Stackable File Systems as a Security Tool

Erez Zadok
Computer Science Department, Columbia University
ezk@cs.columbia.edu

CUCS-036-99

Abstract

Programmers often prefer to use existing system security services, such as file system security, rather than implement their own in an application. Traditional Unix security is generally considered inadequate and few operating systems offer enhanced security features such as ACLs[24] or immutable files[12]. Additional file system security features are always sought, but implementing them is a difficult task because modifying and porting existing file systems is costly or not possible.

We advocate adding new security features using stackable file systems. As a starting point, we propose a portable, stackable template called the *wrapper file system* or Wrapfs, a minimal file system from which a wide range of file systems can be prototyped. A developer can modify a copy of the template and add only that which is necessary to achieve the desired functionality. The Wrapfs template takes care of kernel internals, freeing the developer from the details of operating system internals. Wrapfs imposes an overhead of only 5–7%.

This paper describes the design, implementation, and porting of Wrapfs. We discuss several prototype file systems written using Wrapfs that illustrate the ease with which security features can be implemented. The examples demonstrate security features such as encryption, access control, intrusion detection, intrusion avoidance, and analysis.

1 Introduction

Security has become and integral part of computing in recent years. To provide the strongest possible security to users, security solutions sometimes consume a large amount of resources, and may inconvenience their users too much. In addition, developers find it difficult to port applications to many different security APIs. A file system can alleviate some of these problems as follows:

Performance: An in-kernel file system will perform better than an out of kernel one, because user-level file systems and applications suffer from context switches. Moreover, a file system could be tailored to perform specific intrusion detection actions faster than general purpose audit systems such as Solaris's BSM[25].

Flexibility: Applications and user-level file servers do not have easy access to all system resources. For example, it is easier and more secure to determine the effective user ID of a process inside the kernel than in a user-level NFS-based file server, where that information can be faked by spoofed RPC calls or trojanized shared libraries.

Scalability: When a facility as basic as the file system offers security features, existing user applications can automatically benefit from such features with little or no modification (e.g., text editors and encryption). In contrast, many applications now must implement their own security. New user applications also gain from such added file system security, rather than have it become a design afterthought.

Standardization: A file system provides a common API to all user applications, speeding up development and porting of new applications. Conversely, today's secure applications often use different security facilities.

Security: Code that runs in the kernel is generally harder to track, replace, or circumvent than user-level code. Access to kernel state and direct manipulation of kernel objects is often more difficult, and involves manipulation of kernel memory (e.g., via `/dev/kmem`). User processes are susceptible to root attacks, but the kernel runs in privileged mode and can even prevent the root user from certain actions until they are authorized. While address space separation does not immediately guarantee better security[19], kernel based file system security features may serve to slow attackers.

In addition, user-level file servers (such as those based on NFS) may be open to protocol vulnerabilities. That is, the protocol between the file server and the rest of the system may be compromised easily, especially if the protocol uses the network.

There are several areas where a kernel level file system may be more suitable for supporting security features than a user-level application or a user-level file system:

Cryptography: A file system can offer transparent encryption and decryption to all applications using it.

Access Control: Additional access methods such as ACLs can be added easily to file systems. The file system can take measures to protect against some root initiated attacks by declaring certain parts of a file system as read-only or immutable, or by requiring authentication before certain executables are modified.

Intrusion Detection: A file system can take measures to detect possible intrusions. For example, if a root process whose controlling terminal is not the console is trying to overwrite the `/bin/login` executable, the file system may elect to deny such a request on the assumption that this could indicate an intrusion. Also, if an attempt is made to turn the setuid bit on a root-owned binary, that request can be rejected by the file system. Moreover, the file system may decide to fake the success of such attempts and at the same time send an alert to a trusted user about a possible intrusion in progress.

Intrusion Avoidance: A file system can decide that no root owned or setuid system binaries can be modified from anywhere other than the console, and even then only after users authenticate themselves to the file system. It could further force the access to a subset of subdirectories in a given file system to read-only, without having to set the whole file system read-only.

Post Break-in Analysis: Attackers often cover their tracks by removing log files and tools used in the break-in. A file system can detect attempts to remove or truncate files and instead rename or make backup copies of them that are hidden from attackers' view.

Several extensible file system interfaces have been proposed in the past decade and a few of those were prototyped[8, 17, 23]. None of these proposals are available in commonly-used Unix operating systems, because of the significant changes that overhauling the file system interface would require, and the impact it would have on performance. A few small enhancements to kernel-based file systems were implemented over the years. These include immutable files in 4.4-BSD's FFS[12] and ACLs in Solaris's UFS[24]. However, none of these features are widely available. Furthermore, making such changes to existing file systems is often a costly proposal: source access is required, and the work involves deep understanding of that operating system's internals. Once the work is accomplished on one platform, a port to another is almost as difficult as the initial port.

Others have resorted to writing file systems at the user level. These file systems work similarly to the Amd automounter[16] or Blaze's Cryptographic file system CFS[2] and are based on an NFS[20] server. While it is easier to develop code for such user-level file servers, they suffer from poor performance due to the high number of context switches that take place in order to serve a user request, and because of limitations of the NFS (V.2) protocol such as synchronous writes. This limits the usefulness of such file systems. Worse, user-level NFS-based file servers are more vulnerable to protocol and network compromises, since they use RPC mechanisms and the network to communicate with the kernel and user processes.

We advocate a different solution to these problems: writing kernel-resident file systems that use existing unchanged native file systems, and expose to the user an interface (Vnode/VFS) that is generally similar even across different operating systems. Doing so results in performance comparable to that of kernel-resident systems, with development effort on par with user-level file systems.

Specifically, we provide a template *Wrapper File System* called Wrapfs. Wrapfs can *wrap* (mount) itself on top of one or more existing directories, and act as an intermediary between the user accessing the mount point and the lower level file system on which it was mounted. Wrapfs can then transparently change the behavior of the file system as seen by users, while keeping the underlying media unaware of the upper-level changes. The templates for Wrapfs take care of many file system internals and operating system bookkeeping, and provide the developer with simple hooks to manipulate or modify the data, names, and attributes of the lower file system's objects.

Wrapfs templates exist for several common operating systems: Solaris, Linux, and FreeBSD. Wrapfs can be ported to any operating system with a Vnode interface that provides a private opaque pointer in each of the data structures comprising the interface. The performance overhead imposed by Wrapfs is only 5–7%.

1.1 The Stackable Vnode Interface

Wrapfs is implemented as a stackable vnode interface. A *Virtual Node* or *vnode* is a data structure used within Unix-based operating systems to represent an open file, directory, device, or other entity (e.g., socket) that can appear in the file system name-space. A vnode does not expose what type of physical file system it implements. The *vnode interface* allows higher level operating system modules to perform operations on vnodes uniformly.

One improvement to the vnode concept is *vnode stacking*[8, 17, 23], a technique for modularizing file system functions by allowing one vnode interface to call another. Before stacking existed, there was only a single vnode interface; higher level operating system code called the vnode interface which in turn called code for a specific file system. With vnode stacking, several vnode interfaces may exist and may call each other in sequence: the code for a certain operation at stack level N typically calls the corresponding operation at level $N - 1$, and so on. Before calling the next level, a layer may modify file objects and file attributes to provide added security. For example, it can encrypt data pages or check if a key was provided before writing certain files.

Figure 1 shows the structure for a simple, single-level stackable wrapper file system. System calls are translated into vnode level calls, and those invoke their Wrapfs equivalents. Wrapfs again invokes generic vnode operations, and the latter call their respective *lower level* file system specific operations such as UFS. Wrapfs can also call the lower level file system directly, by invoking the respective vnode operations of the lower vnode. It accomplishes that without knowing who or what type of lower file system it is calling.

