rights management (DRM) schemes, such as those embedded in iTunes.

Providing OS extensibility in a manner that does not also open attack vectors is a problem for future work. We address DRM below.

5.1 DRM's Needs

DRM entails a software publisher relying on invariants established in its software while that software is running on platforms under the control of others.

Assuming "a person having physical access to the hardware has absolute control of the machine", then DRM cannot be fully solved. There will always be some level at which the DRM can be subverted. The largest deployer of DRM in the world, Apple, has said "DRMs haven't worked, and may never work" [14].

However, publishers are satisfied with making DRM difficult to subvert rather than absolutely secure.

Hereinafter, let us assume the DRM of concern is media content DRM, which we will define as protection of passive (non-executable) data which is transferred to the user's environment in a protected form, then transformed and presented to the user using the services of the user's environment. In this case, DRM focuses on maintaining the data in its protected form (usually encrypted), enforcing access control, and then transforming and presenting the data only across a trusted path. (This path recently has been extended to video display devices, by using the HDCP scheme[12].)

Establishment of this trusted transform-and-present path means that many conventional operating system functions, such as I/O redirection, must be disabled or incapacitated. This is the source of the tension — users want the full functions of their environment, but DRM-using publishers want to cripple users' functions. In other words, does the platform's provision of "trusted components" mean trusted by the user or trusted by third parties? When the user's and third parties' wishes conflict, whom does the platform serve? This raises ethical and legal questions, which are not examined here [17].

Computing has a number of solutions to the secure information transfer problem, for example TLS/SSL for secure information transfer over TCP/IP. In this situation, the problem is to move data from one trusted endpoint to another while not disclosing the information to other (untrusted) observers. In contrast to this problem, DRM's purpose is to reveal the information to the (untrusted) user, but to do so with a number of constraints. For example, iTunes ensures that the movie the user has purchased is only viewed from the purchaser's computer or iPod. This attempt to control disclosure leads to problems such as those that prompted HDCP. A user could play an iTunes movie into a digital video recorder (software or hardware), and end up with a near perfect copy of the protected movie. HDCP attempts to avert this by requiring authentication of the display hardware.

Since DRM software relies on the OS to communicate with the display hardware, it requires a trusted path through the OS. This is where we cross paths with DRM. This project, and many other systems projects require the ability to redirect I/O and otherwise change the behavior of systems functions. DRM attempts to "hard code" the endpoints of its output, so as to prevent misuse of its content while still allowing users to use the content in certain ways.

5.2 Current State in Mac OS X

A threat to DRM is normal debugging tools. This threat is twofold:

1. The protected data could easily be inspected during its transit through the trusted path using system tracing tools such as `dtruss`.
2. The protection scheme, which generally uses encryption, can be inspected and its private keys compromised using common debugging tools such as `gdb`.

Mac OS X v10.3 introduced the `PT_DENY_ATTACH` request to the `ptrace` system call as a defense to the second threat. This is known to be an imperfect defense, which Apple calls a "cat-and-mouse game" of private key breach and protection [14]. To prevent simple subversion of `PT_DENY_ATTACH`, Apple has removed the ability to hook system calls. In more recent versions of Mac OS X, system call tracing is prohibited on all processes that have requested `PT_DENY_ATTACH`.

Basically, `PT_DENY_ATTACH` has become a flag indicating that the process wants to exempt itself from multiple standard system capabilities deemed threatening to DRM. However, these are the capabilities needed for debugging and security analysis. And ironically, programs using DRM have proven to be one of the least trustworthy (by the user) categories of software. (For example, review the security and privacy breaches committed by Real Networks and Sony BMG.) A broad "you can't look here; you must just trust me" flag is not an acceptable solution.

5.3 A Way Out?

Observe that the present solution is overconstraining relative to the needs. The needs can be specified as:

1. Protection secrets (i.e., private keys) must remain confidential — not known outside of the owning process.
2. The data streaming through the trusted transform-and-present path must only end up on certain "allowable" types of devices.

For the first requirement, `PT_DENY_ATTACH` is unnecessarily broad. While debugger commands that inspect the state of certain threads and the process address space should be disabled during the presence of private keys, other debugger commands, such as the ability to pause process execution need not be disabled, Additionally, the part of the process needing protection is quite small compared to the whole application.

The needed confidentiality could be implemented by extending the basic mandatory access control scheme now present in most OSes [22, 16] (including Mac OS X v10.5) with 1) labels, 2) the ability to label address spaces and thread state, and 3) a policy regarding these labels that is not reconfigurable by the user.

