Consh: Confined Execution Environment for Internet Computations

Albert Alexandrov, Paul Kmiec, Klaus Schauser

Department of Computer Science
University of California, Santa Barbara
Santa Barbara, CA 93106

{*berto,virus,schauser*}*@cs.ucsb.edu*

Abstract

The recent rapid growth of the Internet made a vast pool of resources available globally and enabled new kinds of applications raising the need for transparent remote access and for protected computing. Currently users need specialized software such as web browsers or FTP clients to access global resources. It is desirable to instead provide OS support for transparent access to these resources so that they can be accessed through standard applications such as text editors or command shells. The new applications made possible by the expanding Internet require provisions for safe and protected computing. For example, global computing projects harness the power of thousands of idle machines to solve complex problems and, similarly, and highly-flexible servers allow users to upload and execute their code on the server to perform otherwise difficult tasks. In both cases, users or servers need to execute applications which they cannot trust completely. Such untrusted applications could potentially corrupt the resources on the machine they are running on or can gain unauthorized access to private information. There are also other emerging applications that require the execution of anonymous or untrusted code. In order for these applications to be widely distributed and used, it is necessary to provide means for safe execution of untrusted applications.

We propose a confined execution environment for Internet computations called Consh. Consh provides security and protection for machines running untrusted applications. What distinguishes Consh from other protection schemes is that it combines protected execution with transparent access to additional remote and alternative local resources. This has two major benefits. First, Consh has a wide applicability in areas such as global computing that benefit from transparent remote access. Second, it allows sensitive local resources to be replaced with safe alternative local or remote ones which greatly relaxes the restrictive nature of protected execution. As a result, a wider range of applications can successfully run under Consh, which would be impossible with other protection schemes.

Consh is implemented using our method for user-level extension of the operating system which works by interposing on system calls. As a result, Consh operates completely at the user level and requires no super-user access and no modification of the existing operating system, system libraries, or applications.

1 Introduction

1.1 Motivation

In its short lifetime, the Internet has grown from a handful of computers at four universities to millions of computers worldwide. This rapid growth enables new types of applications such as: (1) distribution and access of user resources on a global scale; (2) flexible servers that can be extended through user-uploadable services; (3) global computing projects that can harness the power of thousands of idle machines throughout the Internet to solve complex problems. All three applications require interaction between remote computers. One form of such interaction is accessing remote resources such as files or FTP and HTTP servers. Another form of interaction is the execution of code obtained from remote machines which cannot always be trusted. It is desirable to facilitate these forms of interaction by providing transparent access to remote resources and protection from untrusted code.

1.1.1 Transparent Remote Access

Currently, users access remote resources, such as files, through specialized software. For example, a person can use a web browser to access the contents of HTTP servers or an FTP client to access files on FTP servers. While specialized software provides adequate means for interactive browsing, it does not allow existing applications such as text editors, compilers, standard operating system commands to directly access global resources. It is desirable to provide means for existing applications to access remote resources transparently as if they were local. Unfortunately, current operating systems do not provide such access capabilities. We propose to provide this functionality to all applications by extending existing operating systems at the user level.

1.1.2 Protected Execution

The need for protected execution is demonstrated by the increasing spread of applications that require the execution of untrusted code. By untrusted code we mean any piece of code whose origin and contents are not completely trustworthy. In the most simple case a user downloads a freeware or shareware application from an anonymous FTP site. The user runs the application, believing that it will work as advertised and that it will not cause any damage to his or her system. The user, though, has no hard guarantees for that. In the worst case, the application may be a malicious program that will destroy the user's data or will secretly send out passwords or credit card numbers found on the user's machine. It may send a death threat to the White House from the user's email account. In other cases the application may contain a virus or it may contain bugs that cause it to accidentally corrupt the user's data. Other sources for untrusted code are plug-ins and active documents retrieved from the Internet.

The new emerging Internet applications also involve execution of untrusted code. Servers that support user-uploadable services allow third parties to extend the server by installing and running their code on the server [Sun98]. Conventional web servers like Apache [Apa98] are starting to support the installation of user-level CGI scripts or servelets. In the future, HTTP servers may allow users to upload arbitrary pieces of code that the server will run, for example, to filter or compress image or sound data. It is dangerous to allow arbitrary code to run on the server since malicious or buggy binaries may corrupt the server machine.

Execution of untrusted code is also part of global computing. A global computing applications is one that requires a large number of participating machines to cooperate in solving a problem. An example of such applications is the Mersenne prime search [Pri98]. This problem involves finding large prime numbers that have a certain property and has become an exciting problem for the computer science community since it requires a very large amount of computation with tens of thousands of machines working for long periods of time. A second example is the breaking of encryption schemes. This also is a very compute-intensive application that requires a large number of participating machines. Other more complex global computing applications can involve the rendering of complex graphics images for a movie, or solving complex scientific problems.

The common part in all global computing applications is that there is an *initiator* of the computation that provides an executable to be downloaded and run by the participating *volunteers*. Although the initiator may be a well-trusted entity, the volunteers do not have hard guarantees that the executable they run will not cause damage to their machine. It may contain bugs or it may accidentally be infected with a virus. Although in some cases relying on trust in the initiator is a good enough option, lack of trust would prevent many people from participating and therefore would limit the potential number of participants.

These examples show that there is need for running of untrusted applications securely. Unfortunately current operating systems do not provide the means for safe execution of untrusted binaries. The problem is that the security policies of the OS do not distinguish between different applications that a given user is running. As a result if a user executes an untrusted binary it will have the same access to the machine's resources as the user's trusted applications. We propose to solve this problem by extending the operating system at the user level with per-process protection capabilities.

1.1.3 OS Extension at the User Level

The natural solution to providing transparent remote access and protected execution to all existing applications is to extend the operating system with this new functionality. Extending the OS typically requires modifications to the OS itself or installing OS modules or device drivers by the super user. This makes such OS extensions cumbersome or impossible to install and use for a typical user.

There are alternative methods for extending OS functionality that can be implemented at the user level without actual changes to the OS. These methods can achieve most of the benefits of an extensible operating system without having to modify the existing OS or design a new one. In these approaches, extensions can be installed and run by an individual user without the help of a system administrator. One advantage of this is that extensions become very simple to install and use. Another major benefit is that each user can choose to install an OS extension without affecting other users.

We chose to rely on a user-level OS extension method to implement our project in order to make it easily accessible to as wide a group of users as possible.

1.2 Our Approach

In this paper we discuss Consh which is a confined environment for Internet computations. Consh can run untrusted binaries under a restricted environment which controls their access to the machine's resources and denies any potentially dangerous access. There is a variety of resources that Consh protects, the most important of which are the file system and the networking capabilities. The protection mechanism in Consh is not a new idea. In fact the protection in Consh is directly based on previous work done on the Janus project [GWTB96] and Consh incorporates a modified version of Janus as one of its modules.