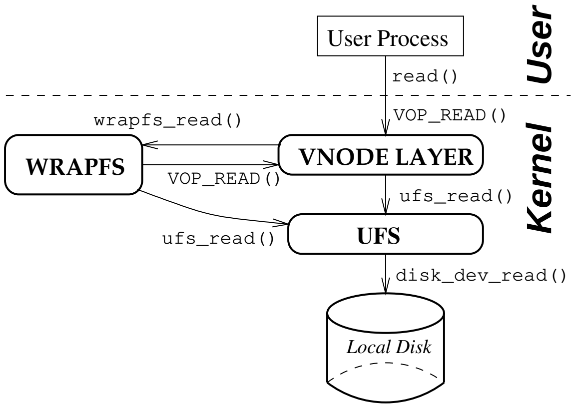

Figure 1: A Vnode Stackable File System

The rest of this paper is organized as follows. Section 2 discusses the design of Wrapfs. Section 3 details Wrapfs's implementation, and issues relating to its portability to other platforms. Section 4 describes nine examples of security file systems written using the Wrapfs templates and illustrating features such as intrusion detection, intrusion avoidance, analysis, access control, and encryption. Section 5 evaluates the performance and portability of our example file system. We survey related works in Section 6 and offer concluding remarks in Section 7. Two appendices follow, covering implementation and portability issues.

2 Design

When designing Wrapfs, we concentrated on the following aspects:

1. What file system objects and file system attributes would developers of security systems want to change.
2. How to simplify the job of developers for the most common file system changes desired.
3. How to allow advanced developers full access to file system internals.
4. User-level issues.
5. Performance.

The first four points are discussed below. Performance is addressed in detail in Section 5.

2.1 What to Change in a File System

Without changing a given Vnode interface, we have identified three items that file system developers want to inspect or manipulate: file data, names, and attributes. Changing file data is the most obvious (e.g., encryption).

Changing file names as part of, say, encryption is also desired. For example, a file system may refuse to create files that contain rarely used characters such as whitespace and other non-printable characters, or visually confusing names such as "..." (three dots) as these are sometimes used by intruders to obscure their tracks. More advanced use can be made by inspecting file names and selectively manipulating them. For example, a file system that adds immutable file support will need to have a list of file names to consider untouchable. Several file systems can place auxiliary information (such as access keys) in files that are hidden from normal view (beyond the obvious "dot" files) and are used only internally by the file system.

Working with file attributes promises some of the most interesting file systems, as attributes fundamentally reflect existing Unix file access control. For example, a file system can perform UID or GID mapping based on the file's ownership. It can exploit seldom used mode bits: the setuid bit on directories can be used to indicate immutable directories. Attempting to modify setuid binaries without prior authentication can be prevented. For every given file F, if a file named `.acl.`F exists, the file system can read the contents of that file and interpret them as additional access grants or revocations to the file F.

Advanced developers may wish to combine changing various aspects of files with additional coding inside specific vnode operations. For example, a file system may wish to track removal of vital files such as logs that can be used to analyze attackers' actions. Attempts to remove the files can be transparently translated into file renaming. The name chosen can be a special name that is hidden from normal view: it will not get listed with the rest of the files in the directory, but be available on the underlying file system. A generalized scheme can include file versioning.

This list of examples is not exhaustive. It should be considered only a hint of what can be accomplished with level of flexibility that Wrapfs offers.

2.1.1 The Wrapfs API

Command	**Input Argument**	**Output Argument**
encode_data	buffer from user space	encoded (same size) buffer to be written
decode_data	buffer read from lower level file system	decoded (same size) buffer to pass to user space
encode_filename	file name passed from user system call	encoded and allocated file name of any length to use in lower level file. Encoded string must be a valid Unix file name. system
decode_filename	file name read from the lower level file system	decoded and allocated file name of any length to pass back to a user process
encode_attr	attributes passed from upper level VFS (ownership, group, modes, etc.).	(optionally) modified attributes to pass to the lower level file system
decode_attr	attributes read from the lower level file system	(optionally) modified attributes to pass to the upper VFS, and possibly out to the user process.

Table 1: Wrapfs Developer API

The API for the Wrapfs developer is summarized in Table 1 and is described here. It consists of six calls to encode and decode file data, names, or attributes. Since it may be necessary to perform more sophisticated operations in these calls, they are passed additional information such as the current vnode, VFS, user credentials, etc.

In order to simplify the manipulation of file data, and to enable MMAP operations (necessary for executing binaries), we perform data manipulations in a size that is native to the operating system, usually 4KB or 8KB. Another compelling reason for manipulating only whole pages is that some file data changes may require it. Some encryption algorithms work on blocks of data of a known fixed size such that bytes within the block depend on preceding bytes[22, 27]. It was therefore important to confine users of Wrapfs to manipulating a fixed size data buffer.

To keep Wrapfs simple, we decided that the data encoding and decoding calls will return a buffer of the same size as the one passed to it. This design decision excludes the possibility of using algorithms such as compression and decompression, because such algorithms change the size of their input data, making file offset calculations costly. Supporting such algorithms would have complicated Wrapfs considerably. Therefore we left this support out of the first implementation of Wrapfs.

We decided that all Vnode calls that write file data will call the function `encode_data` before writing the data to the lower level file system. Then, all Vnode calls that read file data will call the function `decode_data` after reading the data from the lower level file system. In a similar fashion, all Vnode functions that manipulate file names or attributes have the appropriate encode or decode function called in the right places.

The user of Wrapfs who wishes to manipulate files and their names or attributes need not worry about which Vnode functions use them, how directory reading (`readdir`) is being accomplished, about holding and releasing locks, updating reference counts, or caching. The file system developer only needs to fill in the relevant encoding and decoding functions. Wrapfs takes cares of all these operating system internals.

We are making the full sources to Wrapfs publicly available. This way it is possible for file system developers to modify every aspect of a prototype, not just through the six API calls. This also allows the security community to validate and improve the templates.

2.2 User Level Issues

There are three important issues relating to the extension of the Wrapfs API to user-level: mount points, caching, and ioctls.

2.2.1 Mount Points

Wrapfs supports two ways of mounting a file system: a regular mount and an overlay mount. In a regular mount two pathnames are given: one for the mount point (say `/mnt`), and one for the directory to stack on (the mounted directory `/usr`). For example `mount -t wrapfs /mnt /usr`. After the mount is complete, there are two ways to access the mounted-on file system. Access via the mounted-on directory (`/usr`) yields the lower level files without going through Wrapfs. However, access via the mount point (`/mnt`) will go through Wrapfs first. This mount style exposes the mounted directory to user processes, and is useful for debugging purposes and for backups to proceed faster. (That users can bypass the mount point is a general property of stacking, not one brought on by Wrapfs). For example, in an encryption file system, a backup utility can backup files faster and safer if it uses the lower file system's files (ciphertext), rather than the ones through the mount point (cleartext).

The second mount style, an overlay mount, is accomplished using `mount -t wrapfs -O /usr`. Here, Wrapfs is mounted directly on top of `/usr`. Accessing files such as in `/usr/ucb` must go though Wrapfs. There is no easy way to get to the original file system's files under `/usr` without passing through Wrapfs first. This mount style makes backups and debugging more difficult, but has the advantage of hiding the lower level file system from user processes.

We consider an overlay mount more secure and thus made it the default mount style in Wrapfs. A sophisticated attacker might be able to overlay another file system whose purpose would be to bypass several layers and get directly into the lowest level file system. Such an attack requires root privileges, source access to all of file systems currently mounted, and understanding of kernel internals. The attacker would have to carefully follow kernel data structures to reach the ones representing the lowest level file system. This attack is therefore no easier than kernel memory manipulation via `/dev/kmem`.

2.2.2 Cache Coherency

An important point that relates to the mount style is that of caching. Most file systems cache pages to improve performance. When a stackable file system is used on top of, say, UFS, both layers may cache pages independently. Cache incoherency could result if pages at different layers are modified independently, but that could only occur in regular mounts; overlay mounts do not let user processes modify data pages at the lower layers. A mechanism for cache synchronization through a centralized cache manager was proposed by Heidemann[7], but that solution involved modifying the rest of the operating system and other file systems.