In the second requirement, the allowable devices are those that do not record the data they are presenting. We will call these ephemeral endpoints. Examples are video displays, speakers, and so forth. We will call recording endpoints stable endpoints.

If processes can declare output streams as requiring ephemeral endpoints, and trust the operating system to enforce this requirement, then other activities of the process could be treated "normally". Note that this level of trust in the OS would be no higher than the processes presently rely upon for their output streams.

An implementation approach is similar to the first requirement: extend existing mandatory access control facilities with 1) labels, 2) the ability to label output streams, and 3) non-reconfigurable policies. With this, the OS could even allow user-controlled redirection of output, but still enforce the requirement that DRMed output streams not flow to a stable endpoint.

6. RELATED WORK

Our original plan was to develop an isolation environment very much like Alcatraz, until we discovered it had already been done in Alcatraz and in the systems listed below.

Solitude. Solitude [13] is a system very similar to Alcatraz out of the University of Toronto, possibly inspired by Alcatraz. In addition to a one-way isolation file system, Solitude adds the ability to do taint-tracking on files after the user

has committed them to the base file system (in case something still goes wrong later).

We first attempted to use this system as the basis of our work, but when we finally got the source code it was in a non-working state, was essentially one giant C source file, and when the providing professor had last tried it, it had "hosed" his file system. Needless to say, we chose to base our work on Alcatraz due to its availability, its clean architecture, and its ability to compile.

Alcatraz 2?. The group from Stony Brook University that produced Alcatraz described what seems to be a kernel implementation of it in an NDSS paper in 2005 [23]. We originally assumed that was the system in the current release of Alcatraz, but we later found that not to be the case. After corresponding with the Alcatraz group, we were told that the NDSS system was a "different" system that nobody was keeping track of anymore.

Unfortunately it seemed to be a more advanced version of Alcatraz that addressed many of its problems and provided more advanced commit algorithms.

ReVirt. The ReVirt system out of the University of Michigan was one of the first systems we looked at. ReVirt is a virtual machine solution to this problem of isolation—their system runs underneath the operating system. We rejected this system because it does not solve the problem we were examining. ReVirt provides isolation at the operating system level (not application), and it focuses on logging, roll-back, and replay of intrusions for an entire VM.

7. FUTURE WORK

This project has suggested several avenues of future work to us. The first, and most obvious, is to *fix Alcatraz*. We managed to break the current system (at least its current implementation) in a number of ways, and the POSIX File System Test Suite uncovers many other problems. It could still be a useful system if it really provided one-way isolation and it was not too difficult to run experimental applications in it.

More experimental work could also be done. One avenue for this is to evaluate Solitude in the same way that Alcatraz was evaluated, and then additionally compare their performance they way we originally intended to compare a Linux and OS X version of Alcatraz. This could also be done with the NDSS 2005 system if we could get source code—there would be a clear goal when evaluating this system, as it is intended to be an improved successor to Alcatraz. Does it significantly improve performance and usability?

A more ambitious idea was suggested to us by our Design Review team. A system like Alcatraz could be used to allow a user to masquerade as having (mainly write) root permissions to an application that insists on having root access. Some applications could be fooled well enough to run, and one of our reviewers said he had a need for something like this right now.

Another possible future project is to implement a new one-way program isolation environment based on the facilities present in Mac OS X v10.5 and in the OS's apparent direction. This would likely use the `kauth` or Seatbelt access control functions and a VFS plug-in. Note that Mac OS X's VFS does not support stacking file systems [2], but MacFUSE does.

8. CONCLUSIONS

We continue to believe that one-way isolation of user programs is an important security feature for modern operating systems.

The system examined here, Alcatraz, is not the solution to providing this feature. The performance overhead can be unacceptable in routine situations and file system operation semantics are not preserved. However, Alcatraz demonstrates the feasibility of one-way isolation of user programs.

In our attempts to port Alcatraz to Mac OS X, we observed that the various security goals present in a modern commodity operating system can be at odds with each other, impeding overall progress on adding security facilities. DRM is a factor in this, but not the only driver of these conflicting goals. In this case, DRM and one-way isolation are not mutually exclusive, and we have proposed one alternative to preserving system adaptability properties while still serving the trusted platform needs of DRM.

A substantially different approach to implementing one-way isolation is probably feasible, as discussed in the Future Work section.

We encourage continued work on integration of one-way isolation into commodity platforms.

9. ACKNOWLEDGMENTS

We would like to thank Professor R. Sekar from Stony Brook University and Professor Ashvin Goel from the University of Toronto for providing us with source code. We would also like to thank Drake Dowsett, Joel Hestness, and Professor Mike Dahlin for reviewing earlier iterations of this work.