What distinguishes Consh from previous protection schemes is that it combines protection with access to alternative local and alternative or additional remote resources. Consh virtualizes all the resources visible to untrusted applications which allows protected resources such as files and directories or networking to be replaced with safe alternative local or remote ones transparently. This greatly relaxes the restrictive nature of protected execution. As a result a wider range of applications can run successfully, which would be impossible with alternative protection schemes. Another benefit of our approach is that it can significantly simplify the design of Internet applications. For example, in global computing applications, the different parts of the computation can transparently access remote resources provided by Consh such as common data files and executables or use remote files to communicate and synchronize with minimum programming effort.

Consh is implemented as a user-level OS extension using the method of System Call Interposition (SCI). Consh is a regular user-level process that is positioned between the applications and the operating system (Figure 1). It works by intercepting the system calls issued by the applications and modifying their behavior. To provide protection, Consh aborts system calls that attempt to access forbidden resources. To provide access to remote resources, Consh handles or modifies the behavior of system calls that access such resources. Consh works transparently to the applications being controlled and appears as part of the operating system. To the OS, Consh appears a regular user-level process.

The major benefit of using SCI for our implementation is that Consh installs and runs completely at the user level. A user can simply download the Consh executable and run it without requiring changes to the OS, existing binaries, or standard libraries, and without requiring assistance from the super user. A potential concern with our approach is its performance overhead. While the cost of intercepting individual system calls is significant, our performance analysis shows that the overhead is acceptable for the target applications. Also, we believe that some performance overhead is a good tradeoff for protection and transparent remote access.

We have implemented Consh for Sun Solaris 2.5.1 using the */proc* interface for system call interposition. We have based our implementation on two previous projects: Ufo [AISS97] and Janus [GWTB96]. Ufo is a user-level extension of the operating system and provides transparent access to remote FTP and HTTP file systems. We have

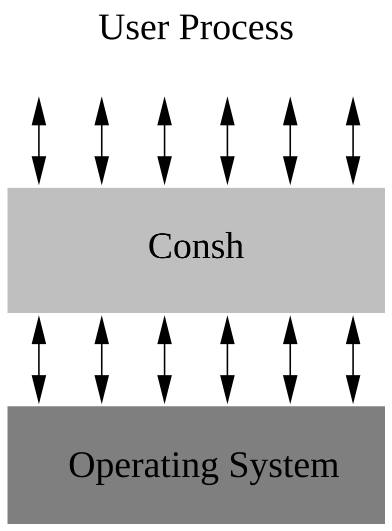

Figure 1: The Consh OS extension.

extended Ufo with new modules most notably a virtual file system and virtual networking. Janus is a user-level confined environment which restricts helper applications' access to local resources. We have also extended Janus to improve its security and to make it less restrictive.

1.3 Usage Scenarios

In this section we will outline several typical scenarios in which Consh can be used. We should note that Consh itself is not a UNIX shell like the name may suggest. It is a stand-alone program that runs as a user-level process. The Consh process can attach to other user-level processes and control them. In particular any standard UNIX shell can be run as a subject process under Consh. The subject processes can access the remote resources that Consh provides but are restricted in their accesses to local resources.

The simplest way to run Consh is to start applications under its control. These applications and any future child processes they spawn will be controlled by Consh. For example:

```
tcsh% consh csh
csh% cat /local/secret/credit_cards.txt
Interrupted system call
csh% grep UCSB http://www.cs.ucsb.edu/index.html
```

Here the user starts a C-shell under the control of Consh. The *cat* and *grep* child processes that are started from within the C-shell are also running under Consh. The example shows the two basic functions of Consh. It denies accesses to sensitive resources such as the */local/secret/credit_cards.txt* file. At the same time it provides transparent access to remote resources such as HTTP servers. The rest of the examples are based on this simple scenario.

Downloaded binaries. A user can use Consh to run any suspicious binary downloaded from the Internet. This is done by simply starting the untrusted application under Consh:

```
tcsh% consh untrusted_binary
```

By configuring Consh to deny access to sensitive resources the user can achieve a high degree of protection from malicious or buggy executables.

User-uploadable code. New extendible HTTP servers allow users to upload custom services that provide new functionality. The user services are implemented in the form of a piece of code that is run on special events. The server can protect itself from untrusted uploaded code by running it under Consh and thus limiting its access to local resources. The uploaded code can use the remote resources provided by Consh to transparently access data files, binaries or for networking.

Global Computing. As a last example, consider how Consh is used in a global computing environment. The initiator distributes binary executables to the volunteers and the volunteers run the binaries under Consh. In this case, Consh is useful to both the volunteer and the initiator. Consh can protect the volunteer's machine from corruption. At the same time, it provides transparent remote access to the untrusted binary which simplifies the design of global computing applications. For example, the binary can access the initiator's file system or, in case that the volunteers disallows networking access to the untrusted binary, Consh can replace it with the initiator's networking capabilities.

2 Related work

Existing work related to our research falls in three main categories: OS extension methods, protected execution, and access to remote resources. In this section we discuss projects from each of these categories and compare them to our work.

2.1 OS Extension Methods

Consh uses system call interposition (SCI) to provide extended OS functionality to untrusted applications. Compared to alternative methods, the main distinction of SCI is that it is a completely user-level method. It allows extensions to install and work at the user level and requires no modifications to the operating system or to the existing binary executables. Here we discuss other projects based on SCI and alternative methods for OS extension at the user level.

The two projects that are closest to Consh in that they use SCI are Interposition Agents [Jon92] and Janus [GWTB96]. Interposition Agents (IA) is a toolbox that provides a high-level interface to the system call interception facilities in Mach [ABB+86]. The focus of this work is to provide a tool for writing OS extensions. In contrast, our work shows that a significant OS extension can be implemented successfully and efficiently using SCI. A second difference is that, in IA, extensions run inside the applications' address space. This allows buggy or malicious applications to interfere with the extension making IA not a good choice for implementing protected execution. In our work, the extension runs in a separate address space and can protect itself from undesirable interference.

The Janus project [GWTB96] uses SCI to provide a confined environment for execution of untrusted binaries and is used as the basis of the protection module in Consh. In contrast to Consh, Janus uses SCI to a limited degree only. Janus operates by simply denying potentially dangerous system calls issued by the untrusted application. Consh, on the other hand, modifies/augments the implementation of the system calls. Consh also adds access to remote resources in addition to providing protection.

Apart from SCI, it is possible to implement user-level OS extensions by replacing static or dynamic link libraries. Most applications do not communicate directly with OS, but instead use standard libraries as intermediaries. By replacing the standard libraries with custom-built ones, it is possible to achieve a similar effect to SCI. Projects that use the library approach include Newcastle Connection [BMR82], Prospero [NAU93], Condor [Con95], Jade [RP93], IFS [EP93] and COLA [KK92].

Changing standard libraries works well for most applications, especially when combined with dynamic linking. The major drawback of this approach is that applications can circumvent the static and dynamic standard libraries and execute system call instructions directly. This makes the approach a poor choice for providing protection from untrusted binaries. Another drawback is that statically linked applications need to be relinked. Commercial applications are often distributed as statically linked binaries and therefore do not “succumb” to the library approach since the user does not have access to the necessary object files.