We decided that Wrapfs will perform its own caching, and may cache pages at the lower layer depending on the mount style. If the mount style was regular, Wrapfs caches pages also at the lower layer, because this improves performance when accessing files directly through the lower layer; in fact there is no way to avoid caching pages at the lower layer in a regular mount style, because processes can access files directly through the lower level file system. We also decided that the higher the layer is, the more authoritative it would be. For example, when writing to disk, cached pages for the same file in Wrapfs would overwrite their UFS counterparts. This policy matches the most common case of cache access, through the uppermost layer.

If an overlay mount style was used, Wrapfs does not cache pages at the lower layer. This cuts memory usage for pages by half, and performance is still very good, as pages are served off of the upper (Wrapfs) layer, where pages are always cached.

2.2.3 Ioctls

The third important user-level issue relates to the `ioctl(2)` system call. Ioctls have been used for years as simple means to extend the API of a file system beyond that which system and Vnode calls offer. Wrapfs allows its user to define new ioctl codes and implement their associated actions. Two ioctls are already defined: one to set a debugging level, and one to query it. Wrapfs comes with many debugging traces that can be turned on or off at run time by a root user. Other possible ioctls that can be implemented by specific file systems include passing and retrieving additional information to and from the file system. An encryption file system (such as the one described in Section 4.9) might use an ioctl mechanism to set encryption keys.

3 Implementation

Wrapfs is the template from which we wrote the example security file systems discussed in Section 4. Our first implementation concentrated on the Solaris 2.5.1 operating system for several reasons. Solaris is a popular commercial operating system, and it includes a standard vnode interface. In addition, we had access to kernel sources. The next two implementations we worked on were for Linux 2.0 and FreeBSD 3.0. We chose them because they are sufficiently different from Solaris and each other, they represent another large section of the Unix market , and they also come with sources.

By implementing Wrapfs and further examples using it for these three operating systems we hope to prove that practical non-trivial stackable file systems are portable to sufficiently different Unix operating systems, and that the effort involved in porting them is small (see Section 5.4). Appendix A provides more details of the first implementation, Solaris. In Appendix B we discuss the differences in implementation for the Linux and FreeBSD ports.

3.1 Stacking

Wrapfs was initially similar to the Solaris loopback file system (lofs)[26]. Lofs passes all Vnode and VFS operations to the lower layer, but it only stacks on directory vnodes. Wrapfs stacks on every vnode, and makes identical copies of data blocks, pages, and file names in its own layer, so they can be changed independently of the lower level file system. Wrapfs does not explicitly manipulate objects in other layers. It appears to the upper VFS as a lower-level file system. Concurrently, Wrapfs treats lower-level file systems as an upper-layer. This allows us to stack multiple instances of Wrapfs or file systems derived thereof on top of each other.

The key point that enables stacking is that the major data structures used in the file system (`struct vnode` and `struct vfs`) contain a field into which file system specific data can be stored. Wrapfs uses that private field to store several pieces of information, most notably a pointer to the corresponding lower level file system's vnode and VFS. When a vnode operation in Wrapfs is called, it finds the lower level's vnode from the current vnode, and repeats the same operation on the lower level vnode.

4 Examples

This section details the design and implementation of several sample security file systems we wrote based on Wrapfs. The examples range from simple to complex. We attempted to cover a range of security features: intrusion detection, intrusion avoidance, analysis, access control, and encryption.

1. **Snoopfs**: detects and warns of attempted access to users' files by other non-root users.
2. **Rofs**: selectively marks subdirectories as read-only.
3. **Wofs**: only allows writing new files.
4. **Aclfs**: adds simple access control lists.
5. **Imfs**: adds immutable file support with key access.
6. **Unrmfs**: allows un-removing files.
7. **Consfs**: allows execution and modification of certain binaries only on the system console.
8. **Rot13fs**: is a trivial encryption file system.
9. **Cryptfs**: is an encryption file system.

It should be noted that these examples are merely experimental file systems intended to illustrate the kinds of file systems that can be written based on Wrapfs. We do not consider them to be complete solutions. Whenever possible, we illustrate potential enhancements to our examples. We hope this would serve to convince the readers of the flexibility and ease of writing new file systems using Wrapfs. Our primary intent is to demonstrate that useful file systems can be written with a small amount of code and rapidly prototyped. Moreover, several such file systems can be stacked together, to form a union of their functionality.

4.1 Snoopfs

User files in their home directories are often considered private and personal. Normally, these files can be read by their owner or by the root user (for example during backups). Other sanctioned file access includes files shared via a common Unix group. Any other access trial can be considered a break-in attempt. For example, a manager might want to know if a subordinate tried to `cd` to the manager's `~/private` directory, or an instructor might wish to be informed when anyone tries to read files containing solutions to homeworks or exams.

The one place in a file system where files are initially searched is the vnode `lookup` routine. In order to detect access problems, we first have to perform the lookup on the lower file system, and then check the resulting status. If the status was one of the error codes "permission denied" or "file not found", we know that someone was trying to read a file they do not have access to, or they were trying to guess the names of files. If we detect one of these two error codes, we also check if the process accessing it belongs to the super-user or the file's owner (by checking user credentials). If it was a root user or the owner, we do nothing. Otherwise, we print a warning using the in-kernel `syslog(3)` facility. The warning contains the file or directory name to which access was denied, and the user ID of the process that tried to access it.

We made one additional design change and allowed for selective directory snooping. That is, we allow for this simple intrusion detection algorithm to be turned on only for directories that their owner wants to. We decided to use a seldom used mode bit for directories—the setuid bit—to flag a directory for snooping. This bit can be easily turned on or off by the `chmod(3)` utility.

The implementation of Snoopfs was very simple. In less than one hour we were able to implement it on all three operating systems we have Wrapfs ported to. The sum total of code added to Wrapfs was less than 10 lines of C code. Without Wrapfs, writing Snoopfs would have required several thousand lines of code

Snoopfs makes the point that even good technology can be abused. A malicious user with source access to Snoopfs and a root ID can modify Snoopfs to silently pry into other users' files.

Snoopfs can serve as a prototype for a more elaborate intrusion detection file system. Such a file system can prohibit or limit the creation or execution of setuid and setgid programs; it can deny an attacker from unlinking opened files (so as to continue using them while they are hidden from others). It can also disallow overwriting certain executables that rarely change (such as `/bin/login`) to prevent an attacker from replacing them with trojan versions.

4.2 Rofs

When users export a file system for remote (NFS) mounting, the whole file system is either given write permission or not. Unix directory modes can be used to deny write access to selected directories, but a remote client with compromised root access can easily change these modes by becoming the owner of the directory. Rofs can be used to deny modification of any files in selected directories.

Rofs is first mounted on a file system to be exported, and then Rofs's mount point is exported for NFS mounts. In that way Rofs controls file access on the NFS server side where it is safer. If anyone tries to change the mode of read-only directory, Rofs denies that request and returns "permission denied." Attempts to modify such directories or files within are refused with an error code "read-only file system." The only way write access can be given to such read-only directories is from the host that exported the file system, and going through the disk-based file system directly, not through Rofs. (Only the Rofs mount point is exported, and it is imperative that the actual lower-level file system is not.)

The implementation of Rofs was easy and took less than 20 lines of C code. For each operation that wants to modify state (such as `unlink` or `write`) we first check the modes on the parent vnode. If the modes do not allow writing, we return an error code instead of performing the given vnode operation.

4.3 Wofs

Attackers will often attempt to remove or truncate files that may reveal their tracks, such as logs and tools used to attack. Wofs is a file system that allows only appending to files and the creation of new files. That way attackers cannot remove existing files, nor can they modify existing data within. However, system log utilities such as `syslog(3)` can still append new information to files. No user process is allowed to read back existing files through Wofs. That way attackers could not find out what information might already be known about them.