10. REFERENCES

[1] Apple Inc. Kernel Programming Guide. http://developer.apple.com/documentation/Darwin/Conceptual/KernelProgramming/, 2006.

[2] Apple Inc. Technical Q&A QA1242: Developing for VFS. http://developer.apple.com/qa/qa2001/qa1242.html, 2006.

[3] Apple Inc. Kernel Extension Programming Topics. http://developer.apple.com/documentation/Darwin/Conceptual/KEXTConcept/, 2007.

[4] Apple Inc. dyld-96.2. Darwin Source Code. http://www.opensource.apple.com/darwinsource/10.5.5/dyld-96.2/, 2008.

[5] Apple Inc. Kernel Framework Reference. http://developer.apple.com/documentation/Darwin/Reference/KernelIOKitFramework/, 2008.

[6] Apple Inc. Libc-498.1.1. Darwin Source Code. http://www.opensource.apple.com/darwinsource/

10.5.5/Libc-498.1.1/, 2008.

[7] Apple Inc. Libsystem-111.1.1. Darwin Source Code. http://www.opensource.apple.com/darwinsource/10.5.5/Libsystem-111.1.1/, 2008.

[8] Apple Inc. xnu-1228.7.58. Darwin Source Code. http://www.opensource.apple.com/darwinsource/10.5.5/xnu-1228.7.58/, 2008.

[9] S. Bellwood. Some VMware Images. http://www.thoughtpolice.co.uk/vmware/#centos4.6, Referenced December 2008.

[10] B. Cantrill, M. Shapiro, and A. Leventhal. Dynamic instrumentation of production systems. In *USENIX Annual Technical Conference*, pages 15–28, 2004.

[11] P. J. Dawidek and NTFS-3G Technology Ltd. POSIX File System Test Suite, Stable release 20080816. http://www.ntfs-3g.org/pjd-fstest.html, Released August 16, 2008.

[12] Intel Corporation. High-bandwidth Digital Content Protection System. http://www.digital-cp.com/, 2006.

[13] S. Jain, F. Shafique, V. Djeric, and A. Goel. Application-level isolation and recovery with solitude. In *Proceedings of the 3rd ACM SIGOPS/EuroSys European Conference on Computer Systems 2008*, pages 95–107. ACM New York, NY, USA, 2008.

[14] S. Jobs. Thoughts on Music. http://www.apple.com/hotnews/thoughtsonmusic/, Feb 2007.

[15] Z. Liang, V. N. Venkatakrishnan, and R. Sekar. Isolated Program Execution: An Application Transparent Approach for Executing Untrusted Programs. In *Proceedings of Annual Computer Security Applications Conference*, 2003.

[16] P. Loscocco and S. Smalley. Integrating flexible support for security policies into the linux operating system. In *Proceedings of the FREENIX Track: 2001 USENIX Annual Technical Conference*, pages 29–42, Berkeley, CA, USA, 2001. USENIX Association.

[17] D. K. Mulligan and A. K. Perzanowski. The Magnificence of the Disaster: Reconstructing the Sony BMG Rootkit Incident. *Berkeley Technology Law Journal*, 22:1157, 2007.

[18] W. D. Norcott. Iozone Filesystem Benchmark. http://www.iozone.org/, Last Updated: Oct 28th 16:00:00 EST 2006.

[19] Secure System Lab, Stony Brook University. Etrace. Technical report, 2005.

[20] Secure System Lab, Stony Brook University. Alcatraz. http://www.seclab.cs.sunysb.edu/alcatraz/, Referenced December 2008.

[21] Secure System Lab, Stony Brook University. Etrace. http://www.seclab.cs.sunysb.edu/etrace/, Referenced December 2008.

[22] R. Spencer, S. Smalley, P. Loscocco, M. Hibler, D. Andersen, and J. Lepreau. The flask security architecture: system support for diverse security policies. In *SSYM'99: Proceedings of the 8th conference on USENIX Security Symposium*, pages 11–11, Berkeley, CA, USA, 1999. USENIX Association.

[23] W. Sun, Z. Liang, R. Sekar, and V. Venkatakrishnan. One-way Isolation: An Effective Approach for Realizing Safe Execution Environments. In *Proceedings of the 12th Annual Symposium on Network and Distributed System Security*, volume 12, pages 265–278. Internet Society, 2005.