Apart from the discussed work on user-level OS extensions, there is a lot of current research on designing new operating systems that allow for easier and more efficient user-level extension. Examples are Mach [ABB+86], Exokernel [EKO95], SPIN [BSP+94] and VINO [SESS94]. Although this research shows a lot of promise, it is slow to make it into the mainstream operating systems. In contrast, we use a method that provides extensibility for existing commercial operating systems.

2.2 Transparent Remote Access

There are a number of systems that provide transparent access to remote resources on the Internet. Examples include NFS [SGK+85], AFS [MSC+86], Coda [SKK+90], ftpFS in Plan 9 [PPTT90] and Linux [Fit96], Sprite

[Wel91, NWO88], WebFS [VDA96], Alex [Cat92], Prospero [NAU93], and Jade [RP93]. They all have one significant drawback: they either require root access or modifications to the existing operating system, applications or libraries. Consh is distinct in that it requires no such modifications to any existing code and runs entirely at the user level.

There are a few systems for global file access that run entirely at the user level and are user-installable. Prospero [NAU93] and Jade [RP93] both provide access to NFS and AFS file systems, and to FTP servers and run at the user level by replacing standard link libraries. This avoids changes to the operating system, but does not work for all applications as discussed in Section 2.1.

Other global file systems also run at the user level, but are not user-installable since they require extensions to the operating system itself, which in turn requires root access. These include the WebFS [VDA96] global file system, UserFS [Fit96] in Linux, ftpfs in Plan 9 [PPTT90]. At least two projects provide access to FTP servers by implementing an NFS server that functions as an FTP-to-NFS gateway. Alex [Cat92] supports read-only access to anonymous FTP servers, while [Gsc94] additionally allows read and write access to authenticated FTP servers.

There are several related projects that provide access to remote resources other than file systems. Condor [Con95] provides process migration and checkpointing for a cluster of workstation. Unlike the SCI approach used in Consh, Condor is implemented by replacing static link libraries which limits the range of applications it works for. Solaris MC [KBM$^+$96] is a research cluster computing operating system which presents the remote resources inside a cluster of workstations as a single-image system to existing unmodified applications. Plan 9 [PPTT90] is a distributed operating system whose main distinction is that all resources, local and remote, are named and accessed like files in a hierarchical file system. Since all resources are files, applications can transparently access remote resources as if they were local by simply configuring their private name space. Both Solaris MC and Plan 9 implement their functionality by redesigning the operating system, Consh does not require any changes to the OS in order to provide access to remote resources.

2.3 Protected Execution

The protection part of Consh is based on Janus [GWTB96] and thus Janus is the closest project to ours. Janus secures the interface boundary between applications and the OS by interposition on the system calls. System calls issued by untrusted applications are intercepted and inspected by Janus before being executed. Any system call making unauthorized or unsafe accesses to resources is aborted. In Consh we have extended Janus to be less restrictive and to have more functionality.

A second project related to Janus and Consh is MAPBox [Raj98]. MAPBox also uses Janus as a starting point but extends its functionality in a direction orthogonal to Consh. MAPBox fine-tunes the restrictions imposed on the untrusted applications based on an application classification. The application classes defined by MAPBox include editor, browser, compiler, etc. Each class has its own set of predefined access controls for resources. For example, the compiler class disallows network access, but the browser class does not. An application first identifies itself as belonging to a particular class, and is then executed with restrictions appropriate for its class.

In comparison to both Janus and MAPBox, Consh adds several items. It provides a customizable virtual file system for the untrusted applications. As a result Consh is able to replace protected resources with safe local or remote alternatives. This allows for a much wider class of applications to run which would not be able to work under Janus or MAPBox due to their accessing of protected resources such as files or networking. The provided access to remote files and networking in Consh also makes it an excellent tool for distributed computing projects.

The CUBA [Erl98] project aims at improving the performance of Java applets by extracting the computation part of an applet and running it into a native executable. CUBA restricts the native executable to only performing computation using a thin and very restrictive protection layer that works similarly to Consh and Janus. Unlike Consh, though, CUBA does not address the problem of running unmodified existing applications.

The projects discussed so far work by securing the application-OS interface boundary at runtime by interposing on the system calls. An alternative approach to protection is to ensure that an application is safe prior to its execution. Proof Carrying Code (PCC) [NL96] requires that binaries contain a formal proof of their safety. This technique requires binaries to be compiled with a special proof-generating compiler and run through a proof validator prior to execution to ensure their safety. Another approach for making the application safe is to modify the existing binary. Software Fault Isolation (SFI) [WLG93] uses binary rewriting to ensure that different applications can run inside the same address space without interfering with one another. This is achieved by inserting safety checks in front of any instruction that could potentially perform an unauthorized access. Originally SFI was only developed to check for address range violations, but the same can be used to also make other access, such as system calls, safe. In contrast to

PCC and SFI, Consh works with unmodified binary executables.

3 Virtualizing and Protecting Resources

In unprotected environments, an untrusted binary can compromise the machine it runs on by destroying files, stealing information, using the machine's identity to attack other computers, or over-using resources to deny service to other applications. To avoid these dangers it is necessary to prevent untrusted applications from accessing resources that can potentially be abused. The protection approach in Consh is to construct a virtual environment in which untrusted applications are executed. The environment is initially empty and the user can explicitly populate it with resources that are safe to access. The two major resources that Consh provides are file systems and networking. Consh provides virtual versions of the local file system and local networking, several remote file systems, and remote networking. These are used as the building blocks in constructing an environment for the untrusted applications.

Consh uses virtualization as a tool for providing protection. Instead of a sensitive local resource, Consh can provide an alternative safe local version of it or a remote resource. For example Consh can provide TCP/IP networking through a remote machine as a replacement for the local networking. Regardless of where the virtual resource is located, the untrusted binary has the illusion that it is accessing the “real” local resource.

In addition to virtualizing whole resources, Consh uses fine-grain control to protect local resources when they are made accessible to untrusted applications. For example, the user can allow access only to certain portions of the local file system and allow network connections only to certain remote addresses.

3.1 File System Resources

The file systems that Consh can provide are: a virtual file system (VFS), the local file system, HTTP and FTP file systems, and a consistent remote file system. VFS serves as the root file system for all untrusted applications and is initially empty. The user can populate it by mounting other file systems at appropriate locations inside VFS. For example, the local file system is typically mounted at */local* and the FTP file system is typically mounted at */ftp*. In addition the user can create directories and symbolic links inside VFS which can be used to construct safe versions of protected portions of the local file system.

The local file system is provided to untrusted applications but the access is controlled at a fine-grain level to ensure that no sensitive files or directories are accessed. By default, the whole local file system is inaccessible, but the user can grant explicit read, write and execute permissions on individual files and directories. For example, the user can expose the */usr/bin* and */usr/lib* directories of the local file system for reading and execution, while protecting everything else.

The HTTP and FTP file systems provide access to FTP and HTTP servers. These file systems can be used to supplement the limited access to the local file system and to easily retrieve information from the Internet. The consistent remote file system provides access to remote files, but, unlike the HTTP and FTP file systems, it guarantees strong consistency for applications that require it. For example, it provides file locking which can be used to synchronize concurrent file accesses in a global computing application.