The implementation of Wofs also took less than 10 lines of C code. The `decode_data` routine always returns an error code such as "permission denied". The `encode_data` routine allows the write to proceed only if the write offset requested was past the end of the file.

Allowing the creation of new files and disallowing their deletion has an added benefit which can be used to deny the installation of trojan programs. A simple enhancement to Wofs can achieve the following: if someone tried to overwrite a root owned file (such as `/usr/ucb/telnetd`), Wofs could implicitly rename the new file to, say `/usr/ucb/.telnetd.trj`. Since this new file now exists, attackers who discovered this would be unable to remove the file. A more clever addition to Wofs will also use `decode_filename` to skip listing files ending with `.trj`. This way such files would not be visible to casual listing through Wofs.

Since Wofs (and Unrmfs, Section 4.6) perform multiple atomic operations, such as renaming files, some guarantees are needed to ensure that multi-layer file system operations work reliably. Wrapfs behaves like a virtual file system (VFS) to the lower file system on which Wrapfs mounts on. Wrapfs takes care of locking file system objects such as vnodes, as mandated by the VFS which called Wrapfs. As such, Wrapfs may be invoked with locked objects (e.g., files, pages, etc.) and will have to lock the underlying, corresponding objects before calling the lower level file system. This implicit ordering in acquiring locks avoids deadlocking, but requires that the lower level file system not be directly accessible: the stackable file system must be overlay mounted.

4.4 Aclfs

Aclfs is used to provide very simple additional access controls. Specifically, it allows for an additional user and Unix group to share access to a directory as if they were the owner and group of the directory. When looking up a file in a directory, Aclfs first performs the normal access checks (in the vnode `lookup` routine). If access to the file was denied but the file exists, Aclfs looks for an additional file named `.acl` in that directory. It then repeats the lookup operation on the originally looked-up file, but using the ownership and group credentials of the `.acl` file. The `.acl` file itself is not modifiable via Aclfs, but only through the lower level file system using normal Unix file creation and access controls. Aclfs assumes that only authorized users would create the `.acl` files before Aclfs is overlay-mounted. Aclfs's implementation used no more than 10 lines of C code over Wrapfs.

There are several possible extensions to this trivial implementation of Aclfs. Instead of using the modes of the `.acl` file, it can contain an arbitrarily long list of user and group IDs to allow access to. The `.acl` file may also include sets of permissions to deny access from, perhaps using negative integers to distinguish them from access permissions. The granularity of Aclfs can be made on a per-file basis; for each file F, access permissions are read from the file `.`F`.acl`, if one exists. Also, ACLs can be used to completely replace regular Unix access control. Finally, new ioctls can be added to ease ACL administration.

4.5 Imfs

Certain files in a Unix system are very important and hardly change, such as the kernel image or system daemons like `/usr/ucb/rshd`. Attackers often try to replace those with trojan versions. Moreover, a careless user or administrator may inadvertently overwrite those by attempting to install pre-packaged software or one built from source. FreeBSD's FFS is able to set an immutable flag on some files so they cannot be overwritten by even the root user, until the immutable flag is turned off. Imfs provides such functionality, but also adds simple key authentication.

For every read-only file F, for which the key file `.`F`.imkey` exists, Imfs will not allow modification to that file unless the key was given. A user-level tool prompts the user for a passphrase, and passes an MD5 hash of it to the kernel via an ioctl on that file. If the key matches the key stored in the `.imkey` file, the process is allowed to modify the file.

Most of the implementation work for Imfs concentrated in the vnode `ioctl` routine. One ioctl is used to set a key initially, while another is used to get a key from a user and compare it to an existing key. Of the 60 lines of C code for Imfs, most were used for general key management.

This initial design does not allow changing or removing keys through Imfs. They have to be removed by removing the `.imkey` file from the lower-level file system. This requires an unmounting of Imfs, as it is implemented as an overlay-only file system (see Section 2.2.1); casual access to the key files by root users is not possible. Clearly, better key management schemes can be used for Imfs, but such discussion is beyond the scope of this paper.

4.6 Unrmfs

Unrmfs can alleviate some of the problems associated with accidental or intentional removal of files, by allowing the reversion of such deletion. Its implementation is simple. When the vnode function `unlink` is called to remove a file F, it instead translates it into a call to `rename` the file to F`.unrm`. An ioctl added to Unrmfs allows the user to un-remove a file by renaming it back. This way a user can avoid accidental removal of files. The code for this file system took 20 lines.

Using a special flag, Unrmfs can be mounted in a more secure mode, intended to help track down attackers. In this mode, Unrmfs will neither list nor allow the removal of the `.unrm` files. If the original file is re-created and removed again, the older `.unrm` file will not be overwritten, so as to keep the old backup file around.

One obvious extension to Unrmfs is to allow versioning. Each time a file F is removed, a new F`.unrm.`N is created, where N is a monotonically increasing integer. Another enhancement may put a limit on the maximum number of versions, while keeping only recent versions. This limits the possibility of filling up a file system with repeated file removals, which could be a form of denial-of-service attack.

4.7 Consfs

Consfs allows root owned binaries to be modified or executed only by a process whose controlling terminal is `/dev/console`. It is intended so that attackers who often come from the outside could not replace vital binaries with trojans, nor could they execute them. This way only administrators with physical access to a machine are able to perform such work.

We wrote Consfs also to illustrate an aspect of Wrapfs that is both a source of functional flexibility and the cause of portability problems. Consfs has to determine—inside the kernel—if the current process has a controlling terminal and that one is the console. Different operating systems use different methods to find this information; different data structures and fields have to be followed. This makes it harder to modify one set of Wrapfs's API calls (Section 2.1.1) in a way that would compile and work on different operating systems. Not surprisingly, Consfs's total code size was about 100 lines of C; about one third of it was unportable code.

4.8 Rot13fs

Before we embarked on a cryptographically strong file system, we wrote one that uses a simple encryption algorithm that works on single bytes—rot13. Rot13fs encrypts only file data, not names. We developed Rot13fs at the same time we worked on Wrapfs. Rot13fs was useful in finding out which features of a stackable file system were generally useful for the Wrapfs template, and which were not.

4.9 Cryptfs

The encryption file system Cryptfs is the most involved file system we have designed and implemented based on Wrapfs. This section summarizes the design and implementation of Cryptfs. More detailed information is available in a separate report[31].

For an encryption algorithm we picked Blowfish[22], a 64 bit block cipher that was designed to be fast, compact, and simple. Blowfish is suitable in applications where the keys do not change often such as in automatic file decryptors. It can use variable length keys as long as 448 bits. We kept the default 128 bit long keys. In relation to the algorithm, we picked the Cipher Block Chaining (CBC)[21] encryption mode because it allows us to encrypt byte sequences of any length, which is suitable for encrypting file names. However, we decided to use CBC only within each block encrypted by Cryptfs. This way ciphertext blocks (of 4–8KB) would not depend on previous ones, allowing us to decrypt each block independently. Moreover, since Wrapfs naturally lets us manipulate file data in aligned units of even page size multiples, encrypting them promised to be simple.

Next, we decided to encrypt file names as well, to provide stronger security. We realized that we should not encrypt the "." and ".." directory names so that the lower level Unix file system remains intact. Furthermore, since encrypting file names may result in characters that are illegal in file names (such as nulls and forward slashes "/"), we settled on further uuencoding the resulting encrypted strings. This eliminated the unwanted characters and guaranteed that all file names consisted of printable characters that are valid in file names.

We decided that Cryptfs mounts will be regular. This facilitates faster and more secure backups of (ciphertext) user files directly from the lower level file system.