APPENDIX

A. FULL EXPERIMENTAL RESULTS

A.1 Case Study 1: Targeted Study Tests

W? = Works/Fails as it should?
C? = Commits/Doesn't as it should?
B? = Broken?
Possible values are Y = yes, N = no, ~ = partial

NO FS MODIFICATION

Test	W?	C?	B?	Notes
ls	~	Y	Y	Can't ls '/', '/dev', '/tmp/Alcatraz.5005', works for most. Just broken.
stat	Y	Y		Works even on '/' and '/dev'
cat	Y	Y		'cat /usr/share/dict/words'
grep	Y	Y		'grep foo /usr/share/dict/words'
find	Y	Y		'find /home/notroot -name *.txt -print'
pipe	Y	Y		'cat /usr/share/dict/words | grep foo | head'
ping	N	Y		Fails as advertised. 'ping www.google.com'
nc	N	N		'nc localhost 3000' Fails as advertised with "Operation not permitted" with server running
unix-socket	N	N		Fails as advertised. "Operation not permitted" on socket connect to socket created in /tmp
dev/zero	N	N		'cat /dev/zero' Operation not permitted as advertised.
dev/random	N	N		'cat /dev/random' Operation not permitted as advertised.
dev/urandom	Y	Y	Y	'cat /dev/urandom' Works? Inconsistent.
kill alc.	N	Y		'kill 13097', 'kill -9 13097', 'killall alcatraz'. Fails as advertised
kill etrace	N	Y		'kill 13097', 'kill -9 13097', 'killall etrace'. Fails as advertised.
kill out	N	Y		'kill 13097', 'kill -9 13097', 'killall top'. Fails as advertised.
kill within	Y	Y		'kill 14271' where 14271 is top run within SEE
renice in	Y	Y		'renice +1 14271' to process inside of SEE
renice out	Y	Y	Y	'renice +1 14524' to process outside of SEE. Breaks one-way isolation.
su	N	N	Y	Produces ptrace error and "Operation not permitted" and "su: incorrect password". Not 'transparent.'
xclock	Y	Y		'xclock -display :2' after outside 'Xnest -ac :2'. Not really transparent.

FS MODIFICATION

Test	W?	C?	B?	Notes
create+ls	~		Y	'touch foo; ls -l' prints the error message 'ls: foo: No such file or directory' before proceeding to correctly list the directory (including foo).
touch	Y	Y		'touch foo' and 'touch /tmp/foo'
cp	Y	Y		'cp foo foo2' w/foo existing
mkdir	Y	Y		'mkdir foodir' and 'mkdir /tmp/foodir'
dirmod1	Y	Y		Using existing directory 'foo', add a file outside the see and a different file inside the SEE.
mkfifo	Y	Y		'mkfifo foofo' and 'mkfifo /tmp/foofo'
symlink	Y	Y		'ln -s ../foo' and 'echo "bar" >> foo"
hardlink	~	~	Y	Using existing file '../foo', 'link ../foo bar'. Appears to work, contents of both agree. However 'stat ../foo' and 'stat ../bar' both show 'Links: 1' Modifying the contents of either does not affect the other. Appears to commit correctly, and the new link appears. However, both 'stat' on both files still shows "Links: 1" and modifying one does not modify the other.

```
touch+chmod Y   Y      'touch foo; chmod 700 foo'
chmod       N   N   Y  'chmod 700 foo' with foo already existing.  Ignores
                       command.  Broken.
chown       N   N   Y  'chown root foo' produces "Operation not permitted".
                       Broken.
chgrp       N   N   Y  'chgrp root foo' produces "Operation not permitted".
                       Broken.
overwrite   Y   Y      'cat /usr/share/dict/words > foo' w/foo existing
append      Y   Y      'echo "test" >> foo' w/foo existing
file rename Y   Y      'mv foo bar'.  Commits as a delete plus a create.
dir rename  N   N   Y  'mv foodir bardir' to existing dir foodir fails with "mv:
                       cannot move ‘foodir’ to ‘bardir’: Is a directory
conflict1   Y   Y   ~  Using existing file 'foo', write to it within SEE (causes
                       copy-on-write), write to it outside SEE (writes to
                       original).  Shows modification at commit time, I hit
                       commit, says "/home/notroot/foo Modified! Discarding
                       changes..." and throws away modifications from within
                       SEE. Inconsistent with conflict2!
conflict2   Y   Y   ~  Using existing directory 'foodir', create a file
                       'foodir/bar' within the SEE, add contents, create a file
                       'foodir/bar' outside the SEE with different contents.
                       Commits fine and overwrites changes from outside SEE.
                       Inconsistent with conflict1!
fifo-comm   Y   N   Y  Using existing fifo 'foofo', from inside SEE 'cat
                       /usr/share/dict/words > foofo', and outside SEE 'cat
                       foofo'.  Breaks one-way isolation.  Shows nothing at
                       commit time.
/proc       ~   ~   Y  'echo 100000 > /proc/sys/fs/file-max'.  Works within SEE
                       and is not visible outside SEE.  Shows up in changes
                       summary, but error on commit "mv: inter-device move
                       failed...;unable to remove target: Operation not
                       permitted".  Broken implementation.
dbus        N   N      in python, 'import dbus;  d = dbus.SessionBus()' fails on
                       connect to UNIX socket.  As advertised, but makes life
                       difficult for some UNIX apps.
```