3.2 Networking Resources

Consh is configurable to either use the local machine or a remote machine to provide networking to untrusted applications. When using the local machine's network capabilities, Consh controls its use at a fine-grain level. Consh can limit the set of IP addresses and the set of ports on which the untrusted binary can connect. It can also limit the number of active connections and it can prohibit the creation of server sockets. This allows untrusted applications to use local networking but only to a limited degree to prevent abuse of this resource. For example if Consh is used to protect a server from user-uploaded code, it can be configured to allow connection only to the host that uploaded the code. In general, an untrusted application can be prevented from using the machine it runs on to host a pornography site, or to spam or hack into other machines,

For tighter security, local networking can be completely prohibited. Instead, Consh can transparently redirect all network-related accesses to a remote machine. This can be used in global computations and in servers that allow user-uploadable code. In these scenarios, the remote machine would be run by the owner of the untrusted binary who benefits from its execution. Networking redirection eliminates the danger that the untrusted applications would attempt to attack third parties using the identity of the user and host executing it.

3.3 Other Resources

In addition to the file system and networking, untrusted applications can abuse other resources by using too much CPU time and memory, or by spawning off large numbers of threads and processes. This can create denial of service problems where the untrusted application monopolizes all available resources, preventing other applications from running. Although Consh does not virtualize these resources, it enforces limits on their usage.

4 Implementation of Consh

To show that our approach works, we have implemented Consh for Solaris 2.5 in C++ using the System V /proc file system to intercept system calls. We have based our implementation of two previous projects: Ufo [AISS97] and Janus [GWTB96].

Ufo is a user-level extension of the operating system and provides transparent access to remote FTP and HTTP file systems. It also provides an infrastructure for additional file system modules to be plugged-in. Ufo serves as the base for adding alternative resources. We have used Ufo's existing functionality and added several new modules.

Janus is a user-level confined environment which restricts helper applications' access to local resources. We use Janus as the base for confining access to the local file system, restricting network, and controlling the environment. We have also extended Janus to improve its security and to make it less restrictive.

4.1 System Architecture

The architecture of Consh is based on our infrastructure that we built for System Call Interposition (SCI). The idea behind SCI is that applications communicate with the OS using system calls. If we put a layer between the applications and the OS that intercepts system calls and modifies their behavior, we can in effect modify or extend the functionality of the OS that the applications perceive.

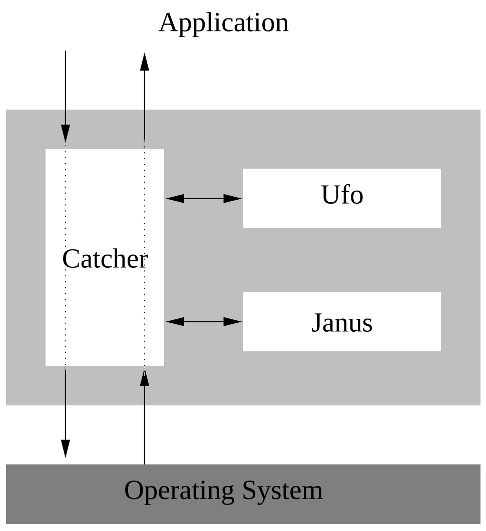

Figure 2: Consh architecture.

Our infrastructure consists of the Catcher and optional extension modules that can be plugged-in (Figure 2). The Catcher is the central component which intercepts system calls and forwards them to the extension modules. In Consh, the extension modules are Ufo, which provides the extended file system and network functionality, and Janus, which provides protection.

4.1.1 Catcher

The Catcher is capable of attaching to user processes and intercepting and modifying the system calls issued by the processes. The Catcher provides a layer between the user's application processes and the original operating system, as shown in Figure 2. This extra layer does not change the existing OS directly, but allows us to control the user's environment, either by modifying system call parameters, or issuing additional service requests.

The Catcher operates as follows: Initially, it connects to the user process and tells the operating system which system calls to intercept. The Catcher can change at the user level the semantics of intercepted system calls to implement extended OS functionality. Whenever a system call of interest begins (or completes), the operating system stops the subject process and notifies the Catcher. The Catcher calls appropriate extension modules, if needed, and then resumes the system call.

In our Solaris implementation, the Catcher monitors user processes using the /proc virtual file system [FG91]. This is the same method used by monitoring programs, such as truss or strace, which are also available on a number of other UNIX platforms, including Digital Unix, IRIX, BSD or Linux. The System V /proc interface allows us to control a process by operating on a file in the /proc directory associated with the process.

4.1.2 Intercepting System Calls

The Catcher intercepts system calls issued by subject processes and forwards them to the Ufo and Janus modules when needed. Inside Consh, the two modules are layered on top of one other so that Ufo is between the subject process and Janus, and Janus is between Ufo and the operating system. This order is important since it guarantees that Janus always has the last word on whether to allow or deny a system call.

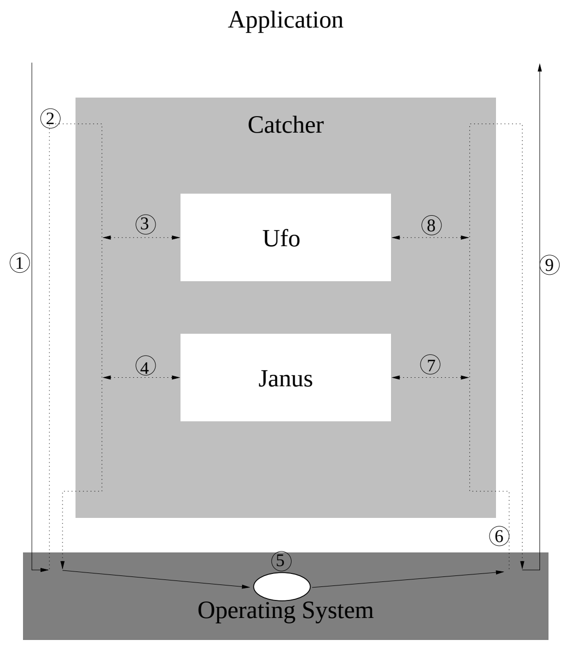

Figure 3: A detailed view of how Consh works.

To illustrate how Consh works we will examine the life of system call, starting with its first interception, going through the Consh modules, the operating system and back to the application (Figure 3). When the subject process issues a system call (1), it can be directly serviced by the operating system or intercepted by the Catcher (2). For intercepted system calls, the Catcher determines that Ufo requested to see this system call on the entry (3), and so the Catcher passes the system call to it and waits until Ufo is finished. Janus has also requested to see this system call on the entry (4), and so the Catcher passes the system call to it and waits until it is finished. Now the system call is allowed to proceed and enter the operating system where it is serviced (5). After the operating system is done servicing the system call, the Catcher can again intercept the system call on the exit (6). Janus has requested the interception of the exit of the system call so it is invoked (7). Then the Catcher passes the system call to Ufo and waits until it is done. Finally, the system call is allowed to proceed and it returns to the subject process (9).