4.9.1 Key Management

The next important design issue for Cryptfs was key management. We decided that only the root user would be allowed to mount an instance of Cryptfs, but could not automatically encrypt or decrypt files. To thwart an attacker who gains access to a user's account or to root privileges, Cryptfs maintains keys in an in-memory data structure that associates keys not with UIDs alone but with the combination of UID and session ID. To succeed in acquiring or changing a user's key, attackers would not only have to break into an account, but also arrange for their processes to have the same session ID as the process that originally received the user's passphrase. This is a more difficult attack, requiring session and terminal *hijacking* or kernel-memory manipulations.

Using session IDs to further restrict key access does not burden users during authentication. Login shells and daemons use `setsid(2)` to set their session ID and detach from the controlling terminal. Forked processes inherit the session ID from their parent. Therefore, users would normally have to authorize themselves only once in a shell. From this shell they could run most other programs that would work transparently and safely with the same encryption key. Note that Cryptfs can be modified easily to use only one key per mount instance. This provides a separation of mechanism, since a separate file system is instantiated for each user. Wrapfs templates, in general, do not impose a restriction on how many times they can be instantiated.

We designed a user tool that prompts users for passphrases that are at least 16 characters long. The tool hashes passphrases using MD5 and passes them to Cryptfs using a special `ioctl(2)`. The tool can also instruct Cryptfs to delete or reset keys.

Our design decouples key possession from file ownership. For example, a group of users who wish to edit a single file would normally do so by having the file group-owned by one Unix group and add each user to that group. However, Unix systems often limit the number of groups a user can be a member of to 8 or 16. Worse, there are often many subsets of users who are all members of one group and wish to share certain files, but are unable to guarantee the security of their shared files because there are other users who are members of the same group; e.g., many sites put all of their staff members in a group called "staff", students in the "student" group, guests in another, etc. With our design, users can further restrict access to shared files only to those users who were given the key.

One disadvantage of this design is reduced scalability with respect to the number of files being encrypted and shared. Users who have many files encrypted with different keys will have to switch their effective key before attempting to access files that were encrypted with a different one. We did not perceive this to be a serious problem for two reasons. First, the amount of Unix file sharing of restricted files has always been limited. Most shared files are generally world readable and thus do not require encryption. Secondly, with the proliferation of windowing systems, users can associate different keys with different windows.

In the current design, Cryptfs uses one Initialization Vector (IV) per mount, used to jump start a sequence of encryption. If not specified, a predefined IV is used. The superuser mounting Cryptfs can choose a different one, but that will make the first eight bytes of all previously encrypted files undecipherable with the new IV. Files that use the same IV and key produce identical ciphertext blocks that are subject to analysis of identical blocks, because using fixed IVs with CFB-mode encryption leaks plaintext. By default, CFS[2] uses no IVs, and we felt that using a fixed one produces a reasonably strong security for this extensible prototype file system.

With the Wrapfs templates, additional security features could be tested and added with relative ease; we expect future developers to implement stronger security mechanisms into Cryptfs. One possible extension to Cryptfs would be to use different IVs for different files, based on the file's inode number and perhaps in combination with the page number (so that each page within a file can be encrypted with a different IV). Making the IV depend on inode numbers or page numbers would not completely secure data pages, as detailed in SFS[5]. Some methods to improve the security of block ciphers are detailed by Luby and Rackoff[10]. Faster construction of block ciphers is outlined by Lucks and use dedicated hash functions[11]. Other more obvious extensions to Cryptfs include the usage of different encryption algorithms, perhaps different ones per user or file.

Cryptfs adds more than 1500 lines of C code to Wrapfs. 70% of it is the Blowfish implementation. Most of the rest is for key management and special file name handling.

5 Evaluation

When evaluating the file systems we have built, we concentrated mostly on the more complex ones: Wrapfs and its derivative Cryptfs. To test the stability of these two, we ran two concurrent loops of the tests described below for over two weeks. We ensured that no errors occurred, the system remained operational, and all file systems involved incurred no corruptions.

In the following sections we evaluate the performance of our file systems, and then their portability. As each of these file systems is based on several others, our performance measurements were aimed at identifying the overhead that each layer adds. The main goal was to prove that the overhead imposed by stacking is small enough and comparable to other stacking work[8, 23].

5.1 Wrapfs

For most of our tests, we included figures for a native disk-based file system because disk hardware performance can be a significant factor. This number should be considered the base to which other file systems compare. We included figures for Wrapfs (our full-fledged stackable file system) and for lofs (the low-overhead simpler one), to be used as a base for evaluating the cost of stacking. When using lofs or Wrapfs, we mounted them over a local disk based file system.

For testing Wrapfs, we decided to use as our performance measure a full build of Am-utils[29], a new version of the Berkeley Amd automounter. The test auto-configures the package and then builds it. Only the sources for Am-utils and built binaries were using the file system being tested; compiler tools and such were left outside. The configuration runs several hundred (600–700) small tests, many of which are small compilations and executions. The build phase compiles about 50,000 lines of C code spread among several dozen files and links about a dozen binaries. The whole procedure contains a fair mix of CPU and I/O bound operations as well as file system operations: many writes, binaries executed, small files created and unlinked, many reads and lookups, and a few directory and symbolic link creations. We felt this test is a more realistic measure of the overall file system performance, and would give users a better feel for the expected impact Wrapfs might have on their workstation. For each file system measured, we ran 12 successive builds on a quiet system, measured the elapsed times of each run, removed the first measure (cold cache) and averaged the remaining 11 measures. The results are summarized in Table 2.[1] The standard deviation for the results reported in this section did not exceed 0.8% of the mean. Finally, there is no native lofs for FreeBSD (and the nullfs available is not fully functional, as discussed in Section B.2).

File System	SPARC 5		Intel P5/90		
	Solaris 2.5.1	Linux 2.0.34	Solaris 2.5.1	Linux 2.0.34	FreeBSD 3.0
ext2/ufs/ffs	1242.3	1097.0	1070.3	524.2	551.2
lofs	1251.2	1110.1	1081.8	530.6	n/a
wrapfs	1310.6	1148.4	1138.8	559.8	667.6
cryptfs	1608.0	1258.0	1362.2	628.1	729.2
crypt-wrap	22.7%	9.5%	19.6%	12.2%	9.2%
nfs	1490.8	1440.1	1374.4	772.3	689.0
cfs	2168.6	1486.1	1946.8	839.8	827.3
cfs-nfs	45.5%	3.2%	41.6%	8.7%	20.1%
crypt-cfs	34.9%	18.1%	42.9%	33.7%	13.5%

Table 2: Time (in seconds) to build a large package on a given platform, and inside a specific file system. The percentage lines show the overhead difference between some file systems.

First, we need to evaluate the performance impact of stacking a file system. Lofs is only 0.7–1.2% slower than the native disk based file system. A single Wrapfs layer adds an overhead of 4.7–6.8% for Solaris and Linux systems, but that is comparable to the 3–10% degradation previously reported for null-layer stackable file systems[8, 23]. On FreeBSD, however, Wrapfs adds an overhead of 21.1% compared to FFS: because of limitations in nullfs, we were forced to use synchronous writes exclusively. Wrapfs is more costly than lofs because it stacks over every vnode and keeps its own copies of data, while lofs stacks only on directory vnodes, and passes all other vnode operations to the lower level verbatim.

5.2 Cryptfs

To measure the performance of Cryptfs and CFS[2], we performed the same tests we did for Wrapfs described in Section 5.1. These results are also summarized in Table 2, for which the standard deviation did not exceed 0.8% of the mean.

[1] All machines used in these tests had 32MB RAM.

We used Wrapfs as the baseline for evaluating the performance impact of the encryption algorithm. The only difference between Wrapfs and Cryptfs is that the latter encrypts and decrypts data and file names. The line marked as "*crypt-wrap*" in Table 2 shows that percentage difference between Cryptfs and Wrapfs for each operating system. Cryptfs adds an overhead of 9.2–22.7% over Wrapfs. That is a significant overhead but is unavoidable. It is the cost of the Blowfish encryption code, which, while designed as a fast software cipher, is still CPU intensive.