A.2 Case Study 2: POSIX File System Test Suite

```
Tests that hang:

chflags - all pass
chmod   - 06 hangs
chown   - 06 hangs on 2/6
link    - 00 hangs on 29/82
        - 10 hangs on 11/14
        - 08 hangs on 2/10
mkdir   - 07 hangs on 2/6
mkfifo  - 07 hangs on 2/6
open    - 12 hangs on 2/6
rename  - 00 hangs on 26/79
        - 11 hangs on 2/10
rmdir   - 05 hangs on 2/6
symlink - 07 hangs on 2/6
truncate- 07 hangs on 2/6
unlink  - 00 hangs on 19/55
        - 07 hangs on 2/6


Summarized output of tests (skipping the above tests that hang):

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/chflags
All tests successful.
Files=14, Tests=14,  2 wallclock secs ( 0.56 cusr +  0.58 csys =  1.14 CPU)
```

```
sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/chmod
Failed 7/11 test scripts, 36.36% okay. 30/136 subtests failed, 77.94% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/chown/
Failed 6/10 test scripts, 40.00% okay. 56/227 subtests failed, 75.33% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/link
Failed 8/15 test scripts, 46.67% okay. 40/103 subtests failed, 61.17% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/mkdir
Failed 6/12 test scripts, 50.00% okay. 21/99 subtests failed, 78.79% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/mkfifo/
Failed 8/12 test scripts, 33.33% okay. 28/99 subtests failed, 71.72% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/open
Failed 9/23 test scripts, 60.87% okay. 70/218 subtests failed, 67.89% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/rename/
Failed 18/19 test scripts, 5.26% okay. 7/9 subtests failed, 22.22% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/rmdir/
Failed 8/15 test scripts, 46.67% okay. 26/111 subtests failed, 76.58% okay.

sh-3.00#  prove -r /home/notroot/src/pjd-fstest-20080816/tests/symlink/
Failed 8/12 test scripts, 33.33% okay. 28/85 subtests failed, 67.06% okay.

sh-3.00#  prove -r /home/notroot/src/pjd-fstest-20080816/tests/truncate/
Failed 5/14 test scripts, 64.29% okay. 16/85 subtests failed, 81.18% okay.

sh-3.00# prove -r /home/notroot/src/pjd-fstest-20080816/tests/unlink
Failed 6/12 test scripts, 50.00% okay. 32/86 subtests failed, 62.79% okay.
```

A.3 Experiment 1: Targeted File System Loads

A.3.1 Load Type = read

Response Performance (MB/s)

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	105	183488605	1747511	31.1094
Error	104	5841993	56173	**Prob > F**
C. Total	209	189330597		<.0001

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
System Type	2	2	119067480	1059.828	<.0001
File Size	13	13	21291907	29.1570	<.0001
Record Size	4	4	18559831	82.6012	<.0001
System Type*File Size	26	26	16614255	11.3757	<.0001
System Type*Record Size	8	8	2750626	6.1209	<.0001
File Size*Record Size	52	52	5204506	1.7818	0.0065

Effect Details, System Type

Least Squares Means Table

Level	Least Sq Mean	Std Error	Mean
bare	2103.4493	28.327928	2103.45
etrace	578.6032	28.327928	578.60
alcatraz	442.3721	28.327928	442.37

LSMeans Differences Tukey HSD

$\alpha = 0.050$ $Q = 2.37776$

Level				Least Sq Mean
bare	A			2103.4493
etrace		B		578.6032
alcatraz			C	442.3721

Levels not connected by same letter are significantly different.