As a more concrete example, if the system call being intercepted is *open(/http/www.cs.ucsb.edu/index.html)*, Ufo will use the opportunity at (3) to download the corresponding file with URL *http://www.cs.ucsb.edu/index.html*, write it to the local system as, say, */tmp/ufo-cache/file0123* and change the file name argument of the system call to */tmp/ufo-cache/file0123*. At point (4) Janus will examine the (already modified) system call to decide whether it is safe or not. In this case */tmp/ufo-cache/file0123* will be determined safe and Janus will allow the system call to proceed. If the file name were */etc/passwd* though, point (4) is where Janus would deny access by aborting the system call. On the exit of the system call, Janus would do nothing at point (7) while Ufo will remember the file descriptor of the newly opened file for future use.

It is important to notice two things. First, this example shows how our infrastructure works. On entry of a system call, services see that system call in one order (Ufo, Janus), and on exit, services see that system call in reverse order (Janus, Ufo). This is very important since it ensures that extension modules can be layered on top of one another. In this example the two extensions are unaware of one another and work independently on the intercepted system calls. Janus can see the modifications that Ufo makes to the system call arguments and Ufo can see the modification that Janus makes to the return values. To Ufo, it appears that Janus is part of the operating system and to Janus, it appears that Ufo is part of the application.

Second, we see why system call interposition introduces performance overhead. Normally, the application would make a system call (1), the operating system would service the system call (5), and the result would be returned to the application (9). Step (1) and (9) both require a context switch. If the system call is intercepted on entry (2) and exit (6), four additional context switches are required. Basically, every line going into or coming out of the operating system is a context switch.

4.2 Ufo Module

The Ufo module provides the HTTP and FTP remote file systems, the Virtual File System (VFS), the Consistent Remote File System (CRFS) and the Remote Networking. The HTTP and FTP modules in Ufo provide transparent access to HTTP servers and to anonymous and authenticated FTP servers. Typically the HTTP and FTP file systems are mounted at */ftp* and */http*. In this case applications can use, for example, */ftp/ftp.netscape.con/public/* and */http/www.cs.ucsb.edu/index.html* to access remote files.

A more detailed discussion of the HTTP and FTP functionality can be found in [AISS97] which discusses the stand-alone Ufo file system. In the rest of this section we will focus on the new additions to Ufo not present in its original version and implemented only in Consh.

4.2.1 Virtual File System

Applications running under Consh have the Virtual File System (VFS) as their root file system. Initially the VFS is empty. It can be populated by adding three types of entries: virtual directories, virtual symbolic links and mountpoints. The directories and symbolic links behave like in a normal file system. The mountpoints are the roots of other file systems provided by Ufo such as the HTTP and FTP file system, the local file system, individual remote FTP or HTTP server and consistent remote file systems. An example configuration of the VFS is shown in Figure 4.

Here */local* is the mountpoint for the local file system. Note that not the whole local file system is accessible, rather only portions of it that are specifically allowed by the user are visible to untrusted applications. The */http* and */ftp* are the mountpoints for the HTTP and FTP file systems. *Leopard* is a mountpoint for a specific authenticated FTP server. There are also several symbolic links shown that help configure a usable file system environment for the applications running under consh. The whole configuration described so far can be specified in the *.uforc* file.

The symbolic links and directories can be also created dynamically. For example, the user can run a shell script to do that before he or she starts the untrusted application. Here is an example of starting a shell under consh and creating a virtual directory and a virtual symbolic link before executing the untrusted application

```
tchs% consh csh
csh% mkdir /mydir
csh% ln -s /ftp/ftp.netscape.com /mydir/mylink
csh% untrusted-application
```

This will create a new directory *mydir* and a new symbolic link inside it, *mylink*, which points to Netscape's FTP server (See Figure 4).

The ability to create symbolic links is an important tool for making applications running under Consh work properly. By creating the */bin*, */usr* and */tmp* symbolic links we provide applications with resources where they would

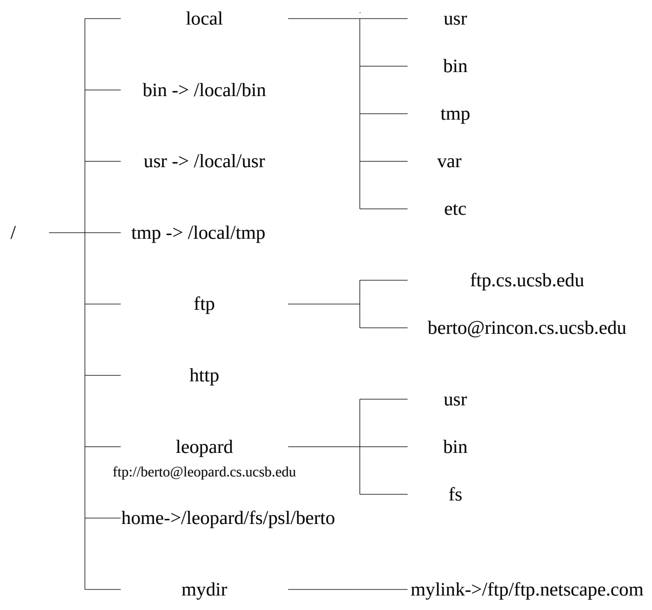

Figure 4: An example virtual file system configuration.

expect them. The ability to create directories is also important to provide proper protection for local resources that does not prevent applications from running.

4.2.2 Remote Networking and Consistent Remote File System

The last two services provided by Ufo are the Consistent Remote File System (CRFS) and Remote networking. We discussed them together in this section since they have similar implementations.

CRFS. CRFS is similar to the HTTP and FTP file systems in that it provides access to remote files, but it differs from them since it provides strong consistency. To achieve stronger consistency, all CRFS related calls are shipped and executed on the remote machine containing the actual files. As a result applications behave as if they were running on the remote server with respect to their CRFS accesses. Consequently, all accesses to CRFS have the same strong consistency guarantees that the CRFS host provides.

Remote Networking. Consh is able to replace the local networking capabilities with these of a remote machine through Ufo's Remote Networking module. This module works by intercepting all system calls operating on the special files */dev/tcp* and */dev/udp* used to implement TCP/IP and UDP communications. The intercepted system calls are then forwarded and executed on the remote machine whose networking Consh uses. As a result, all networking activity of the controlled applications behaves as if it occurs on the remote machine.

Remote System Call Server. Both CRFS and the Remote Networking module require a specific server to be running on the remote machine. Since both services rely on forwarding of file-related system calls, we use a single server for both of them. The server is responsible for executing system calls submitted by Consh from other machines. When Consh intercepts a system call operating on a remote resource, it marshals the system call arguments and sends them to the system call server. The server unmarshals the arguments, executes the system call and sends the result back.

Unlike the HTTP and FTP modules, CRFS and the remote networking modules rely on our custom system call server running on the remote machine whose resources they provide. The idea is that the owner of the untrusted binary would provide a machine running this server if the binary uses CRFS or remote networking services. Since the server works by forwarding system calls, it is required that both the machine on which Consh is running and the machine on which the server is running use the same operating system. Lastly, since Consh is only concerned with the protection of the machine on which it is running, the remote system call server does not require any security checks. Security and protection for the remote machine is an orthogonal issue to our work, and can be an optional addition in the future.