Measuring the encryption overhead of CFS was more difficult. CFS is implemented as a user-level NFS file server, and we also ran it using Blowfish. We expected CFS to run slower due to the number of additional context switches that must take place when a user-level file server is called by the kernel to satisfy a user process request, and due to NFS V.2 protocol overheads such as synchronous writes. While CFS is based on NFS, it does *not* use the NFS server code of the given operating system. CFS serves user requests directly to the kernel. Since NFS server code is implemented in general inside the kernel, it means that the difference between CFS and NFS is not just that of the encryption, but also due to context switches. However, the NFS server in Linux 2.0 is implemented in user-level, and is thus also affected by context switching overheads. If we ignore the fact that CFS and Linux's NFS are two different implementations, and just compare their performance, we see that CFS is 3.2–8.7% slower than NFS on Linux. This is likely to be the overhead of the encryption in CFS. That overhead is somewhat smaller than the encryption overhead of Cryptfs because CFS is more optimized than our Cryptfs prototype; CFS precomputes large stream ciphers for its encrypted directories.

We have performed more finer-grained tests on the file systems listed in Table 2, specifically reading and writing of small and large files. These tests were designed to isolate and show the performance differences for specific file system operations. They show that Cryptfs is anywhere from 43% to an order of magnitude faster than CFS. Since the encryption overhead is roughly 3.2–22.7%, we can assume that rest of the difference comes from the reduction in number of context switches. Details of these additional measurements are available in a separate report[31].

5.3 Other Wrapfs-Based File Systems

The other file systems we developed using Wrapfs are simple, so we did not perform such rigorous performance measurements on them as we did with Wrapfs and Cryptfs. Only Rot13fs and Cryptfs have any significant impact on performance over that which was reported for Wrapfs. We ran similar the same (Am-utils) package build on these and have noted that in all cases, performance degraded by no more than an additional 8% over Wrapfs. We are convinced that with additional optimization time, these prototypes can be made to run even faster.

5.4 Portability

We first developed Wrapfs and Cryptfs on Solaris 2.5.1. As seen it Table 3, it took us almost a year to initially develop Wrapfs and Cryptfs together for Solaris. As we gained more experience, the time to port the same file system to a new operating system grew significantly shorter. Developing these file systems for Linux 2.0 was a matter of days to a couple of weeks, not months. The Linux 2.0 port would have been faster had it not been for Linux's rather different vnode interface.

File System	**Solaris 2.x**	**Linux 2.0**	**FreeBSD 3.0**	**Linux 2.2**
wrapfs	9 months	2 weeks	5 days	1 week
cryptfs	3 months	1 week	2 days	1 day
all others	$\leq$1 day	$\leq$1 day	$\leq$1 day	$\leq$1 day

Table 3: Time to Develop and Port File Systems

The following port, FreeBSD 3.0, was even faster. This was mostly due to the great similarities between the vnode interfaces of Solaris and FreeBSD. We recently also accomplished these ports to the Linux 2.2 kernel. The Linux 2.2 vnode interface makes significant changes to the 2.0 kernel, which is why we list it as another porting effort. We held off on this port until the kernel became more stable (it only recently entered final development phase).

Another metric of the effort involved in porting Wrapfs is the size of the code. Table 4 shows the total number of source lines for Wrapfs, and breaks it down to three categories: common code that needs no porting, code that is easy to port by simple inspection of system headers, and code that is hard to port. The hard-to-port code accounts for more than two-thirds of the total and is the one involving the implementation of each Vnode and VFS operation (operating system specific).

Porting Difficulty	**Solaris 2.x**	**Linux 2.0**	**FreeBSD 3.0**	**Linux 2.2**
Hard	80%	88%	69%	79%
Easy	15%	7%	26%	10%
None	5%	3%	5%	11%
Total Lines	3431	2157	2882	3279

Table 4: Wrapfs Code Size and Porting Difficulty

The difficulty of porting file systems written using Wrapfs depends on several factors. If plain C code was used in the encoding and decoding routines alone, the porting effort would be minimal or none. Wrapfs, however, does not restrict the user from calling any in-kernel operating system specific function. Calling such functions could make portability more difficult.

Wrapfs can be ported to other operating systems that fulfill these prerequisites, such as AIX, HP-UX, Irix, Digital Unix, NetBSD, OpenBSD, BSDI, and SunOS 4.x. However, in practice we found that source access to an operating system's VFS is necessary to port Wrapfs, because many kernel APIs are unknown outside the kernel.

6 Related Work

6.1 Other Stackable File Systems

Vnode stacking was first implemented by Rosenthal (in SunOS 4.1) around 1990 [18]. A few other works followed Rosenthal, such as further prototypes for extensible file systems in SunOS [23], and the Ficus layered file system [6, 9] at UCLA.

Several newer operating systems offer a stackable file system interface. Such operating systems have the potential of easy development of file systems offering a wider range of services. Their main disadvantages are that they are not portable enough, not sufficiently developed or stable, and they are not available for common use. Also, they require significant changes to existing operating systems and existing file systems to support stacking. Finally, new operating systems with new file system interfaces are not likely to perform as well as ones that are several years older. Wrapfs, on the other hand, offers an API that is portable across operating systems, and requires no changes to existing operating systems or other file systems.

The *Herd of Unix-Replacing Daemons* (HURD)[3] from the Free Software Foundation (FSF) is a set of servers running on the Mach 3.0 microkernel[1] that collectively provide a Unix-like environment. HURD file systems are implemented at the user level. The novel concept introduced by HURD is that of the translator. A translator is a program that can be attached to a pathname and perform specialized services when that pathname is accessed. Writing a new translator is a matter of implementing a well defined file access interface and filling in such operations as opening files, looking up file names, creating directories, etc.

Spring is an object-oriented research operating system built by Sun Microsystems Laboratories[13]. It was designed as a set of cooperating servers on top of a microkernel. Spring provides several generic modules that offer services useful for a file system: caching, coherency, I/O, memory mapping, object naming, and security. Writing a file system for Spring requires defining the operations to be applied on the file objects. Operations not defined are inherited from their parent object.

6.2 Other Encryption File Systems

CFS[2] is a portable user-level cryptographic file system based on NFS. It is used to encrypt any local or remote directory on a system, accessible via a different mount point and a user-attached directory. Users first create a secure directory and choose the encryption algorithm and key to use. A wide choice of ciphers is available and great care was taken to ensure a high degree of security. CFS's performance is limited by the number of context switches that must be performed and the encryption algorithm used.

TCFS[4] is a modified client-side NFS kernel module that communicates with a remote NFS server. TCFS is available only for Linux systems, and both client and server must run on Linux. TCFS allows finer grained control over encryption—individual files or directories can be encrypted by turning on or off a special flag. Unfortunately, TCFS does not support the Blowfish cipher.

Linux comes with a special *loop* device driver which allows users to use regular files as block devices, and even create a file system on such looped files[28]. Optional encryption modules can be used to encrypt file data on a block basis. The loop device, however, is not a replacement for a real file system. It does not truly stack on top of all other file systems, it does not support 100% Unix file system semantics (such as access to block and character devices, as well as advisory locking). This is mostly because the loop device does not call the underlying file system functions of the native file system it loops over. For example, the loop device cannot be used on top of NFS, and no more than 7 loop devices may be used at once. Wrapfs is a full featured file system which is independent of the file systems below it; any number of instances of Wrapfs can be mounted on top of Ext2fs, NFS, or even other instances of Wrapfs.

7 Conclusions

Wrapfs and the examples we built from it prove that useful, non-trivial, stackable security file systems can be implemented on modern operating systems without having to change the rest of the system. Many practical security features were added using small amounts of code and implemented in less than a day. Better performance and enhanced security was achieved by running the file systems in the kernel instead of at user-level. File systems built from Wrapfs are more portable than other kernel-based file systems because they interact directly with a (mostly) standard vnode interface, as the quick ports to Linux and FreeBSD showed.