A.3.2 Load Type = write

Response Performance (MB/s)

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	105	4612831.9	43931.7	20.8454
Error	104	219180.3	2107.5	**Prob > F**
C. Total	209	4832012.2		<.0001

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
System Type	2	2	503464.6	119.4458	<.0001
File Size	13	13	2231522.9	81.4497	<.0001
Record Size	4	4	1118906.8	132.7290	<.0001
System Type*File Size	26	26	178262.7	3.2533	<.0001
System Type*Record Size	8	8	140504.9	8.3336	<.0001
File Size*Record Size	52	52	440169.8	4.0165	<.0001

Effect Details, System Type

Least Squares Means Table

Level	Least Sq Mean	Std Error	Mean
bare	288.78884	5.4870019	288.789
etrace	207.87229	5.4870019	207.872
alcatraz	171.66303	5.4870019	171.663

LSMeans Differences Tukey HSD

$\alpha = 0.050$ $Q = 2.37776$

Level				Least Sq Mean
bare	A			288.78884
etrace		B		207.87229
alcatraz			C	171.66303

Levels not connected by same letter are significantly different.

A.3.3 Load Type = reread

Response Performance (MB/s)

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	105	264617977	2520171	15.5527
Error	104	16852284	162041	**Prob > F**
C. Total	209	281470261		<.0001

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
System Type	2	2	164571184	507.8066	<.0001
File Size	13	13	31985259	15.1838	<.0001
Record Size	4	4	21489402	33.1542	<.0001
System Type*File Size	26	26	32194573	7.6416	<.0001
System Type*Record Size	8	8	5022139	3.8741	0.0005
File Size*Record Size	52	52	9355419	1.1103	0.3216

Effect Details, System Type

Least Squares Means Table

Level	Least Sq Mean	Std Error	Mean
bare	2398.1077	48.113140	2398.11
etrace	561.2983	48.113140	561.30
alcatraz	481.6431	48.113140	481.64

LSMeans Differences Tukey HSD

$\alpha = 0.050$ $Q = 2.37776$

Level			Least Sq Mean
bare	A		2398.1077
etrace		B	561.2983
alcatraz		B	481.6431

Levels not connected by same letter are significantly different.

A.3.4 Load Type = rewrite

Response Performance (MB/s)

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	105	32101803	305731	10.3688
Error	104	3066528	29486	**Prob > F**
C. Total	209	35168332		<.0001

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
System Type	2	2	12161889	206.2326	<.0001
File Size	13	13	8979330	23.4254	<.0001
Record Size	4	4	3542176	30.0328	<.0001
System Type*File Size	26	26	3577689	4.6668	<.0001
System Type*Record Size	8	8	1318706	5.5904	<.0001
File Size*Record Size	52	52	2522013	1.6449	0.0162

Effect Details, System Type

Least Squares Means Table

Level	Least Sq Mean	Std Error	Mean
bare	851.67259	20.523802	851.673
etrace	358.32091	20.523802	358.321
alcatraz	325.59560	20.523802	325.596

LSMeans Differences Tukey HSD

$\alpha = 0.050$ $Q = 2.37776$

Level			Least Sq Mean
bare	A		851.67259
etrace		B	358.32091
alcatraz		B	325.59560

Levels not connected by same letter are significantly different.

A.3.5 *Load Type = random read*

Response Performance (MB/s)

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	105	241057911	2295790	31.0379
Error	104	7692600	73967	**Prob > F**
C. Total	209	248750511		<.0001

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
System Type	2	2	158971146	1074.604	<.0001
File Size	13	13	22185792	23.0723	<.0001
Record Size	4	4	24154012	81.6375	<.0001
System Type*File Size	26	26	24966985	12.9823	<.0001
System Type*Record Size	8	8	4839580	8.1786	<.0001
File Size*Record Size	52	52	5940394	1.5444	0.0309

Effect Details, System Type

Least Squares Means Table

Level	Least Sq Mean	Std Error	Mean
bare	2226.3798	32.506551	2226.38
etrace	419.0594	32.506551	419.06
alcatraz	344.5993	32.506551	344.60

LSMeans Differences Tukey HSD

$\alpha = 0.050$ $Q = 2.37776$

Level			Least Sq Mean
bare	A		2226.3798
etrace		B	419.0594
alcatraz		B	344.5993

Levels not connected by same letter are significantly different.