4.3 Janus Module

The Janus module provides the protection of local resources from unauthorized access. It works by intercepting and denying any system call issued by the untrusted application that might be potentially dangerous. The Janus module in Ufo is based on the Janus project developed at UC Berkeley and a detailed discussion of its implementation is beyond the scope of this paper. For more information on Janus the reader is referred to [GWTB96].

Compared to the original Janus project, though, we have made some modifications and improvements in Consh to improve the security and to allow for a wider range of untrusted applications to run successfully.

The original Janus implements its protection of the local file system by only analyzing the symbolic names of the accessed files. It does not attempt to resolve the name and find out the real file that is being accessed. This is a potential security concern as illustrated by the following example: Assume that the user has specified that accesses to */tmp/** are legal. Janus would allow all accesses to files with prefix */tmp/*. If there is a symbolic link */tmp/security_hole* that points to */etc/passwd* the original Janus would allow access to the password file if the untrusted application opens */tmp/security_hole*. To solve this problem, the Janus module in Consh first resolves all symbolic links in a file name and then performs the security check on the fully resolved name. This eliminates the above security hole.

In Consh we have also added support for safe changing of the current working directory and for safe execution of a class of system calls that operate on open file descriptors. Changing of the current working directory was not possible in the original Janus and the set of system calls were simply denied which made certain applications unable to run.

4.4 Configuration

Since Consh is designed to function in different settings, it is important to allow the user to configure it accordingly. Consh can be configured in two ways: through its configuration files and through a shell script executed before the untrusted application is started.

Consh uses two configuration files: *.uforc* and *.janus*. The first file is used to configure the file system environment for the untrusted application. In it, the user can mount Ufo file system at the desired location and create VFS directories and symbolic links as explained in Section 4.2.1. In the second file, the user can specify and fine-tune the desired protection policy for the local resources. It is possible to explicitly allow and deny access to specific files, directories or subtrees of the file system. It is also possible to allow and deny connections to sets of IP addresses and port numbers as well as to limit the number of open sockets and other aspects of the networking capabilities of the untrusted applications. A more detailed description of all the configuration options can be found in [Kmi98].

The user can further configure the environment by executing a shell script before executing the untrusted binary as shown in Section 4.2.1. In addition to creating VFS directories and symbolic links, the script can set environmental variables for the untrusted binary which is currently not possible to achieve through the configuration files.

5 Experimental Results

The goal of our performance analysis is to measure the overhead introduced by Consh when running applications under its control. This information is necessary to determine the viability of our confined execution environment.

We first present the results of several micro benchmarks which measure the overhead of intercepting individual system calls. To demonstrate the overall impact of this overhead on whole applications, we also run a set of real-life applications. While the micro benchmarks show that intercepting of individual system calls is expensive, the real-life applications exhibit much-lower overhead.

All tests were run on a 167 Mhz Sun Ultra 2 workstation with 128 MB of RAM running Solaris 2.5.1. Tests that required a second machine used an additional identical workstation connected with 100 Mbit/sec Ethernet network.

5.1 Micro Benchmarks

We present two sets of micro benchmarks in order to measure the effect of Consh on accessing the two types of resources provided by it: file systems and networking. The micro benchmarks were run on a lightly loaded workstation by taking the wall-clock time just before and just after the system call. Timing was done using the *gethrtime* system call. Since individual system calls are very fast, normal system activity such as interrupts and context switches distorts some of the measurements. This produces a small percentage of outliers that are several times larger than the rest of the measurements. To ensure we do not include unrelated system call activity in our measurements, in each test run we recorded 100 measurements and discarded the highest 10% of them. The remaining times were then averaged. The numbers in the table are the arithmetic mean of five such runs.

5.1.1 File System Related

The results of the file-related micro benchmarks of individual system calls present the application-perceived run times (measured as wall clock times) for *open*, *close*, *stat*, *read*, *write* and *getpid* system calls. In order to measure the cost of the system calls themselves and not the speed of the network, we used the local file system as opposed to a remote NFS server. Table 1 presents our results.

System Call	Standard OS		Catcher only		Consh local FS		Consh CRFS	
open	221 μs	(1.0)	654 μs	(2.9)	1344 μs	(6.0)	3230 μs	(15)
close	11 μs	(1.0)	360 μs	(32)	715 μs	(65)	1685 μs	(153)
stat	222 μs	(1.0)	644 μs	(2.9)	1172 μs	(5.3)	1903 μs	(8.6)
getpid	3 μs	(1.0)	4 μs	(1.3)	4 μs	(1.3)	4 μs	(1.3)
write 1b	20 μs	(1.0)	38 μs	(1.9)	42 μs	(2.1)	1327 μs	(66)
read 1b	26 μs	(1.0)	37 μs	(1.4)	41 μs	(1.6)	1324 μs	(51)
write 8K	103 μs	(1.0)	128 μs	(1.2)	134 μs	(1.3)	2692 μs	(26)
read 8K	79 μs	(1.0)	91 μs	(1.2)	93 μs	(1.2)	2626 μs	(33)

Table 1: Run times in microseconds for various system calls accessing files. The numbers in parentheses represent the ratio normalized to the standard Solaris OS.

The *Catcher only* column shows the cost of intercepting system calls without passing them to any extension. The Catcher simply intercepts system calls which Consh would normally intercept, and lets them continue immediately without modifying them. Note, that the Catcher does not unnecessarily intercept system calls such as *getpid*. Even though, most overhead comes from the intercepted system calls, the unintercepted system calls still incur a small overhead because the operating system takes a different execution path if at least one system call is intercepted.

The *Consh local FS* column shows how much extra overhead the rest of the extensions introduce in addition to the Catcher. The benchmark program runs under Consh and accesses local files only. This column is important since it shows the overhead needed to safe guard the local file system. This overhead comes from the analysis of the parameters of the intercepted system calls. For system calls that take file name arguments, Consh determines whether the file is indeed local or remote. Since a system call does not necessarily take an absolute path name as an argument, Consh must determine it. Once the path is resolved, Consh must check if the accessed path is safe by matching it against a list of safe paths.

The last column shows the numbers for accessing files using the consistent remote file system. Any system call that is executed on the system call server incurs at least an extra 1 millisecond of overhead due to the network. The *getpid* system call remains at 4 microseconds since it is not shipped to the system call server. The rest of the numbers include the time to marshal and unmarshal the system calls and also the time to intercept both the entry and exit of those system calls. The *open* takes more time because it needs to be executed on the local machine as well as on the system call server. We also see that for the 8K versions of the *read* and *write* system calls, the time needed to transfer 8K block contributes into the overall overhead.

5.1.2 Network Related

Next, we performed benchmarks to see the performance impact of Consh on network-related system calls. We measured the application-perceived run times (measured as wall clock times) of the *socket*, *connect*, *read*, and *write* library functions. For the *read*, we made sure that there is enough data in the stream so that the call would not block. Note that a single library call can execute multiple system calls. Table 2 shows our results.