Estimating the complexity of software is a difficult task. Kernel development in particular is slow and costly because of the hostile development environment. Furthermore, personal experience of the developers figure heavily in the cost of development and testing of file systems. Nevertheless, it is our assertion that once Wrapfs is ported to a new operating system, other non-trivial file systems built from it can be prototyped in a matter of hours or days. We estimate that Wrapfs can be ported to any operating system in less than one month, as long as it has a Vnode interface that provides a private opaque field for each of the major data structures of the file system. In comparison, traditional file system development often takes many months to several years and requires large teams of programmers.

Wrapfs saves the developers from dealing with kernel internals, and allows them to concentrate on the specifics of the security file system they are developing. We hope that with Wrapfs and the example security file systems we have built, other developers would be able to prototype new file systems to try new security ideas, develop fully working ones, and port them to various operating systems—bringing the complexity of file system development down to the level of common user-level software.

7.1 Future Work

As mentioned in Section 2.1.1, this first implementation of Wrapfs does not support algorithms that change the input size (e.g., compression). We have outlined several possible designs for supporting such algorithms efficiently and plan to implement a few of them to find out which one best balances the added code complexity with performance.

We also plan to port our system to Windows NT. NT has a different file system interface than Unix's vnode interface. NT's I/O subsystem defines its file system interface. NT *Filter Drivers* are optional software modules that can be inserted above or below existing file systems[14]. Their task is to intercept and possibly extend file system functionality. One example of an NT filter driver is its virus signature detector. It is therefore possible to emulate file system stacking under NT.

8 Acknowledgments

We thank the following people for their help reviewing this paper: Jerry Altzman, Fuat Baran, Gail Kaiser, Jason Nieh, and Sal Stolfo. This work was partially made possible thanks to NSF infrastructure grant number CDA-96-25374.

A Stacking Implementation Details

This appendix explains some of the more difficult parts of the implementation of Wrapfs and unexpected problems we encountered. Additional details are beyond the scope of this paper and are available elsewhere[32].

A.1 Reading and Writing

By design, we perform reading and writing on whole blocks of size matching the native page size. Whenever a read for a range of bytes is requested, we compute the extended range of bytes up to the next page boundary, and apply the operation to the lower file system using the extended range. Upon successful completion, the exact number of bytes requested are returned to the caller of the vnode operation.

Writing a range of bytes is more complicated than reading. Within one page, bytes may depend on previous bytes (e.g., encryption), so we have to read and decode parts of pages before writing other parts of them. The example depicted in Figure 2 shows what happens when a process asks to write bytes of an existing file from byte 9000 until byte 25000. Let us also assume that the file in question has a total of 4 pages (32768) of bytes in it. First, compute the extended range of bytes that covers pages 1–3, read those three pages in, and decode them. Then, write into these three (in memory) pages the new bytes passed from the user, at the proper offsets. Next, encode these three pages. Finally, write out to the lower level file system only those bytes that could have changed.

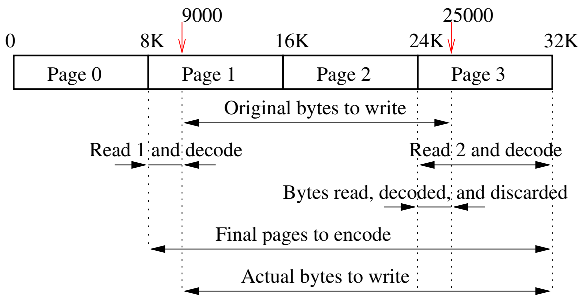

Figure 2: Writing Bytes in Wrapfs

A.2 File Names and Directory Reading

The `readdir` vnode operation is implemented in the kernel as a restartable function. A user process calls the readdir C library call, which is translated into repeated calls to the `getdents(2)` system call, passing it a buffer of a given size. The buffer is filled by the kernel with enough bytes representing files in a directory being read, but no more. If the kernel has more bytes to offer the process (i.e., the directory has not been completely read) it will set a special EOF flag to false. As long as the C library call sees that the flag is false, it must call `getdents(2)` again. Each time it does so, it will read more bytes starting at the file offset of the opened directory as was left off during the last read.

The important issue with directory reading is how to continue reading the directory from exactly the offset it was left off the last time. This is accomplished by recording the last position and ensuring that it is returned to us upon the next invocation. We implemented `readdir` by reading a number of bytes from the lower level directory, breaking these bytes into individual records representing one directory entry at a time (`struct dirent`), calling `decode_filename` on each name, and then composing a new block of `dirent` data structures containing the decoded names. This new block is returned to the caller. If there is more data to read, then the EOF flag is set to false before returning from this function, and the last read offset is recorded.

A.3 Memory Mapping

To support MMAP operations and execute binaries we had to implement memory-mapping vnode functions. As discussed in Section 2.2.1, Wrapfs maintains its own cached (decoded) pages, while the lower file system keeps cached encoded pages.

When a page fault occurs, the kernel calls the vnode operation getpage. This function retrieves one or more pages from a file. For simplicity, we implemented it as repeatedly calling a function which retrieves a single page, getapage. The implementation of getapage appeared simple. We first look for the page in the cache and return it if found. Otherwise we allocate a new page, call the getpage routine on the lower level file system, and then decode the bytes in the page just read into the new page. The new page now contains decoded bytes. It is added to the page cache and Wrapfs returns it to the caller.

The implementation of putpage was similar to getpage. In practice we also had to carefully handle two additional details, to avoid deadlocks and data corruption. First, pages contain several types of locks, and these locks must be held and released in the right order and at the right time. Secondly, the MMU keeps mode bits indicating status of pages in hardware, especially the referenced and modified bits. We have to update and synchronize the hardware version of these bits with their software version kept in the pages' flags. For a file system to have to know and handle all of these low-level details blurs the distinction between the file system and the VM system, and further complicates porting.

B Portability to Other Operating Systems

This appendix describes the differences in the implementation of Wrapfs from the initial port (Solaris) to the ports that followed: Linux and FreeBSD.

B.1 Linux

When we began the Solaris work we referred to the implementation of other file systems such as lofs. Linux 2.0 did not have a loopback file system as part of standard distributions, but we were able to locate a prototype[2] and use it in our port.

The Linux Vnode and VFS interfaces contains a different set of functions and data structures than Solaris, but it operates in a similar fashion. In Linux, much of the common file system code was extracted and moved to a generic (higher) level. Many generic file system functions exist that can be used by default if the file system does not define its own version thereof. This leaves the file system developer to deal with only the core issues of the file system. For example, Solaris User I/O (uio) structures contain various fields that must be updated carefully and consistently. Linux simplifies data movement by passing vnode functions such as read and write a simple allocated (char *) buffer and an integer describing how many bytes to read into or write out of the buffer passed.

Memory mapped operations are also easier in Linux. The vnode interface in Solaris includes functions that must be able to manipulate one or more pages. In Linux, a file system handles one page at a time, leaving page clustering and multiple-page operations to the higher and more generic code.

Directory reading was much simpler in Linux. In Solaris, we had to read a number of raw bytes from the lower level file system, and parse them into chunks of sizeof(struct dirent), set the proper fields in this structure, and append the file name bytes to the end of the structure. In Linux, we provided the kernel with a callback function for iterating over the entries in a directory. This function was called by higher level code and asked us to simply process one file name at a time.

There was only one caveat to the portability of the Linux code. Most of the structures used in the file system (inode, super_block, and file) include a private field into which file system specific opaque data could be placed. We used this field to store information pertinent for stacking. We had to add a private field to only one structure which was missing it, the vm_area_struct. This structure represents custom per-process virtual memory manager page-fault handlers. Since Wrapfs is the first fully stackable file system for Linux, we feel that these changes are small and acceptable, given that more stackable file systems are likely to be developed.[3]

[2] http://www.kvack.org/~blah/lofs/

[3] We are currently working with the maintainers to include stackable file system support in a future version of Linux.