A.3.6 Load Type = random write

Response Performance (MB/s)

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio
Model	105	63637110	606068	22.6845
Error	104	2778597	26717	**Prob > F**
C. Total	209	66415707		<.0001

Effect Tests

Source	Nparm	DF	Sum of Squares	F Ratio	Prob > F
System Type	2	2	30009465	561.6116	<.0001
File Size	13	13	14302134	41.1780	<.0001
Record Size	4	4	4839567	45.2850	<.0001
System Type*File Size	26	26	11156234	16.0602	<.0001
System Type*Record Size	8	8	1091905	5.1086	<.0001
File Size*Record Size	52	52	2237804	1.6107	0.0202

Effect Details, System Type

Least Squares Means Table

Level	Least Sq Mean	Std Error	Mean
bare	1093.6324	19.536515	1093.63
etrace	307.1972	19.536515	307.20
alcatraz	277.0948	19.536515	277.09

LSMeans Differences Tukey HSD

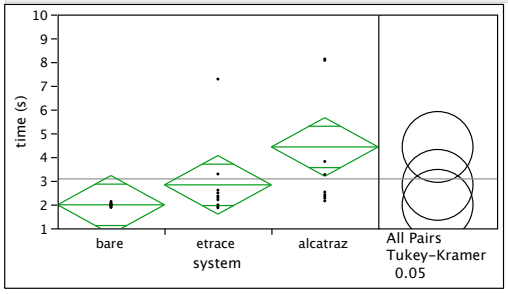
$\alpha = 0.050$ $Q = 2.37776$

Level			Least Sq Mean
bare	A		1093.6324
etrace		B	307.1972
alcatraz		B	277.0948

Levels not connected by same letter are significantly different.

A.4 Experiment 2: Real Applications

Oneway Analysis of time By system app=bzip



Oneway Anova

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio	Prob > F
system	2	30.63766	15.3188	4.2427	0.0250*
Error	27	97.48711	3.6106		
C. Total	29	128.12477			

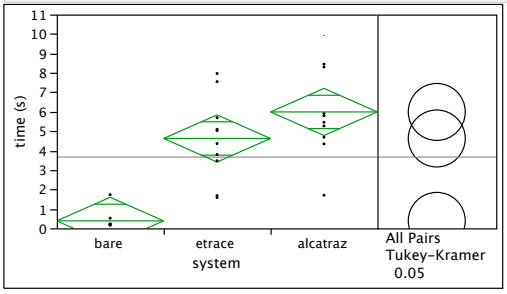
Means Comparisons

Comparisons for all pairs using Tukey-Kramer HSD

Level		Mean
alcatraz	A	4.4252000
etrace	A B	2.8289000
bare	B	1.9886000

Levels not connected by same letter are significantly different.

Oneway Analysis of time By system app=tar



Oneway Anova

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio	Prob > F
system	2	170.91495	85.4575	24.4585	<.0001*
Error	27	94.33747	3.4940		
C. Total	29	265.25243			

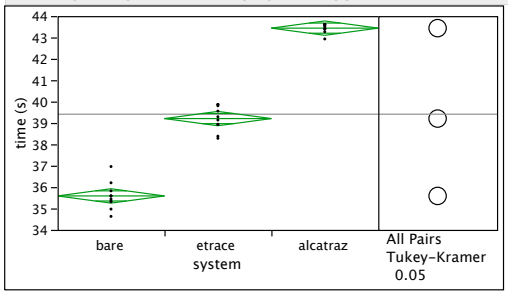
Means Comparisons

Comparisons for all pairs using Tukey-Kramer HSD

Level		Mean
alcatraz	A	5.9885000
etrace	A	4.6228000
bare	B	0.3824000

Levels not connected by same letter are significantly different.

Oneway Analysis of time By system app=make



Oneway Anova

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio	Prob > F
system	2	308.64214	154.321	576.2974	<.0001*
Error	27	7.23007	0.268		
C. Total	29	315.87221			

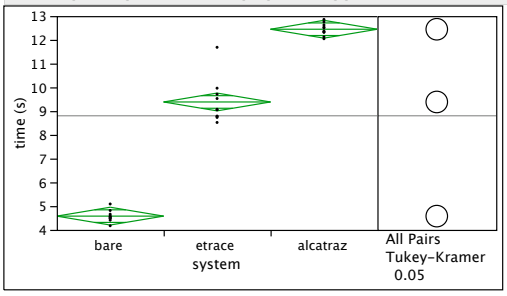
Means Comparisons

Comparisons for all pairs using Tukey-Kramer HSD

Level		Mean
alcatraz	A	43.435700
etrace	B	39.205400
bare	C	35.586900

Levels not connected by same letter are significantly different.

Oneway Analysis of time By system app=makeinstall



Oneway Anova

Analysis of Variance

Source	DF	Sum of Squares	Mean Square	F Ratio	Prob > F
system	2	314.60247	157.301	471.7454	<.0001*
Error	27	9.00302	0.333		
C. Total	29	323.60549			

Means Comparisons

Comparisons for all pairs using Tukey-Kramer HSD

Level		Mean
alcatraz	A	12.445900
etrace	B	9.383700
bare	C	4.577800

Levels not connected by same letter are significantly different.