As before, the *Catcher only* column shows the cost of intercepting system calls that a particular library call issues without passing them to any extension. The *Consh local* column shows the times for accessing socket using the */dev/tcp* on the local file system. This column shows the overhead needed to safe guard the local network capabilities. Notice that only *socket* and *connect* incur significant overhead; they are the only two library functions which make system calls that Consh intercepts.

Library Call	Standard OS		Catcher only		Consh local		Consh remote	
socket	403 μs	(1.0)	1976 μs	(4.9)	3288 μs	(8.1)	10639 μs	(26)
connect	692 μs	(1.0)	1129 μs	(1.6)	1186 μs	(1.7)	7811 μs	(11)
write 1b	110 μs	(1.0)	113 μs	(1.0)	113 μs	(1.0)	1897 μs	(17)
read 1b	32 μs	(1.0)	35 μs	(1.0)	32 μs	(1.0)	1816 μs	(37)
write 8K	367 μs	(1.0)	368 μs	(1.0)	383 μs	(1.0)	3585 μs	(9.6)
read 8K	194 μs	(1.0)	203 μs	(1.0)	198 μs	(1.0)	3250 μs	(16)

Table 2: Run times in microseconds for various network library functions. The numbers in parentheses represent the ratio normalized to the standard Solaris OS.

The *Consh remote* column shows the times for accessing the network using the remote network communication. The times have increased by at least 1 milliseconds. Notice *socket* takes 26.4 times longer than standard OS. This is because the *socket* library function is implemented in terms of one *open* system call and four *ioctl* system calls (all of which have to be executed on the system call server). As before, transferring 8K blocks adds more overhead.

We can observe that intercepting individual system calls is expensive. System calls which have a filename as one of their arguments are especially costly. The reason is that Consh resolves all filenames to eliminate any symbolic links. Since Solaris 2.5.1 does not have any system call that resolves paths, Consh needs to execute many *stat* system calls, similar to *pwd* command. We foresee that this overhead will decrease significantly in Solaris 2.6 where paths can be resolved with a single system call.

5.2 Application Benchmarks

In addition to micro benchmarks, we also benchmarked real applications. We wanted to see what is the overall impact of Consh on typical applications it is designed to run. The tables report absolute execution times in seconds.

5.2.1 Common Applications

We started by measuring the *make* and *latex* benchmarks. The *make* test compiles 40 files (about 20,000 lines of code) using *g++* and produces about 6 MB of object files. The *latex* test is to latex three times a 20-page paper consisting of 8 tex files and to produce a postscript from the dvi file. Table 3 presents our results.

Benchmark	Standard OS		Consh local FS		Consh CRFS	
make	102 s	(1.0)	122 s	(1.2)	124 s	(1.2)
latex	11.8 s	(1.0)	14.3 s	(1.2)	16.2 s	(1.4)
IOStone	3 s	(1.0)	52 s	(17)	188 s	(63)

Table 3: Run times in seconds for application benchmarks. The numbers in parentheses represent the ratio normalized to the standard Solaris OS.

Notice that with *make*, Consh introduces only a 20% slowdown when using the local file system. When using the consistent remote file system, the overhead increases only slightly. That is because *g++* writes the object files using a single system call. This results in the number of system calls executed on the system call server being relatively small. The *write* system calls that are executed on the system call server have a big buffer (60,000 bytes or one object file) as one of their arguments. Since we are using 100Mbit/s Ethernet network, transferring those big buffers is very fast.

The *latex* benchmark issues more system calls that operate on small temporary files. Although Consh only introduces 20% slowdown when using the local file system, the slowdown increases to 36% when using the consistent remote file system. That is because the computation to communication ratio is less favorable.

As an extreme test, we used IOStone to benchmark the consistent remote file system. IOStone is a standard file system benchmark. We chose this as an example of applications that execute a large number of file system calls that Consh intercepts, handles, and possibly ships to the system call server. The IOStone benchmark performs thousands of file accesses by issuing *open*, *close*, and *lseek* and other system calls. Most of these have to be intercepted by Consh. Thus, Consh runs about 17 times slower on the local file system. When using the consistent file system, Consh runs about 62 times slower. Notice that nearly all the system calls that are intercepted have to be executed on the system call server. The reader should note that IOStone is not a typical application intended to run under Consh and is included here to give a complete picture of the introduced performance overhead.

In an effort to reduce the overall overhead, we have made optimizations that decrease the number of system calls that are intercepted. For example, system calls like *read* and *write* are only intercepted whenever there is at least one file opened on the CRFS.

5.2.2 Global Computing

To test Consh in the global computing setting, we set up a Mersenne Prime computation. We took the well-known *lucas* test program and wrapped it in a Perl script. The Perl script reads numbers to test from a file located on the consistent remote file system, runs the *lucas* program on those numbers, and writes the results in the file on the consistent remote file system using file locking for synchronization. The script loops until there are no more numbers to test. The file on the consistent remote file system simply contains a list of numbers and their status (untested, being tested, not a Mersenne prime, and bingo).

	NFS			Consh CRFS		
Participants	1	2	3	1	2	3
Fine Granularity of Data	128 s	65 s	45 s	127 s	66 s	44 s
Coarse Granularity of Data	126 s	64 s	44 s	125 s	68 s	45 s

Table 4: Run times in seconds for Mersenne Prime computation.

We benchmarked the performance of the consistent remote file system against NSF. In each instance, the file containing the list of number which clients use to synchronize, is placed on one of those two file systems. We varied the number of hosts (one, two, and three) and how many numbers each client fetches at a time (one and ten, representing fine and coarse granularity of data). The results are presented in Table 4.

There are two important observations. First, the consistent remote file system provided by Consh enabled us to quickly use an off-the-shelf program, *lucas*, in the global computation. We took advantage of the synchronization feature of the consistent remote file system and wrote a simple script to manage data for the main computation engine. The whole project took less than an hour to set up, including writing the Perl script. Second, global computations are typically structured to minimize communication between different hosts. Therefore, the performance of the consistence remote file system has little impact on the overall time.

5.2.3 User-Uploadable Services

Another important use for Consh is for protection in servers allowing user-uploadable services. In this test, we measured the performance of two services that are likely to be uploaded in an HTTP server. First, we used the *sox* audio conversion program to convert a 5 MB .wav file to an .au file. Second, we used the *djpeg* utility to transcode a 200K jpeg file into a grayscale .gif file three times. We measured and compared the performance in three cases: using the standard operating system without protection, using Consh for protection with the local file system as the location of the input and output files, and using Consh with the consistent remote file system as the location of the input and output files. Table 5 shows our results.

Service	Standard OS	Consh local FS	Consh CRFS
sox (audio conversion)	10.16 s	10.23 s	13.81 s
djepg (image transcoding)	2.19 s	2.26 s	2.94 s

Table 5: Run times in seconds for user-uploadable service.

In both cases, the overhead incurred by running either service inside Consh using the local file system is very small (less than 4%). When using the consistent remote file system, the additional overhead comes from transferring the input and output files from, and to, the remote machine. As we expected, the computation is more dominant than communication and thus the overhead is low.