B.2 FreeBSD

FreeBSD 3.0 is based on BSD-4.4Lite. We chose it as the third port because it represents another major section of Unix operating systems—the BSD ones. FreeBSD's vnode interface is very similar to Solaris's and the port was straightforward. FreeBSD's version of the loopback file system is called *nullfs*[15], a useful template for writing stackable file systems. Unfortunately, ever since the merging of the VM and Buffer Cache in FreeBSD 3.0, stackable file systems stopped working because of the inability of the VFS to correctly map data pages of stackable file systems to their on-disk locations. We had to work around these deficiencies in nullfs by forcing all writes to be synchronous and by implementing `getpages` and `putpages` using `read` and `write`, respectively.

References

[1] M. Accetta, R. Baron, W. Bolosky, D. Golub, R. Rashid, A. Tevanian, and M. Young. Mach: A New Kernel Foundation for UNIX Development. *USENIX Conference Proceedings* (Atlanta, GA), pages 93–112. USENIX, Summer 1986.

[2] M. Blaze. A Cryptographic File System for Unix. *Proceedings of the first ACM Conference on Computer and Communications Security* (Fairfax, VA). ACM, November, 1993.

[3] M. I. Bushnell. The HURD: Towards a New Strategy of OS Design. *GNU's Bulletin*. Free Software Foundation, January 1994. Copies are available by writing to `gnu@prep.ai.mit.edu`.

[4] G. Cattaneo and G. Persiano. Design and Implementation of a Transparent Cryptographic File System for Unix. Unpublished Technical Report. Dip. Informatica ed Appl, Università di Salerno, 8 July 1997. Available via ftp in ftp://edu-gw.dia.unisa.it/pub/tcfs/docs/tcfs.ps.gz.

[5] P. Gutmann. Secure FileSystem (SFS) for DOS/Windows. Online Documentation. September 1996. Available via the WWW in `http://www.cs.auckland.ac.nz/~pgut001/sfs/`.

[6] R. G. Guy, J. S. Heidemann, W. Mak, T. W. Page Jr., G. J. Popek, and D. Rothmeier. Implementation of the Ficus replicated file system. *USENIX Conference Proceedings*, pages 63–71. USENIX, June 1990.

[7] J. Heidemann and G. Popek. Performance of cache coherence in stackable filing. *Fifteenth ACM Symposium on Operating Systems Principles* (Copper Mountain Resort, CO). Association for Computing Machinery SIGOPS, 3–6 December 1995.

[8] J. S. Heidemann and G. J. Popek. File System Development with Stackable Layers. *Transactions on Computing Systems*, **12**(1):58–89. (New York, New York), ACM, February, 1994.

[9] J. S. Heidemann and G. J. Popek. A layered approach to file system development. Technical report CSD-910007. University of California, Los Angeles, March 1991.

[10] M. Luby and C. W. Rackoff. How to construct pseudo-random permutations from pseudo-random functions. *SIAM Journal on Computing*, **17**(2):373–86, April 1988.

[11] S. Lucks. Faster Luby-Rackoff Ciphers. In *Fast Software Encryption*, pages 189–203. Springer LNCS 1039, 1996. Available via the WWW in `http://th.informatik.uni-mannheim.de/People/Lucks/papers.html`.

[12] M. K. McKusick, K. Bostic, M. J. Karels, and J. S. Quarterman. The Berkeley Fast Filesystem. In *The Design and Implementation of the 4.4BSD Operating System*, pages 269–84. Addison-Wesley, May 1996.

[13] J. G. Mitchell, J. J. Gibbons, G. Hamilton, P. B. Kessler, Y. A. Khalidi, P. Kougiouris, P. W. Madany, M. N. Nelson, M. L. Powell, and S. R. Radia. An Overview of the Spring System. *CompCon Conference Proceedings* (San Francisco, California). CompCon, 1994.

[14] R. Nagar. Filter Drivers. In *Windows NT File System Internals: A developer's Guide*, pages 615–67. O'Reilly, 1997.

[15] J.-S. Pendry and M. K. McKusick. Union mounts in 4.4BSD-Lite. *Conference Proceedings of the USENIX 1995 Technical Conference on UNIX and Advanced Computing Systems* (New Orleans), pages 25–33. Usenix Association, 16–20 January 1995.

[16] J.-S. Pendry and N. Williams. Amd – The 4.4 BSD Automounter. User Manual, edition 5.3 alpha. Imperial College of Science, Technology, and Medicine, London, England, March 1991.

[17] D. S. H. Rosenthal. Requirements for a "Stacking" Vnode/VFS Interface. Unix International document SD-01-02-N014. UNIX International, 1992.

[18] D. S. H. Rosenthal. Evolving the Vnode Interface. *USENIX Conference Proceedings* (Anaheim, CA), pages 107–18. USENIX, Summer 1990.

[19] J. H. Saltzer and M. D. Schroeder. The protection of information in computer systems. *Proceedings of IEEE*, **63**(9):1278–308, September 1975.

[20] R. Sandberg, D. Goldberg, S. Kleiman, D. Walsh, and B. Lyon. Design and implementation of the Sun Network Filesystem. *USENIX Association Summer Conference Proceedings of 1985* (11-14 June 1985, Portland, OR), pages 119–30. USENIX Association, El Cerrito, CA, 1985.

[21] B. Schneier. Algorithm Types and Modes. In *Applied Cryptography, Second Edition*, pages 189–97. John Wiley & Sons, 1996.

[22] B. Schneier. Blowfish. In *Applied Cryptography, Second Edition*, pages 336–9. John Wiley & Sons, 1996.

[23] G. C. Skinner and T. K. Wong. "Stacking" Vnodes: A Progress Report. *USENIX Conference Proceedings* (Cincinnati, OH), pages 161–74. USENIX, Summer 1993.

[24] SMCC. acl(2). SunOS 5.6 Reference Manual, Section 2. Sun Microsystems, Incorporated, 18 March 1996.

[25] SMCC. SunSHIELD Basic Security Module Guide. Solaris 2.6 System Administrator Collection Vol 1. Sun Microsystems, Incorporated, 1997. Available via http://docs.sun.com/ab2/coll.47.4/SHIELD/@Ab2TocView?

[26] SMCC. lofs – loopback virtual file system. SunOS 5.5.1 Reference Manual, Section 7. Sun Microsystems, Incorporated, 20 March 1992.

[27] M. E. Smid and D. K. Branstad. The Data Encryption Standard: Past and future. *IEEEPROC.*, **76**:550–9, 1988.

[28] T. Ts'o, W. Almesberger, E. Young, and M. DSouza. losetup(8). RedHat Linux 6.1 Reference Manual, Section 8. RedHat Systems, Incorporated, 24 November 1993.

[29] E. Zadok. Am-utils (4.4BSD Automounter Utilities). User Manual, for Am-utils version 6.0a16. Columbia University, 22 April 1998. Available http://www.cs.columbia.edu/~ezk/am-utils/.

[30] E. Zadok. *FiST: A File System Component Compiler*. PhD thesis, published as Technical Report CUCS-033-97 (Ph.D. Thesis Proposal). Computer Science Department, Columbia University, 27 April 1997. Available http://www.cs.columbia.edu/~library/.

[31] E. Zadok, I. Badulescu, and A. Shender. Cryptfs: A Stackable Vnode Level Encryption File System. Technical Report CUCS-021-98. Computer Science Department, Columbia University, 28 July 1998. Available http://www.cs.columbia.edu/~library/.

[32] E. Zadok, I. Badulescu, and A. Shender. Extending File Systems Using Stackable Templates. *USENIX Conference Proceedings* (Monterey, California). USENIX, 6-11 June 1999.

C Author Information

Erez Zadok is an Ph.D. candidate in the Computer Science Department at Columbia University. His primary interests include operating systems and file systems. The work described in this paper was first mentioned in his Ph.D. thesis proposal[30]. To access source code for the file systems described in this paper see *http://www.cs.columbia.edu/~ezk/research/s*