B. Mac OS X v10.5 SYSTEM CALL FLOW

When a user program invokes a system call that requires kernel processing, for example `open()`, the call flows through a number of system components:

1. The programming language library (potentially).
2. The system's run time library, known as `libc` on many UNIX variants.
3. The syscall stubs library, which invokes the software trap (on IA32, the `SYSENTER` instruction) to switch to kernel mode.
4. The kernel low-level support code, which receives the trap, saves the user mode state, sets up the kernel environment.
5. The kernel syscall handler, which marshals arguments and dispatches the call to the kernel function designated by the syscall number passed from the syscall stub.

Note: On Mac OS X, `libc` and the syscall stubs library are subsumed by `libSystem`, sometimes referred to the `System` framework.

The return path from the kernel function to the user program is through these same components in reverse (LIFO) order.

In the following, *syscall_name* stands for the function name of the UNIX system call, such as `open`, `chmod`, or `gettimeofday`. The procedure is given for Intel IA32 processes, but other processors (PowerPC and ARM) and 64 bit processes are handled analogously.

PROCEDURE 1. *System call — user mode side*

syscall_name in the `libc` part of `libSystem`:
1: System call user mode wrapper code — *if the system call needs it*
`__`*syscall_name* in `__`*syscall_name*`.s` in the `libsyscall` part of `libSystem`:
2: *EAX* register ← syscall number (`SYS_`*syscall_name*) to
3: Call `__sysenter_trap`
`__sysenter_trap` in `libsyscall/custom/custom.s`:
4: *EDX* register ← Return address
5: *ECX* register ← Stack pointer
6: `SYSENTER` (software trap to kernel mode)
7: **if** *CF* (carry flag) not set, **then**
8: **return** — *syscall returned with no error*
else — *syscall returned with error code*
9: Jump to `cerror`
`cerror` in `libsyscall/custom/custom.s`:
10: `errno` ← *EAX* register — *Returned error code*
11: Call `cthread_set_errno_self(`*EAX*`)`
12: *EAX* register ← -1 — *Function return value*
13: **return**
end if

PROCEDURE 2. *System call — kernel mode side*

`hi_sysenter` in `xnu/osfmk/i386/idt.s`:
1: Save user mode state (stack pointer, segment base registers, general registers)
2: Restore kernel environment
`lo_sysenter` in `xnu/osfmk/i386/locore.s`:
3: **if** *EAX* < 0, **then** jump to Mach syscall handler **end if**
4: Switch to kernel stack and unmask interrupts
`unix_syscall` in `xnu/bsd/dev/i386/systemcalls.c`:
5: Get pointers to task, proc, caller registers, syscall args, etc.
6: Look up syscall number in `sysent` table
7: **if** this syscall takes args, **then**
8: `copyin()` args from userspace to kernelspace
9: **if not** a `kdebug_trace` syscall **and** `kdebug` enabled, **then**
10: write syscall start trace record to `kdebug` buffer
end if
11: Use syscall argument munger for this syscall (from `sysent` table) to normalize all syscall args to 64-bit process style layout
else

12: **if** kdebug enabled, **then**
13: write syscall start trace record to kdebug buffer
end if
end if
14: Set the thread's `kauth` credentials to the proc's
15: Set `UT_NOTCANCELPT` flag — *thread cannot be canceled*
16: **if** auditing enabled, **then**
17: write an syscall entry audit record
end if
18: Call the syscall function pointed to by the `sysent` record
19: **if** auditing enabled, **then**
20: write an syscall exit audit record
end if
21: **if** syscall function returned `ERESTART`, **then**
22: Decrement user thread's *EIP* (instruction pointer) appropriately
end if
23: **if** syscall function returned `EJUSTRETURN`, **then**
24: User thread's *EAX* register ← Return value
else
25: User thread's *EAX* register ← Error code
26: User thread's *CF* (carry flag) ← Set
end if
27: Clear `UT_NOTCANCELPT flag` — *thread can be canceled again*
28: **if not** a `kdebug_trace` syscall **and** `kdebug` enabled, **then**
29: write syscall end trace record to `kdebug` buffer
end if
`thread_exception_return` in `xnu/osfmk/i386/locore.s`:
30: Mask interrupts; switch stack pointer to PCB stack
`return_from_trap` in `xnu/osfmk/i386/locore.s`:
31: Check for pending asynchronous system traps, and call handlers
`hi_ret_to_user` in `xnu/osfmk/i386/idt.s`:
32: Restore user mode state (stack pointer, segment base registers, general registers)
33: Restore flags register (including carry flag)
34: Unmask interrupts
35: `SYSEXIT`