6 Conclusions

In this paper we discussed Consh, a confined environment for Internet computations. Consh can run untrusted binaries under a restricted environment which controls their access to all of the machine's resources and denies any potentially dangerous access. There is a variety of resources that Consh protects, the most important of which are the file system and the networking capabilities. What distinguishes Consh from other protection schemes is that it combines protected execution with transparent access to additional remote and alternative local resources. Consh provides access to a virtual file system, to remote FTP and HTTP file servers, to a remote consistent file system and to remote networking.

Combining protection and access to additional resources has two major benefits. First, it allows sensitive local resources to be replaced with safe alternative local or remote ones which greatly relaxes the restrictive nature of protected execution. As a result a wider range of applications can run successfully, which would be impossible under other protection schemes. Second, Consh has a wide applicability in areas such as global computing that benefit from transparent remote access.

We implemented Consh as a user-level OS extension using system call interposition. This makes Consh very simple to use since it can be installed and run by an individual user without the help of a system administrator and without affecting other users. One concern with using system call interposition in Consh was the potential performance penalty. Although our measurements show that the performance overhead of intercepting individual system calls is large, the overall performance overhead of the target applications is acceptable.

References

[ABB+86] M. Acetta, R. Baron, W. Bolowsky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A new kernel foundation for Unix development. In *Proceedings of the USENIX Summer '86 Conference*, July 1986.

[AISS97] A. D. Alexandrov, M. Ibel, K. E. Schauser, and C. J. Scheiman. Extending the Operating System at the User Level: the Ufo Global File System. In *1997 Annual Technical Conference on UNIX and Advanced Computing Systems (USENIX'97)*, January 1997.

[Apa98] Apache Group. Apache HTTP Server Project. http://www.apache.org/, 1998.

[BMR82] D. R. Brownbridge, L. F. Marshall, and B. Randell. The Newcastle Connection, or UNIXes of the world unite! *Software — Practice and Experience*, 12, 1982.

[BSP+94] B. N. Bershad, S. Savage, P. Pardyak, E. F. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers. Extensibility, safety and performance in the SPIN operating system. In *Proceedings of the 15th Symposium on Operating System Principles*, 1994.

[Cat92] V. Cate. Alex — a global filesystem. In *Proceedings of the 1992 USENIX File System Workshop*, Ann Arbor, MI, May 1992.

[Con95] The Condor Team. Checkpoint & migration of UNIX processes in the Condor distributed processing system. *Dr. Dobbs Journal*, February 1995.

[EKO95] D. Engler, F. Kaashoek, and J. O'Toole. Exokernel: An operating system architecture for application-level resource managment. In *Proceedings of the 15th ACM Symposium on Operating System Principles*, December 1995.

[EP93] P. R. Eggert and D. S. Parker. File systems in user space. In *Proceedings of the Usenix Winter 1993 Technical Conference*, Berkeley, CA, 1993. Usenix Association.

[Erl98] U. Erlingsson. *High-Performance Binary Applets*, 1998.

[FG91] R. Faulkner and R. Gomes. The process file system and process model in UNIX system V. In *Proceedings of the 1991 USENIX Winter Conference*, 1991.

[Fit96] J. Fitzhardinge. Userfs: A user file system for Linux. ftp://sunsite.unc.edu/pub/Linux/ALPHA/userfs/userfs-0.9.tar.gz, 1996.

[Gsc94] M. Gschwind. FTP — access as a user-defined file system. *ACM Operating Systems Review*, 1994.

[GWTB96] I. Goldberg, D. Wagner, R. Thomas, and E. A. Brewer. A Secure Environment for Untrusted Helper Applications — Confining the Wily Hacker. In *Proceedings of the 1996 USENIX Security Symposium*, 1996.

[Jon92] M. B. Jones. *Transparently Interposing User Code at the System Interface*. PhD thesis, Carnegie Mellon University, 1992.

[KBM+96] Y. Khalidi, J. Barnabeu, V. Matena, K. Shirriff, and M Thadani. Solaris MC: A Multicomputer Operating System. In *Proceedings of the 1996 Usenix Technical Conference*, January 1996.

[KK92] E. Krell and B. Krishnamurthy. COLA: Customized overlaying. In USENIX Association, editor, *Proceedings of the Winter 1992 USENIX Conference:Z January 20 — January 24, 1992, San Francisco, California*, jan 1992.

[Kmi98] P. Kmiec. Consh: User-level confined execution shell. Master's thesis, University of California at Santa Barbara, 1998.

[MSC+86] J. Morris, M. Satyanarayananan, M. H. Conner, J. H. Howard, D. S. Rosenthal, and F. D. Smith. Andrew: A distributed personal computing environment. *Communications of the ACM*, 29(3), 1986.

[NAU93] B. C. Neumann, S. S. Augart, and S. Upasani. Using Prospero to support integrated location-independent computing. In *Proceedings of the Symposium on Mobile and Location-Independent Computing*, Cambridge, MA, 1993.

[NL96] G. Necula and P. Lee. Safe kernel extensions without run-time checking. In USENIX, editor, *2nd Symposium on Operating Systems Design and Implementation (OSDI '96), October 28–31, 1996. Seattle, WA*, pages 229–243, Berkeley, CA, USA, October 1996. USENIX.

[NWO88] M. Nelson, B. Welch, and J. Ousterhout. Caching in the Sprite network file system. *ACM Transactions on Computer Systems*, 6(1), February 1988.

[PPTT90] R. Pike, D. Presotto, K. Thompson, and H. Trickey. Plan 9 from Bell labs. In *Proceedings of the UKUUG Conference*, July 1990.

[Pri98] PrimeNet. Great Internet Mersenne Prime Search. http://www.mersenne.org/prime.htm, 1998.

[Raj98] M. Raje. Safe execution of untrusted binaries. Master's thesis, University of California at Santa Barbara, 1998.

[RP93] H. C. Rao and L. L. Peterson. Accessing files in an internet: The JADE file system. *IEEE Transactions on Software Engineering*, 19(6), June 1993.

[SESS94] M. Seltzer, Y. Endo, C. Small, and K. Smith. An introduction to the VINO architecture. Technical Report TR34-94, Harvard University, 1994.

[SGK+85] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun network file system. In *Proceedings of the Summer USENIX conference*, June 1985.

[SKK+90] M. Satyanarayananan, J. J. Kistler, P. Kumar, M. E. Okasaki, E. H. Siegel, and D. C. Steere. Coda: A highly available file system for a distributed workstation. *IEEE Transactions on Computers*, 39(4), 1990.

[Sun98] Sun Microsystems. Java web server. http://jserv.java.sun.com/products/webserver/index.html, 1998.

[VDA96] A. Vahdat, M. Dahlin, and T. Anderson. Turning the web into a computer. Technical report, University of California, Berkeley, 1996.

[Wel91] B. B. Welch. Measured performance of caching in the Sprite network file system. *Computer Systems*, 3(4), 1991.

[WLG93] R. Wahbe, S. Lucco, and S. Graham. Efficient software-based fault-isolation. In *Proceedings of the 14th Symposium on Operating System Principles*, 1993.